

ELSEVIER

Abraham
SILBERSCHATZ

Peter Baer
GALVIN

Greg
GAGNE

SISTEMAS OPERACIONAIS **COM JAVA**



8^a Edição

SISTEMAS OPERACIONAIS COM JAVA

8^a EDIÇÃO

ABRAHAM SILBERSCHATZ

PETER BAER GALVIN

GREG GAGNE

ELSEVIER

SISTEMAS OPERACIONAIS COM JAVA

8^a EDIÇÃO

ABRAHAM SILBERSCHATZ

PETER BAER GALVIN

GREG GAGNE

ELSEVIER

Sumário

Capa

Folha de rosto

Caderno zero

direitos autorais

Dedicatória

Os autores

Prefácio

Agradecimentos

PARTE I: VISÃO GERAL

Introdução a Visão geral

Capítulo 1: Introdução

- 1.1 O que os sistemas operacionais fazem
 - 1.2 Organização do computador
 - 1.3 Arquitetura do sistema computadorizado
 - 1.4 Estrutura do sistema operacional
 - 1.5 Operações do sistema operacional
 - 1.6 Gerência de processos
 - 1.7 Gerência de memória
 - 1.8 Gerência de armazenamento
 - 1.9 Proteção e segurança
 - 1.10 Sistemas distribuídos
 - 1.11 Sistemas de uso específico
 - 1.12 Ambientes de computação
 - 1.13 Sistemas operacionais de fonte aberto
 - 1.14 Resumo
- Exercícios práticos
- Exercícios
- Notas bibliográficas

Capítulo 2: Estruturas do sistema operacional

- 2.1 Serviços do sistema operacional

2.2 Interface usuário-sistema operacional
2.3 Chamadas de sistema
2.4 Tipos de chamadas de sistema
2.5 Programas do sistema
2.6 Projeto e implementação do sistema operacional
2.7 Estrutura do sistema operacional
2.8 Máquinas virtuais
2.9 Java
2.10 Depuração do sistema operacional
2.11 Geração do sistema operacional
2.12 Boot do sistema
2.13 Resumo
Exercícios práticos
Exercícios
Problemas de programação
Projetos de programação
Notas bibliográficas

PARTE II: GERÊNCIA DE PROCESSOS

Introdução a Gerência de processos

Capítulo 3: Processos

3.1 Conceito de processo
3.2 Escalonamento de processos
3.3 Operações sobre processos
3.4 Comunicação entre processos
3.5 Exemplos de sistemas de IPC
3.6 Comunicação em sistemas cliente-servidor
3.7 Resumo
Exercícios práticos
Exercícios
Problemas de programação
Projetos de programação
Visão geral
Parte 1: Criando um processo externo
Parte 2: Mudando diretórios
Parte 3: Incluindo um recurso de histórico
Notas bibliográficas

Capítulo 4: Threads

4.1 Visão geral
4.2 Modelos de múltiplas threads (multithreading)
4.3 Bibliotecas de threads
4.4 Threads em Java
4.5 Aspectos do uso de threads
4.6 Exemplos em sistemas operacionais
4.7 Resumo

[Exercícios práticos](#)
[Exercícios](#)
[Problemas de programação](#)
[Projetos de programação](#)
[Projeto 1: Projeto de serviço de nomes](#)
[Projeto 2: Multiplicação de matriz](#)
[Passando parâmetros a cada thread](#)
[Esperando que as threads terminem](#)
[Notas bibliográficas](#)

Capítulo 5: Escalonamento de CPU

[5.1 Conceitos básicos](#)
[5.2 Critérios de escalonamento](#)
[5.3 Algoritmos de escalonamento](#)
[5.4 Escalonamento de thread](#)
[5.5 Escalonamento em múltiplos processadores](#)
[5.6 Exemplos de sistema operacional](#)
[5.7 Escalonamento em Java](#)
[5.8 Avaliação de algoritmo](#)
[5.9 Resumo](#)
[Exercícios práticos](#)
[Exercícios](#)
[Problemas de programação](#)
[Notas bibliográficas](#)

Capítulo 6: Sincronismo de processos

[6.1 Fundamentos](#)
[6.2 O problema da seção crítica](#)
[6.3 Solução de Peterson](#)
[6.4 Hardware de sincronismo](#)
[6.5 Semáforos](#)
[6.6 Problemas clássicos de sincronismo](#)
[6.7 Monitores](#)
[6.8 Sincronismo em Java](#)
[6.9 Exemplos de sincronismo](#)
[6.10 Transações atômicas](#)
[6.11 Resumo](#)
[Exercícios práticos](#)
[Exercícios](#)
[Problemas de programação](#)
[Projetos de programação](#)

Capítulo 7: Deadlocks

[7.1 Modelo do sistema](#)
[7.2 Caracterização do deadlock](#)
[7.3 Métodos para tratamento de deadlocks](#)
[7.4 Prevenção de deadlock](#)
[7.5 Evitar deadlock](#)
[7.6 Detecção de deadlock](#)

[7.7 Recuperação do deadlock](#)

[7.8 Resumo](#)

[Exercícios práticos](#)

[Exercícios](#)

[Notas bibliográficas](#)

PARTE III: GERÊNCIA DE MEMÓRIA

[Introdução a Gerência de memória](#)

[Capítulo 8: Memória principal](#)

[8.1 Conceitos básicos](#)

[8.2 Swapping](#)

[8.3 Alocação de memória contígua](#)

[8.4 Paginação](#)

[8.5 Estrutura da tabela de página](#)

[8.6 Segmentação](#)

[8.7 Exemplo: Intel Pentium](#)

[8.8 Resumo](#)

[Exercícios práticos](#)

[Exercícios](#)

[Problemas de programação](#)

[Notas bibliográficas](#)

[Capítulo 9: Memória virtual](#)

[9.1 Aspectos básicos](#)

[9.2 Paginação por demanda](#)

[9.3 Cópia na escrita](#)

[9.4 Substituição de página](#)

[9.5 Alocação de quadros](#)

[9.6 Thrashing](#)

[9.7 Arquivos mapeados na memória](#)

[9.8 Alocando memória do kernel](#)

[9.9 Outras considerações para sistemas de paginação](#)

[9.10 Exemplos de sistema operacional](#)

[9.11 Resumo](#)

[Exercícios práticos](#)

[Exercícios](#)

[Problemas de programação](#)

[Projetos de programação](#)

[Tradução de endereços](#)

[Tratamento das falhas de página](#)

[Notas bibliográficas](#)

PARTE IV: GERÊNCIA DE ARMAZENAMENTO

[Introdução a Gerência de armazenamento](#)

Capítulo 10: Interface do sistema de arquivos

10.1 Conceito de arquivo

10.2 Métodos de acesso

10.3 Estrutura de diretório e disco

10.4 Montagem do sistema de arquivos

10.5 Compartilhamento de arquivos

10.6 Proteção

10.7 Resumo

Exercícios práticos

Exercícios

Notas bibliográficas

Capítulo 11: Implementação do sistema de arquivos

11.1 Estrutura do sistema de arquivos

11.2 Implementação do sistema de arquivos

11.3 Implementação do diretório

11.4 Métodos de alocação

11.5 Gerenciamento do espaço livre

11.6 Eficiência e desempenho

11.7 Recuperação

11.8 NFS

11.9 Exemplo: O sistema de arquivos WAFL

11.10 Resumo

Exercícios práticos

Exercícios

Projetos de programação

Notas bibliográficas

Capítulo 12: Estrutura de armazenamento em massa

12.1 Visão geral da estrutura de armazenamento em massa

12.2 Estrutura do disco

12.3 Conexão de disco

12.4 Escalonamento de disco

12.5 Gerenciamento de disco

12.6 Gerenciamento do swap space

12.7 Estrutura RAID

12.8 Implementação do armazenamento estável

12.9 Estrutura do armazenamento terciário

12.10 Resumo

Exercícios práticos

Exercícios

Problemas de programação

Notas bibliográficas

Capítulo 13: Sistemas de E/S

13.1 Visão geral

13.2 Hardware de E/S

13.3 Interface de E/S da aplicação

- 13.4 Subsistema de E/S do kernel
- 13.5 Transformando requisições de E/S em operações de hardware
- 13.6 STREAMS
- 13.7 Desempenho
- 13.8 Resumo
- Exercícios práticos
- Exercícios
- Notas bibliográficas

PARTE V: PROTEÇÃO E SEGURANÇA

Introdução a Proteção e segurança

Capítulo 14: Proteção

- 14.1 Objetivos da proteção
- 14.2 Princípios de proteção
- 14.3 Domínio de proteção
- 14.4 Matriz de acesso
- 14.5 Implementação da matriz de acesso
- 14.6 Controle de acesso
- 14.7 Revogação de direitos de acesso
- 14.8 Sistemas baseados em capacidade
- 14.9 Proteção baseada na linguagem
- 14.10 Resumo
- Exercícios práticos
- Exercícios
- Notas bibliográficas

Capítulo 15: Segurança

- 15.1 O problema da segurança
- 15.2 Ameaças ao programa
- 15.3 Ameaças ao sistema e à rede
- 15.4 Criptografia como ferramenta de segurança
- 15.5 Autenticação do usuário
- 15.6 Implementando defesas de segurança
- 15.7 Uso de firewalls para proteger sistemas e redes
- 15.8 Classificações de segurança de computador
- 15.9 Um exemplo: Windows XP
- 15.10 Resumo
- Exercícios
- Notas bibliográficas

PARTE VI: SISTEMAS DISTRIBUÍDOS

Introdução a Sistemas distribuídos

Capítulo 16: Estruturas de sistemas distribuídos

16.1 Motivação

16.2 Tipos de sistemas operacionais em rede

16.3 Estrutura de rede

16.4 Topologia de rede

16.5 Estrutura de comunicação

16.6 Protocolos de comunicação

16.7 Robustez

16.8 Aspectos de projeto

16.9 Um exemplo: redes

16.10 Resumo

Exercícios práticos

Exercícios

Projetos de programação

Notas bibliográficas

Capítulo 17: Sistemas de arquivos distribuídos

17.1 Aspectos básicos

17.2 Nomeação e transparência

17.3 Acesso a arquivo remoto

17.4 Serviço Stateful e serviço Stateless

17.5 Replicação de arquivos

17.6 Um exemplo: AFS

17.6.4 Implementação

17.7 Resumo

Exercícios

Notas bibliográficas

Capítulo 18: Coordenação distribuída

18.1 Ordenação de eventos

18.2 Exclusão mútua

18.3 Atomicidade

18.4 Controle de concorrência

18.5 Tratamento de deadlock

18.6 Algoritmos de eleição

18.7 Chegando ao acordo

18.8 Resumo

Exercícios

Notas bibliográficas

PARTE VII: SISTEMAS DE USO GERAL

Introdução a Sistemas de uso geral

Capítulo 19: Sistemas de tempo real

19.1 Aspectos básicos

19.2 Características do sistema

19.3 Recursos dos kernels de tempo real

19.4 Implementando sistemas operacionais de tempo real

[19.5 Escalonamento de CPU em tempo real](#)

[19.6 Um exemplo: VxWorks 5.x](#)

[19.7 Resumo](#)

[Exercícios](#)

[Notas bibliográficas](#)

Capítulo 20: Sistemas multimídia

[20.1 O que é multimídia?](#)

[20.2 Compactação](#)

[20.3 Requisitos dos kernels de multimídia](#)

[20.4 Escalonamento de CPU](#)

[20.5 Escalonamento de disco](#)

[20.6 Gerenciamento de rede](#)

[20.7 Um exemplo: CineBlitz](#)

[20.8 Resumo](#)

[Exercícios](#)

[Notas bibliográficas](#)

PARTE VIII: ESTUDOS DE CASO

Introdução a Estudos de caso

Capítulo 21: O sistema Linux

[21.1 História do Linux](#)

[21.2 Princípios de projeto](#)

[21.3 Módulos do kernel](#)

[21.4 Gerência de processos](#)

[21.5 Escalonamento](#)

[21.6 Gerência de memória](#)

[21.7 Sistemas de arquivos](#)

[21.8 Entrada e saída](#)

[21.9 Comunicação entre processos](#)

[21.10 Estrutura de rede](#)

[21.11 Segurança](#)

[21.12 Resumo](#)

[Exercícios práticos](#)

[Exercícios](#)

[Notas bibliográficas](#)

Capítulo 22: Windows XP

[22.1 História](#)

[22.2 Princípios de projeto](#)

[22.3 Componentes do sistema](#)

[22.4 Subsistemas de ambiente](#)

[22.5 Sistema de arquivos](#)

[22.6 Redes](#)

[22.6.5 Domínios](#)

[22.6.6 Active Directory](#)

[22.7 Interface do programador](#)

[22.8 Resumo](#)

[Exercícios práticos](#)

[Exercícios](#)

[Notas bibliográficas](#)

Capítulo 23: Sistemas operacionais marcantes

[23.1 Migração de recursos](#)

[23.2 Primeiros sistemas](#)

[23.3 Atlas](#)

[23.4 XDS-940](#)

[23.5 THE](#)

[23.6 RC 4000](#)

[23.7 CTSS](#)

[23.8 MULTICS](#)

[23.9 IBM OS/360](#)

[23.10 TOPS-20](#)

[23.11 CP/M e MS-DOS](#)

[23.12 Macintosh Operating System e Windows](#)

[23.13 Mach](#)

[23.14 Outros sistemas](#)

[Exercícios](#)

Bibliografia

Índice

Caderno zero

SISTEMAS OPERACIONAIS COM JAVA

8^a Edição

ABRAHAM SILBERSCHATZ

PETER BAER GALVIN

GREG GAGNE

TRADUÇÃO

Daniel Vieira

Presidente da Multinet Informática

Programador e tradutor especializado em Informática

REVISÃO TÉCNICA

Sérgio Guedes de Souza

Pesquisador - Núcleo de Computação Eletrônica - NCE - UFRJ

Professor Colaborador - Departamento de Ciência da Computação

Instituto de Matemática - DCC/IM - UFRJ

ELSEVIER



CAMPUS

direitos autorais

Do original: *Operating System with Java Concepts*

Tradução autorizada do idioma inglês da edição publicada por John Wiley & Sons, Inc.

Copyright © 2010 John Wiley & Sons, Inc. All rights reserved.

© 2016, Elsevier Editora Ltda.

Todos os direitos reservados e protegidos pela Lei nº 9.610, de 19/02/1998.

Nenhuma parte deste livro, sem autorização prévia por escrito da editora, poderá ser reproduzida ou transmitida sejam quais forem os meios empregados: eletrônicos, mecânicos, fotográficos, gravação ou quaisquer outros.

Copidesque: Edna da Silva Cavalcanti

Revisão: Carla Camargo Martins

Editoração Eletrônica: Estúdio Castellani

Elsevier Editora Ltda.

Conhecimento sem Fronteiras

Rua Sete de Setembro, 111 - 16º andar
20050-006 - Centro - Rio de Janeiro - RJ - Brasil

Rua Quintana, 753 - 8º andar
04569-011 - Brooklin - São Paulo - SP - Brasil
Serviço de Atendimento ao Cliente
0800-0265340

atendimento1@elsevier.com

ISBN 978-85-352-8367-9

ISBN (versão digital): 978-85-352-8368-6

ISBN (edição original): 978-0-470-50949-4

Nota

Muito zelo e técnica foram empregados na edição desta obra. No entanto, podem ocorrer erros de digitação, impressão ou dúvida conceitual. Em qualquer das hipóteses, solicitamos a comunicação ao nosso Serviço de Atendimento ao Cliente, para que possamos esclarecer ou encaminhar a questão.

Nem a editora nem o autor assumem qualquer responsabilidade por eventuais danos ou perdas a pessoas ou bens, originados do uso desta publicação.



CIP-Brasil. Catalogação na Publicação
Sindicato Nacional dos Editores de Livros, RJ

S576s Silberschatz, Abraham
8. ed. Sistemas operacionais com Java/Abraham
Silberschatz, Peter Baer Galvin, Greg Gagne; tradução
Daniel Vieira; revisor técnico Sérgio Guedes de Souza.
- 8. ed. - Rio de Janeiro: Elsevier, 2016.

il.; 28 cm.

Tradução de: Operating system concepts

Inclui índice

ISBN 978-85-352-8367-9

1. Java (Linguagem de programação de computador). 2. Software de aplicação

- Desenvolvimento. I. Galvin, Peter Baer. II. Gagne, Greg. III. Título.
15-28270 CDD: 005.133
CDU: 004.43

Dedicatória

À minha Valerie

Avi Silberschatz

Aos meus pais, Brendan e Ellen Galvin

Peter Baer Galvin

A Steve, Ray e Bobbie

Greg Gagne

Os autores

ABRAHAM SILBERSCHATZ é professor da cátedra Sidney J. Weinberg e diretor do Departamento de Ciência da Computação na Universidade de Yale. Antes de entrar para Yale, foi vice-presidente do Information Sciences Research Center na Bell Laboratories, e professor e diretor do Departamento de Ciência da Computação na Universidade do Texas, em Austin.

É membro da ACM e IEEE, e da Connecticut Academy of Science and Engineering. Recebeu o IEEE Taylor L. Booth Education Award de 2002, o ACM Karl V. Karlstrom Outstanding Educator Award de 1998 e o ACM SIGMOD Contribution Award de 1997. Em reconhecimento ao seu fantástico nível de inovação e excelência técnica, recebeu também o prêmio do presidente da Bell Laboratories por três diferentes projetos - o QTM Project (1998), o DataBlitz Project (1999) e o NetInventory Project (2004).

Os textos do Professor Silberschatz fizeram parte de diversas publicações da ACM e IEEE, além de outras conferências e periódicos profissionais. É coautor do livro *Conceitos de sistemas de banco de dados*. Também escreveu artigos para *New York Times*, *Boston Globe* e *Hartford Courant*, dentre outros.

PETER BAER GALVIN é tecnólogo chefe da Corporate Technologies (www.cp.tech.com), uma empresa de revenda e integração de instalações de computação. Foi gerente de sistemas no Departamento de Ciência da Computação da Brown University. Também é colunista da Sun para a revista; *login*; e escreveu artigos para a *Byte* e outras revistas, e colunas para a *SunWorld* e a *SysAdmin*. Como consultor e treinador, tem realizado palestras e tutoriais de ensino sobre segurança e administração de sistemas em todo o mundo.

GREG GAGNE é diretor do Departamento de Ciência da Computação no Westminster College, em Salt Lake City, onde leciona desde 1990. Além de ensinar sobre sistemas operacionais, também ministra cursos sobre redes de computadores, sistemas distribuídos e engenharia de software, e realiza workshops para educadores de Ciência da Computação e profissionais do setor.

Prefácio

Os sistemas operacionais são uma parte essencial de qualquer computador. Da mesma forma, a disciplina de sistemas operacionais é obrigatória em qualquer curso de ciência da computação. Essa área está passando por rápidas mudanças, uma vez que os computadores agora estão presentes praticamente em cada aplicação, desde os aplicativos para crianças em celulares até as ferramentas de planejamento mais sofisticadas para governos e empresas multinacionais. Mesmo assim, os conceitos fundamentais continuam sendo bastante claros, e é nesta tese que fundamentamos este livro.

Escrevemos este livro como texto para curso introdutório em sistemas operacionais em nível de graduação básico ou profissional, ou no nível de primeiro ano de pós-graduação. É bem provável que também seja útil para profissionais. Ele provê uma descrição clara dos *conceitos* em que se baseiam os sistemas operacionais. Como pré-requisitos, consideramos que o leitor esteja familiarizado com as estruturas de dados básicas, organização do computador e linguagem de alto nível, preferivelmente Java. Os tópicos de hardware exigidos para o conhecimento dos sistemas operacionais estão incluídos no [Capítulo 1](#). Para os exemplos de código, usamos principalmente Java, com um pouco de C, mas o leitor conseguirá entender os algoritmos sem precisar ter conhecimento profundo dessas linguagens.

Os conceitos são apresentados por meio de descrições intuitivas. São abordados resultados teóricos importantes, mas as provas formais são omitidas. As notas bibliográficas contêm indicações para trabalhos de pesquisa, nos quais os resultados foram inicialmente apresentados e comprovados, além de referências a materiais para leitura adicional. No lugar de provas, figuras e exemplos são usados para sugerir por que devemos esperar que o resultado em questão seja verdadeiro.

Os conceitos e algoritmos fundamentais abordados no livro normalmente têm como base aqueles usados em sistemas operacionais comerciais existentes. Nossa objetivo é apresentar esses conceitos e algoritmos em um ambiente geral que não esteja ligado a qualquer sistema operacional específico. Apresentamos grande quantidade de exemplos que pertencem aos sistemas operacionais mais populares e inovadores, incluindo o Solaris, da Sun Microsystems; o Linux; o Microsoft Windows Vista, o Windows 2000 e o Windows XP; e o Apple Mac OS X. Quando nos referirmos ao Windows XP como exemplo de sistema operacional, estaremos considerando o Windows Vista, o Windows XP e o Windows 2000. Quando houver um recurso específico de um sistema, indicaremos isso explicitamente.

A organização deste livro

A organização deste livro reflete nossos muitos anos de ensino de sistemas operacionais. Também demos atenção ao retorno fornecido pelos revisores do texto, além dos comentários submetidos pelos leitores das edições anteriores. Além disso, o conteúdo do texto corresponde às sugestões dos *Curriculos de Computação de 2005* para o ensino sobre sistemas operacionais, publicado pela força-tarefa conjunta da IEEE Computing Society e da Association for Computing Machinery (ACM).

O conteúdo deste livro

O texto está organizado em oito partes principais:

- **Visão geral.** Os [Capítulos 1 e 2](#) explicam o que são sistemas operacionais, o que eles fazem e como são projetados e construídos. E discutem quais são seus recursos comuns, o que ele faz para o usuário e o que faz para o operador do sistema computadorizado. A apresentação é motivadora e explicativa por natureza. Nesses capítulos, evitamos uma discussão sobre como as coisas funcionam internamente. Portanto, são adequados para indivíduos ou alunos em classes de nível mais básico, que querem aprender o que é um sistema operacional, mas sem entrar nos detalhes dos algoritmos internos.
- **Gerência de processos.** Os [Capítulo 3 a 7](#) [Capítulo 4](#) [Capítulo 5](#) [Capítulo 6](#) [Capítulo 7](#) descrevem o conceito de processo e a concorrência como o núcleo dos sistemas operacionais modernos. Um *processo* é a unidade de trabalho em um sistema. Tal sistema consiste em uma coleção de processos em execução *concorrente*, alguns deles, processos do sistema operacional (aqueles que executam código do sistema) e o restante, processos do usuário (aqueles que executam código do usuário). Esses capítulos abordam os métodos para escalonamento de processos, comunicação entre processos, sincronismo de processos e tratamento deadlock. Também incluem uma discussão sobre threads, bem como uma análise de questões relacionadas com sistemas multicore.
- **Gerência de memória.** Os [Capítulos 8 e 9](#) tratam da gerência da memória principal durante a execução de um processo. Para melhorar a utilização da CPU e a velocidade da resposta aos seus usuários, o computador precisa manter vários processos na memória. Existem diversos esquemas de gerenciamento de memória que refletem diversas técnicas de gerência de memória, e a eficácia de determinado algoritmo depende da situação.
- **Gerência de armazenamento.** Os [Capítulos 10 a 13](#) [Capítulo 11](#) [Capítulo 12](#) [Capítulo 13](#) descrevem como o sistema de arquivos, o armazenamento em massa e a E/S são tratados em um computador moderno. O sistema de arquivos provê o mecanismo para o armazenamento on-line e o acesso a dados e programas residentes nos discos. Esses capítulos descrevem os algoritmos internos clássicos e as estruturas da gerência de armazenamento. Eles fornecem um conhecimento prático firme dos algoritmos utilizados – as propriedades, vantagens e desvantagens. Como os dispositivos de E/S que se conectam a um computador variam bastante, o sistema operacional precisa prover uma grande gama de funcionalidade às aplicações, para permitir que elas controlem todos os aspectos dos dispositivos. Discutimos a E/S do sistema com profundidade, incluindo projeto do sistema de E/S, interfaces e estruturas e funções internas do sistema. De várias maneiras, os dispositivos de E/S também são os componentes principais mais lentos do computador. Por serem um gargalo no desempenho, as questões de desempenho são examinadas.
- **Proteção e segurança.** Os [Capítulos 14 e 15](#) discutem sobre os mecanismos necessários para a proteção e a segurança dos sistemas de computação. Os processos em um sistema operacional que precisam ser protegidos das atividades uns dos outros. Para fins de proteção e segurança, usamos mecanismos que garantem que apenas os processos que obtiveram autorização própria do sistema operacional podem operar sobre os arquivos, segmentos de memória, CPU e outros recursos do sistema. Proteção é o mecanismo que controla o acesso dos programas, processos ou usuários aos recursos definidos para um sistema computadorizado. Esse mecanismo precisa prover um meio de especificar os controles a serem impostos, bem como um meio de imposição. A segurança protege a integridade das informações armazenadas no sistema (dados e código), além dos recursos físicos do sistema computadorizado, contra acesso não autorizado, destruição ou alteração maliciosa e introdução accidental de incoerência.
- **Sistemas distribuídos.** Os [Capítulos 16 a 18](#) [Capítulo 17](#) [Capítulo 18](#) tratam de uma coleção de processadores que não compartilham a memória ou um relógio – um *sistema distribuído*. Fornecendo ao usuário o acesso aos diversos recursos que ele mantém, um sistema distribuído pode melhorar a velocidade de computação e a disponibilidade e a confiabilidade dos dados. Tal sistema também fornece ao usuário um sistema de arquivos distribuído, que é o sistema de serviço de arquivo cujos usuários, servidores e dispositivos de armazenamento estão dispersos entre os locais de um sistema distribuído. Um sistema distribuído precisa prover diversos mecanismos para o sincronismo e a comunicação entre os processos, além de lidar com o problema de deadlock e uma série de falhas que não existem em um sistema centralizado.
- **Sistemas de uso geral.** Os [Capítulos 19 e 20](#) lidam com sistemas utilizados para finalidades específicas, incluindo sistemas de tempo real e sistemas de multimídia. Esses sistemas possuem requisitos específicos que diferem dos sistemas de uso geral, que são o foco do restante do texto. Os sistemas de tempo real podem exigir não apenas que os resultados calculados estejam “corretos”, mas também que os resultados sejam produzidos dentro de um prazo especificado. Os sistemas de multimídia exigem garantias de qualidade de serviço que assegurem que os dados de multimídia sejam entregues aos clientes dentro de um período específico.
- **Estudos de caso.** Os [Capítulos 21 a 23](#) [Capítulo 22](#) [Capítulo 23](#) integram os conceitos abordados

neste livro, descrevendo sistemas operacionais reais. Esses sistemas incluem Linux, Windows XP, FreeBSD, Mach e Windows 2000. Escolhemos Linux e FreeBSD porque o UNIX - em uma época - era pequeno o suficiente para ser compreendido, embora não fosse um sistema operacional de "brinquedo". A maior parte dos seus algoritmos internos foi selecionada por *simplicidade*, em vez de velocidade ou sofisticação. Tanto o Linux quanto o FreeBSD estão prontamente disponíveis nos departamentos de ciência da computação, de modo que muitos alunos têm acesso a eles. Escolhemos Windows XP e Windows 2000 porque oferecem uma oportunidade de estudar um sistema operacional moderno, com design e implementação muito diferentes do que existe no mundo do UNIX. O [Capítulo 23](#) descreve rapidamente alguns outros sistemas operacionais marcantes.

Ambientes de sistema operacional

Este livro utiliza exemplos de muitos sistemas operacionais do mundo real para ilustrar conceitos fundamentais do sistema operacional. Porém, deve-se prestar uma atenção particular à família Microsoft de sistemas operacionais (incluindo Windows Vista, Windows 2000 e Windows XP) e diversas versões do UNIX (incluindo Solaris, BSD e Mac OS X). Este livro utiliza a Java para ilustrar muitos conceitos do sistema operacional, como multithreading, escalonamento de CPU, sincronismo de processos, deadlock, gerência de memória e arquivo, segurança, redes e sistemas distribuídos. Java é mais uma tecnologia do que uma linguagem de programação, e por isso é um excelente veículo para demonstrações.

Grande parte do material relacionado com Java neste texto foi desenvolvida e testada nas aulas de sistemas operacionais nos cursos de graduação. Segundo nossa experiência, os alunos que participam dessas aulas e que não possuem conhecimento de Java - mas possuem experiência com o uso de C++ e princípios básicos de orientação a objetos - não costumam ter muito trabalho com Java. Na verdade, a maior parte das dificuldades está em entender conceitos como multithreading e compartilhamento de dados por vários threads concorrentes. Esses conceitos são sistemáticos e não específicos à Java; até mesmo alunos com bom conhecimento de Java provavelmente terão alguma dificuldade com eles. Portanto, enfatizamos a concorrência e a passagem de referências de objeto para vários threads, em vez de nos concentrarmos na sintaxe. Todos os programas em Java neste texto foram compilados com o Java Software Development Kit (SDK), versões 1.5 e 1.6. O Java 1.5 fornece vários recursos novos, tanto no nível de linguagem quanto no nível de API, que melhoraram a utilidade da linguagem para estudar os sistemas operacionais. Fizemos vários novos acréscimos à API Java 1.5 no decorrer deste texto. Muitos programas incluídos neste texto não serão compilados com versões anteriores do Java SDK; aconselhamos todos os leitores a usarem Java 1.5 como configuração mínima para a Java.

O texto também oferece alguns programas de exemplo escritos em C, cujo propósito é que sejam executados nos ambientes de programação Windows e POSIX. POSIX (*Portable Operating System Interface* - interface portável de sistema operacional) representa um conjunto de padrões implementados principalmente para sistemas operacionais baseados em UNIX. Embora sistemas Windows Vista, Windows XP e Windows 2000 também possam rodar certos programas POSIX, nosso estudo de POSIX focaliza principalmente sistemas UNIX e Linux.

A oitava edição

Enquanto escrevíamos esta oitava edição, fomos guiados pelos muitos comentários e sugestões que recebemos de leitores das nossas edições anteriores, além de nossas próprias observações sobre os campos em constante mudança dos sistemas operacionais e das redes. Reescrevemos o material da maior parte dos capítulos, atualizando a parte mais antiga e removendo aquilo que não era mais de interesse ou relevância.

Fizemos várias revisões e mudanças organizacionais em muitos dos capítulos. O mais importante é que acrescentamos a cobertura dos sistemas operacionais de fonte aberto no [Capítulo 1](#), e também incluímos mais exercícios práticos. A seguir, um esboço das principais mudanças nos diversos capítulos deste livro:

- O [Capítulo 1](#) foi expandido para incluir CPUs multicore, computadores em clusters e sistemas operacionais de fonte aberto.
- O [Capítulo 2](#) oferece uma cobertura bastante atualizada das máquinas virtuais, bem como CPUs multicore, carregador de boot GRUB e depuração do sistema operacional.
- O [Capítulo 4](#) acrescenta nova cobertura da programação para sistemas multicore e atualiza a cobertura dos estados de threads Java.
- O [Capítulo 5](#) acrescenta a cobertura do escalonamento de máquina virtual e multithreaded, e arquiteturas multicore. Ele também inclui novos recursos de escalonamento na Java 1.5.
- O [Capítulo 6](#) acrescenta uma discussão sobre locks (trancas) de exclusão mútua, inversão de prioridade e memória transacional.
- O [Capítulo 8](#) inclui uma discussão sobre NUMA.
- O [Capítulo 9](#) atualiza o exemplo do Solaris para incluir a gerência de memória do Solaris 10.
- O [Capítulo 10](#) foi atualizado com tecnologias e capacidades atuais.
- O [Capítulo 11](#) inclui uma descrição completa do sistema de arquivos ZFS da Sun e expande a explicação sobre volumes e diretórios.
- O [Capítulo 12](#) acrescenta o tratamento de iSCSI, volumes e pools ZFS.
- O [Capítulo 13](#) acrescenta a cobertura de PCIX, PCI Express e HyperTransport.
- O [Capítulo 16](#) acrescenta a cobertura das redes sem fio 802.11.
- O [Capítulo 21](#) foi atualizado para incluir a versão mais recente do kernel do Linux.
- O [Capítulo 23](#) aumenta a cobertura de computadores muito抗igos, bem como TOPS-20, CP/M, Windows e Mac OS original.

Exercícios e projetos de programação

Para enfatizar os conceitos apresentados no texto, acrescentamos ou modificamos 12 exercícios de programação e projetos usando Java. Em geral, *projetos* de programação são mais detalhados e exigem maior comprometimento de tempo do que os *exercícios* de programação. Esses exercícios e projetos enfatizam processos e comunicação entre processos, threads, sincronismo de processo, memória virtual, sistemas de arquivos e redes. Novos exercícios e projetos de programação incluem a implementação de programação de socket, o uso da chamada de método remoto (RMI) do Java, o trabalho com programação de classificação multithreaded, a listagem de threads na máquina virtual Java (JVM), o projeto de um sistema de gerência de identificador de processo e a gerência da memória virtual.

Agradecimentos

Este livro é derivado das edições anteriores, sendo que as três primeiras tiveram James Peterson como coautor. Outros que nos ajudaram com as edições anteriores foram Hamid Arabnia, Rida Bazzi, Randy Bentson, David Black, Joseph Boykin, Jeff Brumfield, Gael Buckley, Roy Campbell, P. C. Capon, John Carpenter, Gil Carrick, Thomas Casavant, Bart Childs, Ajoy Kumar Datta, Joe Deck, Sudarshan K. Dhall, Thomas Doeppner, Caleb Drake, M. Racsit Eskicioglu, Hans Flack, Robert Fowler, G. Scott Graham, Richard Guy, Max Hailperin, Rebecca Hartman, Wayne Hathaway, Christopher Haynes, Don Heller, Bruce Hillyer, Mark Holliday, Dean Hougen, Michael Huangs, Ahmed Kamel, Richard Kieburtz, Carol Kroll, Morty Kwestel, Thomas LeBlanc, John Leggett, Jerrold Leichter, Ted Leung, Gary Lippman, Carolyn Miller, Michael Molloy, Euripides Montagne, Yoichi Muraoka, Jim M.Ng, Banu Özden, Ed Posnak, Boris Putanec, Charles Qualline, John Quartermann, Mike Reiter, Gustavo Rodriguez-Rivera, Carolyn J. C. Schauble, Thomas P. Skinner, Yannis Smaragdakis, Jesse St. Laurent, John Stankovic, Adam Stauffer, Steven Stepanek, John Sterling, Hal Stern, Louis Stevens, Pete Thomas, David Umbaugh, Steve Vinoski, Tommy Wagner, Larry L. Wear, John Werth, James M. Westall, J. S. Weston e Yang Xiang.

Partes do [Capítulo 12](#) foram derivadas de um artigo de Hillyer e Silberschatz [1996]. Partes do [Capítulo 17](#) foram derivadas de um artigo de Levy e Silberschatz [1990]. O [Capítulo 21](#) foi derivado de um manuscrito não publicado de Stephen Tweedie. O [Capítulo 22](#) foi derivado de um manuscrito não publicado de Dave Probert, Cliff Martin e Avi Silberschatz. Mike Shapiro, Bryan Cantrill e Jim Mauro responderam a várias perguntas relativas ao Solaris. Jason Belcher ofereceu auxílio com os assuntos genéricos sobre Java. Josh Dees e Rob Reynolds contribuíram com o material sobre .NET da Microsoft.

Nossa editora executiva, Beth Golub, forneceu orientação especializada enquanto preparávamos esta edição. Ela foi auxiliada por Mike Berlin, que conduziu muitos detalhes deste projeto com tranquilidade. O editor de produção sênior, Ken Santor, foi responsável por lidar com todos os detalhes da produção. A ilustradora da capa foi Susan Cyr, e o projetista da capa foi Howard Grossman. Beverly Peavler realizou a revisão do manuscrito. A revisora autônoma foi Katrina Avery; a indexadora autônoma foi a WordCo, Inc.

Finalmente, gostaríamos de acrescentar algumas notas pessoais. Avi gostaria de agradecer a Valerie por seu apoio durante a preparação desta nova edição. Peter gostaria de agradecer aos seus colegas da Corporate Technologies. Greg gostaria de agradecer a três colegas do Westminster College, que foram responsáveis por sua contratação em 1990: seu reitor, Ray Ownbey; o diretor da área, Bobbie Fredsall; e o vice-presidente acadêmico, Stephen Baar.

ABRAHAM SILBERSCHATZ, *New Haven, CT*

2009

PETER BAER GALVIN, *Burlington, MA*

2009

GREG GAGNE, *Salt Lake City, UT*

2009

PARTE I

VISÃO GERAL

ESBOÇO

[Capítulo 1: Introdução a Visão geral](#)

[Capítulo 2: Introdução](#)

[Capítulo 3: Estruturas do sistema operacional](#)

Introdução a Visão geral

Um *sistema operacional* atua como intermediário entre o usuário de um computador e o hardware do computador. A finalidade do sistema operacional é prover um ambiente no qual o usuário possa executar programas de forma *conveniente* e *eficiente*.

Um sistema operacional é o software que controla o hardware do computador. O hardware precisa prover mecanismos apropriados para assegurar a operação correta e impedir que os programas do usuário interfiram no funcionamento correto do sistema.

Internamente, os sistemas operacionais variam bastante em sua composição, pois são organizados ao longo de muitas linhas diferentes. O projeto de um novo sistema operacional é uma tarefa muito grande. É importante que os objetivos do sistema sejam bem definidos antes que o projeto inicie. Os objetivos formam a base para as escolhas entre diversos algoritmos e estratégias.

Como o sistema operacional é grande e complexo, ele precisa ser criado parte por parte. Cada uma dessas partes deverá ser uma porção bem delineada do sistema, com entradas, saídas e funções definidas cuidadosamente.

CAPÍTULO 1

Introdução

Um **sistema operacional** é um programa que gerencia o hardware do computador. Ele também fornece a base para os programas aplicativos e atua como intermediário entre o usuário e o hardware do computador. Um aspecto importante dos sistemas operacionais é como eles podem variar na realização dessas tarefas. Os sistemas operacionais dos computadores de grande porte (mainframes) são projetados principalmente para otimizar a utilização do hardware. Os sistemas operacionais para computadores pessoais (PC) aceitam jogos complexos, aplicações comerciais e tudo o que se encontra entre eles. Já os sistemas operacionais para computadores portáteis são projetados para fornecer um ambiente em que o usuário possa se comunicar facilmente com o computador para executar os programas. Portanto, alguns sistemas operacionais são projetados para serem *convenientes*, outros para serem *eficientes* e outros para alguma combinação disso.

Antes de podermos explorar os detalhes da operação do sistema computadorizado, precisamos conhecer algo sobre a estrutura do sistema. Começamos discutindo as funções básicas da partida do sistema, E/S e armazenamento. Também descrevemos a arquitetura básica do computador, que torna possível a escrita de um sistema operacional funcional.

Como um sistema operacional é grande e complexo, ele precisa ser criado parte por parte. Cada parte deverá ser uma porção bem delineada do sistema, com entradas, saídas e funções definidas cuidadosamente. Neste capítulo, fornecemos uma visão geral dos principais componentes de um sistema operacional.

OBJETIVOS DO CAPÍTULO

- Prover um passeio pelos principais componentes dos sistemas operacionais.
- Prover uma abordagem sobre a organização básica do sistema computadorizado.

1.1 O que os sistemas operacionais fazem

Vamos iniciar nossas discussões examinando o papel do sistema operacional em um sistema computadorizado genérico, que pode ser dividido basicamente em quatro componentes: o *hardware*, o *sistema operacional*, os *programas aplicativos* e os *usuários* (Figura 1.1).

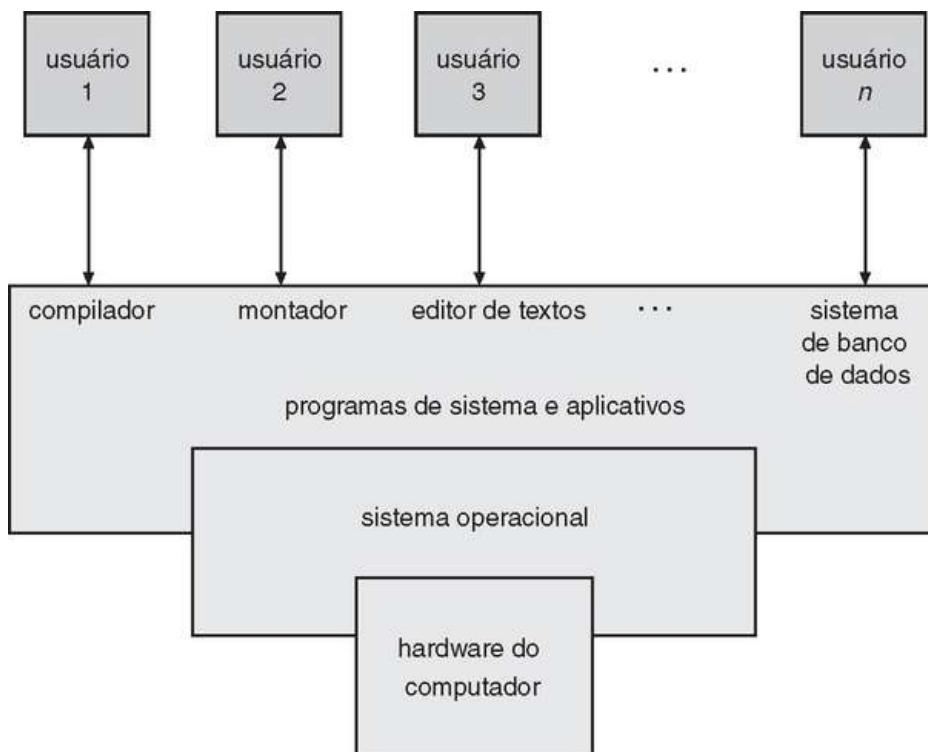


FIGURA 1.1 Visão abstrata dos componentes de um sistema computadorizado.

O **hardware** – a **unidade central de processamento (Central Processing Unit - CPU)**, a **memória e os dispositivos de entrada/saída (E/S)** – provê recursos computacionais básicos para o sistema. Os **programas aplicativos** – como processadores de textos, planilhas, compiladores e navegadores da Web – definem as formas como esses recursos são usados para solucionar os problemas computacionais dos usuários. O **sistema operacional controla e coordena o uso do hardware entre os diversos programas aplicativos para os diversos usuários**.

Também podemos considerar um sistema computadorizado a combinação de hardware, software e dados. O sistema operacional fornece os meios para o uso adequado desses recursos na operação do computador. Um sistema operacional é semelhante a um *governo*. Assim como um governo, ele não realiza qualquer função útil por si só. Ele fornece um *ambiente* dentro do qual os programas podem realizar um trabalho útil.

Para entender melhor seu papel, exploramos, a seguir, os sistemas operacionais sob dois pontos de vista: do usuário e do sistema.

1.1.1 Visão do usuário

A visão do computador pelo usuário varia de acordo com a interface utilizada. A maioria dos usuários de computador se senta à frente de um PC, que consiste em monitor, teclado, mouse e unidade do sistema. Esse sistema foi projetado para o usuário monopolizar seus recursos. O objetivo é agilizar o trabalho (ou jogo) realizado. Nesse caso, o sistema operacional foi projetado principalmente para **facilidade de uso**, com alguma atenção ao desempenho e nenhuma à **utilização de recursos** – como os diversos recursos de hardware e software são compartilhados. É natural que o desempenho seja importante para o usuário; mas esses sistemas são otimizados, não para utilização de recursos, mas para a experiência do único usuário.

Em alguns casos, o usuário se senta à frente de um terminal conectado a um **mainframe** ou **minicomputador**. Outros usuários estão acessando o mesmo computador, por meio de outros terminais. Esses usuários compartilham recursos e podem trocar informações. O sistema operacional, nesses casos, foi projetado para facilitar a utilização de recursos – para garantir que todo o tempo de CPU, memória e E/S disponível seja usado de modo eficiente e que nenhum usuário individual ocupe mais do que sua justa fatia de tempo.

Ainda em outros casos, os usuários se sentam à frente de **estações de trabalho** conectadas a redes de outras estações de trabalho e **servidores**. Esses usuários possuem recursos dedicados à sua disposição, mas também compartilham recursos como rede e servidores de arquivos, processamento e impressão. Portanto, seu sistema operacional é projetado para um compromisso entre a facilidade de utilização individual e a utilização de recursos compartilhados.

Recentemente, muitas variedades de computadores portáteis se tornaram moda. A maioria desses dispositivos é de unidades independentes para usuários individuais. Alguns estão conectados a redes, seja diretamente por fio ou (com mais frequência) por modems sem fio compondo uma rede. Devido a limitações de potência, velocidade e interface, eles realizam poucas operações remotas. Seus sistemas operacionais são projetados principalmente para facilitar a utilização individual, mas o desempenho por tempo de vida da bateria também é muito importante.

Alguns computadores possuem pouca ou nenhuma visão do usuário. Por exemplo, os computadores embutidos nos dispositivos domésticos e em automóveis podem ter teclados numéricos e acender e apagar luzes indicadoras, para mostrar seu status, mas, em sua maior parte, eles e seus sistemas operacionais são projetados para serem executados sem a intervenção do usuário.

1.1.2 Visão do sistema

Do ponto de vista do computador, o sistema operacional é o programa envolvido mais intimamente com o hardware. Nesse contexto, podemos ver um sistema operacional como um **alocador de recursos**. Um sistema computadorizado possui muitos recursos que podem ser necessários para a solução de um problema: tempo de CPU, espaço de memória, espaço para armazenamento de arquivos, dispositivos de E/S, e assim por diante. O sistema operacional atua como o gerenciador desses recursos. Enfrentando inúmeras requisições de recursos, possivelmente em conflito, o sistema operacional precisa decidir como alocá-los a programas e usuários específicos, de modo que possa operar o sistema computadorizado de forma eficiente e justa. Como já vimos, a alocação de recursos é importante, em especial onde muitos usuários acessam o mesmo mainframe ou minicomputador.

DEFINIÇÕES DE ARMAZENAMENTO E NOTAÇÃO

Um **bit** é a unidade básica de armazenamento no computador. Ele pode conter um dentre dois valores, 0 e 1. Todo o armazenamento restante em um computador é baseado em coleções de bits. Com bits suficientes, é incrível a quantidade de coisas que um computador pode representar: números, letras, imagens, filmes, sons, documentos e programas, para citar apenas alguns. Um **byte** é composto de 8 bits, e na maioria dos computadores esse é o menor trecho de armazenamento conveniente. Por exemplo, a maioria dos computadores não possui uma instrução para mover um bit, mas sim para mover um byte. Um termo menos comum é **word**, que é a unidade de armazenamento nativa da arquitetura de determinado computador. Uma word geralmente é composta de um ou mais bytes. Por exemplo, um computador pode ter instruções para mover words de 64 bits (8 bytes).

Um kilobyte, ou KB, contém 1.024 bytes; um megabyte, ou MB, contém 1.024² bytes; e um gigabyte, ou GB, contém 1.024³ bytes. Os fabricantes de computador normalmente arredondam esses números e dizem que um megabyte contém 1 milhão de bytes e um gigabyte corresponde a 1 bilhão de bytes.

Uma visão um pouco diferente de um sistema operacional enfatiza a necessidade de controlar os diversos dispositivos de E/S e programas do usuário. Um sistema operacional é um programa de controle. Um **programa de controle** administra a execução dos programas do usuário para impedir erros e o uso impróprio do computador. Ele se preocupa, em especial, com a operação e o controle de dispositivos de E/S.

1.1.3 Definição de sistemas operacionais

Examinamos o papel do sistema operacional sob os pontos de vista do usuário e do sistema. Como, então, podemos definir o que é um sistema operacional? Em geral, não temos uma definição totalmente adequada de um sistema operacional. Os sistemas operacionais existem porque fornecem um modo razoável de solucionar o problema de criação de um sistema de computação utilizável. A meta fundamental dos computadores é executar programas e facilitar a solução dos problemas do usuário. Para esse objetivo, o hardware do computador é construído. Como o hardware puro não é fácil de utilizar, programas aplicativos são desenvolvidos. Esses programas exigem determinadas operações comuns, como aquelas que controlam os dispositivos de E/S. As funções comuns do controle e alocação de recursos são então reunidas em um software: o sistema operacional.

Além disso, não temos uma definição aceita universalmente para aquilo que faz parte do sistema

operacional. Um ponto de vista simples é que ele inclui tudo o que um vendedor entrega quando você pede “o sistema operacional”. No entanto, os recursos incluídos variam muito entre os sistemas. Alguns sistemas ocupam menos de 1 megabyte de espaço e nem sequer possuem um editor que use a tela inteira (full-screen), enquanto outros exigem gigabytes de espaço e são inteiramente baseados em sistemas gráficos com janelas. Uma definição mais comum, e aquela que normalmente seguimos, é que o sistema operacional é o único programa que executa o tempo todo no computador (normalmente chamado de **kernel**). (Além do kernel, existem dois outros tipos de programas: *programas de sistemas*, que estão associados ao sistema operacional mas não fazem parte do kernel, e *programas de aplicação*, que incluem todos os programas não associados à operação do sistema.)

A questão do que realmente constitui um sistema operacional tornou-se cada vez mais importante. Em 1998, o Departamento de Justiça dos Estados Unidos moveu uma ação contra a Microsoft, afirmando que ela incluía muita funcionalidade em seus sistemas operacionais e, portanto, impedia que vendedores de aplicativos competissem com ela. Por exemplo, um navegador Web fazia parte integral do sistema operacional da Microsoft. Como resultado, a Microsoft foi considerada culpada por usar seu monopólio de sistema operacional para limitar a concorrência.

1.2 Organização do computador

Antes de explorar os detalhes da operação do sistema computadorizado, precisamos ter o conhecimento geral da estrutura de um sistema computadorizado. Nesta seção, veremos várias partes dessa estrutura. A seção trata principalmente da organização do sistema computadorizado. Portanto, se você já comprehende bem os conceitos, pode passar por ele superficialmente ou então pulá-lo.

O ESTUDO DOS SISTEMAS OPERACIONAIS

Nunca houve uma época mais interessante para estudar os sistemas operacionais, e nunca foi tão fácil. O movimento de fonte aberto alcançou os sistemas operacionais, tornando muitos deles disponíveis em formato fonte e binário (executável). Essa lista inclui Linux, BSD UNIX, Solaris e parte do Mac OS X. A disponibilidade do código-fonte nos permite estudar os sistemas operacionais no seu interior. Perguntas que anteriormente só poderiam ser respondidas examinando a documentação ou o comportamento de um sistema operacional agora podem ser respondidas examinando o próprio código.

Além disso, o aumento da virtualização como uma função básica (e frequentemente gratuita) do computador possibilita executar muitos sistemas operacionais em cima de um sistema central. Por exemplo, VMware (<http://www.vmware.com>) oferece um “player” gratuito no qual centenas de “aparelhos virtuais” gratuitos podem ser executados. Usando esse método, os estudantes podem experimentar centenas de sistemas operacionais dentro dos seus sistemas operacionais existentes sem custo algum.

Sistemas operacionais que não são mais economicamente viáveis também tiveram o código aberto, permitindo-nos estudar como os sistemas operavam em uma época com menos recursos de CPU, memória e armazenamento. Uma lista extensa de projetos de sistema operacional de fonte aberto, embora não completa, pode ser encontrada em http://dmoz.org/Computers/Software/Operating_Systems/Open_Source/. Simuladores de hardware específico também estão disponíveis em alguns casos, permitindo que o sistema operacional rode em hardware “nativo”, tudo dentro dos confins de um computador moderno e um sistema operacional moderno. Por exemplo, um simulador DECSYSTEM-20 rodando no Mac OS X pode dar partida no TOPS-20, carregar as fitas de fonte e modificar e compilar um novo kernel TOPS-20. Um estudante interessado pode pesquisar na Internet para encontrar os artigos originais que descrevem o sistema operacional e os manuais originais.

O surgimento dos sistemas operacionais de código-fonte aberto também facilita a transição de estudante para desenvolvedor de sistema operacional. Com algum conhecimento, algum esforço e uma conexão com a Internet, um estudante pode até mesmo criar uma nova distribuição de sistema operacional! Há alguns anos, era difícil ou impossível ter acesso ao código-fonte. Agora, o acesso é limitado apenas pelo tempo e espaço em disco que um estudante possui.

1.2.1 Operação do computador

Um computador de uso geral consiste em uma CPU e uma série de controles de dispositivos e adaptadores, conectados por meio de um barramento comum, fornecendo acesso à memória compartilhada (Figura 1.2). Cada controle de dispositivo está encarregado de um tipo específico de dispositivo (por exemplo, unidades de disco, dispositivos de áudio e monitores de vídeo). A CPU e os controladores de dispositivos podem ser executados simultaneamente, competindo pelos ciclos de memória. Para garantir o acesso ordenado à memória compartilhada, um controlador de memória é fornecido, cuja função é exatamente sincronizar esse acesso.

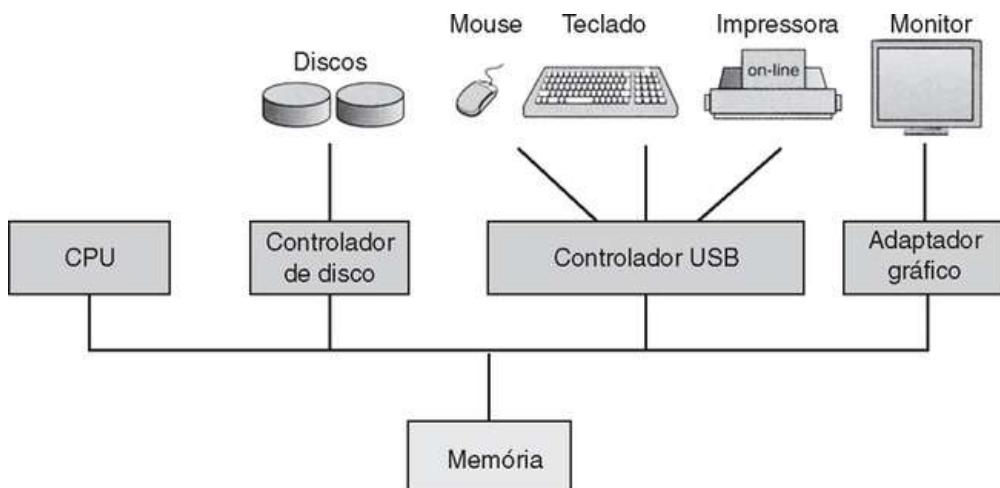


FIGURA 1.2 Um sistema computadorizado moderno.

Para o computador começar a funcionar – por exemplo, quando ele é ligado ou reinicializado –, é preciso que haja um programa inicial para ser executado. Esse programa inicial, ou **bootstrap**, costuma ser simples. Em geral, é armazenado na memória somente de leitura (**ROM**) ou na memória somente de leitura apagável programaticamente (**EPPROM**), conhecida pelo termo geral **firmware**, dentro do hardware do computador. Ele inicializa todos os aspectos do sistema, desde registradores da CPU até controladores de dispositivos e conteúdo de memória. O programa de boot precisa saber como carregar o sistema operacional e como começar a executar esse sistema. Para conseguir esse objetivo, ele precisa localizar e carregar na memória o kernel do sistema operacional. O sistema operacional, então, começa a executar o primeiro processo, como “init”, e espera que ocorra algum evento.

A ocorrência de um evento normalmente é sinalizada por uma **interrupção** do hardware ou do software. O hardware pode disparar uma interrupção a qualquer momento, enviando um sinal à CPU, em geral por meio do barramento do sistema. O software pode disparar uma interrupção executando uma operação especial, denominada **system call (chamada de sistema)** ou **monitor call (chamada ao monitor)**.

Quando a CPU é interrompida, ela para o que está fazendo e imediatamente transfere a execução para uma locação fixa de memória. Essa locação fixa contém o endereço inicial no qual está localizada a rotina de atendimento da interrupção. A rotina de atendimento da interrupção é executada; ao terminar, a CPU retoma a computação interrompida. Uma linha de tempo dessa operação é mostrada na [Figura 1.3](#).

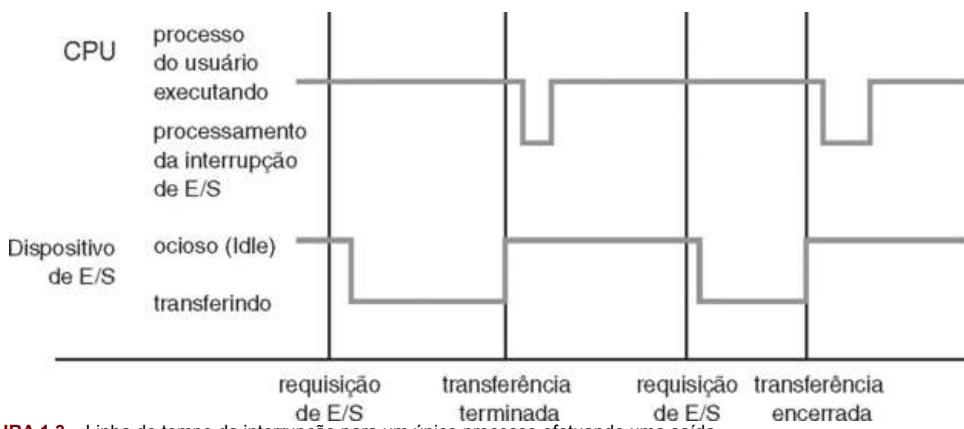


FIGURA 1.3 Linha de tempo da interrupção para um único processo efetuando uma saída.

As interrupções são uma parte importante de uma arquitetura de computador. Cada projeto de computador possui seu próprio mecanismo de interrupção, mas diversas funções são comuns. A interrupção precisa transferir o controle para a rotina de atendimento de interrupção apropriada. O método simples para o tratamento dessa transferência seria chamar uma rotina genérica para examinar a informação da interrupção; a rotina, por sua vez, chamaria o tratador específico da interrupção. Contudo, as interrupções precisam ser tratadas com rapidez. Como só pode haver uma quantidade predefinida de interrupções, uma tabela de ponteiros para rotinas de interrupção pode ser usada no lugar desse enfoque para fornecer a velocidade necessária. A rotina de interrupção, então, é chamada indiretamente pela tabela, sem uma rotina intermediária, agilizando o tratamento

de interrupções. Em geral, a tabela de ponteiros é armazenada na memória baixa (a primeira centena, ou mais, de locações de memória). Essas locações mantêm os endereços das rotinas de atendimento de interrupção para os diversos dispositivos. Esse vetor de endereços, ou **vetor de interrupção**, é indexado pelo número exclusivo do dispositivo, fornecido com a requisição de interrupção, para produzir o endereço da rotina de atendimento de interrupção para o dispositivo que está interrompendo. Sistemas operacionais tão diferentes quanto Windows e UNIX despacham interrupções dessa forma.

A arquitetura de interrupção também precisa salvar o endereço da instrução interrompida. Muitos projetos antigos armazenavam o endereço da interrupção em uma locação fixa ou em um local indexado por um número de dispositivo. As arquiteturas mais recentes armazenam o endereço de retorno na pilha do sistema. Se a rotina de interrupção precisar modificar o estado do processador – por exemplo, modificando valores de registrador –, ela precisará salvar o estado atual explicitamente e depois restaurar esse estado antes de retornar. Depois que a interrupção for atendida, o endereço de retorno salvo é carregado no contador de programa e o processamento interrompido continua como se a interrupção não tivesse acontecido.

1.2.2 Estrutura de armazenamento

A CPU pode carregar instruções somente da memória, de modo que quaisquer programas a serem executados precisam ser armazenados lá. Computadores de uso geral executam a maioria de seus programas por meio da memória regravável, chamada **memória principal** (também chamada **memória de acesso aleatório** ou **Random Access Memory - RAM**). A memória principal é implementada tipicamente em uma tecnologia de semicondutores chamada **memória de acesso aleatório dinâmica (Dynamic Random Access Memory - DRAM)**. Os computadores também utilizam outras formas de memória. Lembre-se de que o programa de bootstrap normalmente é armazenado na memória somente de leitura (ROM) ou na memória somente de leitura apagável programaticamente (EEPROM). Como a ROM não pode ser alterada, somente programas estáticos são armazenados lá. A natureza imutável da ROM é utilizada em cartuchos de jogos, de modo que os fabricantes podem distribuir jogos que não poderão ser modificados. A EEPROM não pode ser alterada com frequência e, portanto, contém a maioria dos programas estáticos. Por exemplo, smartphones utilizam EEPROM para armazenar seus programas instalados de fábrica.

Todas as formas de memória oferecem um array de words, ou unidades de armazenamento. Cada word possui seu próprio endereço. A interação é obtida por meio de uma sequência de instruções para carregar ou armazenar (load ou store), para especificar endereços de memória específicos. A instrução load move uma word da memória principal para um registrador interno dentro da CPU, enquanto a instrução store move o conteúdo de um registrador para a memória principal. Além de carregar e armazenar explicitamente, a CPU carrega instruções da memória principal de forma automática, para serem executadas.

A maioria dos computadores modernos é baseada na arquitetura **von Neumann**. Nessa arquitetura, programas e dados são armazenados na memória principal, que é gerenciada por uma CPU. Uma instrução típica – ciclo de execução, conforme executado em tal sistema – primeiro apanha uma instrução da memória e armazena essa instrução no **registrador de instruções**. A instrução é, então, decodificada e pode fazer operandos serem apanhados da memória e armazenados em algum registrador interno. Após a execução da instrução sobre os operandos, o resultado pode ser armazenado de volta na memória. Observe que a unidade de memória vê apenas um fluxo de endereços de memória; ela não sabe como são gerados (pelo contador de instruções, indexação, indireção, endereços literais ou outros meios) ou para que servem (instruções ou dados). Como consequência, podemos ignorar como um endereço de memória é gerado por um programa. Só estamos interessados na sequência de endereços de memória gerada pelo programa em execução.

O ideal é que os programas e os dados residam na **memória principal permanentemente**. No entanto, esse esquema não é possível por dois motivos:

1. A memória principal costuma ser muito pequena para armazenar todos os programas e dados permanentemente.
2. A memória principal é um dispositivo de armazenamento volátil, que perde seu conteúdo quando a alimentação é cortada ou o sistema é reinicializado de alguma outra maneira.

Assim, a maioria dos sistemas computadorizados fornece um **armazenamento secundário** como uma extensão da memória principal. O requisito principal para o armazenamento secundário é que ele seja capaz de manter grandes quantidades de dados permanentemente.

O dispositivo de armazenamento secundário mais comum é o **disco magnético**, que fornece um espaço para armazenamento para programas e dados. A maioria dos programas (sistema e aplicação) é armazenada em um disco até que sejam carregados para a memória. Muitos programas utilizam o disco como uma origem e um destino de informações para processamento. Logo, a gerência apropriada do armazenamento em disco é de grande importância para um sistema computadorizado, conforme discutiremos no [Capítulo 12](#).

Contudo, em um sentido mais amplo, a estrutura de armazenamento descrita – consistindo em

registradores, memória principal e discos magnéticos - é apenas um dentre muitos sistemas de armazenamento possíveis. Outras possibilidades incluem memória cache, CD-ROM, fitas magnéticas e assim por diante. Cada sistema de armazenamento fornece as funções básicas de armazenamento de um dado e de manutenção desse dado até que ele seja recuperado mais tarde. As principais diferenças entre os diversos sistemas de armazenamento estão em velocidade, custo, tamanho e volatilidade.

A grande variedade de sistemas de armazenamento em um sistema computadorizado pode ser organizada em uma hierarquia ([Figura 1.4](#)), de acordo com a velocidade e o custo. Os níveis mais altos são caros, mas velozes. À medida que descemos na hierarquia, o custo por bit diminui, enquanto o tempo de acesso aumenta. Essa compensação é razoável; se determinado sistema de armazenamento fosse mais rápido e mais barato do que outro - com as outras propriedades sendo iguais -, não haveria motivo para usar uma memória mais lenta, mais cara. Na verdade, muitos dispositivos de armazenamento antigos, incluindo fita de papel e memórias core, são relegados a museus, agora que a fita magnética e a **memória de semicondutores** se tornaram mais rápidas e mais baratas. Os quatro níveis superiores da memória na [Figura 1.4](#) podem ser construídos com a utilização de memória de semicondutores.

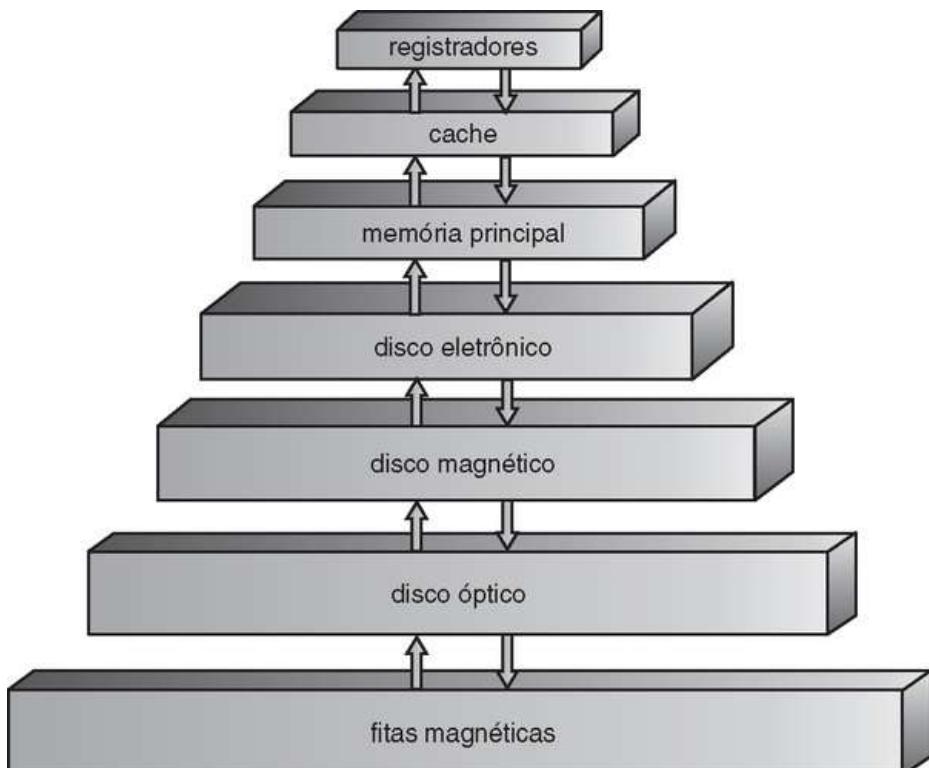


FIGURA 1.4 Hierarquia de dispositivo de armazenamento.

Além de ter diferentes velocidades e custos, os diversos sistemas de armazenamento são voláteis ou não voláteis. Como já mencionado, o **armazenamento volátil** perde seu conteúdo quando a alimentação do dispositivo é removida. Na ausência de sistemas caros de reserva por bateria e gerador, os dados precisam ser gravados no **armazenamento não volátil** por proteção. Na hierarquia mostrada na [Figura 1.4](#), os sistemas de armazenamento acima do **disco eletrônico** são voláteis, enquanto os de baixo não são voláteis. Um disco eletrônico pode ser projetado para ser volátil ou não. Durante a operação normal, o disco eletrônico armazena dados em um grande conjunto de DRAMs, que são voláteis. Entretanto, muitos dispositivos de disco eletrônico contêm um disco rígido magnético oculto e uma bateria para alimentação de reserva. Se a alimentação externa for interrompida, o controlador do disco eletrônico copia os dados da RAM para o disco magnético. Quando a energia externa retorna, o controlador copia os dados de volta para a RAM. Outra forma de disco eletrônico é a memória flash, que é muito popular em câmeras e **assistentes digitais pessoais (PDAs)**, em robôs e cada vez mais como armazenamento removível em computadores de uso geral. A memória flash é mais lenta do que a DRAM, mas não precisa de energia para reter seu conteúdo. Outra forma de armazenamento não volátil é a **NVRAM**, que é a DRAM com fonte de reserva com bateria. Essa memória pode ser tão rápida quanto a DRAM e (desde que a bateria dure) é não volátil.

O projeto de um sistema de memória completo precisa equilibrar todos esses fatos: ele utiliza apenas a quantidade de memória cara necessária, enquanto fornece o máximo de memória não

volátil mais barata possível. Quando houver divergência de tempo de acesso ou taxa de transferência entre dois componentes, os caches podem ser instalados para melhorar o desempenho.

1.2.3 Estrutura de E/S

O armazenamento é apenas um dos muitos tipos de dispositivos de E/S dentro de um computador. Grande parte do código do sistema operacional é dedicada ao gerenciamento de E/S, tanto por sua importância para confiabilidade e desempenho de um sistema quanto pela natureza variável dos dispositivos. Portanto, agora fornecemos uma visão geral da E/S.

Um sistema computadorizado de uso geral consiste em uma CPU e vários controladores de dispositivos conectados por um barramento comum. Cada controlador de dispositivo está encarregado de um tipo específico de dispositivo. Dependendo do controlador, pode haver mais de um dispositivo conectado. Por exemplo, sete ou mais dispositivos podem ser conectados ao controlador **SCSI (Small Computer-Systems Interface)**. Um controlador de dispositivo mantém algum armazenamento em buffer local e um conjunto de registradores de uso especial. O controlador de dispositivo é responsável por mover os dados entre os dispositivos periféricos que controla e seu armazenamento em buffer local. Em geral, os sistemas operacionais têm um **driver de dispositivo (device driver)** para cada controlador de dispositivo. Esse driver entende o controlador de dispositivo e apresenta uma interface uniforme do dispositivo para o restante do sistema operacional.

Para iniciar uma operação de E/S, o driver de dispositivo carrega os registradores apropriados para dentro do controlador de dispositivo. O controlador de dispositivo, por sua vez, examina o conteúdo desses registradores para determinar que ação deve ser realizada (como “ler um caractere do teclado”). O controlador começa a transferir dados do dispositivo para o seu buffer local. Quando a transferência de dados estiver concluída, o controlador de dispositivo informará ao driver de dispositivo via uma interrupção que terminou a operação. O driver de dispositivo, então, retorna o controle ao sistema operacional, possivelmente retornando os dados ou um ponteiro para os dados, se a operação foi uma leitura. Para outras operações, o driver de dispositivo retorna a informação de status.

Essa forma de E/S controlada por interrupção funciona para mover pequenas quantidades de dados, mas pode produzir um grande overhead quando usada para movimento de dados em quantidade, como na E/S de disco. Para resolver esse problema, utiliza-se o **acesso direto à memória (Direct Memory Access - DMA)**. Depois de configurar buffers, ponteiros e contadores para o dispositivo de E/S, o controlador de dispositivo transfere um bloco inteiro de dados diretamente para ou do seu próprio buffer de armazenamento para a memória, sem qualquer intervenção da CPU. Somente uma interrupção é gerada por bloco, para dizer ao driver de dispositivo que a operação foi concluída, em vez de uma interrupção por byte, gerada para os dispositivos de baixa velocidade. Enquanto o controlador de dispositivo está realizando essas operações, a CPU está livre para realizar outras tarefas.

Alguns sistemas de última geração utilizam a arquitetura de switch, em vez de barramento. Nesses sistemas, vários componentes podem interagir com outros componentes ao mesmo tempo, em vez de competir pelos ciclos em um barramento compartilhado. Nesse caso, o DMA é ainda mais eficiente. A [Figura 1.5](#) mostra a interação de todos os componentes de um sistema computadorizado.

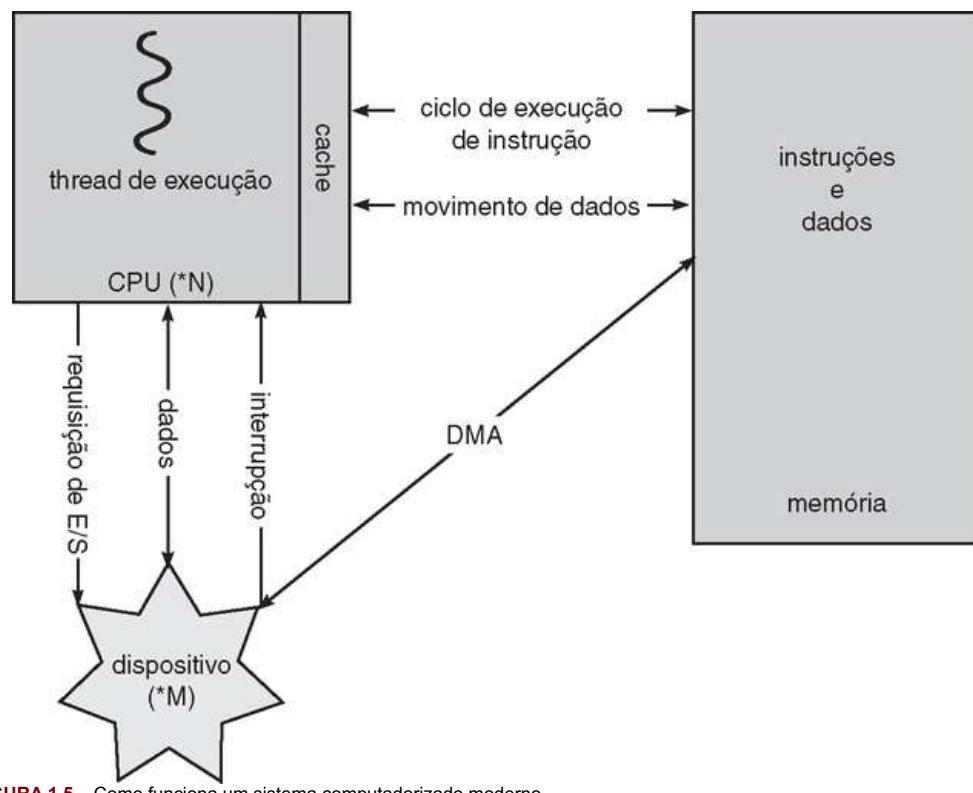


FIGURA 1.5 Como funciona um sistema computadorizado moderno.

1.3 Arquitetura do sistema computadorizado

Na Seção 1.2, introduzimos a estrutura geral de um sistema computadorizado típico. Um sistema computadorizado pode ser organizado de diversas maneiras, que podemos categorizar aproximadamente de acordo com o número de processadores de uso geral utilizados.

1.3.1 Sistemas de processador único

A maioria dos sistemas utiliza um único processador. Porém, a variedade de sistemas de processador único pode ser surpreendente, pois esses sistemas variam desde PDAs até mainframes. Em um sistema de processador único, existe uma CPU principal capaz de executar um conjunto de instruções de uso geral, incluindo instruções de processos do usuário. Quase todos os sistemas também possuem outros processadores de uso especial. Eles podem vir na forma de processadores específicos ao dispositivo, como disco, teclado e controladores gráficos; ou, em mainframes, eles podem vir na forma de processadores de uso mais geral, como processadores de E/S, que movem dados rapidamente entre os componentes do sistema.

Todos esses processadores de uso geral executam um conjunto de instruções limitado e não executam processos do usuário. Às vezes, eles são gerenciados pelo sistema operacional, pois o sistema operacional envia informações sobre sua próxima tarefa e monitora seu status. Por exemplo, um microprocessador controlador de disco recebe uma sequência de solicitações da CPU principal e implementa sua própria fila de disco e algoritmo de escalonamento. Esse arranjo alivia a CPU principal da sobrecarga do escalonamento de disco. Os PCs contêm um microprocessador no teclado para converter os toques de tecla em códigos a serem enviados à CPU. Em outros sistemas ou circunstâncias, processadores de uso especial são componentes de baixo nível embutidos no hardware. O sistema operacional não pode se comunicar com esses processadores; eles realizam suas tarefas de modo autônomo. O uso de microprocessadores de uso especial é comum, e não transforma um sistema de único processador em um multiprocessador. Se houver apenas uma CPU de uso geral, então o sistema é um sistema de processador único.

1.3.2 Sistemas multiprocessados

Embora a maioria dos sistemas atuais seja de processador único, os **sistemas multiprocessados (multiprocessor systems)**, também conhecidos como **sistemas paralelos (parallel systems)** ou **sistemas fortemente acoplados (tightly coupled systems)**, têm importância cada vez maior. Esses sistemas possuem mais de um processador em perfeita comunicação, compartilhando o barramento do computador e, às vezes, o relógio, a memória e os dispositivos periféricos.

Os sistemas multiprocessados possuem três vantagens principais.

1. **Maior vazão (throughput).** Aumentando o número de processadores, esperamos realizar mais trabalho em menos tempo. A taxa de aumento de velocidade com N processadores, porém, não é N , e sim inferior a N . Quando vários processadores cooperam em uma tarefa, um custo adicional é contraído para fazer todas as partes funcionarem corretamente. Esse custo adicional (ou overhead), mais a disputa pelos recursos compartilhados, reduz o ganho esperado dos demais processadores. De modo semelhante, um grupo de N programadores trabalhando em conjunto não produz N vezes a quantidade de trabalho que um único programador produziria.
2. **Economia de escala.** Os sistemas multiprocessados podem custar menos do que múltiplos sistemas de processador único equivalentes, pois compartilham periféricos, armazenamento em massa e fonte de alimentação. Se vários programas operam sobre o mesmo conjunto de dados, é mais barato armazenar esses dados em um disco e fazer todos os processadores compartilhá-los do que ter muitos computadores com discos locais e muitas cópias dos dados.
3. **Maior confiabilidade.** Se as funções podem ser distribuídas corretamente entre diversos processadores, então a falha de um processador não interromperá o sistema, só o atrasará. Se tivermos dez processadores e um falhar, então cada um dos nove processadores restantes terá de apanhar uma fatia do trabalho do processador que falhou. Assim, o sistema inteiro é executado apenas 10% mais lento, em vez de travar completamente.

Uma maior confiabilidade do sistema computadorizado é crucial em muitas aplicações. A capacidade de continuar fornecendo serviço proporcional em nível do hardware sobrevivente é chamada de **degradação controlada**. Alguns sistemas vão além da degradação controlada e são considerados **tolerantes a falhas (fault tolerant)**, pois podem sofrer uma falha de qualquer componente isolado e ainda continuar operando. Observe que a tolerância a falhas exige um mecanismo para permitir que a falha seja detectada, diagnosticada e, se possível, corrigida. O sistema HP NonStop (anteriormente, Tandem) utiliza duplicação de hardware e software para garantir a operação contínua apesar de falhas. O sistema consiste em múltiplos pares de CPUs trabalhando em cooperação. Os dois processadores em um par executam cada instrução e comparam os resultados. Se os resultados diferirem, então uma CPU do par tem uma falha, e ambas são interrompidas. O processo que estava sendo executado é passado para outro par de CPUs, e a

instrução que falhou é reiniciada. Essa solução é eficaz, porém dispendiosa, pois envolve um hardware especial e uma duplicação de hardware considerável.

Dois tipos de sistemas multiprocessados estão em uso atualmente. Alguns sistemas utilizam **multiprocessamento assimétrico**, no qual cada processador recebe uma tarefa específica. Um processador mestre controla o sistema; os outros procuram instruções com o mestre ou possuem tarefas predefinidas. Esse esquema define um relacionamento mestre-escravo. O processador mestre escalona e aloca trabalho para os processadores escravos.

Os sistemas mais comuns usam o **multiprocessamento simétrico (SMP)**, no qual cada processador executa todas as tarefas em um sistema operacional. SMP significa que todos os processadores são iguais; não existe um relacionamento mestre-escravo entre os processadores. A [Figura 1.6](#) ilustra uma arquitetura SMP típica. Observe que cada processador tem seu próprio conjunto de registradores, bem como um cache privado - ou local; porém, todos os processadores compartilham a memória física. Um exemplo de sistema SMP é o Solaris, uma versão comercial do UNIX projetada pela Sun Microsystems. O Solaris pode ser configurado para empregar dezenas de processadores, todos executando o Solaris. O benefício desse modelo é que muitos processos podem ser executados simultaneamente - N processos podem estar em execução se houver N CPUs -, sem causar deterioração significativa do desempenho. Contudo, temos de controlar a E/S com cuidado, para garantir que os dados cheguem ao processador correto. Além disso, como as CPUs são separadas, uma pode estar ociosa enquanto outra está sobrecarregada, resultando em ineficiências. Essas ineficiências podem ser evitadas se os processadores compartilharem determinadas estruturas de dados. Um sistema multiprocessado dessa forma permitirá que os processos e os recursos - como a memória - sejam compartilhados dinamicamente entre os diversos processadores, podendo reduzir a variância entre os processadores. Esse tipo de sistema precisa ser escrito cuidadosamente, como veremos no [Capítulo 6](#). Quase todos os sistemas operacionais modernos - incluindo Windows, Mac OS X e Linux - agora fornecem suporte para SMP.

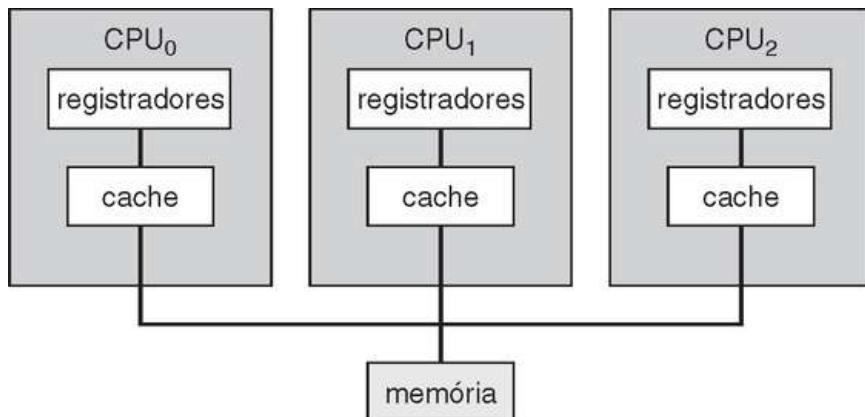


FIGURA 1.6 Arquitetura de multiprocessamento simétrico.

A diferença entre o multiprocessamento simétrico e o assimétrico pode ser resultante do hardware ou do software. Um hardware especial pode diferenciar os diversos processadores ou, então, o software pode ser escrito para permitir apenas um mestre e vários escravos. Por exemplo, o sistema operacional SunOS versão 4 da Sun fornece multiprocessamento assimétrico, enquanto a versão 5 (Solaris) é simétrica no mesmo hardware.

O multiprocessamento acrescenta CPUs para aumentar o poder de computação. Se a CPU tiver um controlador de memória integrado, então o acréscimo de CPUs também pode aumentar a quantidade de memória endereçável no sistema. De qualquer forma, o multiprocessamento pode fazer um sistema mudar seu modelo de acesso à memória de um acesso uniforme à memória (**Uniform Memory Access - UMA**) para um acesso não uniforme à memória (**Non-Uniform Memory Access - NUMA**). UMA é definido como a situação em que o acesso a qualquer RAM a partir de qualquer CPU leva a mesma quantidade de tempo. Com NUMA, algumas partes da memória podem exigir um tempo de acesso maior do que outras partes, criando uma penalidade sobre o desempenho. Os sistemas operacionais podem minimizar a penalidade do acesso NUMA por meio da gerência de recursos, conforme discutimos na [Seção 9.5.4](#).

Uma tendência recente em projeto de CPU é incluir múltiplos **núcleos** (ou *multicore*) de computação em um único chip. Basicamente, esses são chips multiprocessados. Eles podem ser mais eficientes do que múltiplos chips com núcleos isolados, pois a comunicação no chip é mais rápida do que a comunicação entre chips. Além disso, um chip com vários núcleos utiliza muito menor energia do que diversos chips com núcleo único. Como resultado, os sistemas multicore são especialmente bem adequados para sistemas servidores, como servidores de banco de dados ou servidores Web.

Na [Figura 1.7](#), mostramos um projeto dual-core com dois núcleos no mesmo chip. Nesse projeto,

cada núcleo tem seu próprio conjunto de registradores, bem como seu próprio cache local; outros projetos poderiam usar um cache compartilhado ou uma combinação de caches local e compartilhado. Além das considerações arquitetônicas, como cache, memória e disputa pelo barramento, essas CPUs de múltiplos núcleos aparecem no sistema operacional como N processadores comuns. Essa característica coloca uma pressão maior sobre projetistas de sistema operacional (e programadores de aplicação) para utilizar essas CPUs.

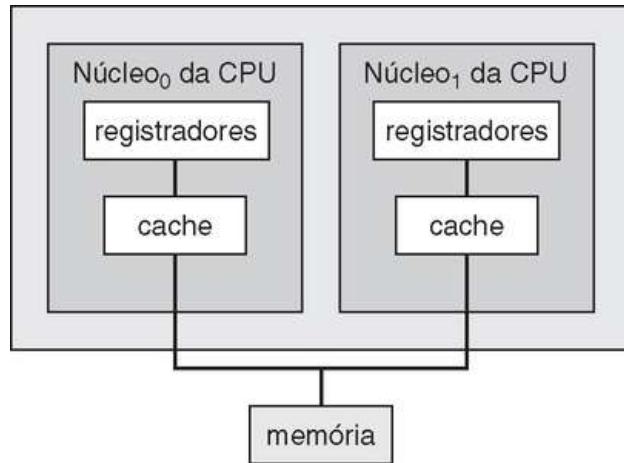


FIGURA 1.7 Um projeto dual-core com dois núcleos colocados no mesmo chip.

Por último, os **servidores em lâmina** são um desenvolvimento recente, em que várias placas de processador, placas de E/S e placas de rede são colocadas no mesmo chassis. A diferença entre estes e os sistemas multiprocessados tradicionais é que cada placa de processador em lâmina tem partida independente e executa seu próprio sistema operacional. Algumas placas de servidor em lâmina também são multiprocessadas, o que torna mais confusa a distinção entre os tipos de computadores. Basicamente, esses servidores consistem em múltiplos sistemas multiprocessados independentes.

1.3.3 Sistemas em clusters

Outro tipo de sistema com diversas CPUs é o **sistema em clusters**. Como os sistemas multiprocessados, os sistemas em clusters reúnem diversas CPUs para realizar trabalho de computação. Entretanto, os sistemas em clusters diferem dos sistemas multiprocessados porque **são compostos de dois ou mais sistemas individuais - ou nós - acoplados**. A definição do termo *em clusters* não é concreta; muitos pacotes comerciais discutem o significado de um sistema em clusters e por que uma forma é melhor do que a outra. A definição geralmente aceita é a de que **computadores em clusters compartilham armazenamento e estão conectados de maneira rígida por meio de redes locais (LANs)** (como descrito na Seção 1.10) ou interconexão mais rápida, como o InfiniBand.

O agrupamento em clusters costuma ser usado para **fornecer um serviço de alta disponibilidade**, ou seja, um serviço continuará mesmo com falha em um ou mais sistemas no cluster. Em geral, a alta disponibilidade é obtida pela inclusão de um nível de redundância no sistema. Uma camada de software de cluster é executada nos nós do cluster. Cada nó pode monitorar um ou mais nós (pela LAN). Se a máquina monitorada falhar, a máquina que a monitora pode assumir o armazenamento e reiniciar as aplicações que estavam sendo executadas na máquina que falhou. Os usuários e clientes das aplicações só perceberiam uma breve interrupção no serviço.

Os sistemas em clusters podem ser estruturados de forma assimétrica ou simétrica. No **modo assimétrico**, uma máquina está no **modo hot-standby**, enquanto a outra está executando as aplicações. A máquina hot-standby apenas monitora o servidor ativo. Se esse servidor falhar, o hot-standby se torna o servidor ativo. No **modo simétrico**, duas ou mais máquinas host estão executando aplicações, e elas estão monitorando uma à outra. Esse modo, obviamente, é mais eficiente, pois utiliza todo o hardware disponível. Ele exige que mais de uma aplicação esteja disponível para ser executada.

Como um cluster consiste em diversos sistemas de computação conectados por uma rede, os clusters também podem ser usados para oferecer ambientes de **computação de alta performance**. Esses tipos de sistemas podem oferecer muito mais poder de computação do que um único processador ou até mesmo sistemas SMP, pois são capazes de executar uma aplicação simultaneamente em todos os computadores no cluster. Porém, as aplicações precisam ser escritas especificamente para tirar proveito do cluster, usando uma técnica conhecida como **parallelismo**, que consiste em dividir um programa em componentes separados, que são executados em paralelo em computadores individuais no cluster. Normalmente, essas aplicações são projetadas de modo

que, quando cada nó de computação no cluster tiver resolvido sua parte do problema, os resultados de todos os nós são combinados em uma solução final.

Outras formas de clusters incluem clusters paralelos e uso de clusters por uma rede remota (WAN) (como descrito na [Seção 1.10](#)). Os clusters paralelos permitem que vários hosts acessem os mesmos dados no armazenamento compartilhado. Como a maioria dos sistemas operacionais não possui suporte para acesso a dados simultâneos por vários computadores, os clusters paralelos normalmente são constituídos pelo uso de versões especiais de software e releases especiais de aplicações. Por exemplo, o Oracle Real Application Cluster é uma versão do banco de dados Oracle que foi concebida para executar em um cluster paralelo. Cada máquina executa o Oracle, e uma camada de software acompanha o acesso ao disco compartilhado. Cada máquina possui acesso total a todos os dados no banco de dados. Para fornecer esse acesso compartilhado aos dados, o sistema também precisa fornecer controle de acesso e lock (trava), para assegurar que não haverá operações em conflito. Essa função, normalmente conhecida como **gerenciador de lock distribuído (Distributed Lock Manager - DLM)**, está incluída em algumas tecnologias de cluster.

A tecnologia de cluster está mudando rapidamente. Alguns produtos de clusters suportam dezenas de sistemas em um cluster e nós que estão separados por vários quilômetros. Muitas dessas melhorias se tornaram possíveis por meio de **sistemas de armazenamento em rede**, conhecidos como **SANs** (Storage-Area Networks), conforme descreveremos na [Seção 12.3.3](#), permitindo que muitos sistemas se conectem a um pool de armazenamento. Se as aplicações e seus dados forem armazenados em SAN, então o software de cluster pode atribuir a aplicação para executar em qualquer host que esteja conectado ao SAN. Se o host travar, então qualquer outro host poderá assumir. Em um cluster de banco de dados, dezenas de hosts podem compartilhar o mesmo banco de dados, aumentando bastante o desempenho e a confiabilidade. A [Figura 1.8](#) representa a estrutura geral de um sistema em clusters.

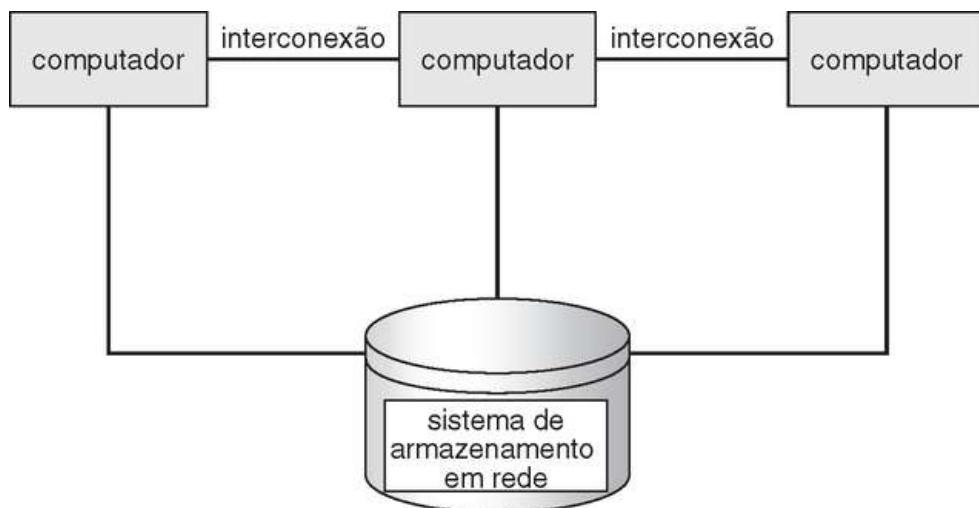


FIGURA 1.8 Estrutura geral de um sistema em clusters.

CLUSTERS BEOWULF

Clusters Beowulf são projetados para resolver tarefas de computação de alta performance. Esses clusters são montados com hardware de consumo – como computadores pessoais de baixo custo – conectados por meio de uma rede local simples. O interessante é que um cluster Beowulf não utiliza um pacote de software específico, mas consiste em um conjunto de bibliotecas de software de fonte aberto, permitindo que os nós de computação no cluster se comuniquem entre si. Assim, existem diversas técnicas para se construir um cluster Beowulf, embora os nós de computação Beowulf normalmente executem o sistema operacional Linux. Como os clusters Beowulf não exigem hardware especial e operam usando software de fonte aberto, disponível gratuitamente, eles oferecem uma estratégia de baixo custo para a criação de um cluster de computação de alta performance. Na verdade, alguns clusters Beowulf montados a partir de conjuntos de computadores pessoais descartados estão usando centenas de nós de computação para resolver problemas que exigem uma computação científica muito rigorosa.

1.4 Estrutura do sistema operacional

Agora que já discutimos as informações básicas sobre a organização do sistema computadorizado e sua arquitetura, estamos prontos para falar sobre os sistemas operacionais. Um sistema operacional provê o ambiente dentro do qual os programas são executados. Internamente, os sistemas operacionais variam bastante em sua composição, pois são organizados ao longo de muitas linhas diferentes. Porém existem muitas semelhanças, que consideramos nesta seção.

Um dos aspectos mais importantes do sistema operacional é a capacidade de multiprogramar. Geralmente, um único usuário não pode manter a CPU ou os dispositivos de E/S ocupados o tempo todo. Entretanto, usuários isolados constantemente possuem vários programas em execução. A **multiprogramação** aumenta a utilização de CPU, organizando as tarefas (código e dados) de modo que a CPU sempre tenha uma para executar.

Vamos explicar melhor essa ideia. O sistema operacional mantém várias tarefas na memória simultaneamente (Figura 1.9). Como geralmente a memória principal é muito pequena para acomodar todas as tarefas, elas são mantidas inicialmente no disco, no **banco de tarefas**. Esse pool de tarefas consiste em todos os processos que residem no disco aguardando a alocação da memória principal.

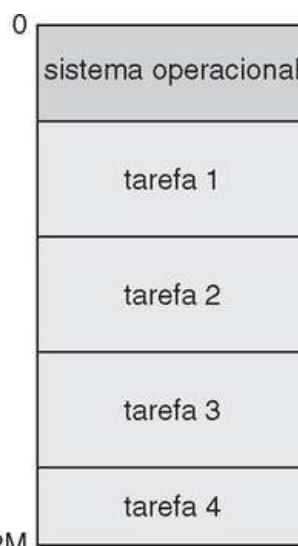


FIGURA 1.9 Diagrama de memória para um sistema de multiprogramação.

Esse conjunto de tarefas na memória pode ser um subconjunto das tarefas mantidas no banco de tarefas. O sistema operacional apanha e começa a executar uma das tarefas na memória. Por fim, a tarefa pode ter de esperar por alguma tarefa, como uma operação de E/S, para completar sua função. Em um sistema monoprogramado, a CPU ficaria ociosa. Em um sistema multiprogramado, o sistema operacional passa para outra tarefa e a executa. Quando essa tarefa precisar esperar, a CPU troca para outra tarefa, e assim por diante. Por fim, a primeira tarefa termina sua espera e recebe a atenção da CPU novamente. Desde que pelo menos uma tarefa precise ser executada, a CPU nunca fica ociosa.

Essa ideia é muito comum em outras situações da vida. Um advogado não trabalha somente para um cliente de cada vez. Por exemplo, enquanto um caso está esperando para ir a julgamento ou enquanto documentos são digitados, o advogado pode trabalhar em outro caso. Se tiver muitos clientes, ele nunca ficará ocioso por falta de trabalho. (Os advogados ociosos costumam se tornar políticos, de modo que existe um valor social em mantê-los ocupados.)

Os sistemas multiprogramados fornecem um ambiente em que os diversos recursos do sistema (por exemplo, CPU, memória, dispositivos periféricos) são utilizados com eficiência, mas não proporcionam interação do usuário com o sistema. **Tempo compartilhado** (time sharing), ou **multitarefa** (multitasking), é uma extensão lógica da multiprogramação. Em sistemas de tempo compartilhado, a CPU executa várias tarefas alternando entre elas, mas as trocas ocorrem com tanta frequência que os usuários podem interagir com cada programa enquanto ele está sendo executado.

O tempo compartilhado exige um **computador interativo**, que providencia a comunicação direta entre o usuário e o sistema. O usuário dá instruções diretamente ao sistema operacional ou a um programa, usando um teclado ou um mouse, e espera receber resultados imediatos em um dispositivo de saída. De acordo com isso, o **tempo de resposta** deverá ser curto - normalmente, menos de um segundo.

Um sistema operacional de tempo compartilhado permite que muitos usuários compartilhem o computador simultaneamente. Como cada ação ou comando em um sistema de tempo compartilhado costuma ser curto, somente um pouco de tempo de CPU é necessário para cada usuário. Visto que o sistema passa rapidamente de um usuário para o seguinte, cada usuário tem a impressão de que o computador inteiro está dedicado ao seu uso, embora ele esteja sendo compartilhado entre muitos usuários.

Um sistema operacional de tempo compartilhado utiliza o escalonamento de CPU e a multiprogramação para fornecer a cada usuário uma pequena fração do tempo de um computador com tempo compartilhado. Cada usuário tem pelo menos um programa separado na memória. Um programa carregado na memória e em execução costuma ser considerado um **processo**. Quando um processo é executado, em geral ele executa por um pequeno período antes de ser interrompido ou precisar realizar E/S. A E/S pode ser interativa, ou seja, a saída vai para o monitor de um usuário e a entrada vem do seu teclado, mouse ou outro dispositivo. Como a E/S interativa normalmente é executada na “velocidade das pessoas”, ela pode levar muito tempo para ser concluída. A entrada, por exemplo, pode estar ligada à velocidade de digitação do usuário; digitar sete caracteres por segundo é rápido para as pessoas, mas incrivelmente lento para os computadores. Em vez de deixar a CPU ociosa enquanto ocorre essa atividade interativa lenta, o sistema operacional passará rapidamente a CPU para o programa de algum outro usuário.

Tempo compartilhado e multiprogramação exigem que várias tarefas sejam mantidas simultaneamente na memória. Se várias tarefas estiverem prontas para serem levadas para a memória, e se não houver espaço suficiente para todas elas, o sistema precisará escolher entre elas. Tomar essa decisão é chamado de **escalonamento de tarefa**, e é discutido no [Capítulo 5](#). Quando o sistema operacional seleciona uma tarefa do conjunto, ele carrega essa tarefa na memória para execução. Manter vários programas na memória ao mesmo tempo exige alguma forma de gerência de memória, o que é explicado nos [Capítulos 8 e 9](#). Além disso, se várias tarefas estiverem prontas para ser executadas ao mesmo tempo, o sistema precisará escolher entre elas. Essa decisão se chama **escalonamento de CPU**, discutido no [Capítulo 5](#). Finalmente, a execução de múltiplas tarefas simultaneamente exige que sua capacidade de afetar umas às outras seja limitada em todas as fases do sistema operacional, incluindo o escalonamento de processo, armazenamento de disco e gerência de memória. Essas considerações são discutidas no decorrer deste livro.

Em um sistema operacional de tempo compartilhado, o sistema operacional deve assegurar um tempo de resposta razoável, que às vezes é obtido por meio de **swapping**, no qual processos são trocados entre a memória principal e o disco. O método mais comum para conseguir esse objetivo é a **memória virtual**, uma técnica que permite a execução de um processo que não está completamente na memória ([Capítulo 9](#)). A principal vantagem do esquema de memória virtual é que ele permite que os usuários executem programas maiores do que a **memória física** real. Além disso, o esquema retira a memória principal para uma sequência de armazenamento grande e uniforme, separando a **memória lógica**, como é vista pelo usuário, da memória física. Essa organização evita que os programadores se preocupem com as limitações de armazenamento na memória.

Os sistemas de tempo compartilhado também fornecem um sistema de arquivos ([Capítulos 10 e 11](#)). O sistema de arquivos reside em uma coleção de discos; logo, é preciso que haja gerenciamento de discos ([Capítulo 12](#)). Os sistemas de tempo compartilhado também fornecem um mecanismo para recursos de proteção por uso inapropriado ([Capítulo 14](#)). Para assegurar a execução em ordem, o sistema também fornece mecanismos para sincronismo de tarefas e comunicação entre elas ([Capítulo 6](#)) e pode garantir que as tarefas nunca ficarão presas por um deadlock, aguardando umas pelas outras eternamente ([Capítulo 7](#)).

1.5 Operações do sistema operacional

Como já vimos, os sistemas operacionais modernos são **controlados por interrupção**. Se não houver processos para executar, nenhum dispositivo de E/S para atender e nenhum usuário para responder, um sistema operacional ficará silencioso, esperando que algo aconteça. Os eventos quase sempre são sinalizados pela ocorrência de uma interrupção ou um **trap**. Um trap (ou uma **exceção**) é a interrupção gerada pelo software, causada por um erro (por exemplo, divisão por zero ou acesso inválido à memória) ou por uma requisição específica de um programa do usuário, para que um serviço do sistema operacional seja realizado. A forma como o sistema operacional controla as interrupções define a estrutura geral desse sistema. Para cada tipo de interrupção, segmentos de código separados no sistema operacional determinam qual ação deve ser realizada, e a rotina de serviço de interrupção é fornecida para tratar dessa interrupção.

Como o sistema operacional e os usuários compartilham os recursos de hardware e software do sistema computadorizado, precisamos ter certeza de que um erro em um programa do usuário só poderá causar problemas para aquele programa que estava sendo executado. Com o compartilhamento, muitos processos poderiam ser afetados adversamente por um bug em um programa. Por exemplo, se um programa ficar preso em um loop infinito, esse loop poderá impedir a operação correta de muitos outros processos. Erros mais sutis podem ocorrer em um sistema multiprogramado, no qual um programa com erro pode modificar outro programa, os dados de outro programa ou até mesmo o próprio sistema operacional.

Sem a proteção contra esses tipos de erros, ou o computador precisa executar apenas um processo de cada vez ou toda a saída precisa ser suspeita. Um sistema operacional projetado corretamente precisa garantir que um programa incorreto (ou malicioso) não possa fazer outros programas serem executados de forma incorreta.

1.5.1 Operação no modo dual

Para garantir a execução apropriada do sistema operacional, devemos poder distinguir entre a execução do código do sistema operacional e o código do usuário definido. A técnica utilizada por muitos sistemas operacionais fornece suporte de hardware que permite diferenciar entre diversos modos de execução.

No mínimo, precisamos de dois **modos** de operação separados: **modo usuário** e **modo kernel** (também chamado **modo supervisor**, **modo do sistema** ou **modo privilegiado**). Um bit, chamado **bit de modo**, é adicionado ao hardware do computador para indicar o modo atual: kernel (0) ou usuário (1). Com o bit de modo, somos capazes de distinguir entre uma tarefa executada em nome do sistema operacional e uma executada em nome do usuário. Quando o sistema computadorizado está executando em favor de uma aplicação do usuário, o sistema está no modo usuário. Porém, quando uma aplicação do usuário requisita um serviço do sistema operacional (por meio de uma chamada de sistema), ele precisa passar do modo usuário para o modo kernel, para atender a solicitação (Figura 1.10). Como veremos, essa melhoria arquitetônica também é útil para muitos outros aspectos do sistema operacional.

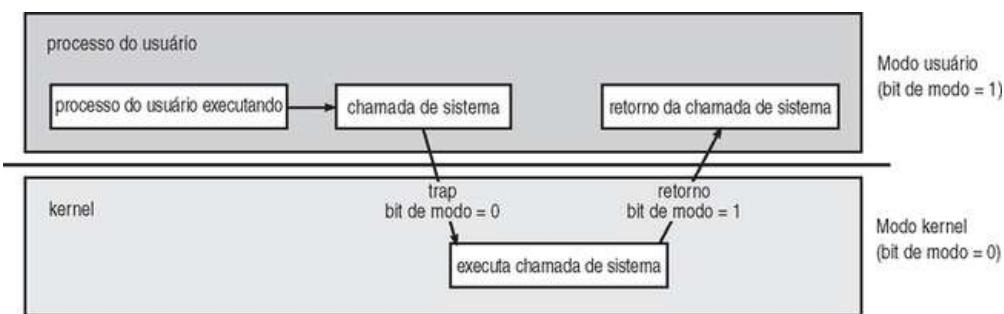


FIGURA 1.10 Transição de modo usuário para modo kernel.

No momento do boot do sistema, o hardware começa no modo kernel. O sistema operacional é, então, carregado e inicia as aplicações do usuário no modo usuário. Sempre que ocorre um trap ou uma interrupção, o hardware passa do modo usuário para o modo kernel (ou seja, ele muda o estado do bit de modo para 0). Assim, sempre que o sistema operacional obtém o controle do computador, ele está no modo kernel. O sistema sempre passa para o modo usuário (definindo o bit de modo como 1) antes de passar o controle para um programa do usuário.

O modo de operação dual fornece os meios para proteger o sistema operacional contra usuários (mal intencionados ou não) e um usuário do outro. Conseguimos essa proteção designando algumas das instruções de máquina que podem causar danos como **instruções privilegiadas**. O hardware

permite que instruções privilegiadas sejam executadas apenas no modo kernel. Se houver uma tentativa de executar uma instrução privilegiada no modo usuário, o hardware não executa a instrução, trata-a como ilegal e chama o sistema operacional.

A instrução para passar para o modo kernel é um exemplo de uma instrução privilegiada. Alguns outros exemplos incluem controle de E/S, gerência de temporizador e gerência de interrupção. Conforme veremos no decorrer do livro, existem muitas outras instruções privilegiadas.

Agora podemos ver o ciclo de vida da execução de instrução em um sistema computadorizado. O controle inicial está dentro do sistema operacional, onde as instruções são executadas no modo kernel. Quando o controle é dado a uma aplicação do usuário, o modo é definido como modo usuário. Por fim, o controle retorna ao sistema operacional por meio de uma interrupção, um trap ou uma chamada de sistema.

As chamadas de sistema fornecem os meios para um programa do usuário pedir ao sistema operacional para realizar tarefas reservadas para o sistema operacional em favor do programa do usuário. Uma chamada de sistema é invocada de diversas maneiras, dependendo da funcionalidade oferecida pelo processador subjacente. De todas as formas, ela é o método utilizado por um processo para solicitar a ação pelo sistema operacional. Uma chamada de sistema normalmente tem a forma de um trap para um local específico no vetor de interrupção. Esse trap pode ser executado por uma instrução trap genérica, embora alguns sistemas (como a família MIPS R2000) tenham uma instrução syscall específica.

Quando uma chamada de sistema é executada, ela é tratada pelo hardware como uma interrupção de software. O controle passa pelo vetor de interrupção até chegar a uma rotina de serviço no sistema operacional, e o bit de modo é definido para o modo kernel. A rotina de serviço da chamada de sistema é uma parte do sistema operacional. O kernel examina a instrução que está interrompendo para determinar qual foi a chamada de sistema que ocorreu; um parâmetro indica que tipo de serviço o programa do usuário está solicitando. Informações adicionais, necessárias para a requisição, podem ser passadas nos registradores, na pilha ou na memória (com ponteiros para os locais da memória passando nos registradores). O kernel verifica se os parâmetros são corretos e válidos, executa a requisição e retorna o controle à instrução após a chamada de sistema. Descrivemos mais a fundo as chamadas de sistema na [Seção 2.3](#).

A falta de um modo dual suportado pelo hardware pode causar sérias limitações em um sistema operacional. Por exemplo, o MS-DOS foi escrito para a arquitetura 8088 da Intel, que não possui bit de modo e, portanto, nenhum modo dual. Um programa do usuário com problemas pode acabar com o sistema operacional, escrevendo dados sobre ele; e vários programas são capazes de escreverem em um dispositivo ao mesmo tempo, possivelmente com resultados desastrosos. As versões recentes da CPU Intel, como o Pentium, fornecem o modo de operação dual. Por conseguinte, sistemas operacionais mais contemporâneos, como Microsoft Vista e Windows XP, bem como Unix, Linux e Solaris – aproveitam esse recurso e fornecem maior proteção para o sistema operacional.

Quando existe a proteção do hardware, os erros de violação de modos são detectados pelo hardware. Esses erros são tratados pelo sistema operacional. Se um programa do usuário falhar de alguma maneira – tentando, por exemplo, executar uma instrução ilegal ou acessar a memória que não está no espaço de endereço do usuário –, então o hardware causará um trap para o sistema operacional. O trap transfere o controle por meio do vetor de interrupção para o sistema operacional, assim como em uma interrupção. Quando ocorre um erro no programa, o software precisa terminar o programa de modo anormal. Essa situação é tratada pelo mesmo código usado para um término anormal solicitado pelo usuário. Uma mensagem de erro apropriada é enviada, e isso pode ocasionar um dump da memória. Esse dump normalmente é gravado em um arquivo, para que o usuário ou programador possa examiná-lo, talvez corrigi-lo e reiniciar o programa.

1.5.2 Temporizador

Temos de garantir que o sistema operacional mantenha o controle da CPU. Precisamos impedir que um programa do usuário fique preso em um loop infinito ou deixe de chamar os serviços do sistema, e nunca retorne o controle ao sistema operacional. Para isso, podemos usar um **temporizador**. Um temporizador (ou timer) pode ser ajustado para interromper o computador após um período especificado. O período pode ser fixo (por exemplo, 1/60 segundo) ou variável (por exemplo, de 1 milissegundo a 1 segundo). Um **temporizador variável**, em geral, é implementado por um relógio de velocidade fixa e um contador. O sistema operacional define o contador. Toda vez que o relógio toca, o contador é decrementado. Quando o contador atinge 0, ocorre uma interrupção. Por exemplo, um contador de 10 bits com um relógio de 1 milissegundo permite interrupções em intervalos de 1 milissegundo a 1.024 milissegundos, em passos de 1 milissegundo.

Antes de passar o controle para o usuário, o sistema operacional garante que o temporizador esteja ajustado para interromper. Se o temporizador interromper, o controle é transferido automaticamente para o sistema operacional, que pode tratar a interrupção como um erro fatal ou dar mais tempo ao programa. É claro que as instruções que modificam a operação do temporizador são privilegiadas.

Assim, podemos utilizar o temporizador para evitar a execução de um programa do usuário por

muito tempo. Uma técnica simples é inicializar um contador com o período durante o qual um programa tem permissão para executar. Um programa com um tempo limite de 7 minutos, por exemplo, teria seu contador inicializado como 420. A cada segundo, o temporizador interrompe e o contador é decrementado em 1. Enquanto o contador for positivo, o controle é retornado ao programa do usuário. Quando o contador se torna negativo, o sistema operacional termina o programa por exceder o limite de tempo atribuído.

1.6 Gerência de processos

Um programa não faz nada, a menos que suas instruções sejam executadas por uma CPU. Um programa em execução, como dissemos, é um **processo**. Um programa do usuário com tempo compartilhado, como um compilador, é um processo. Um programa de processamento de textos, executado por um usuário individual em um PC, também é um processo. Uma tarefa do sistema, como o envio de saída para uma impressora, também pode ser um processo (ou, pelo menos, parte de um). Por enquanto, você pode considerar um processo uma tarefa ou um programa de tempo compartilhado, mas aprenderá mais tarde que o conceito é mais geral. Neste ponto, é importante lembrar que um programa, por si só, não é um processo; um programa é uma entidade *passiva*, como o conteúdo de um arquivo armazenado no disco, enquanto um processo é uma entidade *ativa*.

Um processo precisa de certos recursos - incluindo tempo de CPU, memória, arquivos e dispositivos de E/S - para realizar sua tarefa. Esses recursos são dados ao processo quando ele é criado ou são alocados enquanto ele está sendo executado. Além dos diversos recursos físicos e lógicos que um processo obtém quando é criado, vários dados de inicialização (entrada) podem ser passados. Por exemplo, considere um processo cuja função seja exibir o status de um arquivo na tela de um terminal. O processo receberá, como entrada, o nome do arquivo, e executará as instruções e chamadas de sistema apropriadas para obter e exibir a informação desejada no terminal. Quando o processo terminar, o sistema operacional retomará quaisquer recursos reutilizáveis.

Um processo de uma única thread possui um **contador de programa** (program counter) que especifica a próxima instrução a ser executada. (As threads serão explicadas no [Capítulo 4](#).) A execução de tal processo precisa ser sequencial. A CPU executa uma instrução do processo após a outra, até que o processo termine. Além do mais, a qualquer momento, no máximo uma instrução é executada em nome do processo. Assim, embora dois processos possam estar associados ao mesmo programa, eles são considerados duas sequências de execução separadas. Um processo com múltiplas threads (multithreaded) possui diversos contadores de programa, cada um apontando para a próxima instrução a ser executada para uma dada thread.

Um processo é a unidade de trabalho em um sistema. Tal sistema consiste em uma coleção de processos - alguns deles são processos do sistema operacional (aqueles que executam código do sistema) e o restante são processos do usuário (aqueles que executam código do usuário). Em potencial, todos esses processos podem ser executados ao mesmo tempo, com uma única CPU, por exemplo, por meio da multiplexação entre eles.

O sistema operacional é responsável pelas seguintes atividades, em conjunto com a gerência de processos:

- Escalonar processos e threads nas CPUs.
- Criar e remover os processos de usuário e de sistema.
- Suspender e retomar os processos.
- Prover mecanismos para o sincronismo de processos.
- Prover mecanismos para a comunicação entre processos.

Discutiremos as técnicas de gerência de processos nos [Capítulos 3 a 6](#) [Capítulo 4](#) [Capítulo 5](#) [Capítulo 6](#).

1.7 Gerência de memória

Conforme discutimos na [Seção 1.2.2](#), a memória principal é fundamental para a operação de um computador moderno. A memória principal é um grande conjunto de words ou bytes, variando em tamanho desde centenas de milhares até bilhões. Cada word ou byte possui seu próprio endereço. A memória principal é um repositório de dados rapidamente acessíveis, compartilhados pela CPU e pelos dispositivos de E/S. O processador central lê instruções da memória principal durante o ciclo de busca de instruções e tanto lê quanto escreve dados da memória principal durante o ciclo de busca de dados (pelo menos em uma arquitetura von Neumann). Como já vimos, a memória principal é o único dispositivo de armazenamento grande que a CPU pode endereçar e acessar diretamente. Por exemplo, para a CPU processar dados do disco, esses dados primeiro precisam ser transferidos para a memória principal pelas chamadas de E/S geradas pela CPU. Da mesma forma, as instruções precisam estar na memória para que a CPU as execute.

Para que um programa seja executado, ele precisa ser mapeado para endereços absolutos e carregado na memória. Enquanto o programa é executado, ele acessa instruções de programa e dados da memória, gerando esses endereços absolutos. Por fim, o programa termina, seu espaço de memória é declarado disponível e o próximo programa pode ser carregado e executado.

Para melhorar a utilização da CPU e a velocidade da resposta do computador aos seus usuários, computadores com objetivos gerais têm de manter vários programas na memória, criando a necessidade de uma gerência de memória. Muitos esquemas de gerência de memória diferentes são utilizados. Esses esquemas refletem diversas técnicas, e a eficácia dos diferentes algoritmos depende da situação em particular. Ao selecionar um esquema de gerência de memória devemos levar em consideração muitos fatores, especialmente no projeto de *hardware* do sistema. Cada algoritmo requer seu próprio suporte de hardware.

O sistema operacional é responsável pelas seguintes atividades relacionadas com a gerência de memória:

- Registrar quais partes da memória estão sendo usadas atualmente e por quem.
- Decidir quais processos (ou parte deles) e dados devem ser colocados e retirados da memória.
- Alocar e desalocar espaço de memória conforme a necessidade.

As técnicas de gerência de memória serão discutidas nos [Capítulos 8 e 9](#).

1.8 Gerência de armazenamento

Para o uso conveniente do sistema computadorizado pelos usuários, o sistema operacional fornece uma visão lógica e uniforme do armazenamento de informações. O sistema operacional se separa das propriedades físicas dos dispositivos de armazenamento para definir uma unidade de armazenamento lógica, o **arquivo**. O sistema operacional mapeia arquivos no meio físico e acessa esses arquivos por meio de dispositivos de armazenamento.

1.8.1 Gerência de sistema de arquivos

A gerência de arquivos é um dos componentes mais visíveis de um sistema operacional. Os computadores podem armazenar informações em vários tipos diferentes de meios físicos, sendo que o disco magnético, o disco óptico e a fita magnética são os mais comuns. Cada um desses meios possui suas próprias características e organização física. Cada meio é controlado por um dispositivo, como uma unidade de disco ou unidade de fita, que também tem suas próprias características exclusivas. Essas propriedades incluem velocidade de acesso, capacidade, taxa de transferência de dados e método de acesso (sequencial ou aleatório).

Um arquivo é uma coleção de informações relacionadas, definidas pelo seu criador. Normalmente, os arquivos representam programas (nos formatos fonte e objeto) e dados. Os arquivos de dados podem ser numéricos, alfabéticos, alfanuméricos ou binários. Os arquivos podem ter formato livre (por exemplo, arquivos de texto) ou podem ser formatados de forma rígida (por exemplo, campos fixos). É claro que o conceito de arquivo é extremamente genérico.

O sistema operacional implementa o conceito abstrato de um arquivo gerenciando meios de armazenamento em massa, como fitas e discos, e os dispositivos que os controlam. Além disso, os arquivos são organizados em diretórios (ou pastas) para facilitar o uso. Finalmente, quando vários usuários têm acesso a arquivos, é importante que se tenha o controle de por quem e de quais maneiras (por exemplo, leitura, escrita, remoção) eles podem ser acessados.

O sistema operacional é responsável pelas seguintes atividades relacionadas com a gerência de arquivos:

- Criação e remoção de arquivos.
- Criação e remoção de diretórios para organizar arquivos.
- Suporte a primitivas para manipulação de arquivos e diretórios.
- Mapeamento de arquivos em armazenamento secundário.
- Backup (cópia de reserva) de arquivos em meios de armazenamento estáveis (não voláteis).

As técnicas de gerência de arquivos serão discutidas nos [Capítulos 10 e 11](#).

1.8.2 Gerência de armazenamento em massa

Como já foi visto, sendo a memória principal muito pequena para acomodar todos os dados e programas, e como os dados que mantêm se perdem quando falta energia, o computador precisa fornecer armazenamento secundário como apoio para a memória principal. A maioria dos computadores modernos utiliza discos como o principal meio de armazenamento on-line para programas e dados. Quase todos os programas - incluindo compiladores, montadores (assemblers), processadores de textos, editores e formatadores - são armazenados em um disco até que sejam carregados para a memória e depois utilizam o disco como origem e destino do seu processamento. Logo, a gerência correta do armazenamento em disco é de importância vital para um computador. O sistema operacional é responsável pelas seguintes atividades relacionadas com o gerenciamento de disco:

- Gerenciamento do espaço livre.
- Alocação do armazenamento.
- Escalonamento do disco.

Como o armazenamento secundário é usado com muita frequência, ele precisa ser usado de forma eficiente. A velocidade de operação inteira de um computador pode depender das velocidades do subsistema de disco e dos algoritmos para manipular esse subsistema.

Porém, existem muitos usos para o armazenamento que são mais lentos e inferiores em custo (e, às vezes, de maior capacidade) do que o armazenamento secundário. Os backups de dados do disco, dados raramente usados e armazenamento por arquivamento por longo prazo são alguns exemplos. Unidades de fita magnética e suas fitas, e unidades e placas de CD e DVD são dispositivos típicos de **armazenamento terciário**. A mídia (fitas e placas ópticas) varia entre os formatos **WORM** (write-once, read-many-times) e **RW** (read-write).

O armazenamento terciário não é crucial para o desempenho do sistema, mas ainda precisa ser gerenciado. Alguns sistemas operacionais assumem essa tarefa, enquanto outros deixam o gerenciamento do armazenamento terciário para os programas de aplicação. Algumas das funções que os sistemas operacionais podem fornecer incluem montagem e desmontagem de mídia nos dispositivos, alocação e liberação dos dispositivos para uso exclusivo pelos processos e migração de

dados do armazenamento secundário para o terciário.

As técnicas para o gerenciamento de armazenamento secundário e terciário serão discutidas no [Capítulo 12](#).

1.8.3 Caching

O **caching** é um princípio importante dos sistemas computadorizados. As informações são mantidas em algum sistema de armazenamento (como a memória principal). À medida que são utilizadas, elas são copiadas para um sistema de armazenamento mais rápido - o **cache** - de modo temporário. Quando precisamos de determinada informação, primeiro verificamos se ela se encontra no cache. Se estiver, usamos as informações diretamente dele; caso contrário, usamos as informações da origem mais lenta, colocando uma cópia no cache, supondo que, em breve, precisaremos dela novamente.

Além disso, registradores programáveis internos, como registradores de índice, fornecem um cache de alta velocidade para a memória principal. O programador (ou compilador) implementa os algoritmos de alocação de registrador e substituição de registrador, para decidir quais informações devem ser mantidas nos registradores e quais devem permanecer na memória principal. Há também caches implementados totalmente no hardware. Por exemplo, a maioria dos sistemas possui um cache de instruções, para manter as próximas instruções que deverão ser executadas. Sem esse cache, a CPU teria de esperar vários ciclos enquanto uma instrução fosse buscada na memória principal. Por motivos semelhantes, a maioria dos sistemas possui um ou mais caches de dados de alta velocidade na hierarquia da memória. Neste texto, não estamos interessados nesses caches apenas de hardware, pois estão fora do controle do sistema operacional.

Como os caches possuem tamanho limitado, o **gerenciamento de cache** é um aspecto de projeto importante. A seleção cuidadosa do seu tamanho e de uma política de substituição pode resultar em um desempenho bastante aumentado. Veja na [Figura 1.11](#) uma comparação de desempenho de armazenamento em grandes estações de trabalho e pequenos servidores que apresentam necessidade de cache. Diversos algoritmos de substituição para caches controlados por software são discutidos no [Capítulo 9](#).

Nível	1	2	3	4
Nome	registradores	cache	memória principal	armazenamento em disco
Tamanho típico	< 1 KB	< 16 MB	< 64 GB	> 100 GB
Tecnologia de implementação	memória personalizada com múltiplas portas, CMOS	SRAM CMOS no chip ou fora do chip	DRAM CMOS	disco magnético
Tempo de acesso (ns)	0,25–0,5	0,5–25	80–250	5.000.000
Largura de banda (MB)	20.000–100.000	5.000–10.000	1.000–5.000	20–150
Gerenciado por	compilador	hardware	sistema operacional	sistema operacional
Backup por	cache	memória principal	disco	CD ou fita

FIGURA 1.11 Desempenho de vários níveis de armazenamento.

A memória principal pode ser vista como um **cache veloz** para o armazenamento secundário, pois os dados no armazenamento secundário precisam ser copiados para a memória principal antes de serem usados e, de forma recíproca, precisam estar na memória principal antes de serem movidos para o armazenamento secundário, por proteção. Os dados do sistema de arquivos, que residem permanentemente no armazenamento secundário, podem aparecer em diversos níveis na hierarquia de armazenamento. No nível mais alto, o sistema operacional pode manter um cache de dados do sistema de arquivo na memória principal. Além disso, discos eletrônicos em RAM (também conhecidos como **discos de estado sólido**) podem ser utilizados para o armazenamento de alta velocidade, os quais são acessados por meio da interface do sistema de arquivos. A maior parte do **armazenamento secundário** se encontra nos **discos magnéticos**. O armazenamento em disco magnético, por sua vez, é copiado para fitas magnéticas ou discos removíveis, para proteção contra a perda de dados no caso de uma falha no disco rígido. Alguns sistemas conseguem arquivar automaticamente antigos dados de arquivo do armazenamento secundário para o armazenamento terciário, como jukeboxes de fita, para reduzir o custo do armazenamento (ver [Capítulo 12](#)).

O movimento de informações entre os níveis de uma hierarquia de armazenamento pode ser explícito ou implícito, dependendo do projeto de hardware e do software de controle no sistema operacional. Por exemplo, a transferência de dados do cache para a CPU e seus registradores é uma função do hardware, sem a intervenção do sistema operacional. Por outro lado, a transferência de dados do disco para a memória normalmente é controlada pelo sistema operacional.

Em uma estrutura de armazenamento hierárquica, os mesmos dados podem aparecer em diferentes níveis do sistema de armazenamento. Por exemplo, suponha que o inteiro A que deverá ser incrementado em 1 esteja localizado no arquivo B, que reside em disco magnético. A operação de incremento prossegue emitindo primeiro uma operação de E/S para copiar o bloco do disco em

que A reside para a memória principal. Essa operação é seguida por A sendo copiado para o cache e para um registrador interno. Assim, a cópia de A aparece em vários lugares: no disco magnético, na memória principal, no cache e no registrador interno (Figura 1.12). Quando o incremento ocorre no registrador interno, os valores de A nos diversos sistemas de armazenamento diferem. O valor de A só se torna igual depois que o novo valor de A for escrito do registrador interno para o disco magnético.

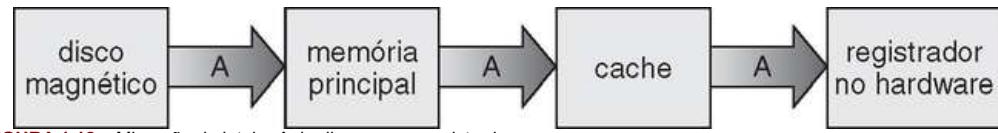


FIGURA 1.12 Migração do inteiro A do disco para o registrador.

Em um ambiente de computação em que somente um processo é executado de cada vez, esse arranjo não impõe qualquer dificuldade, pois um acesso ao inteiro A sempre será para a cópia no nível de hierarquia mais alto. Contudo, em um ambiente de multitarefa, no qual a CPU é alternada entre diversos processos, é preciso haver um cuidado extremo para garantir que, se vários processos desejarem acessar A, cada um desses processos poderá obter o valor de A atualizado mais recentemente.

A situação se torna ainda mais complicada em um ambiente com multiprocessadores (ver Figura 1.6), no qual, além de manter registradores internos, cada uma das CPUs também contém um cache local. Nesse tipo de ambiente, uma cópia de A pode existir simultaneamente em vários caches. Como todas as diversas CPUs podem executar ao mesmo tempo, precisamos nos certificar de que uma atualização no valor de A em um cache seja refletida imediatamente em todos os outros caches em que A reside. Essa situação é chamada de **coerência de cache**, e normalmente é um problema do hardware (tratado abaixo do nível do sistema operacional).

Em um ambiente distribuído, a situação torna-se ainda mais complexa. Nesse tipo de ambiente, várias cópias (ou réplicas) do mesmo arquivo podem ser mantidas em diversos computadores distribuídos no espaço. Como as diversas réplicas podem ser acessadas e atualizadas ao mesmo tempo, alguns sistemas distribuídos garantem que, quando uma réplica for atualizada em um lugar, todas as outras réplicas sejam atualizadas o mais cedo possível. Existem várias maneiras de alcançar isso, conforme discutiremos no Capítulo 17.

1.8.4 Sistemas de E/S

Uma das finalidades de um sistema operacional é ocultar do usuário as peculiaridades dos dispositivos de hardware. Por exemplo, no UNIX, as peculiaridades dos dispositivos de E/S são escondidas do resto do sistema operacional pelo **subsistema de E/S**. O subsistema de E/S consiste em vários componentes:

- Um componente da gerência de memória que inclui o uso de buffers, caches e spools.
- Uma interface genérica controladora de dispositivos.
- Drivers para dispositivos de hardware específicos.

Somente o driver de dispositivo conhece as peculiaridades do dispositivo específico ao qual está atribuído.

Na Seção 1.2.3, discutimos como os tratadores de interrupção e os drivers de dispositivos são usados na construção de subsistemas de E/S eficientes. No Capítulo 13, discutiremos como o subsistema de E/S realiza a interface com outros componentes do sistema, gerencia dispositivos, transfere dados e detecta o término da E/S.

1.9 Proteção e segurança

Se um computador possui diversos usuários e permite a execução simultânea dos diversos processos, então o acesso aos dados precisa ser regulado. Para esse propósito, os mecanismos garantem que os arquivos, segmentos de memória, CPU e outros recursos possam ser operados somente pelos processos que tiveram a devida autorização do sistema operacional. Por exemplo, o hardware de endereçamento de memória garante que um processo só possa ser executado dentro do seu próprio espaço de endereços. O temporizador assegura que nenhum processo possa obter o controle da CPU sem abrir mão do controle. Os registradores de controle de dispositivo não são acessíveis aos usuários, de modo que a integridade dos diversos dispositivos periféricos é protegida.

A **proteção**, portanto, é qualquer mecanismo que controle o acesso dos processos ou usuários aos recursos definidos por um sistema computadorizado. Esse mecanismo precisa fornecer meios para a especificação dos controles a serem impostos e os meios para a imposição.

A proteção pode melhorar a confiabilidade, detectando erros latentes nas interfaces entre os subsistemas componentes. A detecção antecipada de erros de interface constantemente impede a contaminação de um subsistema saudável por outro subsistema que não esteja funcionando bem. Além disso, um recurso desprotegido não pode se defender contra o uso (ou mau uso) por parte de um usuário não autorizado ou não habilitado. Um sistema orientado à proteção fornece meios para distinguir entre uso autorizado e não autorizado, conforme discutiremos no [Capítulo 14](#).

Apesar de um sistema ter proteção adequada, pode ser possível de falhas e permitir acesso inapropriado. Imagine um usuário cuja informação de autenticação (seus meios de identificar-se ao sistema) seja roubada. Seus dados poderiam ser copiados ou excluídos, embora a proteção de arquivo e memória estejam funcionando. É tarefa da **segurança** defender um sistema contra ataques externos e internos. Esses ataques podem ser de vários tipos, incluindo vírus e worms, ataques por recusa de serviço (que utilizam todos os recursos de um sistema e mantêm usuários legítimos fora do sistema), roubo de identidade e roubo de serviço (uso não autorizado de um sistema). A prevenção de alguns desses ataques é considerada uma função do sistema operacional em alguns sistemas, enquanto outros deixam a prevenção para um software de política ou outro. Devido ao crescimento alarmante de incidentes de segurança, os recursos de segurança do sistema operacional representam uma área de rápido crescimento da pesquisa e implementação. A segurança é discutida no [Capítulo 15](#).

Proteção e segurança precisam que o sistema seja capaz de distinguir entre todos os seus usuários. A maioria dos sistemas operacionais mantém uma lista de nomes de usuário e **identificadores de usuário (user IDs)** associados. No palavreado do Windows Vista, esse é um **ID de segurança (Security ID - SID)**. Esses IDs numéricos são exclusivos, um por usuário. Quando um usuário entra no sistema, o estágio de autenticação determina o ID de usuário apropriado para o usuário. Esse ID de usuário está associado a todos os processos e threads do usuário. Quando um ID precisa ser lido pelo usuário, ele é traduzido de volta para o nome do usuário por meio da lista de nomes de usuário.

Em algumas circunstâncias, queremos distinguir entre conjuntos de usuários e usuários individuais. Por exemplo, o proprietário de um arquivo em um sistema UNIX pode ter permissão para emitir todas as operações sobre esse arquivo, enquanto um conjunto de usuários selecionado pode ter permissão para ler o arquivo. Para conseguir isso, temos de definir um nome de grupo e o conjunto de usuários pertencentes a esse grupo. A funcionalidade do grupo pode ser implementada como uma lista de nomes de grupo em nível de sistema e **identificadores de grupo**. Um usuário pode estar em um ou mais grupos, dependendo das decisões de projeto do sistema operacional. As IDs de grupo do usuário também estão incluídas em cada processo e thread associado.

No curso do uso normal de um sistema, a ID de usuário e a ID de grupo para um usuário são suficientes. Porém, um usuário às vezes precisa **escalar privilégios** para ganhar permissões extras para uma atividade. O usuário pode precisar de acesso a um dispositivo que esteja restrito, por exemplo. Os sistemas operacionais fornecem diversos métodos para permitir a escalada de privilégios. No UNIX, por exemplo, o atributo `setuid` em um programa faz o programa executar com a ID de usuário do proprietário do arquivo, em vez de com a ID do usuário atual. O processo é executado com essa **UID efetiva** até que desative os privilégios extras ou termine sua execução.

1.10 Sistemas distribuídos

Um sistema distribuído é uma coleção de sistemas fisicamente dispersos, talvez heterogêneos, em rede, fornecendo ao usuário o acesso aos diversos recursos mantidos pelo sistema. O acesso a um recurso compartilhado aumenta a velocidade de computação, a funcionalidade, a disponibilidade de dados e a confiabilidade. Alguns sistemas operacionais generalizam o acesso à rede como uma forma de acesso a arquivos, com os detalhes da rede contidos no driver de dispositivo da interface de rede. Outros fazem os usuários usarem funções de rede específicas. Geralmente, os sistemas contêm um mix dos dois modos - por exemplo, FTP e NFS. FTP normalmente é uma ferramenta interativa, da linha de comandos, para copiar arquivos entre os sistemas em rede. NFS é um protocolo para permitir que o armazenamento em um servidor remoto apareça e se comporte exatamente como se o armazenamento estivesse ligado ao computador local. Os protocolos que criam um sistema distribuído podem afetar bastante a utilidade e a popularidade desse sistema.

Uma **rede**, em termos simples, é uma via de comunicação entre dois ou mais sistemas. Os sistemas distribuídos dependem de conexões de redes para a sua funcionalidade. As redes variam de acordo com os protocolos utilizados, as distâncias entre os nós e a mídia de transporte. Elas também variam em sua performance e confiabilidade. TCP/IP é o protocolo de rede mais comum, embora ATM e outros protocolos sejam bastante utilizados. Da mesma forma, o suporte aos protocolos pelo sistema operacional também varia. A maioria dos sistemas operacionais provê suporte ao TCP/IP, incluindo os sistemas Windows e UNIX. Alguns sistemas proveem suporte a protocolos próprios, de acordo com suas necessidades. Para um sistema operacional, um protocolo de rede precisa de um dispositivo de interface - um adaptador de rede, por exemplo - com um driver de dispositivo para gerenciá-lo, bem como o software para tratar dos dados. Esses conceitos serão discutidos no decorrer deste livro.

As redes são caracterizadas com base nas distâncias entre seus nós. Uma **rede local (Local-Area Network - LAN)** conecta computadores dentro de uma sala, um pavimento ou um prédio. Uma **rede de longa distância (Wide-Area Network - WAN)** normalmente liga prédios, cidades ou países. Uma empresa global pode ter uma WAN para conectar seus escritórios no mundo inteiro. Essas redes podem trabalhar com um protocolo ou vários. O advento contínuo de novas tecnologias motiva o aparecimento de novas formas de redes. Por exemplo, uma **rede metropolitana (Metropolitan-Area Network - MAN)** poderia conectar prédios dentro de uma cidade. Dispositivos BlueTooth e 802.11 utilizam a tecnologia sem fio para a comunicação por uma distância de vários metros, criando uma **rede doméstica**, como a que poderia existir dentro de uma casa.

A mídia de transporte das redes é igualmente variada. Ela inclui fios de cobre, fios de fibra e transmissões sem fio entre satélites, torres de micro-ondas e rádio. Quando dispositivos de computação são conectados a telefones celulares, eles criam uma rede. Até mesmo a comunicação por infravermelho, de curto alcance, pode ser usada para as redes. Em um nível rudimentar, sempre que os computadores se comunicam, eles utilizam ou criam uma rede.

Alguns sistemas operacionais levaram o conceito de redes e sistemas distribuídos além da noção de fornecer conectividade de rede. Um **sistema operacional de rede** é um sistema operacional que provê recursos como compartilhamento de arquivos pela rede e inclui um esquema de comunicação que permite que diferentes processos em diferentes computadores troquem mensagens. Um computador executando um sistema operacional de rede atua de modo independente de todos os outros computadores na rede, embora esteja cliente da rede e seja capaz de se comunicar com outros computadores em rede. Um sistema operacional distribuído provê um ambiente menos isolado. Os diferentes sistemas operacionais se comunicam a ponto de darem a ilusão de que um único sistema operacional controla a rede.

Explicamos sobre as redes de computadores e sistemas distribuídos nos [Capítulos 16 a 18](#) [Capítulo 17](#) [Capítulo 18](#).

1.11 Sistemas de uso específico

A discussão até aqui focou os sistemas computadorizados de uso geral, que todos nós estamos acostumados a usar. Porém, existem diferentes classes de sistemas computadorizados cujas funções são mais limitadas e cujo objetivo é lidar com domínios de computação limitados.

1.11.1 Sistemas de tempo real embutidos

Computadores embutidos são a forma prevalente de computadores atualmente. Esses dispositivos são encontrados em toda parte, desde motores de carro e robôs de manufatura até videocassetes e fornos de micro-ondas. Eles costumam ter tarefas muito específicas. Os sistemas em que eles executam normalmente são primitivos, e assim os sistemas operacionais fornecem recursos limitados. Eles têm pouca ou nenhuma interface com o usuário, preferindo gastar seu tempo monitorando e gerenciando dispositivos de hardware, como motores de automóvel e braços robóticos.

Esses sistemas embutidos variam muito. Alguns são computadores de uso geral, executando sistemas operacionais padrão – como UNIX – com aplicações de uso específico para implementar a funcionalidade. Outros são dispositivos de hardware com um sistema operacional de uso especial embutido, fornecendo apenas a funcionalidade desejada. Ainda outros são dispositivos de hardware com circuitos integrados específicos (**ASICs**) que realizam as tarefas sem um sistema operacional.

O uso de sistemas embutidos continua a se expandir. O poder desses dispositivos, tanto como unidades isoladas quanto como membros de redes e da Web, certamente também aumentará. Até mesmo agora, casas inteiras podem ser computadorizadas, de modo que um computador central – seja um computador de uso geral ou um sistema embutido – pode controlar o aquecimento e a iluminação, sistemas de alarme e até mesmo cafeteiras. O acesso à Web pode permitir que uma proprietária diga à sua casa para se aquecer antes de chegar nela. Algum dia, o refrigerador poderá ligar para o supermercado quando notar que o leite está acabando.

Sistemas embutidos quase sempre executam **sistemas de tempo real (real-time systems)**. Um sistema de tempo real é usado quando existem requisitos de tempo rígidos na operação de um processador ou do fluxo de dados; assim, ele é usado como dispositivo de controle em uma aplicação dedicada. Sensores trazem dados para o computador. O computador precisa analisar os dados e possivelmente ajustar os controles para modificar as entradas do sensor. Os sistemas que controlam experiências científicas, sistemas de imagens médicas, sistemas de controle industrial e certos sistemas de vídeo são sistemas de tempo real. Alguns sistemas de injeção de combustível de motor de automóvel, controladores de eletrodomésticos e sistemas de armas também são sistemas de tempo real.

Um sistema de tempo real possui restrições bem definidas e com tempo fixo. O processamento *precisa* ser feito dentro das restrições definidas, ou o sistema falhará. Por exemplo, não há sentido em comandar o braço de um robô de modo que pare *depois* de amassar o carro que estava montando. Um sistema de tempo real só funciona corretamente se retornar o resultado correto dentro de suas restrições de tempo. Compare esse sistema com um sistema de tempo compartilhado, no qual é desejável (mas não obrigatório) responder com rapidez, ou um sistema batch. Em um sistema batch, tarefas de longa duração são submetidas ao computador e executadas em interação humana até que sejam concluídas. Esses sistemas podem não ter quaisquer restrições de tempo.

Embora Java ([Seção 2.9.2](#)) normalmente não esteja associada a fornecer características de tempo real, existe uma especificação para Java em tempo real que estende as especificações para a linguagem Java, bem como a máquina virtual Java. A especificação de tempo real para Java (Real-Time Specification for Java - RTSJ) identifica uma interface de programação para criar programas Java que precisam ser executados dentro de demandas de tempo real. Para tratar das restrições de temporização dos sistemas de tempo real, a RTSJ enfrenta vários aspectos que podem afetar o tempo de execução de um programa Java, incluindo escalonamento de CPU e gerência de memória.

No [Capítulo 19](#), explicamos os sistemas embutidos de tempo real com muito mais detalhes. No [Capítulo 5](#), consideraremos o recurso de escalonamento necessário para implementar a funcionalidade de tempo real em um sistema operacional. No [Capítulo 9](#), descrevemos o projeto de gerência de memória para a computação de tempo real. Finalmente, no [Capítulo 22](#), descrevemos os componentes de tempo real do sistema operacional Windows XP.

1.11.2 Sistemas multimídia

A maioria dos sistemas operacionais é projetada para lidar com dados convencionais, como arquivos de texto, programas, documentos de processamento de textos e planilhas. Porém, uma tendência recente na tecnologia é a incorporação de **dados de multimídia** aos sistemas computadorizados. Os dados de multimídia consistem em arquivos de áudio e vídeo, além de arquivos convencionais. Esses dados diferem dos dados convencionais porque os dados de multimídia – como frames de

vídeo – precisam ser entregues (streamed) de acordo com certas restrições de tempo (por exemplo, 30 frames por segundo).

A multimídia descreve uma grande variedade de aplicações que estão em uso popular hoje em dia. Entre elas estão arquivos de áudio, como MP3, filmes de DVD, videoconferência e clipes de vídeo curtos com prévias de filmes e notícias baixadas pela Internet. As aplicações multimídia também podem incluir webcasts ao vivo (broadcasting pela World Wide Web) ou discursos ou eventos esportivos e até mesmo webcams ao vivo, permitindo que um espectador em Manhattan observe clientes em uma lanchonete em Paris. As aplicações multimídia não precisam ser áudio ou vídeo; em vez disso, uma aplicação multimídia normalmente inclui uma combinação de ambos. Por exemplo, um filme pode consistir em trilhas de áudio e vídeo separadas. As aplicações multimídia também não precisam ser enviadas apenas aos computadores pessoais de desktop. Cada vez mais, elas estão sendo direcionadas para dispositivos menores, como PDAs e telefones celulares. Por exemplo, um negociador de ações pode ter informações sobre ações entregues sem fio e em tempo real ao seu PDA.

No [Capítulo 20](#), exploramos as demandas das aplicações multimídia, como os dados de multimídia diferem dos dados convencionais e como a natureza desses dados afeta o projeto dos sistemas operacionais que dão suporte aos requisitos dos sistemas multimídia.

1.11.3 Sistemas portáteis

Sistemas portáteis incluem assistentes digitais pessoais (Personal Digital Assistants – PDAs), como Palm, Pocket-PCs ou telefones celulares, muitos dos quais usam sistemas operacionais embutidos, de uso especial. Os desenvolvedores de sistemas portáteis e suas aplicações encaram muitos desafios, sendo a maioria relacionada com o tamanho limitado desses dispositivos. Por exemplo, um PDA tem, em geral, cerca de 12,5 cm de altura e 7,5 cm de largura, e pesa em torno de 200 g. Devido ao seu tamanho, a maioria dos dispositivos portáteis possui pequena quantidade de memória, processadores lentos e telas de vídeo pequenas. Vejamos cada uma dessas limitações.

A capacidade de memória física em um dispositivo portátil depende do dispositivo, mas em geral está entre 1 MB e 1 GB de memória. (Compare isso com um PC típico ou estação de trabalho, que pode ter vários gigabytes de memória.) Como resultado, o sistema operacional e as aplicações precisam gerenciar a memória de forma eficiente. Isso inclui retornar toda a memória alocada de volta ao gerenciador de memória, quando ela não estiver sendo usada. No [Capítulo 9](#), exploraremos a memória virtual, que permite aos desenvolvedores escreverem programas que se comportam como se o sistema tivesse mais memória do que realmente existe fisicamente. Hoje, não existem muitos dispositivos portáteis que utilizam técnicas de memória virtual, de modo que os desenvolvedores de programas precisam trabalhar dentro dos confins da memória física limitada.

Um segundo motivo de preocupação para os desenvolvedores de dispositivos portáteis é a velocidade do processador usado nos dispositivos. Os processadores da maioria desses dispositivos trabalham em uma fração da velocidade de um processador do PC. Os processadores mais rápidos exigem mais energia. Para incluir um processador mais rápido em um dispositivo portátil, seria necessária uma bateria maior, a qual precisaria ser substituída (ou recarregada) com mais frequência. A maioria dos dispositivos portáteis usa processadores menores e mais lentos, que consomem menos energia. Portanto, o sistema operacional e as aplicações precisam ser elaborados para não sobrecarregar o processador.

A última questão que os projetistas de programas confrontam para dispositivos portáteis é E/S. A falta de espaço físico limita os métodos de entrada a pequenos teclados, reconhecimento de escrita manual ou pequenos teclados baseados em tela. As pequenas telas limitam as opções de saída. Enquanto um monitor para um computador doméstico pode medir até 30 polegadas, a tela de um dispositivo portátil não costuma ter mais de 3 polegadas. Tarefas comuns, como ler o correio eletrônico ou navegar pelas páginas Web, precisam ser condensadas em telas menores. Uma técnica para exibir o conteúdo em páginas Web é a **Web clipping**, na qual somente um pequeno subconjunto de uma página Web é entregue e exibido no dispositivo portátil.

Alguns dispositivos portáteis utilizam a tecnologia sem fio, como BlueTooth ou 802.11, permitindo o acesso remoto ao correio eletrônico e à navegação Web. Telefones celulares com conectividade com a Internet entram nessa categoria. Entretanto, para PDAs que não proveem acesso sem fio, o download de dados normalmente exige que o usuário primeiro faça o download dos dados para um PC ou estação de trabalho e depois faça o download dos dados para o PDA. Alguns PDAs permitem que os dados sejam copiados diretamente de um dispositivo para outro, usando um enlace infravermelho.

Em geral, as limitações na funcionalidade dos PDAs é compensada por sua conveniência e portabilidade. Seu uso continua a se expandir à medida que as conexões da rede se tornam mais disponíveis e outras opções, como câmeras digitais e players MP3, expandem sua utilidade.

1.12 Ambientes de computação

Até agora fornecemos uma visão geral da organização do sistema computadorizado e os principais componentes do sistema operacional. Agora daremos uma rápida visão geral de como esses sistemas são usados em uma série de ambientes de computação.

1.12.1 Computação tradicional

À medida que a computação amadurece, as linhas que separam muitos dos ambientes de computação tradicionais estão se tornando menos nítidas. Considere o “ambiente de escritório típico”. Há poucos anos, esse ambiente consistia em PCs conectados a uma rede, com servidores fornecendo serviços de arquivo e impressão. O acesso remoto era desajeitado, e a portabilidade era obtida com o uso de laptops. Terminais conectados a mainframes também eram comuns em muitas empresas, com ainda menos opções de acesso remoto e portabilidade.

A tendência atual é fornecer mais formas de acessar esses ambientes computacionais. As tecnologias Web estão alargando as fronteiras da computação tradicional. As empresas estabelecem **portais**, que oferecem acessibilidade da Web a seus servidores internos. **Network computers (NCs)** são basicamente terminais que entendem a computação baseada na Web. Os computadores portáteis podem sincronizar com os PCs para permitir o uso bastante portável das informações da empresa. Os PDAs portáteis também podem se conectar a **redes sem fio** para usar o portal Web da empresa (além dos inúmeros outros recursos da Web).

Em casa, a maioria dos usuários tinha um único computador com uma conexão lenta, via modem, com o escritório, a Internet ou ambos. Hoje, velocidades de conexão de rede que só existiam a um grande custo são relativamente baratas, dando a alguns usuários mais acesso a mais dados. Essas conexões de dados rápidas estão permitindo que computadores domésticos enviem páginas Web e utilizem redes que incluem impressoras, PCs clientes e servidores. Algumas casas possuem até mesmo **firewalls** para proteger esses ambientes domésticos contra brechas de segurança. Esses firewalls custavam milhares de dólares há alguns anos e nem sequer existiam até a década de 1980.

Na segunda metade do século passado, os recursos de computação eram escassos. (Antes disso, eles não existiam!) Por um período, os sistemas eram em lote (batch) ou interativos. Os sistemas em batch processavam jobs em massa, com entrada predeterminada (de arquivos ou outras fontes de dados). Os sistemas interativos esperavam a entrada dos usuários. Para otimizar o uso dos recursos computacionais, diversos usuários compartilhavam tempo nesses sistemas. Os sistemas de tempo compartilhado usavam um temporizador e algoritmos de escalonamento para percorrer os processos rapidamente na CPU, dando a cada usuário uma fatia dos recursos.

Hoje, os sistemas tradicionais de tempo compartilhado são incomuns. A mesma técnica de escalonamento ainda está em uso nas estações de trabalho e servidores, mas constantemente todos os processos pertencem ao mesmo usuário (ou a um único usuário e ao sistema operacional). Os processos do usuário, e os processos do sistema que fornecem serviços ao usuário, são gerenciados de modo que cada um frequentemente apanhe uma fatia do tempo do computador. Considere as janelas criadas enquanto um usuário está trabalhando em um PC, por exemplo, e o fato de que elas podem estar realizando diferentes tarefas ao mesmo tempo.

1.12.2 Sistemas cliente-servidor

À medida que os PCs se tornaram mais rápidos, mais poderosos e mais baratos, os projetistas saíram da arquitetura de sistema centralizada. Os terminais conectados a sistemas centralizados agora estão sendo suplantados por PCs. De modo correspondente, a funcionalidade da interface com o usuário, uma vez tratada diretamente pelos sistemas centralizados, está sendo cada vez mais tratada pelos PCs também. Como resultado, os sistemas centralizados atuam hoje como sistemas **servidores** para satisfazer as requisições geradas por **sistemas clientes**. A estrutura geral de um sistema cliente-servidor (**client-server**) está representada na [Figura 1.13](#).

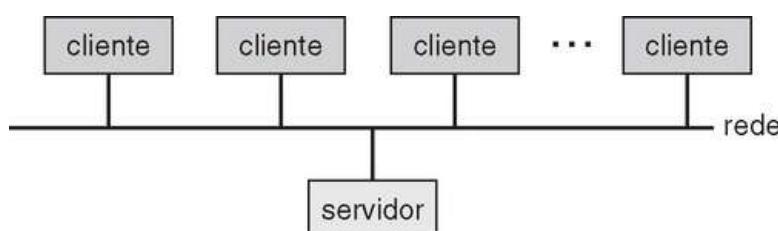


FIGURA 1.13 Estrutura geral de um sistema cliente-servidor.

Os sistemas servidores podem ser categorizados, de modo geral, como servidores de computação

e servidores de arquivos:

- O **sistema servidor de processamento (compute-server)** fornece uma interface para a qual um cliente pode enviar requisições para realizar uma ação (por exemplo, leitura de dados); em resposta, o servidor executa a ação e envia os resultados de volta ao cliente. Um servidor executando um banco de dados que responde a requisições de dados do cliente é um exemplo desse tipo de sistema.
- O **sistema servidor de arquivos (file-server system)** fornece uma interface para o sistema de arquivos, na qual o cliente pode criar, atualizar, ler e excluir arquivos. Um exemplo desse sistema é um servidor Web que fornece arquivos aos clientes que utilizam navegadores Web.

1.12.3 Sistemas peer-to-peer

Outra estrutura para um sistema distribuído é o modelo de sistema **peer-to-peer** (P2P). Nesse modelo, clientes e servidores não são diferenciados um do outro; em vez disso, todos os nós dentro do sistema são considerados iguais, e cada um pode atuar tanto como um cliente quanto como um servidor, dependendo de estar solicitando ou fornecendo um serviço. Os sistemas peer-to-peer fornecem vantagem em relação aos sistemas cliente-servidor tradicionais. Em um sistema cliente-servidor, o servidor é um gargalo, mas em um sistema peer-to-peer os serviços podem ser fornecidos por vários nós, distribuídos por meio da rede.

Para participar de um sistema peer-to-peer, um nó precisa primeiro se conectar à rede. Quando ele estiver conectado à rede, poderá começar a fornecer serviços para outros nós na rede (e requisitar serviços dela). Para determinar quais serviços estão disponíveis, existem duas maneiras gerais:

- Quando um nó se conecta a uma rede, ele registra seu serviço em um serviço de pesquisa centralizado na rede. Qualquer nó que deseja um serviço específico deve contatar primeiro esse serviço de pesquisa centralizado para determinar qual nó fornece o serviço. O restante da comunicação ocorre entre o cliente e o provedor do serviço.
- Um nó que esteja atuando como cliente precisa primeiro descobrir qual nó fornece o serviço desejado, enviando um broadcast de requisição de serviço a todos os outros nós na rede. O nó (ou nós) que está(ão) fornecendo esse serviço responde(m) ao nó que fez a requisição. Para dar suporte a essa técnica, um *protocolo de descoberta* (discovery protocol) precisa ser fornecido para permitir que os nós descubram os serviços fornecidos por outros nós na rede.

As redes peer-to-peer foram muito populares no final da década de 1990, com vários serviços de compartilhamento de música, como Napster e Gnutella, permitindo que os nós trocassem arquivos entre si. O sistema Napster utiliza uma técnica semelhante ao primeiro tipo descrito anteriormente; um servidor centralizado mantém um índice de todos os arquivos armazenados nos diversos nós da rede Napster, e a troca de arquivos real ocorre entre esses nós. O sistema Gnutella utiliza uma técnica semelhante ao segundo tipo; um cliente envia requisições de arquivo por broadcast para outros nós no sistema, e os nós que podem atender à requisição respondem diretamente ao cliente. O futuro da troca de arquivos continua incerto, pois muitos dos arquivos têm direito autoral (música, por exemplo) e há as leis que controlam a distribuição de material com direito autoral. No entanto, de qualquer forma, a tecnologia peer-to-peer sem dúvida terá um papel importante no futuro de muitos serviços, como pesquisa, troca de arquivos e correio eletrônico.

1.12.4 Computação baseada na Web

A Web tornou-se onipresente, ocasionando mais acesso por uma maior variedade de dispositivos do que se sonhava alguns anos atrás. Os PCs ainda são os dispositivos de acesso mais prevalentes, com estações de trabalho, PDAs portáteis e até mesmo telefones celulares também fornecendo acesso.

A computação via Web aumentou a ênfase nas redes. Dispositivos que anteriormente não estavam interligados em rede agora incluem acesso com ou sem fio. Dispositivos que estavam ligados em rede agora possuem conectividade de rede mais rápida, oferecida por tecnologia de rede melhorada, código de implementação de rede otimizado ou ambos.

A implementação da computação baseada na Web deu origem a novas categorias de dispositivos, como **balanceadores de carga**, que distribuem as conexões da rede entre um banco de servidores semelhantes. Sistemas operacionais como Windows 95, que atuavam como clientes Web, evoluíram para Windows 2000 e Windows 7, que podem atuar como servidores Web e também como clientes. Em geral, a Web aumentou a complexidade dos dispositivos, pois seus usuários exigem que estejam preparados para a Web.

1.13 Sistemas operacionais de fonte aberto

O estudo dos sistemas operacionais, como já observamos, é facilitado pela disponibilidade de uma grande quantidade de versões de fonte aberto. Os **sistemas operacionais de fonte aberto** são aqueles disponibilizados no formato de código-fonte, em vez de um código binário compilado. Linux é o sistema operacional de fonte aberto mais famoso, enquanto Microsoft Windows é um exemplo bem conhecido da abordagem oposta, de **fonte fechado**. Nesta seção, discutimos diversos aspectos dos sistemas de fonte aberto em geral e depois descrevemos como os estudantes podem acessar códigos-fonte para os sistemas operacionais Linux, UnixBSD e Solaris.

1.13.1 Benefícios dos sistemas de fonte aberto

O trabalho com sistemas de fonte aberto oferece muitos benefícios a programadores e estudantes de programação. Começar com o código-fonte permite que o programador produza um código binário que possa ser executado em um sistema. Fazer o oposto - **engenharia reversa** do código-fonte a partir dos binários - é muito trabalhoso, e itens úteis, como comentários, nunca são recuperados. Além disso, aprender sistemas operacionais examinando o código-fonte real, em vez de ler resumos desse código, pode ser extremamente útil. Com o código-fonte em mãos, o estudante pode modificar o sistema operacional e depois compilar e executar o código para experimentar essas mudanças, o que é uma excelente ferramenta de aprendizado. Este texto inclui projetos que envolvem a modificação do código-fonte do sistema operacional, enquanto também descreve algoritmos em um alto nível, para garantir que tópicos importantes sobre sistema operacional sejam abordados. No decorrer do texto, oferecemos indicações de exemplos de código de fonte aberto para um estudo mais aprofundado.

Outro benefício dos sistemas operacionais de fonte aberto é a grande comunidade de programadores interessados (e normalmente não pagos) que contribuem para o código, ajudando a depurá-lo, analisá-lo, oferecer suporte e sugerir mudanças. Comprovadamente, o código de fonte aberto é mais seguro do que o código de fonte fechado, pois muito mais olhos estão examinando o código. Certamente, o código de fonte aberto possui bugs, mas seus defensores argumentam que os bugs costumam ser localizados e reparados mais rapidamente, devido ao número de pessoas usando e examinando o código. Empresas que ganham receita com a venda de seus programas costumam hesitar em tornar seu código de fonte aberto, mas Red Hat, SUSE, Sun e diversas outras empresas estão fazendo exatamente isso e mostrando que as empresas comerciais se beneficiam, em vez de perder, quando abrem seu código. A receita pode ser gerada por meio de contratos de suporte e venda do hardware no qual o software é executado, por exemplo.

1.13.2 História dos sistemas de fonte aberto

Nos primeiros dias da computação moderna (ou seja, na década de 1950), havia muito software disponível no formato de fonte aberto. Os hackers originais (entusiastas por computador) no Tech Model Railroad Club do MIT deixavam seus programas em gavetas para que outros trabalhassem neles. Grupos de usuários “caseiros” trocavam código durante seus encontros. Mais tarde, grupos de usuários específicos da empresa, como os da Digital Equipment Corporation (DEC), aceitavam contribuições de programas de fonte aberto, coletavam-nos em fitas e distribuíam as fitas aos membros interessados.

Empresas de computador e software por fim buscaram limitar o uso de seu software a computadores autorizados e clientes pagantes. A divulgação apenas dos arquivos binários compilados a partir do código-fonte, ao invés do próprio código-fonte, ajudou-os a conseguir esse objetivo, além de proteger seu código e suas ideias dos seus concorrentes. Outra questão envolvia o material protegido por direito autoral. Os sistemas operacionais e outros programas podem limitar a capacidade de reproduzir filmes e música ou exibir livros eletrônicos a computadores autorizados. Essa **proteção contra cópia**, ou gerência digital de direitos (Digital Rights Management - DRM), não seria eficaz se o código-fonte que implementava esses limites fosse publicado. As leis de muitos países, incluindo o U.S. Digital Millennium Copyright Act (DMCA), tornam ilegal realizar a engenharia reversa do código protegido ou tentar de alguma outra forma burlar a proteção contra cópia.

Para combater o movimento para limitar o uso e a redistribuição de software, Richard Stallman iniciou, em 1983, o projeto GNU, para criar um sistema operacional gratuito, de fonte aberto, compatível com UNIX. Em 1985, ele publicou o GNU Manifesto, que argumenta que todo o software deveria ser livre e com fonte aberto. Ele também formou a **Free Software Foundation (FSF)** com o objetivo de encorajar a livre troca de código-fonte de software e o uso gratuito desse software. Em vez de proteger seu software com direito autoral, a FSF permite a cópia (“copyleft”) do software para encorajar o compartilhamento e a melhoria. A **Licença Pública Geral (General Public License - GPL)** GNU codifica a permissão de cópia e é uma licença comum sob a qual o software gratuito é liberado. Fundamentalmente, a GPL requer que o código-fonte seja distribuído com

quaisquer binários e que quaisquer mudanças feitas ao código-fonte sejam lançadas sob a mesma licença GPL.

1.13.3 Linux

Como um exemplo de um sistema operacional de fonte aberto, considere o **GNU/Linux**. O projeto GNU produziu muitas ferramentas compatíveis com UNIX, incluindo compiladores, editores e utilitários, mas nunca lançou um kernel. Em 1991, um estudante na Finlândia, Linus Torvalds, lançou um kernel rudimentar tipo UNIX usando compiladores e ferramentas GNU, convidando colaborações no mundo inteiro. O advento da Internet significou que qualquer um interessado poderia baixar o código-fonte, modificá-lo e submeter as alterações a Torvalds. A divulgação de atualizações uma vez por semana permitiu que esse sistema operacional, denominado Linux, crescesse rapidamente, aprimorado por milhares de programadores.

O sistema operacional GNU/Linux resultante gerou centenas de **distribuições** exclusivas, ou builds customizados do sistema. As principais distribuições incluem RedHat, SUSE, Fedora, Debian, Slackware e Ubuntu. As distribuições variam em função, utilidade, aplicações instaladas, suporte do hardware, interface do usuário e propósito. Por exemplo, RedHat Enterprise Linux é voltado para uso comercial amplo. PCLinuxOS é um **LiveCD** - um sistema operacional que pode ser inicializado e executado a partir de um CD-ROM sem ser instalado no disco rígido de um sistema. Uma variante do PCLinuxOS, o "PCLinuxOS Supergamer DVD", é um **LiveDVD** que inclui drivers gráficos e jogos. Um jogador pode executá-lo em qualquer sistema compatível dando um boot a partir do DVD. Quando o jogador termina e reinicia o sistema, este retorna ao sistema operacional instalado.

O acesso ao código-fonte do Linux varia por release. Aqui, consideraremos o Ubuntu Linux. Ubuntu é uma distribuição popular do Linux, que vem em diversos tipos, incluindo aqueles preparados para desktops, servidores e estudantes. Seu fundador paga pela impressão e remessa dos DVDs contendo o binário e o código-fonte (o que ajuda a torná-lo popular). As seguintes etapas explicam uma forma de explorar o código-fonte do kernel Ubuntu em sistemas que admitem a ferramenta gratuita "VMware Player":

- Faça o download do player em <http://www.vmware.com/download/player/> e instale-o no seu sistema.
- Faça o download de uma máquina virtual contendo Ubuntu. Centenas de "aparelhos" ou imagens de máquina virtual, pré-instaladas com sistemas operacionais e aplicações, estão disponíveis na VMware, em <http://www.vmware.com/appliances/>.
- Dê partida na máquina virtual dentro do VMware Player.
- Adquira o código-fonte do release do kernel de seu interesse, como 2.6, executando wget <http://www.kernel.org/pub/linux/kernel/v2.6/linux-2.6.18.1.tar.bz2> de dentro da máquina virtual Ubuntu.
- Descompacte e faça o "untar" do arquivo baixado por meio do comando tar xjf linux-2.6.18.1.tar.bz2.
- Explore o código-fonte do kernel Ubuntu, que agora está em ./linux-2.6.18.1.

Para saber mais sobre o Linux, leia o [Capítulo 21](#). Para saber mais sobre máquinas virtuais, consulte a [Seção 2.8](#).

1.13.4 BSD UNIX

UnixBSD possui uma história mais longa e mais complicada do que Linux. Ele começou em 1978 como uma derivação do UNIX da AT&T. As releases da Universidade da Califórnia em Berkeley vieram em formato de fonte e binário, mas não eram fonte aberto, pois isso exigiria uma licença da AT&T. O desenvolvimento do UnixBSD foi atrasado por um processo movido pela AT&T, mas por fim, em 1994, foi lançada uma versão de fonte aberto, totalmente funcional, a 4.4BSD-lite.

Assim como o Linux, existem muitas distribuições do UnixBSD, incluindo FreeBSD, NetBSD, OpenBSD e DragonFly BSD. Para explorar o código-fonte do FreeBSD, basta baixar a imagem da máquina virtual da versão de interesse e inicializá-la de dentro do VMware, conforme explicamos para o Ubuntu Linux. O código-fonte vem com a distribuição e está armazenado em /usr/src/. O código-fonte do kernel está em /usr/src/sys. Por exemplo, para examinar o código de implementação da memória virtual no kernel do FreeBSD, consulte os arquivos em /usr/src/sys/vm.

Darwin, o componente do kernel do Mac OS X, é baseado no UnixBSD, e também é um código de fonte aberto. Esse código-fonte está disponível em <http://www.opensource.apple.com/darwinsource/>. Toda versão do Mac OS X possui seus componentes de fonte aberto postados nesse local. O nome do pacote que contém o kernel é "xnu". O código-fonte para o kernel do Mac OS X revisão 1228 (o código-fonte do Mac OS X Leopard) pode ser encontrado em www.opensource.apple.com/darwinsource/tarballs/apsl/xnu-1228.tar.gz. A Apple também oferece diversas ferramentas para o desenvolvedor, documentação e suporte em <http://connect.apple.com>.

1.13.5 Solaris

Solaris é o sistema operacional comercial da Sun Microsystems, baseado no UNIX. Originalmente, o sistema operacional **SunOS** da Sun era baseado no UnixBSD. Em 1991, a Sun passou para o System V UNIX da AT&T como sua base. Em 2005, a Sun abriu o fonte de parte do seu código do Solaris e, com o tempo, a empresa acrescentou cada vez mais a essa base de código de fonte aberto. Infelizmente, nem todo o Solaris é um código de fonte aberto, pois parte dele ainda pertence à AT&T e outras empresas. Porém, o Solaris pode ser compilado a partir do fonte aberto e vinculado aos binários dos componentes de fonte fechado, de modo que ainda pode ser explorado, modificado, compilado e testado.

O código-fonte está disponível em <http://opensolaris.org/os/downloads/>. Também estão disponíveis as distribuições pré-compiladas, baseadas no código-fonte, a documentação e grupos de discussão. Não é preciso baixar o conjunto inteiro do código-fonte a partir do site, pois a Sun permite que os visitantes explorem o código-fonte on-line, por meio de um navegador de código-fonte.

1.13.6 Conclusão

O movimento de software livre está levando legiões de programadores a criarem milhares de projetos de fonte aberto, incluindo sistemas operacionais. Sites como <http://freshmeat.net/> e <http://distrowatch.com/> oferecem portais para muitos desses projetos. Como já dissemos, os projetos de fonte aberto permitem que estudantes utilizem o código-fonte como uma ferramenta de aprendizado. Os estudantes podem modificar programas e testá-los, ajudar a encontrar e corrigir erros, além de explorar de outras maneiras os sistemas operacionais maduros, com todos os recursos, compiladores, ferramentas, interfaces de usuário e outros tipos de programas. A disponibilidade do código-fonte para projetos históricos, como o Multics, pode ajudar os estudantes a compreender esses projetos e acumular conhecimento que ajudará na implementação de novos projetos.

GNU/Linux, UnixBSD e Solaris são sistemas operacionais de fonte aberto, mas cada um tem seus próprios objetivos, utilidade, licenças e propósito. Às vezes, as licenças não são mutuamente exclusivas, além de haver polinização cruzada, permitindo melhorias rápidas em projetos de sistema operacional. Por exemplo, diversos componentes importantes do Solares foram portados para o UnixBSD. As vantagens do software livre e do código de fonte aberto provavelmente aumentarão a quantidade e a qualidade dos projetos de fonte aberto, levando a um aumento no número de indivíduos e empresas que utilizam esses projetos.

1.14 Resumo

Um sistema operacional é o software que gerencia o hardware do computador e também fornece um ambiente em que os programas de aplicação podem ser executados. Talvez o aspecto mais visível de um sistema operacional seja a interface com o sistema computadorizado que ele provê ao usuário humano.

Para um computador realizar sua tarefa de executar programas, estes precisam estar na memória principal. A memória principal é a única área de armazenamento grande que o processador pode acessar diretamente. Ela consiste em um arranjo de words ou bytes, variando em tamanho de centenas de milhares para bilhões. Cada word na memória tem seu próprio endereço. A memória principal normalmente é um dispositivo de armazenamento volátil, que perde seu conteúdo quando a alimentação é desligada ou perdida. A maioria dos sistemas computadorizados fornece armazenamento secundário como uma extensão da memória principal. O armazenamento secundário provê uma forma de armazenamento não volátil que é capaz de manter grande quantidade de dados permanentemente. O dispositivo de armazenamento secundário mais comum é um disco magnético, que fornece armazenamento de programas e dados.

A grande variedade de sistemas de armazenamento em um sistema computadorizado pode ser organizada em uma hierarquia, de acordo com a velocidade e o custo. Os níveis mais altos são caros, mas são rápidos. Enquanto descemos na hierarquia, o custo por bit geralmente diminui, ao passo que o tempo de acesso geralmente aumenta.

Existem várias estratégias diferentes para se projetar um sistema computadorizado. Os sistemas de processador único possuem, evidentemente, apenas um único processador, enquanto os sistemas multiprocessados contêm dois ou mais processadores que compartilham memória física e dispositivos periféricos. O projeto multiprocessado mais comum é o multiprocessamento simétrico (ou SMP), onde todos os processadores são considerados peers e executados independentemente uns dos outros. Os sistemas em clusters são uma forma especializada de multiprocessador e consistem em múltiplos sistemas de computador conectados por uma rede local.

Para melhor utilizar a CPU, os sistemas operacionais modernos empregam a multiprogramação, que permite que várias tarefas estejam na memória ao mesmo tempo, garantindo, assim, que a CPU sempre tenha uma tarefa a executar. Os sistemas de tempo compartilhado são uma extensão da multiprogramação na qual os algoritmos de escalonamento de CPU alternam rapidamente entre as tarefas, dando a ilusão de que cada tarefa está sendo executada simultaneamente.

O sistema operacional precisa garantir a operação correta do sistema computadorizado. Para impedir que os programas do usuário interfiram na operação correta do sistema, o hardware tem dois modos: modo usuário e modo kernel. Determinadas instruções são privilegiadas e só podem ser executadas no modo kernel. A memória em que o sistema operacional reside também precisa ser protegida contra modificação pelo usuário. Um temporizador impede loops infinitos. Esses recursos (modo dual, instruções privilegiadas, proteção de memória e interrupção de temporizador) são blocos de montagem básicos usados pelos sistemas operacionais para obter a operação correta.

Um processo (ou tarefa) é a unidade de trabalho fundamental em um sistema operacional. A gerência de processos inclui a criação e exclusão de processos e o fornecimento de mecanismos para os processos se comunicarem e se sincronizarem entre si. Um sistema operacional gerencia a memória registrando quais partes da memória estão sendo usadas e por quem. O sistema operacional também é responsável por alocar dinamicamente e liberar o espaço da memória. O espaço do armazenamento também é gerenciado pelo sistema operacional, incluindo fornecer os sistemas de arquivo para representar arquivos e diretórios e gerenciar o espaço nos dispositivos de armazenamento em massa.

Os sistemas operacionais também precisam se preocupar com a proteção e a segurança do sistema operacional e dos usuários. Medidas de proteção são mecanismos que controlam o acesso de processos ou usuários aos recursos que estão disponíveis pelo sistema computadorizado. As medidas de segurança são responsáveis por defender um sistema de computador de ataques externos ou internos.

Os sistemas distribuídos permitem que os usuários compartilhem recursos em hosts espalhados geograficamente, conectados por uma rede de computadores. Os serviços podem ser fornecidos pelo modelo cliente-servidor ou peer-to-peer. Em um sistema em clusters, diversas máquinas podem realizar cálculos sobre os dados que residem no armazenamento compartilhado, e a computação pode continuar mesmo quando algum subconjunto de membros do cluster falha.

LANs e WANs são os dois tipos básicos de redes. LANs permitem que os processadores distribuídos por uma pequena área geográfica se comuniquem, enquanto as WANs permitem que os processadores distribuídos por uma área maior se comuniquem. As LANs normalmente são mais rápidas que as WANs.

Existem vários sistemas computadorizados que atendem a finalidades específicas. Eles incluem sistemas operacionais de tempo real projetados para ambientes embutidos, como dispositivos de consumidor, automóveis e robótica. Os sistemas operacionais de tempo real têm restrições de tempo bem definidas, fixadas. O processamento *precisa* ser feito dentro das restrições definidas ou então o

sistema fracassará. Os sistemas multimídia envolvem a remessa de dados multimídia e normalmente têm requisitos especiais de exibir ou tocar um áudio, vídeo ou streams de áudio e vídeo sincronizados.

Recentemente, a influência da Internet e da World Wide Web encorajou o desenvolvimento de sistemas operacionais que incluem navegadores Web e software de rede e comunicação como recursos integrados.

O movimento do software livre criou milhares de projetos de fonte aberto, incluindo sistemas operacionais. Devido a esses projetos, os estudantes podem usar o código-fonte como uma ferramenta de aprendizado. Eles podem modificar programas e testá-los, ajudar a encontrar e reparar erros e explorar de outras maneiras sistemas operacionais maduros, completos, além de compiladores, ferramentas, interfaces de usuário e outros tipos de programas.

GNU/Linux, UnixBSD e Solaris são sistemas operacionais de fonte aberto. As vantagens do software livre e do código de fonte aberto provavelmente aumentarão a quantidade e a qualidade dos projetos de fonte aberto, levando a um aumento no número de indivíduos e empresas que utilizam esses projetos.

Exercícios práticos

- 1.1. Quais são as três principais finalidades de um sistema operacional?
- 1.2. Quais são as principais diferenças entre os sistemas operacionais para computadores mainframe e computadores pessoais?
- 1.3. Relacione as quatro etapas que são necessárias para executar um programa em uma máquina completamente dedicada – um computador que esteja executando apenas esse programa.
- 1.4. Enfatizamos a necessidade de um sistema operacional fazer uso eficiente do hardware de computação. Quando é apropriado que o sistema operacional abra mão desse princípio e “desperdice” recursos? Por que esse tipo de sistema não é realmente desperdiçador?
- 1.5. Qual é a principal dificuldade que um programador deverá contornar na escrita de um sistema operacional para um ambiente de tempo real?
- 1.6. Considere as diversas definições de *sistema operacional*. Em seguida, considere se o sistema operacional deverá incluir aplicações como navegadores Web e programas de e-mail. Argumente tanto que ele deve quanto que não deve e dê suporte às suas respostas.
- 1.7. Como a distinção entre o modo kernel e o modo usuário pode funcionar como uma forma rudimentar de sistema de proteção (segurança)?
- 1.8. Quais das seguintes instruções deverão ser privilegiadas?
 - a. Definir o valor do temporizador.
 - b. Ler o valor do relógio.
 - c. Apagar a memória.
 - d. Emitir uma instrução de trap.
 - e. Desativar interrupções.
 - f. Modificar entradas na tabela de status de dispositivo.
 - g. Passar do modo usuário para o modo kernel.
 - h. Acessar dispositivo de E/S.
- 1.9. Alguns computadores antigos protegiam o sistema operacional colocando-o em uma partição da memória que não pudesse ser modificada ou pelo job do usuário ou pelo próprio sistema operacional. Descreva duas dificuldades que poderiam surgir com esse tipo de esquema.
- 1.10. Algumas CPUs oferecem mais de dois modos de operação. Indique dois usos possíveis para esses diversos modos.
- 1.11. Temporizadores poderiam ser usados para calcular a hora atual. Informe resumidamente como isso poderia ser realizado.
- 1.12. A Internet é uma LAN ou uma WAN?

Exercícios

- 1.13. Em um ambiente de multiprogramação e tempo compartilhado, diversos usuários compartilham o sistema simultaneamente. Essa situação pode resultar em vários problemas de segurança.
 - a. Quais são dois desses problemas?
 - b. Podemos garantir o mesmo grau de segurança em uma máquina de tempo compartilhado e em uma máquina dedicada? Explique sua resposta.
- 1.14. O problema da utilização de recursos aparece de diferentes formas, em diferentes tipos de sistemas operacionais. Liste quais recursos precisam ser gerenciados cuidadosamente nos seguintes ambientes:
 - a. Sistemas de mainframe ou minicomputador
 - b. Estações de trabalho conectadas a servidores
 - c. Computadores portáteis
- 1.15. Sob quais circunstâncias seria melhor para um usuário utilizar um sistema de tempo compartilhado em vez de um PC ou estação de trabalho monousuário?
- 1.16. Quais das funcionalidades listadas a seguir precisam ter suporte do sistema operacional para os seguintes: (a) dispositivos portáteis e (b) sistemas de tempo real?
 - a. Programação em batch
 - b. Memória virtual
 - c. Tempo compartilhado
- 1.17. Descreva as diferenças entre o multiprocessamento simétrico e assimétrico. Cite três vantagens e uma desvantagem dos sistemas multiprocessados.
- 1.18. Como os sistemas em clusters diferem dos sistemas multiprocessados? O que é exigido para que duas máquinas pertencentes a um cluster cooperem para fornecer um serviço altamente disponível?
- 1.19. Faça a distinção entre os modelos cliente-servidor e peer-to-peer dos sistemas distribuídos.
- 1.20. Considere um cluster de computadores consistindo em dois nós executando um banco de dados. Descreva duas maneiras como o software do cluster pode gerenciar o acesso aos dados no disco. Discuta os benefícios e as desvantagens de cada um.
- 1.21. Como os computadores em rede diferem dos computadores pessoais tradicionais? Descreva alguns cenários de uso em que é vantajoso utilizar computadores em rede.
- 1.22. Qual é a finalidade das interrupções? Quais são as diferenças entre um trap e uma interrupção? Os traps podem ser gerados intencionalmente por um programa do usuário? Nesse caso, para que finalidade?
- 1.23. O acesso direto à memória é usado para dispositivos de E/S de alta velocidade a fim de evitar o aumento da carga de execução da CPU.
 - a. Como a CPU realiza a interface com o dispositivo para coordenar a transferência?
 - b. Como a CPU sabe quando as operações com a memória foram concluídas?
 - c. A CPU tem permissão para executar outros programas enquanto o controlador de DMA está transferindo dados. Esse processo interfere na execução dos programas do usuário? Nesse caso, descreva que formas de interferência são causadas.
- 1.24. Alguns sistemas computadorizados não fornecem um modo de operação privilegiado no hardware. É possível construir um sistema operacional seguro para esses sistemas computadorizados? Dê argumentos para isso ser possível e impossível.
- 1.25. Dê duas razões para a utilidade dos caches. Que problemas eles resolvem? Que problemas eles causam? Se um cache puder se tornar tão grande quanto o dispositivo para o qual ele está guardando dados (por exemplo, um cache tão grande quanto um disco), por que não usar esse tamanho e eliminar o dispositivo?
- 1.26. Considere um sistema SMP semelhante ao que mostramos na [Figura 1.6](#). Ilustre, com um exemplo, como os dados que residem na memória poderiam realmente ter dois valores diferentes em cada um dos caches locais.
- 1.27. Discuta, com exemplos, como o problema de manter a coerência dos dados em cache se manifesta nos seguintes ambientes de processamento:
 - a. Sistemas de processador único
 - b. Sistemas multiprocessados
 - c. Sistemas distribuídos
- 1.28. Descreva um mecanismo para impor a proteção de memória, a fim de impedir que um programa modifique a memória associada a outros programas.
- 1.29. Que configuração de rede seria mais adequada aos ambientes a seguir?
 - a. Um andar de dormitório
 - b. Um campus universitário
 - c. Um estado
 - d. Uma nação
- 1.30. Defina as propriedades essenciais dos seguintes tipos de sistemas operacionais:

- a. Batch
- b. Interativo
- c. Tempo compartilhado
- d. Tempo real
- e. Rede
- f. Paralelo
- g. Distribuído
- h. Em clusters
- i. Portátil

1.31. Quais são as opções inerentes aos computadores portáteis?

1.32. Identifique diversas vantagens e desvantagens dos sistemas operacionais de fonte aberto.

Inclua os tipos de pessoas que considerariam cada aspecto uma vantagem ou uma desvantagem.

Notas bibliográficas

[Brookshear \[2003\]](#) fornece uma visão geral da ciência da computação em geral.

Uma visão geral do sistema operacional Linux é apresentada em [Bovet e Cesati \[2002\]](#). [Solomon e Russinovich \[2000\]](#) fornecem uma visão geral do Microsoft Windows e detalhes técnicos consideráveis sobre os detalhes internos e componentes do sistema. [Russinovich e Solomon \[2005\]](#) atualizam essa informação para o Windows Server 2003 e Windows XP. [McDougall e Mauro \[2007\]](#) abordam os detalhes internos do sistema operacional Solaris. O Mac OS X é apresentado em <http://www.apple.com/macosx>. Os detalhes internos do Mac OS X são discutidos em [Singh \[2007\]](#).

O tratamento de sistemas peer-to-peer inclui [Parameswaran e outros \[2001\]](#), [Gong \[2002\]](#), [Ripeanu e outros \[2002\]](#), [Balakrishnan e outros \[2003\]](#) e [Loo \[2003\]](#). Uma discussão sobre sistemas de compartilhamento de arquivos peer-to-peer poderá ser encontrada em [Lee \[2003\]](#). Uma boa abordagem sobre computação em clusters é apresentada por [Buyya \[1999\]](#). Avanços recentes sobre computação em clusters são descritos por [Ahmed \[2000\]](#). Um estudo sobre questões relacionadas com suporte dos sistemas operacionais para sistemas distribuídos pode ser encontrado em [Tanenbaum e Van Renesse \[1985\]](#).

Muitos livros gerais abordam sistemas operacionais, incluindo [Stallings \[2000b\]](#), [Nutt \[2004\]](#) e [Tanenbaum \[2001\]](#).

[Hamacher e outros \[2002\]](#) descrevem a organização do computador, e [McDougall e Laudon \[2006\]](#) discutem sobre processadores com múltiplos núcleos. [Hennessy e Patterson \[2007\]](#) fornecem um tratamento sobre sistemas de E/S e barramentos, além de arquitetura de sistemas em geral.

Memórias cache, incluindo memória associativa, são descritas e analisadas por [Smith \[1982\]](#). Esse artigo também inclui uma extensa bibliografia sobre o assunto.

Discussões referentes à tecnologia de disco magnético são apresentadas por [Freedman \[1983\]](#) e por [Harker e outros \[1981\]](#). Os discos ópticos são explicados por [Kenville \[1982\]](#), [Fujitani \[1984\]](#), [O'Leary e Kitts \[1985\]](#), [Gait \[1988\]](#) e [Olsen e Kenley \[1989\]](#). As discussões sobre disquetes são oferecidas por [Pechura e Schoeffler \[1983\]](#) e por [Sarisky \[1983\]](#). Discussões gerais referentes à tecnologia de armazenamento em massa são oferecidas por [Chi \[1982\]](#) e por [Hoagland \[1985\]](#).

[Kurose e Ross \[2005\]](#) e [Tanenbaum \[2003\]](#) fornecem material de visão geral sobre redes de computadores. [Fortier \[1989\]](#) apresenta uma discussão detalhada sobre hardware e software de rede. [Kozierok \[2005\]](#) discute sobre TCP em detalhes. [Mullender \[1993\]](#) oferece uma visão geral dos sistemas distribuídos. [Wolf \[2003\]](#) discute os desenvolvimentos recentes em desenvolvimento de sistemas embutidos. As questões relacionadas com dispositivos portáteis podem ser encontradas em [Myers e Beigl \[2003\]](#) e [Di Pietro e Mancini \[2003\]](#).

Uma discussão completa da história do fonte aberto e seus benefícios e desafios pode ser encontrada em [Raymond \[1999\]](#). A história do hacking é discutida em [Levy \[1994\]](#). A Free Software Foundation publicou sua filosofia em seu site: <http://www.gnu.org/philosophy/free-software-for-freedom.html>. As instruções detalhadas sobre como fazer o build do kernel Linux Ubuntu estão em http://www.howtoforge.com/kernel_compilation_ubuntu. Os componentes de código de fonte aberto do Mac OS X estão disponíveis em <http://developer.apple.com/open-source/index.html>.

Wikipedia (http://en.wikipedia.org/wiki/Richard_Stallman) possui informações a respeito de Richard Stallman.

O código-fonte do Multics está disponível em http://web.mit.edu/multicshistory/source/Multics_Internet_Server/Multics_sources.html.

CAPÍTULO 2

Estruturas do sistema operacional

Um sistema operacional fornece o ambiente dentro do qual os programas são executados. Internamente, os sistemas operacionais variam muito em sua composição, pois são organizados ao longo de muitas linhas diferentes. O projeto de um novo sistema operacional é uma grande tarefa. Antes de iniciar o projeto, os objetivos do sistema devem ser muito bem definidos. Esses objetivos formam a base das opções entre diversos algoritmos e estratégias.

Podemos ver um sistema operacional sob diversos pontos de vista. Um deles focaliza os serviços fornecidos pelo sistema; outro, a interface colocada à disposição de usuários e programadores; e um terceiro, baseado em seus componentes e suas interconexões. Neste capítulo, exploramos todos esses três aspectos, mostrando os pontos de vista de usuários, programadores e projetistas de sistema operacional. Consideraremos quais serviços são fornecidos por um sistema operacional, como são fornecidos, como são depurados e quais são as diversas metodologias para o projeto de tais sistemas. Finalmente, descrevemos como os sistemas operacionais são criados e como um computador dá boot no seu sistema operacional.

OBJETIVOS DO CAPÍTULO

- Descrever os serviços que um sistema operacional provê aos usuários, processos e outros sistemas.
- Discutir as diversas formas de estruturar um sistema operacional.
- Explicar como os sistemas operacionais são instalados e personalizados e como eles dão boot.

2.1 Serviços do sistema operacional

Um sistema operacional provê um ambiente para a execução de programas. Ele fornece determinados serviços aos programas e aos usuários desses programas. Esses serviços do sistema operacional são fornecidos para a conveniência do programador, para facilitar a tarefa de programação. A [Figura 2.1](#) mostra uma visão dos diversos serviços do sistema operacional e como eles estão relacionados.

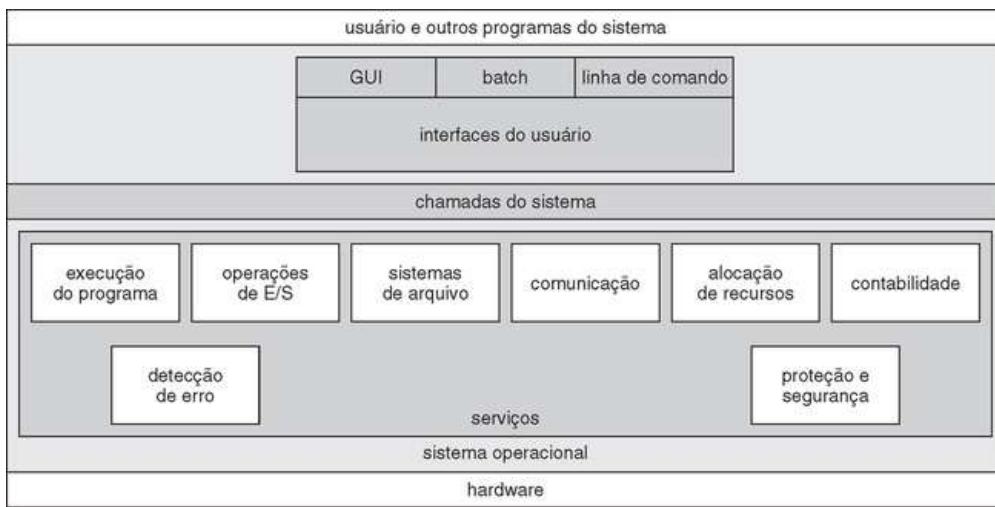


FIGURA 2.1 Uma visão dos serviços do sistema operacional.

Um conjunto de serviços do sistema operacional fornece funções úteis ao usuário.

- **Interface do usuário.** Quase todos os sistemas operacionais possuem uma **interface do usuário (User Interface - UI)**. Essa interface pode assumir várias formas. Uma é a **interface de linha de comando**, que utiliza comandos em modo texto e um método para sua entrada (digamos, um programa para permitir a entrada e a edição de comandos). Outra é uma **interface batch**, em que os comandos e diretivas para controlar esses comandos são introduzidos em arquivos, e esses arquivos são executados. Normalmente, é utilizada uma **interface gráfica com o usuário (Graphical User Interface - GUI)**. Aqui, a interface é um sistema de janela com um dispositivo de apontamento para direcionar a E/S, escolher entre os menus e fazer seleções, e um teclado para a entrada de texto. Alguns sistemas fornecem duas ou essas três variações.
- **Execução de programa.** O sistema precisa ser capaz de **carregar um programa para a memória e executar esse programa**. O programa precisa ser capaz de encerrar sua execução, normalmente ou não (indicando erro).
- **Operações de E/S.** Um programa em execução pode **exigir E/S, o que pode envolver um arquivo ou um dispositivo de E/S**. Para dispositivos específicos, funções especiais podem ser desejadas (como gravar em uma unidade de CD ou DVD ou apagar uma tela de vídeo). Por questão de eficiência e proteção, os usuários normalmente não podem controlar os dispositivos de E/S de forma direta. Portanto, o sistema operacional precisa prover meios para realizar a E/S.
- **Manipulação do sistema de arquivos.** O sistema de arquivos é de interesse particular. É claro que os programas precisam ler e gravar arquivos e diretórios. Eles também precisam criá-los e removê-los por nome, procurar determinado arquivo e listar informações dele. Finalmente, alguns programas incluem gerência de permissões para permitir ou negar o acesso a arquivos ou diretórios com base na propriedade do arquivo. Muitos sistemas operacionais oferecem diversos sistemas de arquivos, às vezes para permitir a escolha pessoal, e outras vezes para oferecer recursos específicos ou características de performance.
- **Comunicações.** Existem muitas circunstâncias em que um processo precisa trocar informações com outro processo. Tal comunicação pode ocorrer entre processos que estão executando no mesmo computador ou entre processos que estão executando em diferentes computadores ligados por uma rede de computadores. As comunicações podem ser implementadas por meio da *memória compartilhada* ou pela *troca de mensagens*, em que os pacotes de informações são movidos entre os processos pelo sistema operacional.
- **Detecção de erro.** O sistema operacional precisa estar ciente de possíveis erros. Os erros podem ocorrer na CPU e no hardware da memória (como um erro de memória ou uma falta de alimentação), nos dispositivos de E/S (como um erro de paridade na fita, uma falha de conexão em uma rede ou a falta de papel na impressora) e no programa do usuário (como um estouro aritmético, uma tentativa de acessar um local de memória ilegal ou o uso de um tempo muito

grande da CPU). Para cada tipo de erro, o sistema operacional deve tomar a medida apropriada para garantir a computação correta e coerente. É claro que existe variação no modo como os sistemas operacionais reagem a esses erros e os corrigem. Os recursos de depuração podem melhorar bastante a capacidade do programador de usar o sistema de modo eficiente.

Outro conjunto de funções do sistema operacional existe não para ajudar o usuário, mas para garantir a operação eficiente do próprio sistema. Os sistemas com muitos usuários podem conseguir eficiência compartilhando os recursos do computador entre os usuários.

■ **Alocação de recursos.** Quando existem muitos usuários ou várias tarefas executando ao mesmo tempo, precisam ser alocados recursos a cada um deles. Muitos tipos diferentes de recursos são controlados pelo sistema operacional. Alguns (como ciclos de CPU, memória principal e armazenamento de arquivos) podem ter código de alocação especial, enquanto outros (como dispositivos de E/S) podem ter código de requisição e liberação muito mais genérico. Por exemplo, ao determinar o melhor uso da CPU, os sistemas operacionais possuem rotinas de escalonamento de CPU que levam em consideração a velocidade da CPU, as tarefas que precisam ser executadas, o número de registradores disponíveis e outros fatores. Pode haver rotinas para alocar impressoras, modems e outros dispositivos periféricos.

■ **Contabilidade.** Queremos registrar quais usuários utilizam quanto e que tipos de recursos do computador. Podemos usar esse registro armazenando para contabilidade (para que os usuários possam ser cobrados) ou para acumular estatísticas de uso. As estatísticas de uso podem ser uma ferramenta valiosa para pesquisadores que querem reconfigurar o sistema, a fim de aprimorar os serviços de computação.

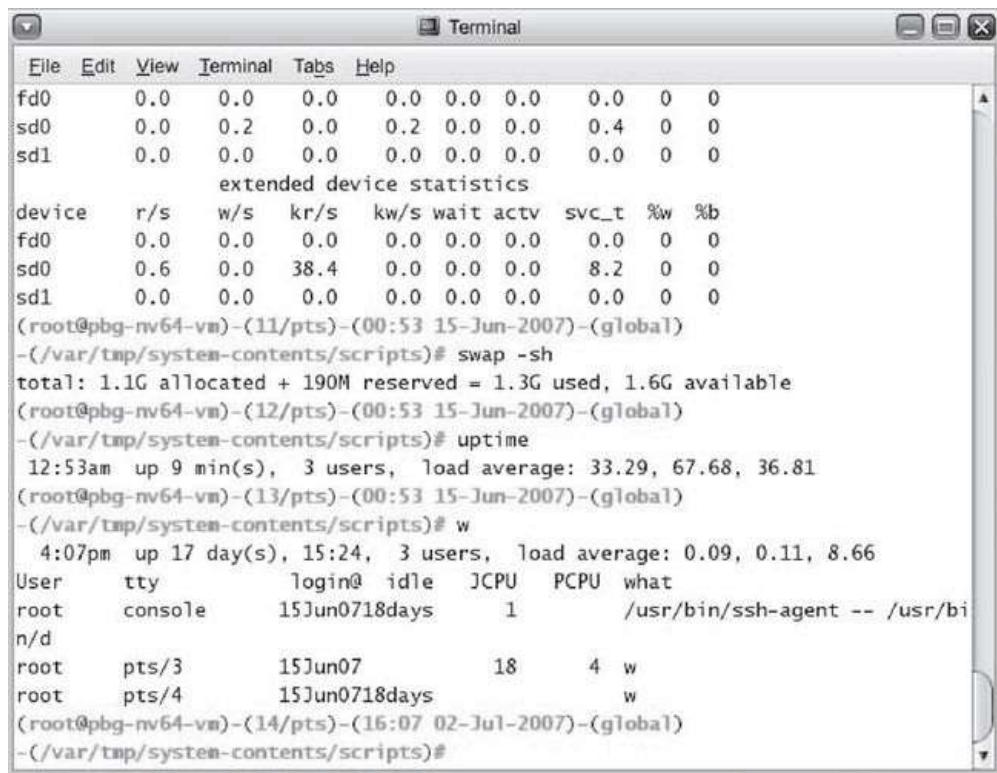
■ **Proteção e segurança.** Os proprietários das informações armazenadas em um sistema computadorizado multiusuário ou em rede podem querer controlar o uso dessas informações. Quando diversos processos separados são executados simultaneamente, não deverá ser possível para um processo interferir com os outros ou com o próprio sistema operacional. A proteção envolve a garantia de controle de todo acesso aos recursos do sistema. A segurança do sistema contra pessoas externas também é importante. Essa segurança começa exigindo que cada usuário se autentique, normalmente por meio de uma senha, para obter acesso aos recursos do sistema. Ela se estende para defender dispositivos de E/S externos, incluindo modems e adaptadores de rede, de tentativas de acesso inválidas até o registro de todas as conexões, para a detecção de invasões. Para um sistema ser protegido e seguro, é preciso instituir precauções por todo o sistema. Uma cadeia é tão forte quanto seu elo mais fraco.

2.2 Interface usuário-sistema operacional

Já mencionamos que existem várias maneiras de os usuários realizarem a interface com o sistema operacional. Aqui, discutimos duas técnicas fundamentais. Uma fornece uma interface de linha de comandos, ou **interpretador de comandos**, que permite que os usuários entrem diretamente com comandos a serem executados pelo sistema operacional. A outra permite que o usuário realize a interface com o sistema operacional por meio de uma interface gráfica com o usuário, ou GUI.

2.2.1 Interpretador de comandos

Alguns sistemas operacionais incluem o interpretador de comandos no kernel. Outros, como o Windows XP e o UNIX, tratam o interpretador de comandos como um programa especial que está sendo executado quando uma tarefa é iniciada ou quando um usuário efetua seu primeiro logon (em sistemas interativos). Em sistemas com múltiplos interpretadores de comandos para escolher, os interpretadores são conhecidos como **shells**. Por exemplo, nos sistemas UNIX e Linux, um usuário pode escolher dentre diversos shells diferentes, incluindo *Bourne shell*, *C shell*, *Bourne-Again shell*, *Korn shell* e assim por diante. Também existem shells de terceiros e shells gratuitos escritos pelo usuário. A maioria dos shells fornece funcionalidade semelhante, e a escolha de um usuário sobre o shell a utilizar geralmente é baseada na preferência pessoal. A [Figura 2.2](#) mostra o interpretador de comandos Bourne shell sendo usado no Solaris 10.



The screenshot shows a terminal window titled "Terminal". The window has a menu bar with "File", "Edit", "View", "Terminal", "Tabs", and "Help". The main area displays command-line output. The user runs several commands: "swap -sh" to start the Bourne shell, "total: 1.1G allocated + 190M reserved = 1.3G used, 1.6G available" to show disk usage, "uptime" to show system load average (33.29, 67.68, 36.81), and "w" to show user activity (root at console since Jun 07 18:00). The terminal window has scroll bars on the right side.

```
fd0      0.0    0.0    0.0    0.0  0.0  0.0    0.0  0  0
sd0      0.0    0.2    0.0    0.2  0.0  0.0    0.4  0  0
sd1      0.0    0.0    0.0    0.0  0.0  0.0    0.0  0  0
              extended device statistics
device   r/s    w/s    kr/s   kw/s wait  activ  svc_t %w  %b
fd0      0.0    0.0    0.0    0.0  0.0  0.0    0.0  0  0
sd0      0.6    0.0   38.4    0.0  0.0  0.0    8.2  0  0
sd1      0.0    0.0    0.0    0.0  0.0  0.0    0.0  0  0
(root@pbg-nv64-vm)-(11/pts)-(00:53 15-Jun-2007)-(global)
-./var/tmp/system-contents/scripts# swap -sh
total: 1.1G allocated + 190M reserved = 1.3G used, 1.6G available
(root@pbg-nv64-vm)-(12/pts)-(00:53 15-Jun-2007)-(global)
-./var/tmp/system-contents/scripts# uptime
12:53am up 9 min(s),  3 users,  load average: 33.29, 67.68, 36.81
(root@pbg-nv64-vm)-(13/pts)-(00:53 15-Jun-2007)-(global)
-./var/tmp/system-contents/scripts# w
 4:07pm up 17 day(s), 15:24,  3 users,  load average: 0.09, 0.11, 8.66
User     tty      login@  idle   JCPU   PCPU what
root    console      15Jun0718days    1      /usr/bin/ssh-agent -- /usr/bi
n/d
root    pts/3      15Jun07          18      4  w
root    pts/4      15Jun0718days          w
(root@pbg-nv64-vm)-(14/pts)-(16:07 02-Jul-2007)-(global)
-./var/tmp/system-contents/scripts#
```

FIGURA 2.2 O interpretador de comandos Bourne shell no Solaris 10.

A função principal do interpretador de comandos é apanhar e executar o próximo comando especificado pelo usuário. Muitos dos comandos emitidos nesse nível manipulam arquivos: criar, excluir, listar, imprimir, copiar, executar etc. Os shells do MS-DOS e do UNIX operam dessa maneira. Esses comandos podem ser implementados de duas maneiras.

Em uma técnica, o próprio interpretador de comandos contém o código para executar o comando. Por exemplo, um comando para excluir um arquivo pode fazer o interpretador de comandos saltar para uma seção do seu código que define os parâmetros e faz a chamada de sistema apropriada. Nesse caso, o número de comandos que podem ser dados determina o tamanho do interpretador de comandos, pois cada comando requer seu próprio código de implementação.

Uma técnica alternativa – usada pelo UNIX, entre outros sistemas operacionais – implementa a maioria dos comandos por meio de programas do sistema. Nesse caso, o interpretador de comandos não entende o comando de forma alguma; ele usa o comando para identificar um arquivo a ser carregado para a memória e executado. Assim, o comando do UNIX para excluir um arquivo

```
rm arquivo.txt
```

```
rm arquivo.txt
```

procuraria um arquivo chamado `rm`, carregaria o arquivo na memória e o executaria com o parâmetro `arquivo.txt`. A função associada ao comando `rm` seria definida completamente pelo código no arquivo `rm`. Desse modo, os programadores podem facilmente incluir novos comandos ao sistema criando novos arquivos com nomes apropriados. O programa interpretador de comandos, que pode ser pequeno, não precisa ser alterado para que novos comandos sejam acrescentados.

2.2.2 Interface gráfica com o usuário

Uma segunda estratégia para a interface com o sistema operacional é por meio de uma interface gráfica com o usuário, ou GUI. Aqui, em vez de inserir comandos diretamente por uma interface da linha de comandos, os usuários empregam um sistema de janela e menu baseado em mouse, caracterizado por uma metáfora de **desktop**. O usuário move o mouse para posicionar seu ponteiro nas imagens, ou **ícones**, na tela (a área de trabalho), que representam programas, arquivos, diretórios e funções do sistema. Dependendo do local do ponteiro do mouse, clicar em um botão no mouse pode ativar um programa, selecionar um arquivo ou diretório - conhecido como **pasta** - ou exibir um menu que contém comandos.

As interfaces gráficas com o usuário apareceram inicialmente, em parte, devido à pesquisa feita no início da década de 1970 na instalação de pesquisa PARC da Xerox. A primeira GUI apareceu no computador Alto da Xerox em 1973. As interfaces gráficas se tornaram mais divulgadas com o advento dos computadores Apple Macintosh, na década de 1980. A interface com o usuário no sistema operacional do Macintosh (Mac OS) passou por várias mudanças no decorrer dos anos, sendo que a mais significativa foi a adoção da interface *Aqua*, que apareceu com o Mac OS X. A primeira versão do Windows da Microsoft - versão 1.0 - era baseada em uma interface GUI para o sistema operacional MS-DOS. Versões mais recentes do Windows fizeram pequenas mudanças na aparência da GUI e várias melhorias em sua funcionalidade, incluindo o Windows Explorer.

Tradicionalmente, os sistemas UNIX usavam interfaces da linha de comandos. Porém, diversas interfaces GUI estão disponíveis, incluindo os sistemas Common Desktop Environment (CDE) e X-Windows, que são comuns em versões comerciais do UNIX, como Solaris e o sistema AIX da IBM. Além disso, desenvolvimentos significativos no projeto de GUI foram feitos por vários projetos **open-source**, como *K Desktop Environment* (ou *KDE*) e o desktop *GNOME*, pelo projeto GNU. Os desktops KDE e GNOME executam no Linux e em vários sistemas UNIX e estão disponíveis sob licenças open-source, o que significa que seu código-fonte está prontamente disponível para leitura e para modificação sob termos de licença específicos.

A escolha entre usar uma interface de linha de comandos ou GUI é principalmente uma questão de gosto pessoal. Em geral, muitos usuários UNIX preferem as interfaces da linha de comandos, pois elas fornecem interfaces de shell poderosas. Por outro lado, a maioria dos usuários do Windows prefere usar o ambiente GUI do Windows e quase nunca utilizam a interface de shell do MS-DOS. As diversas mudanças sofridas pelos sistemas operacionais do Macintosh fornecem um bom estudo. Historicamente, o Mac OS não tem oferecido uma interface da linha de comandos, sempre exigindo que seus usuários se comuniquem com o sistema operacional por meio da sua GUI. Porém, com o lançamento do Mac OS X (que, em parte, é implementado usando um kernel UNIX), o sistema operacional agora oferece uma nova interface *Aqua* e uma interface da linha de comandos. A [Figura 2.3](#) é uma captura de tela da GUI do Mac OS X.

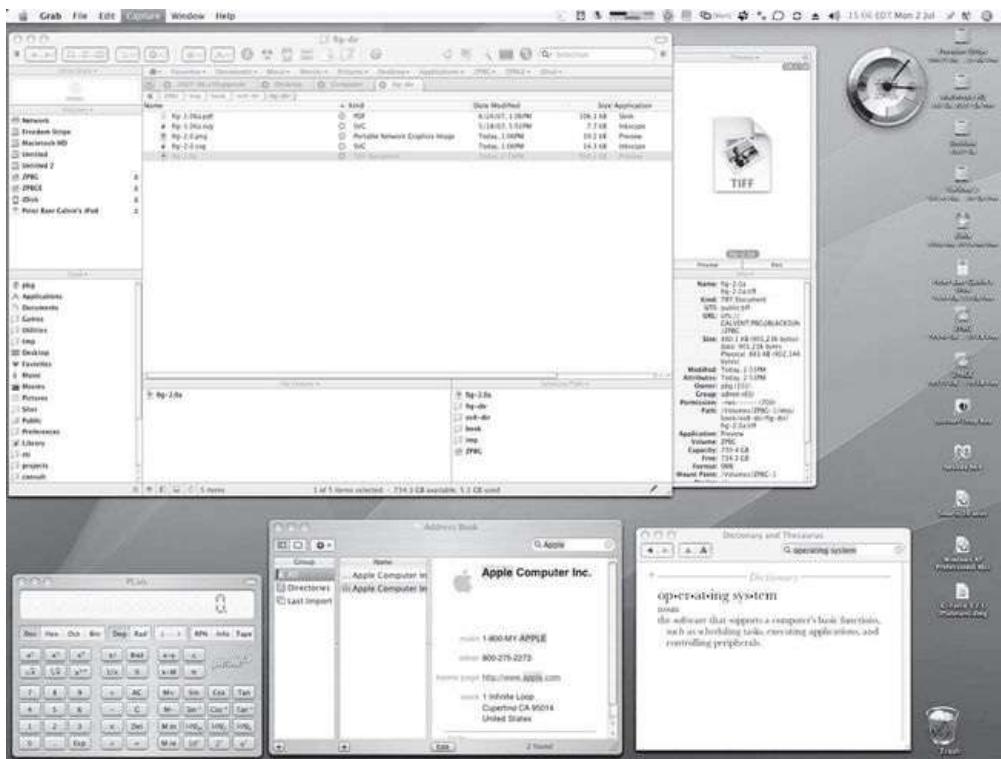


FIGURA 2.3 A GUI do Mac OS X.

A interface com o usuário pode variar de um sistema para outro e até mesmo de um usuário para outro dentro de um sistema. Ela normalmente é removida da estrutura real do sistema. O projeto de uma interface com o usuário útil e amigável, portanto, não é uma função direta do sistema operacional. Neste livro, nos concentraremos nos problemas fundamentais de fornecer serviço adequado aos programas do usuário. Do ponto de vista do sistema operacional, não distinguimos entre programas do usuário e programas do sistema.

2.3 Chamadas de sistema

As **chamadas de sistema (system calls)** proveem uma interface com os serviços disponibilizados por um sistema operacional. Essas chamadas estão disponíveis como rotinas escritas em C e C++, embora certas tarefas de nível mais baixo (por exemplo, tarefas onde o hardware precisa ser acessado diretamente) podem ter que ser escritas por meio de instruções em linguagem assembly.

Antes de discutir como o sistema operacional torna disponível as chamadas de sistema, vamos usar um exemplo de como as chamadas de sistema são usadas: escreva um programa simples para ler dados de um arquivo e copiá-los para outro. A primeira entrada de que o programa precisa são os nomes dos dois arquivos: o arquivo de entrada e o arquivo de saída. Esses nomes podem ser especificados de muitas maneiras, dependendo do projeto do sistema operacional. Uma técnica é fazer o programa pedir ao usuário os nomes dos dois arquivos. Em um sistema interativo, essa técnica exigirá uma sequência de chamadas de sistema, primeiro para escrever uma mensagem de requisição na tela e depois para ler do teclado as características que definem os dois arquivos. Em sistemas baseados em mouse e ícone, um menu de nomes de arquivo é apresentado em uma janela. O usuário pode, então, utilizar o mouse para selecionar o nome da origem, e uma janela pode ser aberta para ser especificado o nome do destino. Essa sequência exigiria muitas chamadas de sistema para E/S.

Quando os dois nomes de arquivo forem obtidos, o programa terá de abrir o arquivo de entrada e criar o arquivo de saída. Cada uma dessas operações exige outra chamada de sistema. Também existem possíveis condições de erro para cada operação. Quando o programa tenta abrir o arquivo de entrada, ele pode descobrir que não existe um arquivo com esse nome ou que o arquivo está protegido contra acesso. Nesses casos, o programa deverá enviar uma mensagem ao console (outra sequência de chamadas de sistema) e depois encerrar de forma anormal (outra chamada de sistema). Se o arquivo de entrada existir, então temos de criar um novo arquivo de saída. Podemos descobrir que já existe um arquivo de saída com o mesmo nome. Essa situação pode levar ao cancelamento do programa (uma chamada de sistema) ou à exclusão do arquivo existente (outra chamada de sistema) e à criação de um novo (outra chamada de sistema). Outra opção, em um sistema interativo, é perguntar ao usuário (por meio de uma sequência de chamadas de sistema, para enviar a mensagem e ler a resposta do terminal) se deseja substituir o arquivo existente ou cancelar o programa.

Agora que os dois arquivos estão prontos, entramos em um loop que lê do arquivo de entrada (uma chamada de sistema) e escreve no arquivo de saída (outra chamada de sistema). Cada leitura e escrita precisam retornar informações de status, dependendo de várias condições de erro possíveis. Na entrada, o programa pode descobrir que o final do arquivo foi atingido ou que houve uma falha de hardware na leitura (como um erro de paridade). A operação de escrita pode encontrar diversos erros, dependendo do dispositivo de saída (falta de espaço no disco, impressora sem papel e assim por diante).

Finalmente, depois de copiar o arquivo inteiro, o programa pode fechar os dois arquivos (outra chamada de sistema), escrever uma mensagem no console ou janela (mais chamadas de sistema) e finalmente terminar de forma normal (a última chamada de sistema). Essa sequência de chamada de sistema é mostrada na [Figura 2.4](#). Como podemos ver, até mesmo os programas simples podem utilizar muito o sistema operacional. Frequentemente, os sistemas executam milhares de chamadas de sistema por segundo.

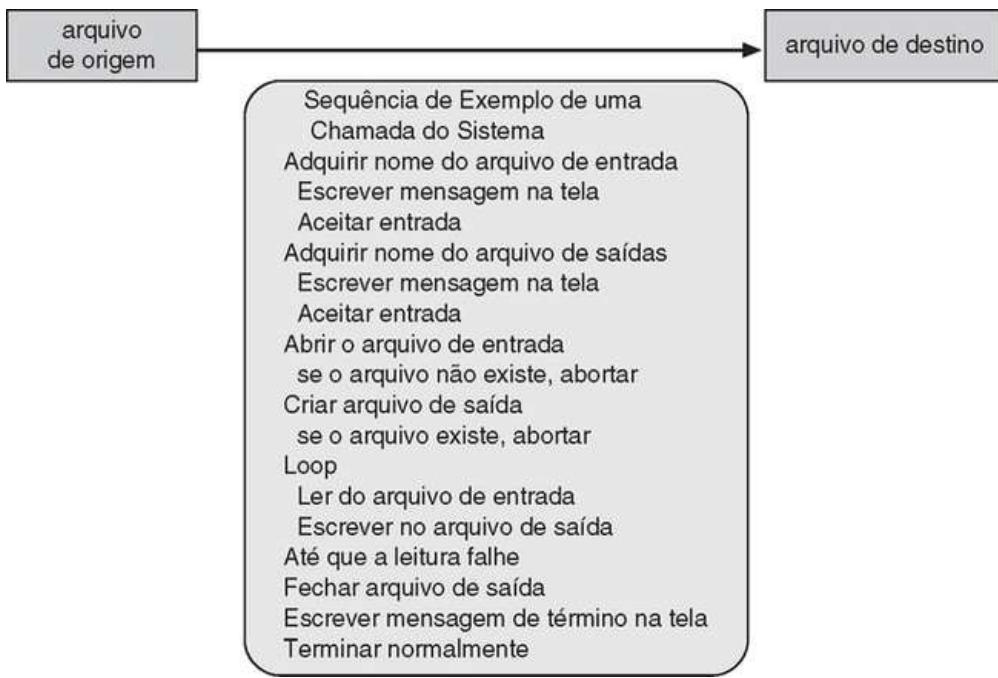


FIGURA 2.4 Exemplo de como são usadas as chamadas de sistema.

Todavia, a maioria dos programadores nunca vê esse nível de detalhe. Normalmente, os desenvolvedores de aplicação projetam programas de acordo com uma **interface de programação de aplicação (Application Programming Interface - API)**. A API especifica um conjunto de funções que estão disponíveis a um programador de aplicações, incluindo os parâmetros que são passados a cada função e os valores de retorno que o programador pode esperar. Três das APIs mais comuns disponibilizadas aos programadores de aplicação são a API Win32 para os sistemas Windows, a API POSIX para sistemas baseados no POSIX (que incluem praticamente todas as versões do UNIX, Linux e Mac OS X) e a API Java para projeto de programas que executam na máquina virtual Java. Observe que – a menos que especificado de outra forma – os nomes de chamada de sistema usados neste livro são exemplos genéricos. Cada sistema operacional tem seu próprio nome para cada chamada do sistema.

Por detrás, as funções que compõem uma API normalmente envolvem as chamadas de sistema reais em favor do programador de aplicação. Por exemplo, a função `CreateProcess()` da API Win32 (que naturalmente é usada para criar um novo processo) normalmente ativa a chamada do sistema `NTCreateProcess()` no kernel do Windows. Por que um programador de aplicação preferiria programar de acordo com uma API em vez de invocar as chamadas de sistema reais? Existem vários motivos para fazer isso. Um benefício de se programar de acordo com uma API refere-se à portabilidade do programa. Um programador de aplicação projetando um programa com uma API pode esperar que seu programa seja compilado e executado em qualquer sistema que admita a mesma API (embora, na realidade, diferenças arquitetônicas normalmente tornem isso mais difícil do que pode parecer). Além do mais, chamadas de sistema reais normalmente podem ser mais detalhadas e difíceis de se trabalhar do que a API disponível a um programador de aplicação. Independentemente disso, normalmente existe uma correlação forte entre invocar uma função na API e sua chamada de sistema associada dentro do kernel. Na verdade, muitas das APIs POSIX e Win32 são semelhantes às chamadas de sistema nativas fornecidas pelos sistemas operacionais UNIX, Linux e Windows.

O sistema de suporte em tempo de execução (um conjunto de funções embutidas em bibliotecas incluídas com um compilador) para a maioria das linguagens de programação fornece uma **interface de chamada de sistema** que serve como ligação com as chamadas do sistema disponibilizadas pelo sistema operacional. A interface da chamada de sistema intercepta as chamadas de função na API e invoca a chamada de sistema necessária dentro do sistema operacional. Normalmente, um número é associado a cada chamada de sistema, e a interface de chamada de sistema mantém uma tabela indexada de acordo com esses números. A interface da chamada de sistema, então, invoca a chamada de sistema desejada no kernel do sistema operacional e retorna o status da chamada de sistema e quaisquer valores de retorno.

Quem chama não precisa saber nada a respeito de como a chamada de sistema é implementada ou o que ela faz durante a execução. Em vez disso, só precisa obedecer à API e entender o que o sistema operacional fará como resultado da execução dessa chamada de sistema. Assim, a maior parte dos detalhes da interface do sistema operacional é escondida do programador pela API e controlada pela biblioteca de suporte em tempo de execução. O relacionamento entre uma API, a interface da chamada de sistema e o sistema operacional é mostrado na [Figura 2.6](#), que ilustra como o sistema operacional trata de uma aplicação do usuário invocando a chamada de sistema `open()`.

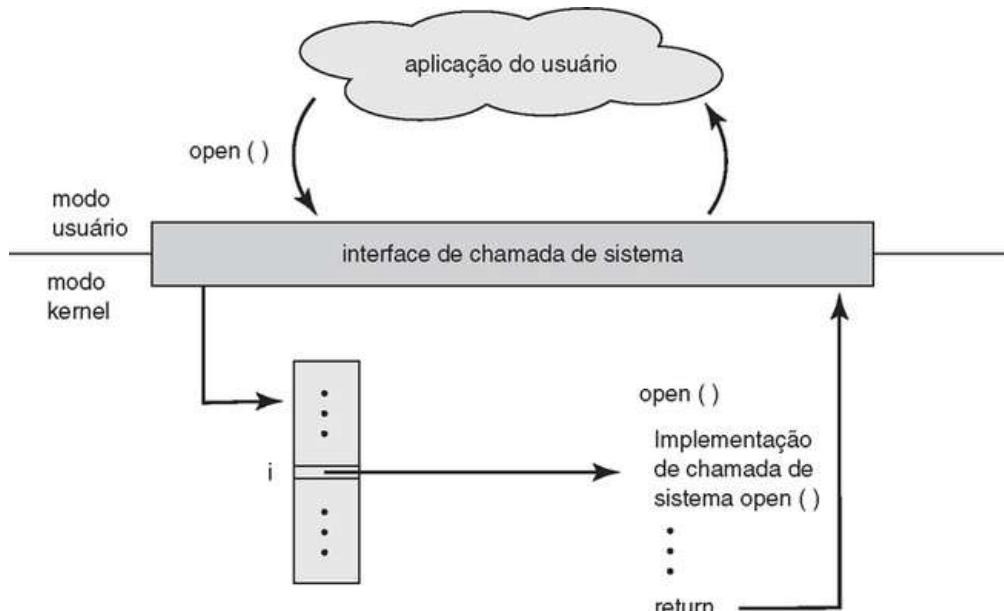
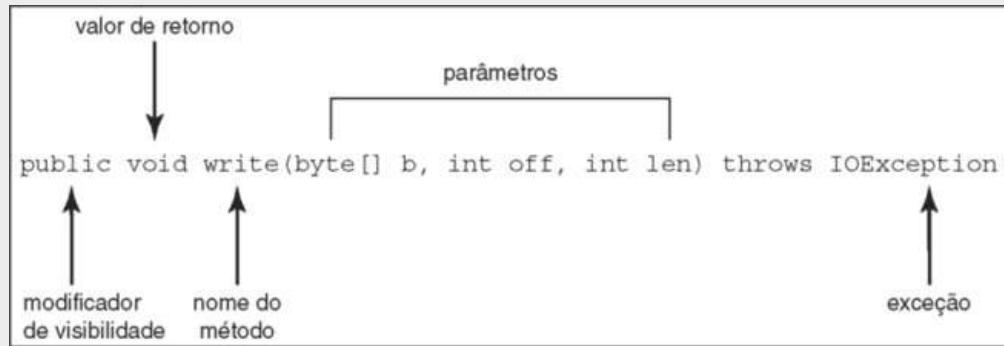


FIGURA 2.6 O tratamento de uma aplicação do usuário invocando a chamada de sistema `open()`.

EXEMPLO DE API PADRÃO

Como um exemplo de uma API padrão, considere o método `write()` na classe `java.io.OutputStream` da API Java, que permite a escrita em um arquivo ou conexão de rede. A API para esse método aparece na Figura 2.5.



Esse método retorna um `void` - ou nenhum valor de retorno. Uma `IOException` é gerada se ocorrer um erro de E/S. Os parâmetros passados a `write()` podem ser escritos da seguinte forma:

- `byte[] b` - os dados a serem escritos.
- `int off` - o offset inicial no array `b` onde os dados serão escritos.
- `int len` - o número de bytes a serem escritos.

As chamadas de sistema ocorrem de diferentes maneiras, dependendo do computador em uso. Normalmente, mais informações são necessárias do que apenas a identidade da chamada de sistema desejada. O tipo exato e a quantidade de informações variam de acordo com o sistema operacional e a chamada em particular. Por exemplo, para obter entrada, podemos ter de especificar o arquivo ou dispositivo a ser usado como origem, bem como o endereço e a extensão do buffer de memória em que a entrada deverá ser lida. O dispositivo ou arquivo e a extensão podem estar implícitos na chamada.

Três métodos gerais são usados para passar parâmetros ao sistema operacional. O método mais simples é passar os parâmetros nos *registradores*. Em alguns casos, porém, pode haver mais parâmetros do que registradores. Nesses casos, os parâmetros são armazenados em um *bloco* (ou tabela) na memória, e o endereço do bloco é passado como parâmetro em um registrador (Figura 2.7). Essa é a técnica utilizada pelo Linux e o Solaris. Os parâmetros também podem ser *colocados* (push) na *pilha* pelo programa e *retirados* (pop) da pilha pelo sistema operacional. Alguns sistemas operacionais preferem os métodos de bloco ou pilha, pois essas técnicas não limitam o número ou a

extensão dos parâmetros passados.

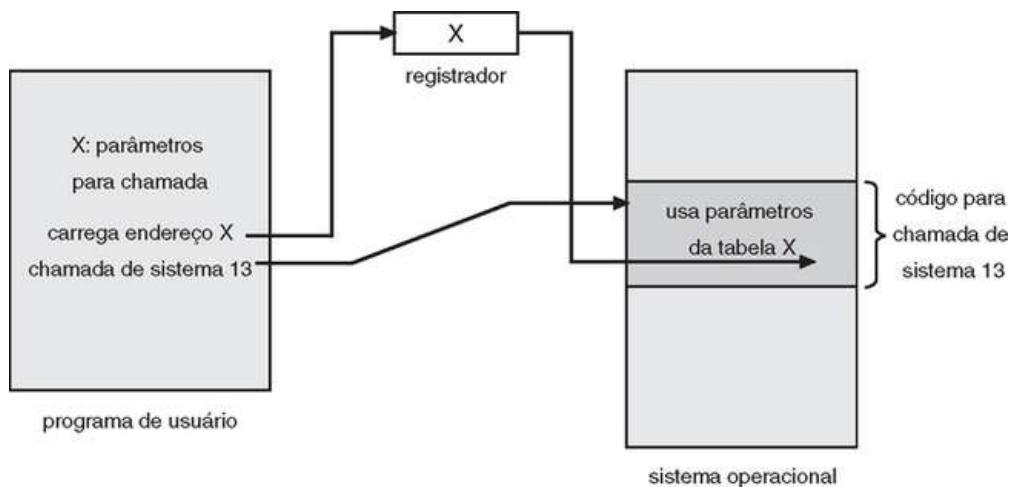


FIGURA 2.7 Passagem de parâmetros como uma tabela.

Como a Java tem por finalidade ser executada em sistemas independentes da plataforma, não é possível fazer chamadas de sistema diretamente de um programa Java. Porém, é possível que um método Java invoque código C ou C++ que seja **nativo** à arquitetura básica em que o programa está sendo executado (por exemplo, Microsoft Windows Vista ou Linux). O código C/C++ pode invocar uma chamada de sistema no sistema hospedeiro, permitindo, assim, que um programa Java faça chamadas de sistema indiretamente. Isso é feito por meio da Java Native Interface (JNI), que permite que um método Java seja declarado como native. Esse método Java native é usado como um marcador de local para a função C/C++ real. Assim, chamar o método Java native na realidade invoca a função nativa escrita em C ou C++. Obviamente, um programa Java que usa métodos native não é considerado portável de um sistema para outro.

2.4 Tipos de chamadas de sistema

As chamadas de sistema podem ser agrupadas de modo geral em seis categorias principais: **controle de processos, manipulação de arquivos, manipulação de dispositivos, manutenção de informações, comunicações e proteção**. Nas [Seções 2.4.1 a 2.4.6](#), discutiremos os tipos de chamadas de sistema oferecidos por um sistema operacional. A maioria delas admite (ou são admitidas por) conceitos e funções discutidos em outros capítulos. A [Figura 2.8](#) resume os tipos de chamadas de sistema fornecidos por um sistema operacional.

- Controle de processos
 - encerrar (end), abortar.
 - carregar, executar.
 - criar processo, terminar processo.
 - obter atributos do processo, definir atributos do processo.
 - esperar por um tempo.
 - esperar evento, sinalizar evento.
 - alocar e liberar memória.
- Gerenciamento de arquivos
 - criar arquivo, excluir arquivo.
 - abrir, fechar.
 - ler, escrever, reposicionar.
 - obter atributos do arquivo, definir atributos do arquivo.
- Gerenciamento de dispositivos
 - solicitar dispositivo, liberar dispositivo.
 - ler, escrever, reposicionar.
 - obter atributos do dispositivo, definir atributos do dispositivo.
 - anexar ou desconectar dispositivos logicamente.
- Manutenção de informações
 - obter hora ou data, definir hora ou data.
 - obter dados do sistema, definir dados do sistema.
 - obter atributos do processo, arquivo ou dispositivo.
 - definir atributos do processo, arquivo ou dispositivo.
- Comunicações
 - criar, excluir conexão de comunicação.
 - enviar, receber mensagens.
 - transferir informações de status.
 - anexar ou desconectar dispositivos remotos.

FIGURA 2.8 Tipos de chamadas de sistema.

2.4.1 Controle de processos

Um programa em execução precisa ser capaz de interromper sua execução normalmente (end) ou de forma anormal (abort). Se for feita uma chamada de sistema para terminar o programa em execução de forma anormal ou se o programa encontrar um problema e causar um trap de erro, às vezes um dump de memória é apanhado e uma mensagem de erro é gerada. O dump é gravado em disco e pode ser examinado por um **depurador** - um programa do sistema designado para auxiliar o programador na localização e correção de bugs - para determinar a causa do problema. Sob certas circunstâncias normais ou anormais, o sistema operacional precisa transferir o controle para o interpretador de comandos que chama. O interpretador de comandos lê, então, o próximo comando. Em um sistema interativo, o interpretador de comandos continua com o próximo comando; considera-se que o usuário emitirá um comando apropriado para responder a qualquer erro. Em um

sistema GUI, uma janela pop-up pode alertar o usuário do erro e pedir orientação. Em um sistema batch, o interpretador de comandos normalmente termina a tarefa inteira e continua com a seguinte. Alguns sistemas permitem que cartões de controle indiquem ações de recuperação especiais caso ocorra um erro. Um **cartão de controle** é um conceito do sistema batch. É um comando para realizar a execução de um processo. Se o programa descobrir um erro em sua entrada e quiser terminar de forma anormal, ele também pode querer definir um nível de erro. Erros mais graves podem ser indicados por um parâmetro de erro de nível mais alto. Então, é possível combinar o término normal e anormal definindo um término normal como um erro no nível 0. O interpretador de comandos ou um programa seguinte poderá usar esse nível de erro para determinar a próxima ação automaticamente.

Um processo ou tarefa executando um programa pode querer carregar (`load`) e executar (`execute`) outro programa. Esse recurso permite que o interpretador de comandos execute um programa conforme instruído, por exemplo, por um comando do usuário, pelo clique de um mouse ou por um comando batch. Uma questão interessante é para onde retornar o controle quando o programa carregado terminar. Essa questão está relacionada com o problema de o programa existente ser perdido, salvo ou ter tido a permissão para continuar a execução concorrentemente com o novo programa.

Se o controle retornar ao programa existente quando o novo programa terminar, temos de salvar a imagem da memória do programa existente; assim, criamos um mecanismo para um programa chamar outro. Se os dois programas continuarem concorrentemente, criamos uma nova tarefa ou processo para ser multiprogramada. Costuma haver uma chamada de sistema específica para essa finalidade (`create process` ou `submit job`).

Se criarmos uma nova tarefa ou processo, ou talvez, ainda, um conjunto de tarefas ou processos, temos de ser capazes de controlar sua execução. Esse controle requer a capacidade de determinar e reiniciar os atributos de uma tarefa ou processo, incluindo a prioridade da tarefa, seu tempo de execução máximo permitível, e assim por diante (`get process attributes` e `set process attributes`). Também podemos querer terminar uma tarefa ou processo criado (`terminate process`) se descobrirmos que está incorreto ou não é mais necessário.

A partir da criação de novas tarefas ou processos, podemos ter de esperar até que terminem sua execução. Podemos esperar por um tempo (`wait time`); mas provavelmente desejaremos esperar até que ocorra um evento específico (`wait event`). As tarefas ou processos deverão, então, sinalizar quando esse evento ocorreu (`signal event`). Frequentemente, dois ou mais processos compartilham dados. Para garantir a integridade dos dados sendo compartilhados, os sistemas operacionais costumam prover chamadas do sistema para permitir que um processo **bloqueie** os dados compartilhados, impedindo, assim, que outro processo acesse os dados até que o bloqueio seja removido. Normalmente, essas chamadas do sistema incluem `acquire lock` e `release lock`. As chamadas de sistema desse tipo, lidando com a coordenação de processos concorrente, serão discutidas em detalhe no [Capítulo 6](#).

EXEMPLOS DE CHAMADAS DO SISTEMA DO WINDOWS E DO UNIX

	Windows	Unix
Controle de processo	<code>CreateProcess()</code>	<code>fork()</code>
	<code>ExitProcess()</code>	<code>exit()</code>
	<code>WaitForSingleObject()</code>	<code>wait()</code>
Manipulação de arquivo	<code>CreateFile()</code>	<code>open()</code>
	<code>ReadFile()</code>	<code>read()</code>
	<code>WriteFile()</code>	<code>write()</code>
	<code>CloseHandle()</code>	<code>close()</code>
Manipulação de dispositivo	<code>SetConsoleMode()</code>	<code>ioctl()</code>
	<code>ReadConsole()</code>	<code>read()</code>
	<code>WriteConsole()</code>	<code>write()</code>
	<code>GetCurrentProcessID()</code>	<code>getpid()</code>
Manutenção de informação	<code>SetTimer()</code>	<code>alarm()</code>
	<code>Sleep()</code>	<code>sleep()</code>
	<code>CreatePipe()</code>	<code>pipe()</code>
Comunicação	<code>CreateFileMapping()</code>	<code>shmem()</code>
	<code>MapViewOfFile()</code>	<code>mmap()</code>
	<code>SetFileSecurity()</code>	<code>chmod()</code>
Proteção	<code>InitializeSecurityDescriptor()</code>	<code>umask()</code>
	<code>SetSecurityDescriptorGroup()</code>	<code>chown()</code>

EXEMPLO DE BIBLIOTECA C PADRÃO

A biblioteca C padrão fornece uma parte da interface de chamada de sistema para muitas versões do UNIX e Linux. Como exemplo, vamos supor que um programa C invoque a instrução `printf()`. A biblioteca C intercepta essa chamada e invoca as chamadas de sistema necessárias no sistema operacional – nesse caso, a chamada de sistema `write()`. A biblioteca C apanha o valor retornado por `write()` e o passa de volta ao programa do usuário. Isso pode ser visto na [Figura 2.9](#).

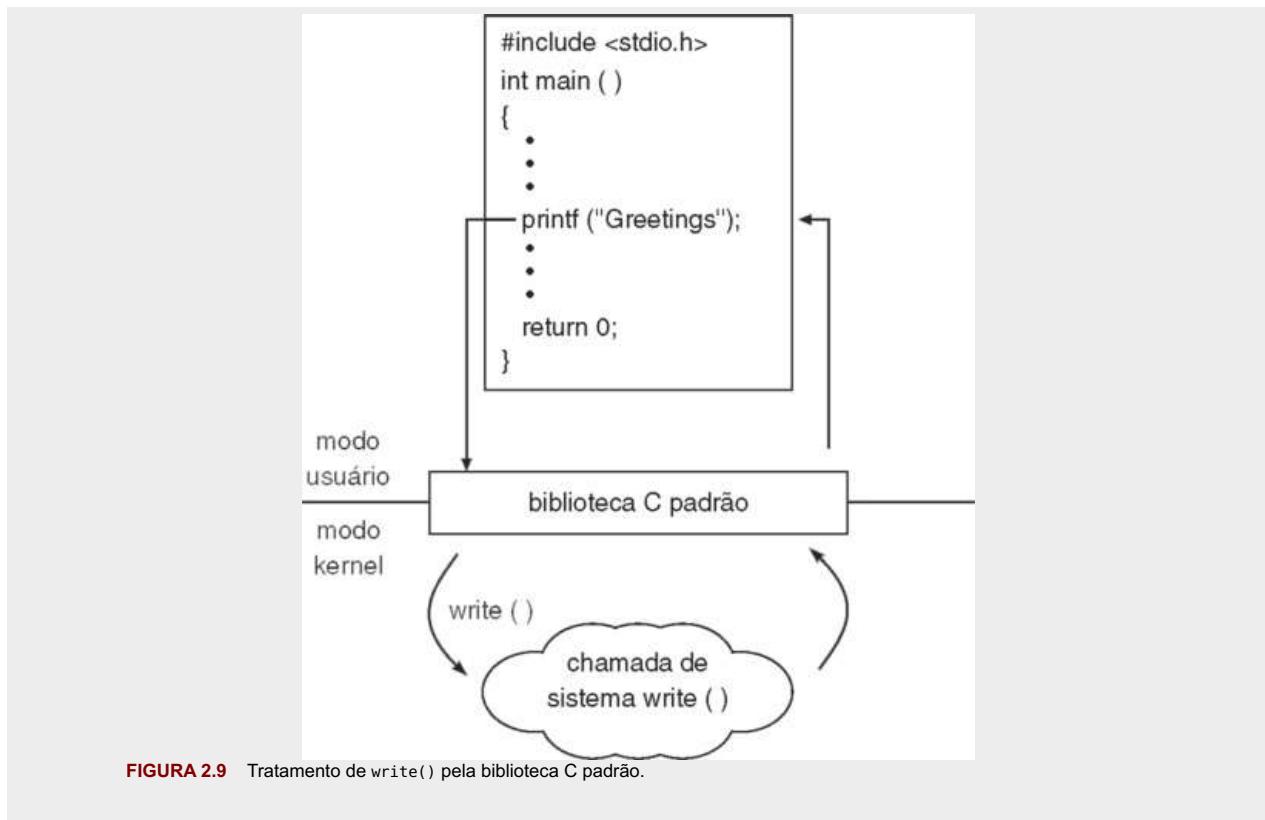


FIGURA 2.9 Tratamento de `write()` pela biblioteca C padrão.

Existem tantas facetas e variações no controle de processos e tarefas que, a seguir, usamos dois exemplos - um envolvendo um sistema monotarefa e outro mostrando um sistema multitarefa - para esclarecer esses conceitos. O sistema operacional MS-DOS é um exemplo de sistema monotarefa. Ele possui um interpretador de comandos que é chamado quando o computador é iniciado ([Figura 2.10a](#)). Como o MS-DOS é um sistema monotarefa, ele usa um método simples para executar um programa e não cria um novo processo. Ele carrega o programa na memória, escrevendo sobre a maior parte de si mesmo para dar ao programa o máximo de memória possível ([Figura 2.10b](#)). Em seguida, ele define o ponteiro de instrução para a primeira instrução do programa. O programa é executado e, em seguida, ou um erro causa um trap ou o programa executa uma chamada de sistema para terminar sua execução. De qualquer forma, o código de erro é salvo na memória do sistema para uso posterior. Após essa ação, a pequena parte do interpretador de comando não modificada retoma a execução. Sua primeira tarefa é recarregar o restante do interpretador de comandos do disco. Em seguida, o interpretador de comandos torna o código de erro anterior disponível ao usuário ou ao programa seguinte.

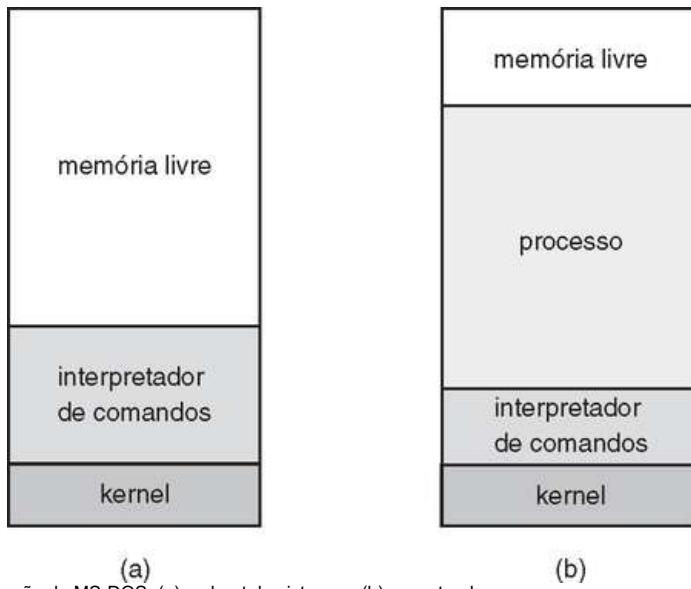


FIGURA 2.10 Execução do MS-DOS: (a) no boot do sistema e (b) executando um programa.

O FreeBSD (derivado do Berkeley UNIX) é um exemplo de um sistema multitarefa. Quando um usuário se conecta ao sistema, o shell escolhido pelo usuário é executado. Esse shell é semelhante ao shell do MS-DOS porque aceita comandos e executa programas requisitados pelo usuário. No entanto, como o FreeBSD é um sistema multitarefa, o interpretador de comandos pode continuar executando enquanto outro programa é executado ([Figura 2.11](#)). Para iniciar um novo processo, o shell executa uma chamada de sistema `fork()`. Depois, o programa selecionado é carregado na memória por meio de uma chamada de sistema `exec()`, e o programa é executado. Dependendo da forma como o comando foi emitido, o shell espera que o processo termine ou executa o processo “em segundo plano”. Nesse último caso, o shell requisita outro comando imediatamente. Quando um processo está executando em segundo plano, ele não pode receber entrada diretamente do teclado, pois o shell está usando esse recurso. A E/S, portanto, é feita por meio de arquivos ou da interface GUI. Nesse ínterim, o usuário está livre para pedir ao shell que execute outros programas, monitorar o progresso do processo em execução, alterar a prioridade desse programa, e assim por diante. Quando o processo conclui, ele executa uma chamada de sistema `exit()` para terminar, retornando ao processo que o chamou um código de status 0 ou um código de erro diferente de zero. Esse código de status ou erro estará disponível ao shell ou a outros programas. Os processos serão discutidos no [Capítulo 3](#), que inclui um exemplo de programa usando as chamadas de sistema `fork()` e `exec()`.

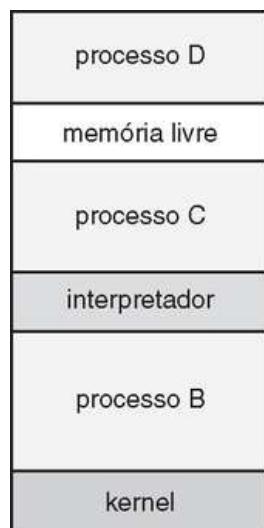


FIGURA 2.11 FreeBSD executando vários programas.

2.4.2 Gerência de arquivos

O sistema de arquivos é discutido com mais detalhes nos [Capítulos 10 e 11](#). Entretanto, podemos

identificar várias chamadas de sistema comuns que lidam com arquivos.

Primeiro, precisamos ser capazes de criar e excluir arquivos. Essas duas chamadas de sistema exigem o nome do arquivo e talvez alguns dos atributos do arquivo. Quando o arquivo é criado, precisamos abri-lo e usá-lo. Também podemos ler, escrever ou reposicionar (retornar ao início ou saltar para o final do arquivo, por exemplo). Finalmente, precisamos fechar o arquivo, indicando que não está mais sendo usado.

Podemos precisar desses mesmos conjuntos de operações para diretórios, se tivermos uma estrutura de diretório para organizar arquivos no sistema de arquivos. Além disso, para arquivos ou diretórios, precisamos determinar os valores de diversos atributos e, talvez, modificá-los, se necessário. Os atributos de arquivo incluem nome do arquivo, tipo do arquivo, códigos de proteção, informações contábeis, e assim por diante. Pelo menos duas chamadas de sistema, `get file attribute` e `set file attribute`, são exigidas para essa função. Alguns sistemas operacionais fornecem muitas outras chamadas, como chamadas para `file move` e `copy`. Outros podem fornecer uma API que realize essas operações usando código e outras chamadas do sistema, e ainda outros podem apenas fornecer programas do sistema para realizar as tarefas. Se os programas do sistema puderem ser chamados por outros programas, então cada um pode ser considerado uma API por outros programas do sistema.

2.4.3 Gerência de dispositivos

Para ser executado, um processo pode precisar de vários recursos - memória principal, unidades de disco, acesso a arquivos, e assim por diante. Se os recursos estiverem disponíveis, eles podem ser concedidos, e o controle pode ser retornado ao processo do usuário. Caso contrário, o processo terá que esperar até que existam recursos suficientes.

Os diversos recursos controlados pelo sistema operacional podem ser imaginados como dispositivos. Alguns desses dispositivos são dispositivos físicos (por exemplo, unidades de disco), enquanto outros são dispositivos abstratos ou virtuais (por exemplo, arquivos). Se houver vários usuários, alguns sistemas exigem que primeiro requisitemos um dispositivo, para garantir seu uso exclusivo. Depois que terminarmos de usar o dispositivo, nós o liberamos (`release`). Essas funções são semelhantes às chamadas de sistema `open` e `close` para arquivos. Outros sistemas operacionais permitem o acesso não gerenciado aos dispositivos. O risco, então, é o potencial de disputa de dispositivo e talvez deadlock, que é descrito no [Capítulo 7](#).

Quando o dispositivo tiver sido requisitado (e alocado para nós), podemos ler (`read`), escrever (`write`) e, possivelmente, reposicionar (`reposition`) o dispositivo, assim como podemos fazer com arquivos comuns. De fato, a semelhança entre os dispositivos de E/S e os arquivos é tão grande que muitos sistemas operacionais, incluindo UNIX, reúnem os dois em uma estrutura combinada de arquivo-dispositivo. Nesse caso, um conjunto de chamadas de sistema é usado em arquivos e dispositivos. Às vezes, os dispositivos de E/S são identificados por nomes de arquivo especiais, posicionamento em um diretório ou atributos de arquivo.

A interface com o usuário também faz arquivos e dispositivos parecerem semelhantes, embora as chamadas de sistema básicas sejam diferentes. Esse é outro exemplo das muitas decisões de projeto que entram na montagem de um sistema operacional e interface com o usuário.

2.4.4 Manutenção de informações

Muitas chamadas de sistema existem com a finalidade de transferir informações entre o programa do usuário e o sistema operacional. Por exemplo, a maioria dos sistemas possui uma chamada de sistema para retornar a hora e a data atuais. Outras chamadas de sistema podem retornar informações sobre o sistema, como o número de usuários atual, o número de versão do sistema operacional, a quantidade livre de memória ou espaço em disco, e assim por diante.

Outro conjunto de chamadas do sistema é útil na depuração de um programa. Muitos sistemas oferecem chamadas do sistema para o `dump` de memória. Essa provisão é útil para a depuração. Um trace do programa lista cada chamada do sistema à medida que é executada. Até mesmo os microprocessadores oferecem um modo da CPU conhecido como *single step*, no qual um trap é executado pela CPU após cada instrução. O trap normalmente é apanhado por um depurador.

Muitos sistemas operacionais oferecem um perfil de tempo de um programa, para indicar a quantidade de tempo que o programa passa executando em determinado local ou conjunto de locais. Um perfil de tempo requer uma facilidade de tracing ou interrupções regulares do temporizador. Em cada ocorrência da interrupção do temporizador, o valor do contador de programa é registrado. Com interrupções de temporizador com frequência suficiente, pode-se obter uma imagem estatística do tempo gasto em diversas partes do programa.

Além disso, o sistema operacional mantém informações sobre todos os seus processos, e existem chamadas de sistema para acessar essa informação. Em geral, também existem chamadas para reiniciar a informação do processo (`get process attributes` e `set process attributes`). Na [Seção 3.1.3](#), discutiremos quais informações normalmente são mantidas.

2.4.5 Comunicações

Existem dois modelos de comunicação: o modelo de troca de mensagens e o modelo de memória compartilhada. No **modelo de troca de mensagens**, os processos em comunicação trocam mensagens entre si para transferir informações. As mensagens podem ser trocadas entre os processos direta ou indiretamente, por meio de uma caixa de correio comum. Antes de ocorrerem as comunicações, é preciso que uma conexão seja aberta. O nome do outro comunicador precisa ser conhecido, seja outro processo no mesmo sistema ou um processo em outro computador conectado por uma rede de comunicações. Cada computador em uma rede possui um *nome de hospedeiro* (ou *host*), pelo qual é conhecido. De modo semelhante, cada processo possui um *nome de processo*, traduzido para um identificador pelo qual o sistema operacional pode se referir a ele. As chamadas de sistema `get hostid` e `get processid` realizam essa tradução. Esses identificadores são, então, passados às chamadas `open` e `close` de uso geral fornecidas pelo sistema de arquivos ou às chamadas de sistema `open connection` e `close connection`, dependendo do modelo de comunicações do sistema. O processo de destino precisa dar sua permissão para ocorrer a comunicação, com uma chamada `accept connection`. A maioria dos processos que estão recebendo conexões é *daemons* de uso especial, programas do sistema fornecidos para essa finalidade. Eles executam uma chamada `wait for connection` e são acordados quando for feita uma conexão. A origem da comunicação, conhecida como *cliente*, e o daemon de recepção, conhecido como *servidor*, trocam mensagens pelas chamadas de sistema `read message` e `write message`. A chamada `close connection` termina a comunicação.

No **modelo de memória compartilhada**, os processos utilizam chamadas de sistema `shared memory create` e `shared memory attach` para criar e obter acesso a regiões da memória de outros processos. Lembre-se de que o sistema operacional tenta evitar o acesso de um processo à memória de outro processo. A memória compartilhada exige que dois ou mais processos concordem em remover essa restrição. Eles podem trocar informações lendo e escrevendo dados nas áreas compartilhadas. O formato dos dados é determinado por esses processos e não está sob o controle do sistema operacional. Os processos também são responsáveis por garantir que não estão escrevendo no mesmo local simultaneamente. Esses mecanismos são discutidos no [Capítulo 6](#). No [Capítulo 4](#), veremos uma variação do modelo de processo - threads - em que a memória é compartilhada por definição.

Os dois modelos discutidos são comuns em sistemas operacionais, e alguns sistemas até mesmo implementam ambos. A troca de mensagens é útil para trocar quantidades de dados menores, pois nenhum conflito precisa ser evitado. Ela também é mais fácil de implementar do que a memória compartilhada para a comunicação entre computadores. A memória compartilhada permite o máximo de velocidade e conveniência de comunicação, pois pode ser feita em velocidades de memória dentro de um computador. Todavia, existem problemas nas áreas de proteção e sincronismo entre os processos de compartilhamento de memória.

2.4.6 Proteção

A proteção oferece um mecanismo para controlar o acesso aos recursos de um sistema de computação. Historicamente, a proteção tem sido uma fonte de preocupação somente nos sistemas de computação multiprogramados, com vários usuários. Porém, com o advento da rede e da Internet, todos os sistemas de computação, de servidores a PDAs, precisam estar atentos à proteção.

Normalmente, as chamadas do sistema que oferecem proteção incluem `set permission` e `get permission`, que manipulam as configurações de permissão de recursos como arquivos e discos. As chamadas do sistema `allow user` e `deny user` especificam se usuários em particular podem ou não ter permissão de acesso a determinados recursos.

Explicaremos sobre a proteção no [Capítulo 14](#), e o aspecto muito mais amplo da segurança no [Capítulo 15](#).

2.5 Programas do sistema

Outro aspecto de um sistema operacional moderno é a coleção de programas inclusos nele. Veja novamente a [Figura 1.1](#), que representa a hierarquia lógica do computador. No nível mais baixo está o hardware. Em seguida está o sistema operacional, depois os programas do sistema e finalmente os programas de aplicação. Os **programas do sistema**, também conhecidos como utilitários do sistema, fornecem um ambiente conveniente para desenvolvimento e execução de programas. Alguns deles são interfaces do usuário para chamadas de sistema; outros são bem mais complexos. Eles podem ser divididos nessas categorias:

- **Gerência de arquivos.** Esses programas criam, removem, copiam, renomeiam, imprimem, fazem dump, listam e geralmente manipulam arquivos e diretórios.
- **Informações de status.** Alguns programas pedem do sistema a data, a hora, quantidade disponível de memória ou espaço em disco, número de usuários ou informações de status semelhantes. Outros são mais complexos, fornecendo informações detalhadas de desempenho, logging e depuração. Normalmente, esses programas formatam e imprimem a saída no terminal ou em outros dispositivos de saída ou arquivos, ou a exibem em uma janela da GUI. Alguns sistemas também admitem um **registro**, que é usado para armazenar e recuperar informações de configuração.
- **Modificação de arquivos.** Diversos editores de textos podem estar disponíveis para criar e modificar o conteúdo dos arquivos armazenados em disco ou outros dispositivos de armazenamento. Também pode haver comandos especiais para procurar o conteúdo de arquivos ou realizar transformações do texto.
- **Suporte para linguagem de programação.** Compiladores, montadores (assemblers), depuradores e interpretadores para linguagens de programação comuns (como C, C++, Java, Visual Basic e PERL) normalmente são fornecidos para o usuário com o sistema operacional.
- **Carga e execução de programas.** Quando um programa é montado ou compilado, ele precisa ser carregado para a memória, a fim de ser executado. O sistema pode fornecer carregadores absolutos (absolute loaders), carregadores relocáveis (relocatable loaders), editores de ligação (linkage editors) e carregadores de overlay (overlay loaders). Sistemas de depuração para linguagens de alto nível ou linguagem de máquina também são necessários.
- **Comunicações.** Esses programas fornecem o mecanismo para criar conexões virtuais entre processos, usuários e sistemas computadorizados. Eles permitem aos usuários enviar mensagens para as telas um do outro, navegar por páginas Web, enviar mensagens de correio eletrônico, efetuar o login remoto ou transferir arquivos de uma máquina para outra.

Além dos programas dos sistemas, a maioria dos sistemas operacionais vem com programas úteis para resolver problemas comuns ou realizar operações comuns. Entre esses **programas de aplicação** estão navegadores Web, processadores e formatadores de textos, planilhas, sistemas de banco de dados, compiladores, pacotes gráficos e de análise estatística, e jogos.

A visão do sistema operacional que pode ter a maioria dos usuários é definida pelos programas de aplicação e do sistema, e não pelas chamadas de sistema reais. Considere o PC de um usuário. Quando o computador de um usuário está executando o sistema operacional Mac OS X, ele pode ver a GUI, usando um mouse e uma interface de janelas. Como alternativa, ou até mesmo em uma das janelas, ele pode ter um shell UNIX da linha de comandos. Ambos utilizam o mesmo conjunto de chamadas de sistema, mas as chamadas de sistema aparecem de forma diferente e atuam de maneiras diferentes. Para confundir ainda mais a visão do usuário, considere um usuário realizando um dual-booting entre o Mac OS X e o Windows Vista. Agora, o mesmo usuário, no mesmo hardware, tem duas interfaces totalmente diferentes e dois conjuntos de aplicações usando os mesmos recursos físicos. No mesmo hardware, então, um usuário pode ser exposto a diversas interfaces de usuário de modo sequencial ou simultâneo.

2.6 Projeto e implementação do sistema operacional

Nesta seção, discutimos problemas que enfrentamos no projeto e implementação de um sistema operacional. Naturalmente, não existem soluções completas para esses problemas, mas existem técnicas que provaram ter sucesso.

2.6.1 Objetivos de projeto

O primeiro problema no projeto de um sistema é definir objetivos e especificações. No nível mais alto, o projeto do sistema será afetado pela escolha do hardware e do tipo de sistema: batch, tempo compartilhado, único usuário, multiusuário, distribuído, tempo real ou uso geral.

Além desse nível de projeto mais alto, os requisitos podem ser muito mais difíceis de especificar. Porém, os requisitos podem ser divididos em dois grupos básicos: objetivos do usuário e objetivos do sistema.

Os usuários desejam certas propriedades óbvias em um sistema: o sistema deve ser conveniente de usar, fácil de aprender e usar, confiável, seguro e rápido. Naturalmente, essas especificações não são particularmente úteis no projeto do sistema, pois não existe um acordo geral sobre como alcançá-las.

Um conjunto de requisitos semelhante pode ser definido pelas pessoas que precisam projetar, criar, manter e operar o sistema. O sistema deverá ser fácil de projetar, implementar e manter; ele deverá ser flexível, confiável, livre de erros e eficiente. Novamente, esses requisitos são vagos e podem ser interpretados de diversas maneiras.

Resumindo, não existe solução única para o problema de definir os requisitos para um sistema operacional. A grande variedade de sistemas em existência mostra que diferentes requisitos podem resultar em uma grande variedade de soluções para ambientes diferentes. Por exemplo, os requisitos para VxWorks, um sistema operacional de tempo real para sistemas embutidos, precisam ter sido substancialmente diferentes dos requisitos para MVS, um grande sistema operacional multiusuário, de múltiplo acesso, para mainframes IBM.

Especificando e projetar um sistema operacional é uma tarefa bastante criativa. Embora nenhum livro possa lhe dizer como fazer isso, princípios gerais têm sido desenvolvidos no setor de **engenharia de software**, e agora passamos para uma discussão sobre alguns desses princípios.

2.6.2 Mecanismos e políticas

Um princípio importante é a separação entre **política** e **mecanismo**. Mecanismos determinam *como* fazer algo; políticas determinam *o que* será feito. Por exemplo, a construção de temporizador ([Seção 1.5.2](#)) é um mecanismo para garantir a proteção da CPU, mas decidir por quanto tempo o timer deve ser configurado para determinado usuário é uma decisão de política.

A separação entre política e mecanismo é importante para a flexibilidade. As políticas provavelmente mudam entre lugares e com o tempo. No pior dos casos, cada mudança na política exigiria uma mudança no mecanismo básico. Um mecanismo não sensível a mudanças na política seria mais desejável. Uma mudança na política, então, exigiria a redefinição apenas de certos parâmetros do sistema. Por exemplo, considere um mecanismo para dar prioridade a certos tipos de programas em relação a outros. Se o mecanismo for devidamente separado da política, ele poderá ser usado para dar suporte a uma decisão política de que os programas com uso intenso de E/S devem ter prioridade em relação àqueles com uso intenso de CPU ou dar suporte à política oposta.

Os sistemas operacionais baseados em microkernel ([Seção 2.7.3](#)) levam à separação entre mecanismo e política a um extremo, implementando um conjunto básico de primitivas tipo blocos de montagem. Esses blocos são quase livres de política, permitindo que mecanismos e políticas mais avançados sejam acrescentados por meio de módulos de kernel criados pelo usuário ou por meio dos próprios programas do usuário. Como exemplo, considere a história do UNIX. Inicialmente, ele tinha um escalonador de tempo compartilhado. Na versão mais recente do Solaris, o escalonamento é controlado por tabelas carregáveis. Dependendo da tabela atualmente carregada, o sistema pode ser de tempo compartilhado, processamento de batch, tempo real, fair share ou qualquer combinação. Tornar o mecanismo de escalonamento de uso geral permite que vastas mudanças de política sejam feitas com um único comando `load-new-table`. No outro extremo existe um sistema como o Windows, em que mecanismo e política são codificados no sistema para impor um estilo global. Todas as aplicações possuem interfaces semelhantes, pois a própria interface está embutida no kernel e nas bibliotecas do sistema. O sistema operacional Mac OS X possui funcionalidade semelhante.

Decisões políticas são importantes para toda alocação de recursos. Sempre que é necessário decidir sobre a alocação ou não de um recurso, uma decisão política precisa ser tomada. Sempre que a pergunta é *como* em vez de *o que*, é um mecanismo que precisa ser determinado.

2.6.3 Implementação

Quando um sistema operacional é projetado, ele precisa ser implementado. Tradicionalmente, os sistemas operacionais têm sido escritos em linguagem assembly. Agora, porém, eles normalmente são escritos em linguagens de nível mais alto, como C ou C++.

O primeiro sistema que não foi escrito em linguagem assembly provavelmente foi o Master Control Program (MCP), para computadores Burroughs. O MCP foi escrito em uma variante do ALGOL. O MULTICS, desenvolvido no MIT, foi escrito principalmente em PL/1. Os sistemas operacionais Linux e Windows XP são escritos principalmente em C, embora incluam algumas pequenas seções de código assembly para drivers de dispositivos e para salvar e restaurar o estado dos registradores.

As vantagens do uso de uma linguagem de nível mais alto, ou pelo menos de uma linguagem de implementação de sistemas, para implementar os sistemas operacionais, são as mesmas resultantes de quando a linguagem é usada para programas de aplicação. O código pode ser escrito mais rapidamente, é mais compacto e é mais fácil de entender e depurar. Além disso, melhorias na tecnologia de compilador melhorarão o código gerado para o sistema operacional inteiro, pela simples recompilação. Finalmente, um sistema operacional é muito mais fácil de ser *portado* – movido para algum outro hardware – se for escrito em uma linguagem de nível mais alto. Por exemplo, o MS-DOS foi escrito na linguagem assembly do Intel 8088. Consequentemente, ele está disponível apenas na família de CPUs X86 da Intel. (Embora o MS-DOS seja executado de forma nativa apenas no Intel X86, emuladores do conjunto de instruções X86 permitem que o sistema operacional funcione de forma não nativa – mais lentamente, usando mais recursos – em outras CPUs. **Emuladores** são programas que duplicam a funcionalidade de um sistema em outro sistema.) O sistema operacional Linux, ao contrário, é escrito principalmente em C, e está disponível de forma nativa em diversas CPUs diferentes, incluindo Intel X86, Sun SPARC e IBMPowerPC.

As únicas desvantagens possíveis de implementar um sistema operacional em uma linguagem de nível mais alto são velocidade reduzida e requisitos de armazenamento aumentados. Contudo, isso não é mais um problema importante nos sistemas de hoje. Embora um programador especialista em linguagem assembly possa produzir rotinas pequenas e eficientes, para programas grandes, um compilador moderno pode realizar uma análise complexa e aplicar otimizações sofisticadas, que produzem um código excelente. Os processadores modernos possuem um pipelining profundo e múltiplas unidades funcionais que podem lidar com os detalhes das dependências complexas muito mais facilmente do que a mente humana.

Como acontece em outros sistemas, as principais melhorias de desempenho nos sistemas operacionais provavelmente são o resultado de melhores estruturas de dados e algoritmos e não de um código excelente em linguagem assembly. Além disso, embora os sistemas operacionais sejam grandes, somente uma pequena quantidade de código é crítica para o alto desempenho; o gerenciador de memória e o escalonador de CPU provavelmente são as rotinas mais críticas. Após esse sistema ser escrito e estar funcionando corretamente, rotinas de gargalo podem ser identificadas e substituídas por equivalentes em linguagem assembly. (Os gargalos serão discutidos mais adiante neste capítulo.)

2.7 Estrutura do sistema operacional

Um sistema tão grande e complexo quanto um sistema operacional moderno precisa ser arquitetado com cuidado para funcionar de modo apropriado e ser modificado com facilidade. Uma técnica comum é partitionar a tarefa em componentes menores, em vez de ter um sistema monolítico. Cada um desses módulos deverá ser uma parte bem definida do sistema, com entradas, saídas e funções bem definidas. Já discutimos rapidamente os componentes comuns dos sistemas operacionais no [Capítulo 1](#). Nesta seção, discutimos como esses componentes são interconectados e ligados ao kernel.

2.7.1 Estrutura simples

Muitos sistemas operacionais comerciais não possuem estruturas bem definidas. Constantemente, esses sistemas operacionais começaram como sistemas pequenos, simples e limitados, e depois cresceram para além do seu escopo original. O MS-DOS é um exemplo desse sistema. Ele foi projetado e implementado originalmente por algumas pessoas que não tinham ideia de que ele se tornaria tão popular. Ele foi escrito para fornecer o máximo de funcionalidade no menor espaço possível, de modo que não foi dividido com cuidado em módulos. A [Figura 2.12](#) mostra sua estrutura.

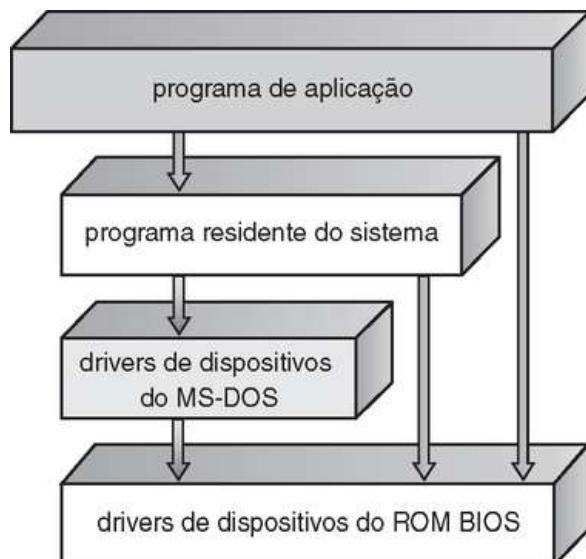


FIGURA 2.12 Estrutura de camadas do MS-DOS.

No MS-DOS, as interfaces e os níveis de funcionalidade não são bem separados. Por exemplo, os programas de aplicação são capazes de acessar as rotinas básicas de E/S para escrever diretamente no vídeo e nas unidades de disco. Essa liberdade deixa o MS-DOS vulnerável a programas com erros (ou maliciosos), derrubando o sistema inteiro quando os programas do usuário falham. É natural que o MS-DOS também fosse limitado pelo hardware de sua época. Como o processador Intel 8088, para o qual ele foi escrito, não provê um modo dual e nenhuma proteção do hardware, os projetistas do MS-DOS não tinham outra escolha além de deixarem o hardware básico acessível.

Outro exemplo de estruturação limitada é o sistema operacional UNIX original. Assim como o MS-DOS, o UNIX inicialmente foi limitado pela funcionalidade do hardware. Ele consiste em duas partes separadas: o kernel e os programas do sistema. O kernel é separado ainda mais em uma série de interfaces e drivers de dispositivos, que foram acrescentados e expandidos com o passar dos anos, enquanto o UNIX evoluía. Podemos ver o sistema operacional UNIX tradicional como sendo em camadas, como mostra a [Figura 2.13](#). Tudo abaixo da interface de chamada de sistema e acima do hardware físico é o kernel. O kernel provê o sistema de arquivos, o escalonamento de CPU, a gerência de memória e outras funções do sistema operacional por meio das chamadas de sistema. Somando tudo, essa é uma grande quantidade de funcionalidade para ser combinada em um nível. Essa estrutura monolítica era difícil de implementar e manter.



FIGURA 2.13 Estrutura do sistema UNIX tradicional.

2.7.2 Enfoque em camadas

Dado o suporte de hardware apropriado, os sistemas operacionais podem ser divididos em partes menores e mais apropriadas do que as permitidas pelos sistemas MS-DOS ou UNIX originais. O sistema operacional pode, então, reter um controle muito maior sobre o computador e sobre as aplicações que utilizam esse computador. Os implementadores possuem mais liberdade na mudança do funcionamento interno do sistema e na criação de sistemas operacionais modulares. Sob o enfoque top-down, a funcionalidade e os recursos gerais são determinados e separados em componentes. A ocultação de informações também é importante, pois deixa os programadores livres para implementar as rotinas de baixo nível como desejarem, desde que a interface externa da rotina permaneça inalterada e que a própria rotina execute a tarefa anunciada.

Pode-se criar um sistema modular de muitas maneiras. Um método é o **enfoque em camadas**, no qual o sistema operacional é dividido em uma série de camadas (níveis). A camada inferior (camada 0) é o hardware; a mais alta (camada N) é a interface com o usuário. A estrutura em camadas é representada na [Figura 2.14](#).

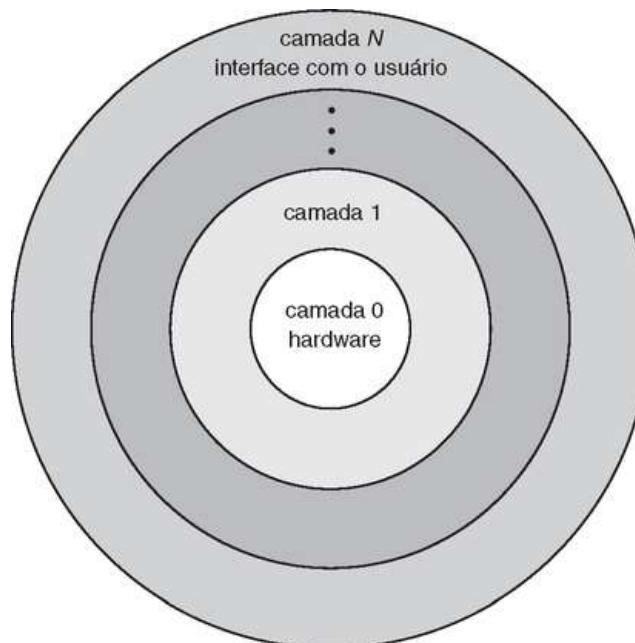


FIGURA 2.14 Um sistema operacional em camadas.

Uma camada do sistema operacional é uma implementação de um objeto abstrato, composta de dados e das operações que podem manipular esses dados. Uma camada típica do sistema

operacional - digamos, a camada M - consiste em estruturas de dados e em um conjunto de rotinas que podem ser invocadas por camadas de nível superior. A camada M , por sua vez, pode invocar operações em camadas de nível mais baixo.

A vantagem principal do enfoque em camadas é a simplicidade da construção e depuração. As camadas são selecionadas de modo que cada uma utilize funções (operações) e serviços apenas de camadas de nível mais baixo. Essa técnica simplifica a depuração e a verificação do sistema. A primeira camada pode ser depurada sem qualquer preocupação com o restante do sistema porque, por definição, utiliza apenas o hardware básico (considerado correto) para implementar suas funções. Quando a primeira camada é depurada, sua funcionalidade correta pode ser assumida, enquanto a segunda camada é depurada, e assim por diante. Se um erro for encontrado durante a depuração de uma camada em particular, o erro precisa estar nessa camada, pois as camadas abaixo dela já estão depuradas. Assim, o projeto e a implementação do sistema são simplificados.

Cada camada é implementada apenas com as operações fornecidas pelas camadas de nível mais baixo. Ela não precisa saber como essas operações são implementadas; só precisa saber o que essas operações fazem. Logo, cada camada oculta a existência de certas estruturas de dados, operações e hardware das camadas de nível mais alto.

A principal dificuldade com o enfoque em camadas envolve a definição apropriada das diversas camadas. Como uma camada só pode usar as camadas de nível mais baixo, é preciso haver um planejamento cuidadoso. Por exemplo, o driver de dispositivo para o armazenamento de apoio (espaço em disco usado pelos algoritmos de memória virtual) precisa estar em um nível inferior ao das rotinas de gerência de memória, pois a gerência de memória exige a capacidade de usar o armazenamento de apoio.

Outros requisitos podem não ser tão óbvios. O driver do armazenamento de apoio normalmente estaria acima do escalonador de CPU, pois o driver pode ter de esperar pela E/S, e a CPU pode ser reescalonada durante esse tempo. No entanto, em um sistema grande, o escalonador de CPU pode ter mais informações sobre todos os processos ativos do que poderiam caber na memória. Portanto, essa informação pode precisar ser colocada e retirada da memória, exigindo que a rotina do driver de armazenamento de apoio fique abaixo do escalonador de CPU.

Um último problema com as implementações em camadas é que costumam ser menos eficientes do que outros tipos. Por exemplo, quando um programa do usuário executa uma operação de E/S, ele executa uma chamada de sistema que é interceptada para a camada de E/S, que chama a camada de gerência de memória, que, por sua vez, chama a camada de escalonamento de CPU, que é passada para o hardware. Em cada camada, os parâmetros podem ser modificados, dados podem precisar ser passados, e assim por diante. Cada camada acrescenta um custo (overhead) adicional à chamada de sistema; o resultado disso é uma chamada de sistema que leva mais tempo do que outra em um sistema que não seja em camadas.

Essas limitações causaram um pequeno recuo contra o enfoque em camadas nos últimos anos. Menos camadas com mais funcionalidade estão sendo projetadas, provendo a maioria das vantagens do código modular enquanto evitam os problemas difíceis da definição e interação da camada.

2.7.3 Microkernels

Vimos que, à medida que o UNIX se expandiu, o kernel se tornou grande e difícil de gerenciar. Em meados da década de 1980, pesquisadores na Carnegie Mellon University desenvolveram um sistema operacional chamado **Mach**, que modularizou o kernel, usando a técnica de **microkernel**. Esse método estrutura o sistema operacional removendo todos os componentes não essenciais do kernel e implementando-os como programas em nível de sistema e usuário. O resultado é um kernel menor. Existe pouco consenso em relação a quais serviços devem permanecer no kernel e quais devem ser implementados no espaço do usuário. Contudo, em geral, os microkernels normalmente fornecem uma gerência mínima de processo e memória, além da facilidade de comunicação.

A função principal do microkernel é fornecer facilidade de comunicação entre o programa cliente e os diversos serviços executados no espaço do usuário. A comunicação é fornecida pela *troca de mensagens*, descrita na [Seção 2.4.5](#). Por exemplo, se o programa cliente quiser acessar um arquivo, ele precisará interagir com o servidor de arquivos. O programa cliente e o serviço nunca interagem diretamente. Em vez disso, comunicam-se de forma indireta trocando mensagens com o microkernel.

Um benefício da abordagem microkernel é a facilidade de extensão do sistema operacional. Todos os novos serviços são acrescentados ao espaço do usuário e, como consequência, não exigem modificação do kernel. Quando o kernel tem de ser modificado, as mudanças costumam ser menores, pois o microkernel é um kernel menor. O sistema operacional resultante é mais fácil de ser passado de um projeto de hardware para outro. O microkernel também provê mais segurança e confiabilidade, pois a maioria dos serviços está executando como processo do usuário, em vez do kernel. Se um serviço falhar, o restante do sistema operacional permanecerá intocável.

Vários sistemas operacionais contemporâneos usaram a abordagem microkernel. O Tru64 UNIX (originalmente Digital UNIX) provê uma interface UNIX para o usuário, mas é implementado com um kernel Mach. O kernel Mach mapeia as chamadas de sistema UNIX em mensagens para os

serviços apropriados no nível do usuário. O kernel do Mac OS X (também conhecido como *Darwin*) também é baseado no microkernel Mach.

Outro exemplo é o QNX, um sistema operacional de tempo real. O microkernel do QNX provê serviços para a troca de mensagens e escalonamento de processo. Ele também trata da comunicação de baixo nível na rede e das interrupções de hardware. Todos os outros serviços no QNX são fornecidos por processos-padrão que executam fora do kernel no modo usuário.

Infelizmente, os microkernels podem sofrer de quedas de desempenho devido ao maior custo adicional da função do sistema. Considere a história do Windows NT. A primeira versão tinha uma organização de microkernel em camadas. Entretanto, essa versão forneceu um desempenho baixo em comparação com o Windows 95. O Windows NT 4.0 revisou parcialmente o problema do desempenho, movendo as camadas do espaço do usuário para o espaço do kernel e integrando-as mais de perto. Quando o Windows XP foi projetado, sua arquitetura foi mais monolítica do que o microkernel.

2.7.4 Módulos

Talvez a melhor metodologia atual de projeto de sistema operacional envolva o uso de técnicas de programação orientada a objeto para criar um kernel modular. Aqui, o kernel possui um conjunto de componentes básicos e vincula serviços adicionais, seja durante o processo de boot ou durante a execução. Essa estratégia utiliza módulos carregáveis dinamicamente e é comum nas implementações modernas do UNIX, como Solaris, Linux e Mac OS X. Por exemplo, a estrutura do sistema operacional Solaris, mostrada na [Figura 2.15](#), é organizada em torno de um kernel básico com sete tipos de módulos de kernel carregáveis:

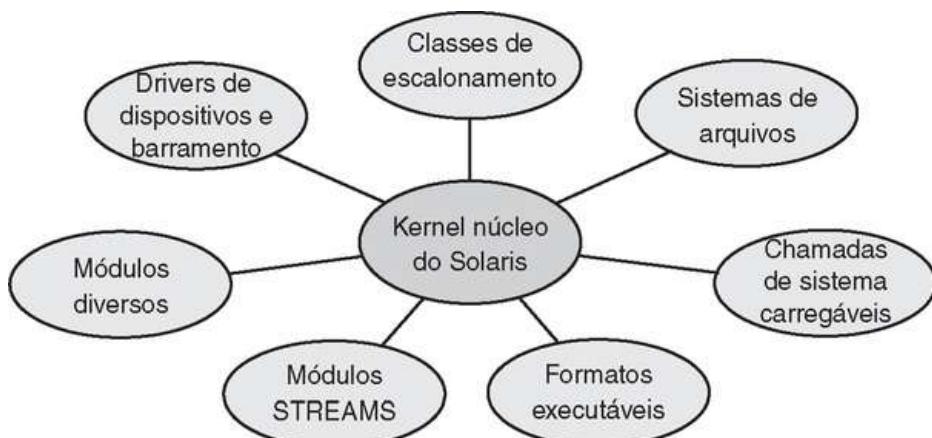


FIGURA 2.15 Módulos carregáveis do Solaris.

1. Classes de escalonamento.
2. Sistemas de arquivos.
3. Chamadas de sistema carregáveis.
4. Formatos executáveis.
5. Módulos STREAMS.
6. Módulos diversos.
7. Drivers de dispositivos e de barramento.

Esse projeto permite que o kernel forneça serviços básicos e também permite que determinados recursos sejam implementados dinamicamente. Por exemplo, os drivers de dispositivos e de barramento para um hardware específico podem ser acrescentados ao kernel, e o suporte para diferentes sistemas de arquivos pode ser acrescentado como módulos carregáveis. O resultado geral é semelhante a um sistema em camadas porque cada seção do kernel possui interfaces definidas e protegidas; mas ele é mais flexível do que um sistema em camadas porque qualquer módulo pode chamar qualquer outro módulo. Além do mais, a técnica é semelhante à do microkernel, porque o módulo primário tem apenas funções básicas e conhecimento de como carregar e se comunicar com os outros módulos; mas ela é mais eficiente, pois os módulos não precisam invocar a troca de mensagens para que possam se comunicar.

O sistema operacional Mac OS X da Apple utiliza uma estrutura híbrida. Ele é um sistema em camadas, no qual uma camada consiste no microkernel Mach. A estrutura do OS X aparece na [Figura 2.16](#). As camadas superiores incluem ambientes de aplicação e um conjunto de serviços que fornecem a interface gráfica para as aplicações. Abaixo dessas camadas está o ambiente do kernel, que consiste principalmente no microkernel Mach e no kernel BSD. O Mach provê gerência de memória, suporte para remote procedures calls (RPCs) e comunicação entre processos (interprocess communication - IPC), incluindo troca de mensagens e escalonamento de threads. O componente

BSD provê uma interface de linha de comandos BSD, suporte para redes e sistemas de arquivos, e uma implementação das APIs POSIX, incluindo Pthreads. Além de Mach e BSD, o ambiente do kernel fornece um kit de E/S para o desenvolvimento de drivers de dispositivos e módulos carregáveis dinamicamente (aos quais o OS X se refere como **extensões do kernel**). Como vemos na figura, as aplicações e os serviços comuns podem utilizar as facilidades Mach e BSD diretamente.

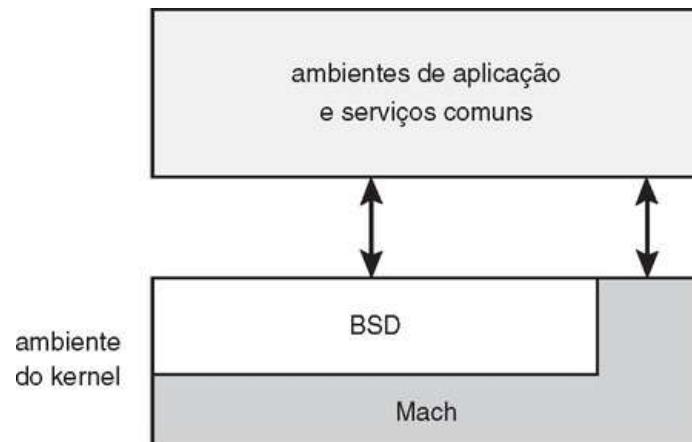


FIGURA 2.16 A estrutura do Mac OS X.

2.8 Máquinas virtuais

A técnica de camadas descrita na [Seção 2.7.2](#) é levada à sua conclusão lógica no conceito de **máquina virtual**. A ideia fundamental por trás de uma máquina virtual é separar o hardware ou um computador isolado (CPU, memória, unidades de disco, placas de interface de rede etc.) em vários ambientes de execução diferentes, criando assim a ilusão de que cada ambiente de execução separado está executando seu próprio computador privado.

Usando técnicas de escalonamento de CPU ([Capítulo 5](#)) e de memória virtual ([Capítulo 9](#)), um sistema operacional **hospedeiro** pode criar a ilusão de que um processo possui seu próprio processador com sua própria memória (virtual). A máquina virtual fornece uma interface *idêntica* à do hardware básico. Cada processo **guest** recebe uma cópia (virtual) do computador subjacente ([Figura 2.17](#)). Normalmente, o processo guest é, de fato, um sistema operacional, e é assim que uma máquina física pode executar vários sistemas operacionais simultaneamente, cada um com sua própria máquina virtual.

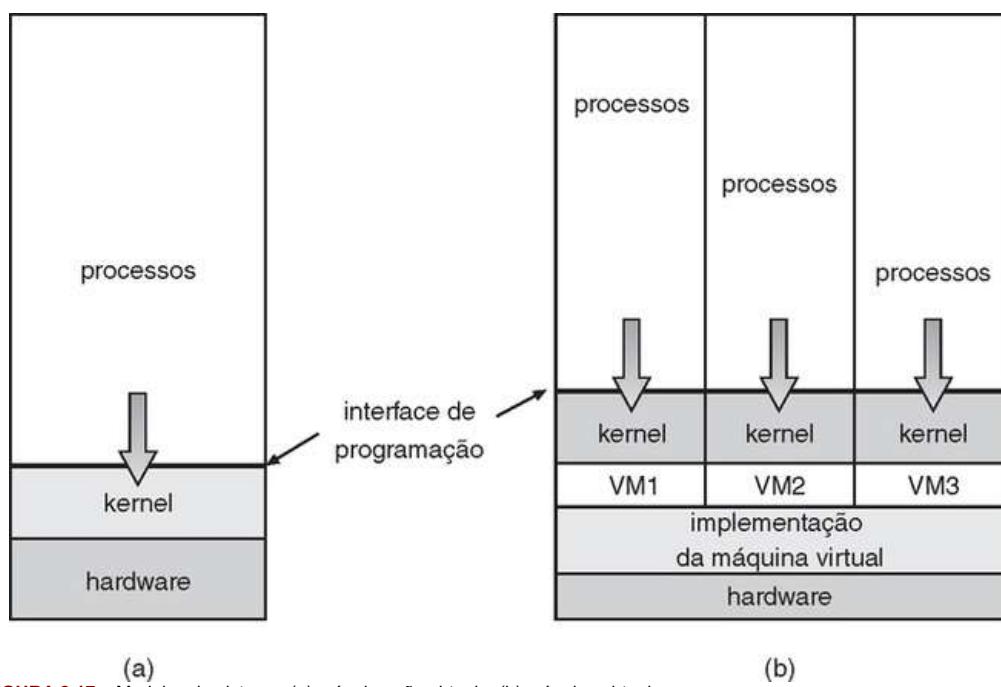


FIGURA 2.17 Modelos de sistema: (a) máquina não virtual e (b) máquina virtual.

2.8.1 Benefícios

Existem vários motivos para se criar uma máquina virtual, todos fundamentalmente relacionados com a capacidade de compartilhar o mesmo hardware executado em vários ambientes de execução diferentes (ou seja, diferentes sistemas operacionais) ao mesmo tempo.

Uma vantagem importante é que o sistema hospedeiro é protegido das máquinas virtuais, assim como as máquinas virtuais são protegidasumas das outras. Um vírus dentro de um sistema operacional guest poderia danificar esse sistema operacional, mas provavelmente não afetaria o hospedeiro ou os outros guests. Como cada máquina virtual é completamente isolada de todas as outras máquinas virtuais, não existem problemas de proteção. Ao mesmo tempo, não existe compartilhamento direto de recursos. Porém, duas técnicas para prover compartilhamento foram implementadas. Primeiro, é possível compartilhar um volume do sistema de arquivos e, portanto, compartilhar arquivos. Segundo, é possível definir uma rede de máquinas virtuais, cada uma podendo enviar informações pela rede de comunicações virtual. A rede é modelada nas redes de comunicação físicas, mas é implementada no software.

Uma vantagem para os desenvolvedores é que um sistema de máquina virtual é um veículo perfeito para pesquisa e desenvolvimento de sistemas operacionais. Normalmente, mudar um sistema operacional é uma tarefa difícil. Os sistemas operacionais são programas grandes e complexos, e é difícil ter certeza de que uma mudança em uma parte não causará bugs obscuros em alguma outra parte. O poder do sistema operacional torna sua mudança particularmente perigosa. Como o sistema operacional é executado no modo kernel, uma mudança errada em um ponteiro poderia causar um erro que destruiria o sistema de arquivos inteiro. Assim, é preciso testar todas as

mudanças no sistema operacional com muito cuidado.

Entretanto, o sistema operacional executa e controla a máquina inteira. Portanto, o sistema atual precisa ser terminado e retirado do uso enquanto as mudanças são feitas e testadas. Esse período é chamado de *tempo de desenvolvimento do sistema*. Como isso torna o sistema indisponível aos usuários, o tempo de desenvolvimento do sistema é marcado para a noite ou para os fins de semana, quando a carga do sistema é baixa.

Um sistema de máquina virtual pode eliminar grande parte desse problema. Os programadores de sistemas recebem sua própria máquina virtual, e o desenvolvimento do sistema é feito na máquina virtual, e não em uma máquina física. A operação normal do sistema raramente precisa ser interrompida para o desenvolvimento do sistema.

Outra vantagem das máquinas virtuais para os desenvolvedores é que diversos sistemas operacionais podem estar rodando na estação de trabalho do desenvolvedor simultaneamente. Essa estação de trabalho virtualizada permite o transporte rápido e o teste de programas em ambientes variados. De modo semelhante, os engenheiros de garantia de qualidade podem testar suas aplicações em diversos ambientes sem ter que comprar, alimentar e manter um computador para cada ambiente.

Uma vantagem importante das máquinas virtuais no uso do centro de dados de produção é a **consolidação** de sistemas, que envolve apanhar dois ou mais sistemas separados e executá-los em máquinas virtuais em um sistema. Essas conversões de físico para virtual resultam em otimização de recursos, pois muitos sistemas pouco usados podem ser combinados para criar um sistema usado com mais intensidade.

Se o uso de máquinas virtuais continuar a se expandir, a implantação de aplicações terá uma evolução correspondente, criando vantagens adicionais. Se um sistema puder facilmente acrescentar, remover e mover uma máquina virtual, então por que instalar aplicações nesse sistema diretamente? Em vez disso, os desenvolvedores de aplicação pré-instalariam a aplicação em um sistema operacional ajustado e customizado para a aplicação. Esse método seria uma melhoria para os desenvolvedores de aplicação; a gerência de aplicações se tornaria mais fácil, menos ajustes seriam necessários e o suporte técnico da aplicação seria mais simples. Os administradores do sistema também achariam o ambiente mais fácil de administrar. A instalação seria mais simples e a mudança da aplicação para outro sistema seria muito mais fácil do que as etapas normais de desinstalação e reinstalação. No entanto, para que haja adoção generalizada dessa metodologia, o formato das máquinas virtuais precisa ser padronizado, de modo que a máquina virtual funcione em qualquer plataforma de virtualização. O “Open Virtual Machine Format” (formato aberto de máquina virtual) é uma tentativa de fazer exatamente isso, com os principais fornecedores concordando em dar suporte a ele; isso poderá ocasionar a unificação dos formatos de máquina virtual.

2.8.2 Implementação

Embora o conceito de máquina virtual seja útil, ele é difícil de implementar. É necessário muito trabalho para oferecer uma duplicata *exata* da máquina utilizada. Lembre-se de que a máquina possui dois modos: o modo usuário e o modo kernel. O software da máquina virtual pode ser executado no modo kernel, pois esse é o sistema operacional. A máquina virtual, em si só, pode ser executada no modo usuário. Todavia, assim como a máquina física possui dois modos, a máquina virtual também precisa ter. Como consequência, precisamos ter um modo usuário virtual e um modo monitor kernel virtual, ambos executando em um modo de usuário físico. As ações que causam uma transferência do modo usuário para o modo kernel em uma máquina real (como uma chamada de sistema ou uma tentativa de executar uma instrução privilegiada) também precisam causar uma transferência do modo usuário virtual para o modo kernel virtual, em uma máquina virtual.

Essa transferência pode ser realizada da seguinte maneira. Quando uma chamada de sistema, por exemplo, é feita por um programa executando em uma máquina virtual no modo de usuário virtual, ela causará uma transferência para o monitor da máquina virtual na máquina real. Quando o monitor da máquina virtual tiver o controle, ele poderá mudar o conteúdo do registrador e do contador de programa para a máquina virtual simular o efeito da chamada de sistema. Ele pode, então, reiniciar a máquina virtual, observando que agora está no modo kernel virtual.

A principal diferença é o tempo. Enquanto uma E/S real poderia ter levado 100 milissegundos, a E/S virtual poderia levar menos tempo (porque está em spool) ou mais tempo (porque é interpretada). Além disso, a CPU está sendo multiprogramada entre muitas máquinas virtuais, atrasando ainda mais as máquinas virtuais de maneiras imprevisíveis. No caso extremo, pode ser necessário simular todas as instruções para fornecer uma máquina virtual verdadeira. O VM funciona para máquinas IBM porque as instruções normais para as máquinas virtuais podem ser executadas diretamente no hardware. Somente as instruções privilegiadas (necessárias principalmente para a E/S) precisam ser simuladas e, portanto, executadas mais lentamente.

Sem esse nível de suporte do hardware, a virtualização seria impossível. Quanto mais suporte do hardware estiver disponível dentro de um sistema, mais ricas em recursos, estáveis e de melhor performance poderão ser as máquinas virtuais. Todas as principais CPUs de uso geral oferecem algum tipo de suporte do hardware para a virtualização. Por exemplo, a tecnologia de virtualização

AMD é encontrada em diversos processadores AMD. Ela define dois novos modos de operação – hospedeiro e guest. O software de máquina virtual pode habilitar o modo hospedeiro (ou *host*), definir as características de cada máquina virtual guest e depois passar o sistema para o modo guest, passando o controle do sistema para o sistema operacional guest que está em execução na máquina virtual. No modo guest, o sistema operacional virtualizado “pensa” que está sendo executado no hardware nativo e vê determinados dispositivos (aqueles incluídos na definição do guest pelo hospedeiro). Se o guest tentar acessar um recurso virtualizado, então o controle é passado para o hospedeiro administrar essa interação.

2.8.3 VMware

Apesar das vantagens das máquinas virtuais, elas receberam pouca atenção por diversos anos após seu desenvolvimento inicial. Hoje, porém, as máquinas virtuais estão voltando a ser usadas como um meio de solucionar problemas de compatibilidade de sistema. Nesta seção, exploramos a máquina virtual VMware Workstation. Conforme veremos neste exemplo, essa máquina virtual pode ser executada em cima de um sistema operacional de qualquer um dos tipos de projeto discutidos anteriormente. Assim, os métodos de projeto do sistema operacional – camadas simples, microkernels, módulos e máquinas virtuais – não são mutuamente excludentes.

A maior parte das técnicas de virtualização discutidas nesta seção exige que a virtualização tenha o suporte do kernel. Outro método envolve escrever a ferramenta de virtualização para rodar no modo usuário, como uma aplicação em cima do sistema operacional. As máquinas virtuais rodando dentro dessa ferramenta acreditam que estão rodando no hardware básico, mas na verdade estão rodando dentro de uma aplicação em nível de usuário.

VMware Workstation é uma aplicação comercial popular, que abstrai o Intel X86 e hardware compatível para máquinas virtuais isoladas. VMware Workstation roda como uma aplicação em um sistema operacional hospedeiro, como Windows ou Linux, e permite que esse sistema execute simultaneamente diversos sistemas operacionais guest diferentes como máquinas virtuais independentes.

A arquitetura desse sistema aparece na [Figura 2.18](#). Nesse cenário, o Linux está sendo executado como sistema operacional hospedeiro; FreeBSD, Windows NT e Windows XP estão sendo executados como sistemas operacionais guest. A camada de virtualização é o núcleo do VMware, pois separa o hardware físico em máquinas virtuais isoladas, rodando como sistemas operacionais guest. Cada máquina virtual possui sua própria CPU virtual, memória, unidades de disco, interfaces de rede, e assim por diante.

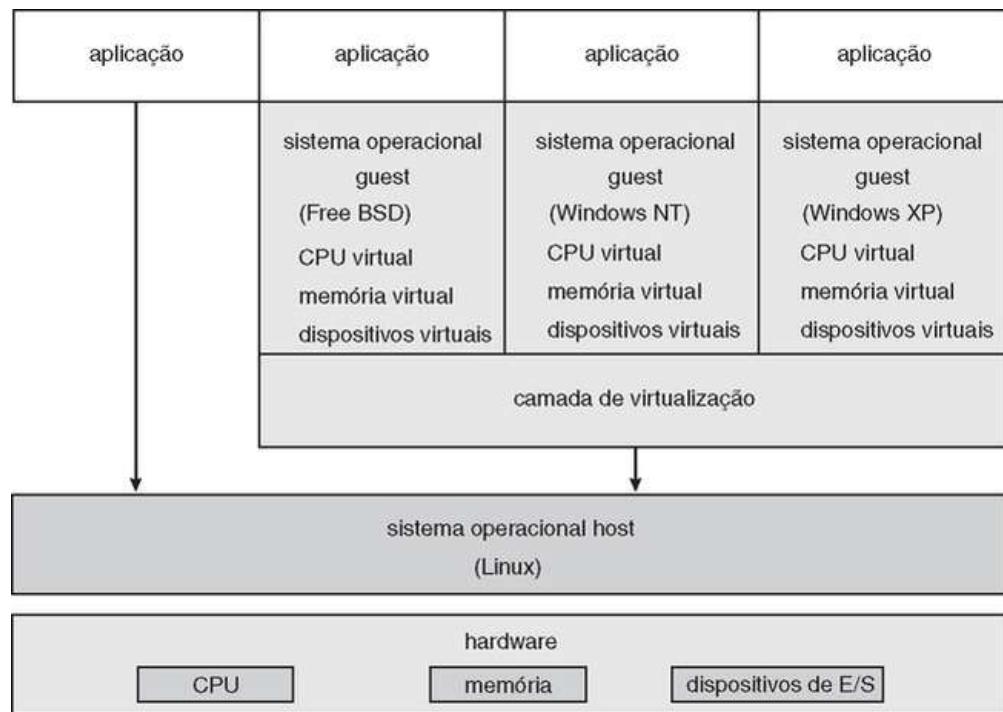


FIGURA 2.18 Arquitetura do VMware.

O disco físico que o guest possui e gerencia, na realidade, é apenas um arquivo dentro do sistema de arquivos do sistema operacional host. Para criar uma instância guest idêntica, podemos copiar o arquivo. A cópia do arquivo para outro local protege a instância guest contra um desastre no local

original. Mover o arquivo para outro local move o sistema guest. Oferecendo essas capacidades, a virtualização pode melhorar a eficiência da administração do sistema, bem como o uso de recursos do sistema.

2.8.4 Alternativas à virtualização de sistemas

A virtualização de sistemas, conforme discutida até aqui, é apenas uma das muitas metodologias de emulação de sistemas. A virtualização é a mais comum, pois faz os sistemas operacionais guest e as aplicações “acreditarem” que estão sendo executados no hardware nativo. Como somente os recursos do sistema precisam ser virtualizados, esses guests são executados quase em velocidade plena. Vamos discutir duas outras opções em seguida: simulação e paravirtualização.

2.8.4.1 Simulação

Outra metodologia é a **simulação**, na qual o sistema hospedeiro possui uma arquitetura de sistema e o sistema guest foi compilado para uma arquitetura diferente. Por exemplo, suponha que uma empresa tenha substituído seu sistema de computação desatualizado por um novo sistema, mas gostaria de continuar a usar certos programas importantes que foram compilados para o sistema antigo. Os programas poderiam ser executados em um emulador que traduz cada uma das instruções do sistema desatualizado para o conjunto de instruções nativo do novo sistema. A simulação pode aumentar a vida dos programas e permitir que exploremos as arquiteturas antigas sem ter uma máquina antiga real, mas seu principal desafio é a performance. A emulação do conjunto de instruções pode ser uma ordem de grandeza mais lenta que as instruções nativas. Assim, a menos que a nova máquina seja dez vezes mais rápida que a antiga, o programa ficará mais lento na nova máquina do que no seu hardware nativo. Outro desafio é que é difícil criar um emulador que funcione corretamente, porque isso envolve essencialmente a escrita de uma CPU inteira no software.

2.8.4.2 Paravirtualização

Paravirtualização é outra variação do tema de emulação do sistema. Em vez de tentar enganar o sistema operacional guest fazendo-o acreditar que possui um sistema para si, a paravirtualização apresenta ao guest um sistema que é semelhante, mas não idêntico ao sistema preferido do guest. O guest precisa ser modificado para funcionar no hardware paravirtualizado. O ganho nesse trabalho extra é o uso mais eficiente dos recursos e uma camada de virtualização menor.

O Solaris 10 contém **contêineres**, ou **zonas**, que criam uma camada virtual entre o sistema operacional e as aplicações. Nesse sistema, apenas um kernel é instalado, e o hardware não é virtualizado. Em vez disso, o sistema operacional e seus dispositivos são virtualizados, dando aos processos dentro de um contêiner a impressão de que são os únicos processos a existir no sistema. Um ou mais contêineres podem ser criados, e cada um pode ter suas próprias aplicações, pilhas de rede, endereço e portas de rede, contas do usuário e assim por diante. Os recursos da CPU podem ser divididos entre os contêineres e os processos no âmbito do sistema. A [Figura 2.19](#) mostra um sistema Solaris 10 com dois contêineres e o espaço de usuário “global” padrão.

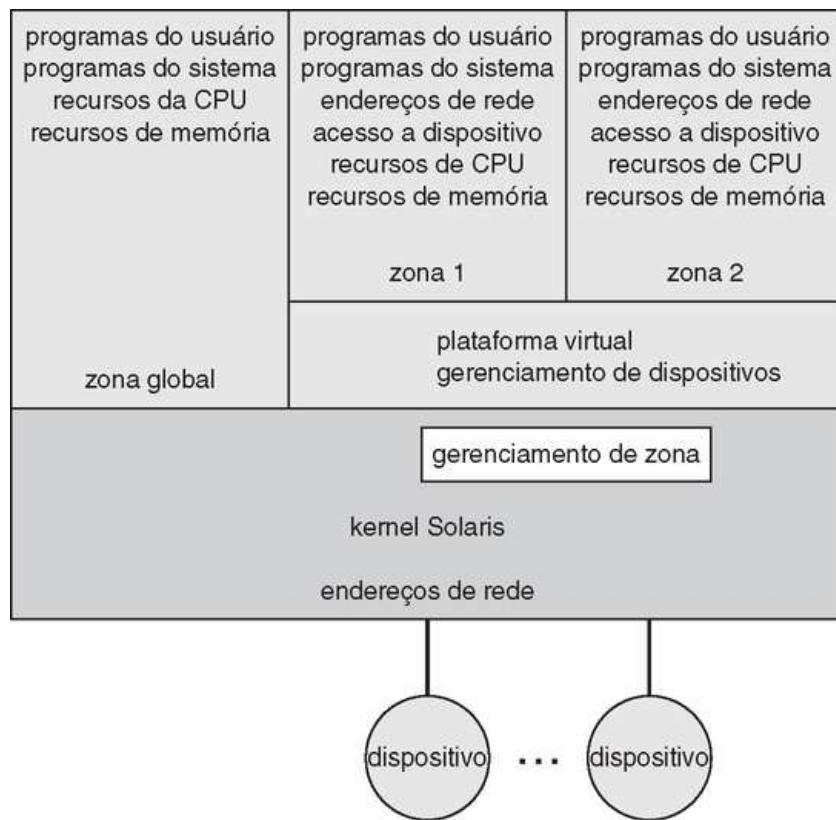


FIGURA 2.19 Solaris 10 com dois contêineres.

2.9 Java

Java é uma tecnologia introduzida pela Sun Microsystems em meados de 1990. Vamos nos referir a ela como uma *tecnologia*, em vez de uma linguagem de programação, porque ela fornece mais do que uma linguagem de programação convencional. A tecnologia Java consiste em dois componentes essenciais:

1. Especificação da linguagem de programação.

2. Especificação de máquina virtual.

Nesta seção, você terá uma visão geral desses dois componentes.

2.9.1 Linguagem de programação Java

Java é uma linguagem de programação de uso geral, orientada a objeto, com suporte para programação distribuída. A Java foi favorecida originalmente pela comunicação de programação da Internet, devido a seu suporte para **applets**, que são programas com acesso limitado aos recursos e executados dentro de um navegador Web. Agora, Java é uma linguagem popular para o projeto de aplicações desktop, aplicações Web cliente-servidor e aplicações que rodam dentro de sistemas embutidos, como smartphones.

Como dissemos, Java é uma linguagem de programação orientada a objeto, o que significa que oferece suporte para o tipo de programação orientada a objeto, discutida anteriormente. Os objetos Java são especificados com a construção **class**; um programa Java consiste em uma ou mais classes. Para cada classe Java, o compilador Java produz um arquivo de saída de **código de bytes** independente da arquitetura (.class), que será executado em qualquer implementação da máquina virtual Java (JVM). A Java também fornece suporte de alto nível para objetos em rede e distribuídos. Essa é uma linguagem multithreaded, significando que um programa Java pode ter várias threads distintas, ou fluxos de controle, permitindo, assim, o desenvolvimento de aplicações concorrentes para tirar proveito de processadores modernos, com múltiplos núcleos de processamento. Abordamos os objetos distribuídos usando a chamada de método remoto (RMI) da Java no [Capítulo 3](#), e discutiremos os programas Java dotados de múltiplas threads no [Capítulo 4](#). A Java é considerada uma linguagem segura. Esse recurso é importante quando se considera que um programa Java pode estar sendo executado por uma rede distribuída. Examinamos a segurança Java no [Capítulo 15](#).

Programas em Java são escritos usando a API Java Standard Edition. Esta é uma API padrão para projetar aplicações desktop e applets com suporte básico da linguagem para gráficos, E/S, segurança, conectividade de banco de dados e redes.

2.9.2 Máquina virtual Java

A máquina virtual Java (JVM) é uma especificação para um computador abstrato. Ela consiste em um **carregador de classes** (class loader) e um interpretador Java que executa os códigos de bytes independentes da arquitetura. O carregador de classes carrega arquivos .class do programa Java e da API Java para execução pelo interpretador Java, conforme mostrado na [Figura 2.20](#). Depois que uma classe é carregada, ela verifica se o arquivo .class é um código de bytes Java válido e não causa um overflow ou underflow da pilha. Ela também garante que o código de bytes não realize aritmética de ponteiro, que poderia oferecer acesso ilegal à memória. Se a classe passar na verificação, ela é executada pelo interpretador Java. A JVM também gerencia a memória automaticamente, realizando **coleta de lixo** (**garbage collection**) - a prática de apanhar a memória utilizada pelos objetos que não estão mais em uso e retorná-la ao sistema. Há muita pesquisa em andamento sobre algoritmos de coleta de lixo para aumentar o desempenho de programas Java na máquina virtual.

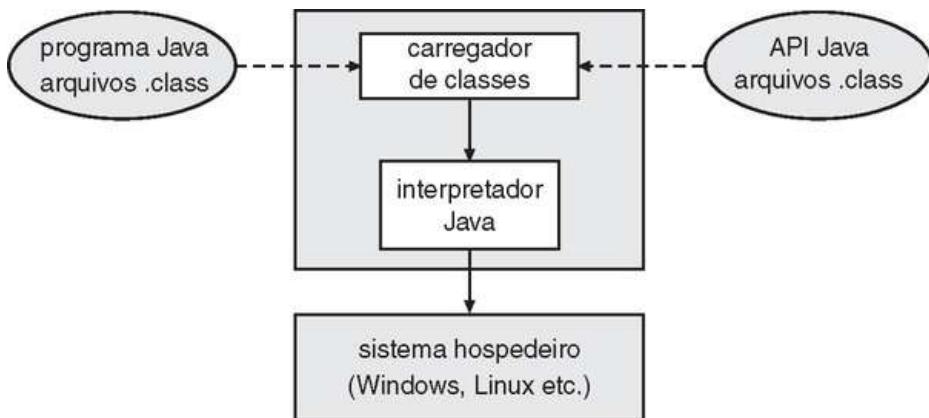


FIGURA 2.20 A máquina virtual Java (JVM).

Uma instância da JVM é criada sempre que uma aplicação Java (ou applet) é executada. Essa instância da JVM começa a ser executada quando o método `main()` de um programa é invocado. Isso também acontece para applets, embora o programador não defina um `main()`. Nesse caso, o navegador executa o método `main()` antes de criar o applet. Se executarmos simultaneamente dois programas Java e um applet Java no mesmo computador, teremos três instâncias da JVM.

A JVM pode ser implementada no software em cima de um sistema operacional hospedeiro, como Windows, Linux ou Mac OS X, ou como parte de um navegador Web. Como alternativa, a JVM pode ser implementada no hardware em um chip projetado especificamente para executar programas Java. Se a JVM for implementada no software, o interpretador Java interpreta as operações do código de bytes um por vez. Uma técnica de software mais rápida é usar o compilador **just-in-time (JIT)**. Aqui, na primeira vez que um método Java é chamado, os códigos de byte para o método são transformados em linguagem de máquina nativa para o sistema hospedeiro. Essas operações são então mantidas em cache, para que chamadas subsequentes de um método sejam realizadas usando as instruções de máquina nativas, e as operações de código de bytes não precisem ser interpretadas novamente. Uma técnica possivelmente mais rápida é executar a JVM no hardware em um chip Java especial, que executa as operações do código de bytes em Java como código nativo, evitando, assim, a necessidade de um interpretador de software ou um compilador just-in-time.

É a plataforma Java que possibilita o desenvolvimento de programas independentes da arquitetura e portáveis. Uma implementação da JVM é específica do sistema, e ela separa o sistema de uma maneira-padrão para o programa Java, provendo uma interface limpa e independente da arquitetura. Essa interface permite que um arquivo `.class` seja executado em qualquer sistema que tenha implementado a JVM de acordo com sua implementação. As máquinas virtuais Java foram projetadas para a maioria dos sistemas operacionais, incluindo Windows, Linux, Mac OS X e Solaris. Quando usamos a JVM neste livro para ilustrar os conceitos do sistema operacional, nos referimos à especificação da JVM, em vez de qualquer implementação em particular.

2.9.3 Ambiente de desenvolvimento Java

O ambiente de desenvolvimento Java (Java Development Kit – JDK) consiste em (1) ferramentas de desenvolvimento, como um compilador e depurador; e (2) um ambiente em tempo de execução (JRE). O compilador transforma um arquivo-fonte Java em um código de bytes (arquivo `.class`). O ambiente em tempo de execução provê a JVM, bem como a API Java para o sistema hospedeiro. O JDK é representado na [Figura 2.21](#).

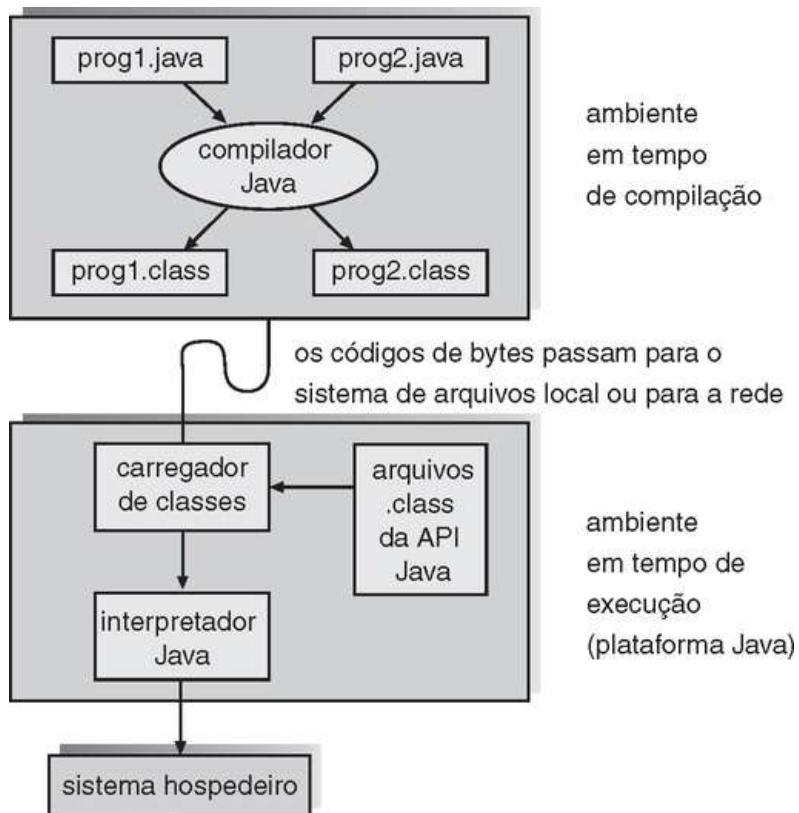


FIGURA 2.21 Ambiente de desenvolvimento Java.

2.9.4 Sistemas operacionais em Java

A maioria dos sistemas operacionais é escrita em uma combinação de C e código em linguagem assembly, principalmente devido aos benefícios de desempenho dessas linguagens e à facilidade de interface com o hardware. No entanto, recentemente foram realizados esforços para escrever sistemas operacionais em Java. Esse tipo de sistema, conhecido como **sistema extensível baseado em linguagem**, é executado em um único espaço de endereços.

Uma das dificuldades no projeto de sistemas baseados em linguagem reside na proteção de memória - proteger o sistema operacional contra programas do usuário maliciosos e também proteger os programas do usuário uns dos outros. Os sistemas operacionais tradicionais contam com os recursos do hardware para fornecer proteção à memória ([Seção 8.1](#)). Os sistemas baseados em linguagem, em vez disso, contam com recursos de segurança de tipo da linguagem para implementar proteção à memória. Como resultado, os sistemas baseados em linguagem são desejáveis em dispositivos de hardware pequenos, que podem não ter recursos de hardware que ofereçam proteção da memória.

O sistema operacional JX foi escrito quase totalmente em Java e também fornece um sistema em tempo de execução para aplicações Java. O JX organiza seu sistema de acordo com **domínios**. Cada domínio representa um JVM independente. Além do mais, cada domínio mantém um heap usado para alocar memória durante a criação de objeto e threads dentro de si mesmo, além da coleta de lixo. O domínio zero é um microkernel ([Seção 2.7.3](#)) responsável pelos detalhes de baixo nível, como inicialização do sistema e salvamento e restauração do estado da CPU. O domínio zero foi escrito em C e linguagem assembly; todos os outros domínios são escritos totalmente em Java. A comunicação entre os domínios ocorre por meio de **portais**, mecanismos de comunicação semelhantes às remote procedure calls (RPCs) usadas pelo microkernel Mach. A proteção dentro e entre os domínios conta com a segurança de tipo da linguagem Java para a sua segurança. Como o domínio zero não é escrito em Java, ele precisa ser considerado **confiável**. A arquitetura do sistema JX é ilustrada na [Figura 2.22](#).

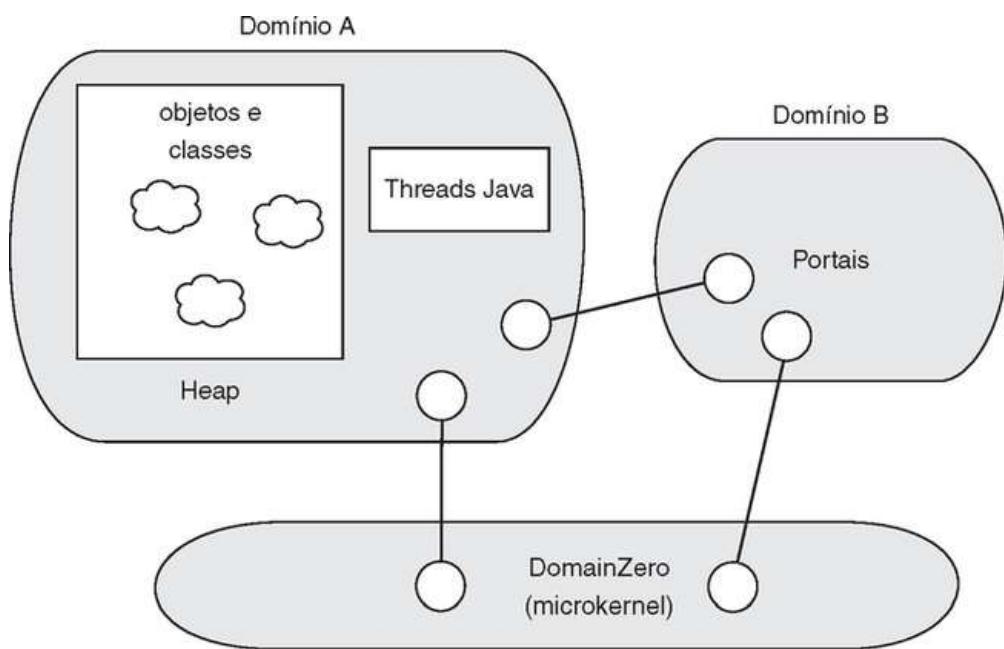


FIGURA 2.22 O sistema operacional JX.

2.10 Depuração do sistema operacional

Em geral, **depuração** é a atividade de localizar e reparar erros, ou **bugs**, em um sistema. A depuração busca localizar e reparar erros tanto no hardware quanto no software. Problemas de performance são considerados bugs, de modo que a depuração também pode incluir **ajuste de performance**, que melhora a performance removendo **gargalos** no processamento que ocorre dentro de um sistema. Uma discussão sobre depuração do hardware está fora do escopo deste livro. Nesta seção, vamos explorar a depuração de erros de kernel e de processo, além de problemas de performance.

2.10.1 Análise de falhas

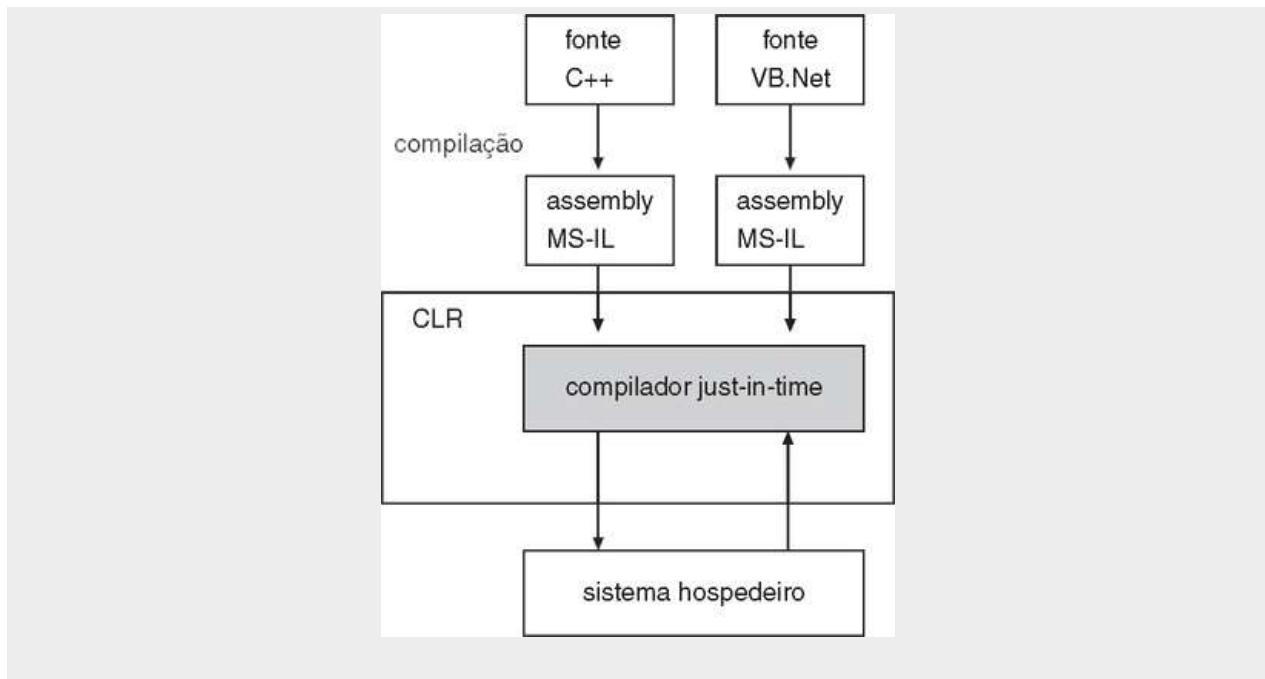
Se um processo falha, a maioria dos sistemas operacionais grava a informação do erro em um **arquivo de log** para alertar os operadores ou usuários do sistema de que o problema ocorreu. O sistema operacional também pode realizar um **core dump** - uma captura da memória (conhecida como “core” nos primeiros dias da computação) do processo. Essa imagem da memória é armazenada em um arquivo para análise posterior. A execução de programas e core dumps pode ser sondada por um **depurador**, uma ferramenta projetada para permitir que um programador explore o código e a memória de um processo.

A depuração do código de processo em nível de usuário é um desafio. A depuração do kernel do sistema operacional é ainda mais desafiadora, devido ao tamanho e à complexidade do kernel, seu controle do hardware e a falta de ferramentas de depuração em nível de usuário. Uma falha no kernel é chamada de **crash**. Assim como uma falha de processo, a informação do erro é salva em um arquivo de log, e o estado da memória é salvo em um **crash dump**.

.NET FRAMEWORK

.NET Framework é uma coleção de tecnologias, incluindo um conjunto de bibliotecas de classe e um ambiente de execução, que se junta para oferecer uma plataforma para desenvolvimento de software. Essa plataforma permite que os programas sejam escritos na .NET Framework, em vez de uma arquitetura específica. Um programa escrito para a .NET Framework não precisa se preocupar com os detalhes do hardware ou do sistema operacional no qual será executado. Assim, qualquer arquitetura implementando .NET será capaz de executar o programa com sucesso. Isso porque o ambiente de execução abstrai esses detalhes e provê uma máquina virtual como um intermediário entre o programa em execução e a arquitetura subjacente.

No núcleo do .NET Framework está o Common Language Runtime (CLR). CLR é a implementação da máquina virtual .NET. Ele oferece um ambiente para a execução de programas escritos em qualquer uma das linguagens visadas no .NET Framework. Os programas escritos em linguagens como C# (pronunciado *C-sharp*) e VB.NET são compilados para uma linguagem intermediária, independente de arquivo, denominada Microsoft Intermediate Language (MS-IL). Esses arquivos compilados, chamados *assemblies*, incluem instruções MS-IL e metadados. Eles possuem extensões de arquivo .EXE ou .DLL. Na execução de um programa, o CLR carrega assemblies no que é conhecido como **domínio de aplicação**. À medida que as instruções são solicitadas pelo programa de aplicação, o CLR converte as instruções MS-IL dentro dos assemblies para código nativo específico à arquitetura subjacente, usando a compilação just-in-time. Quando as instruções tiverem sido convertidas para código nativo, elas são mantidas e continuarão a ser executadas como código nativo para a CPU. A arquitetura do CLR para o .NET Framework pode ser vista a seguir.



A depuração do sistema operacional frequentemente usa ferramentas e técnicas diferentes da depuração de processos, devido à natureza muito diferente dessas duas tarefas. Imagine que uma falha no kernel no código do sistema de arquivos tornaria arriscado para o kernel tentar salvar seu estado em um arquivo do sistema de arquivos antes de reinicializar. Uma técnica comum é salvar o estado de memória do kernel em uma seção do disco separada para esse propósito, que não contém o sistema de arquivos. Se o kernel detectar um erro irrecuperável, ele gravará o conteúdo inteiro da memória, ou pelo menos as partes da memória do sistema pertencentes ao kernel, na área do disco. Quando o sistema for reinicializado, um processo será executado para colher os dados dessa área e os gravará em um arquivo de crash dump dentro de um sistema de arquivos, para serem analisados.

LEI DE KERNIGHAN

"A depuração tem o dobro da dificuldade da escrita do código em primeiro lugar. Portanto, se você escrever o código da forma mais inteligente possível, por definição, você não é inteligente o bastante para depurá-lo."

2.10.2 Ajuste de performance

Para identificar gargalos, precisamos conseguir monitorar a performance do sistema. É preciso acrescentar um código para calcular e exibir métricas de comportamento do sistema. Em diversos sistemas, o sistema operacional realiza essa tarefa produzindo listagens de rastreio do comportamento do sistema. Todos os eventos interessantes são registrados em log, com seu horário e parâmetros importantes, e são gravados em um arquivo. Mais tarde, um programa de análise pode processar o arquivo de log para determinar a performance do sistema e identificar gargalos e ineficiências. Esses mesmos rastreios podem ser executados como entrada para a simulação de uma melhoria sugerida para o sistema. Rastreios também podem ajudar as pessoas a localizar erros no comportamento do sistema operacional.

Outra técnica para o ajuste de performance é incluir ferramentas interativas com o sistema que permitem que usuários e administradores questionem o estado de diversos componentes do sistema, para procurar gargalos existentes. O comando `top` do UNIX apresenta os recursos usados no sistema, bem como uma lista classificada dos "maiores" processos consumidores de recursos. Outras ferramentas apresentam o estado da E/S de disco, alocação de memória e tráfego de rede. Os autores dessas ferramentas de único propósito tentam descobrir o que um usuário gostaria de ver enquanto analisa um sistema e prover essa informação.

Uma área de pesquisa e implementação ativa em sistemas operacionais é a preparação de sistemas operacionais que sejam fáceis de entender, depurar e ajustar. O ciclo de permitir o rastreio à medida que ocorrem problemas no sistema e analisar os rastreios posteriormente está sendo quebrado por uma nova geração de ferramentas de análise de performance habilitadas pelo kernel. Além do mais, essas ferramentas não estão limitadas a uma única finalidade ou a seções de código que foram escritas para produzir dados de depuração. A facilidade de tracing dinâmico do Solaris 10 DTrace é o principal exemplo de uma ferramenta desse tipo.

2.10.3 DTrace

DTrace é uma facilidade que acrescenta sondagens dinamicamente a um sistema em execução, tanto em processos do usuário quanto no kernel. Essas sondagens podem ser consultadas por meio da linguagem de programação D para determinar muita coisa sobre o kernel, o estado do sistema e as atividades do processo. Por exemplo, a [Figura 2.23](#) acompanha uma aplicação enquanto ela executa uma chamada do sistema (ioctl) e mostra ainda as chamadas de função dentro do kernel enquanto executam a chamada do sistema. As linhas que terminam com “U” são executadas no modo usuário, e as linhas que terminam com “K” no modo kernel.

```
# ./all.d 'pgrep xclock' XEventsQueued
dtrace: script './all.d' matched 52377 probes
CPU FUNCTION
  0 -> XEventsQueued                                U
  0  -> _XEventsQueued                             U
  0  -> _X11TransBytesReadable                      U
  0  <- _X11TransBytesReadable                      U
  0  -> _X11TransSocketBytesReadable                U
  0  <- _X11TransSocketBytesReadable                U
  0  -> ioctl                                      U
  0    -> ioctl                                    K
  0      -> getf                                     K
  0        -> set_active_fd                         K
  0          <- set_active_fd                        K
  0          <- getf                                 K
  0          -> get_udatamodel                     K
  0          <- get_udatamodel                     K
  ...
  0          -> releaseef                           K
  0            -> clear_active_fd                   K
  0              <- clear_active_fd                 K
  0              -> cv_broadcast                     K
  0              <- cv_broadcast                   K
  0              <- releaseef                     K
  0      <- ioctl                               K
  0      <- ioctl                               U
  0  <- _XEventsQueued                          U
  0 <- XEventsQueued                           U
```

FIGURA 2.23 DTrace no Solaris 10 acompanha uma chamada do sistema dentro do kernel.

A depuração das interações entre o código no nível usuário e no nível kernel é quase impossível sem um conjunto de ferramentas que compreenda os dois conjuntos de código e possa instrumentar as interações. Para que esse conjunto de ferramentas seja verdadeiramente útil, ele precisa ser capaz de depurar qualquer área de um sistema, incluindo as áreas que não foram escritas visando à depuração, e isso sem afetar a confiabilidade do sistema. O conjunto de ferramentas também precisa ter um impacto mínimo na performance – o ideal é que não tenha impacto quando não estiver em uso e um impacto proporcional durante o uso. A ferramenta DTrace atende a esses requisitos e oferece um ambiente de depuração dinâmico, seguro e de baixo impacto.

Antes que o framework e as ferramentas de DTrace estivessem disponíveis com o Solaris 10, a depuração do kernel normalmente era coberta de mistério e realizada por meio do acaso e com código e ferramentas arcaicas. Por exemplo, as CPUs possuem um recurso de ponto de interrupção (breakpoint) que interromperá a execução e permitirá que um depurador examine o estado do sistema. Depois, a execução pode continuar até o próximo ponto de interrupção ou de término. Esse método não pode ser usado no kernel de um sistema operacional multiusuário sem afetar negativamente todos os usuários no sistema. O **profiling**, que tira amostras periódicas do ponteiro de instrução para determinar qual código está sendo executado, pode mostrar tendências estatísticas, mas não atividades individuais. Pode-se incluir um código no kernel para emitir dados específicos sob circunstâncias específicas, mas esse código atrasa a execução do kernel e costuma não ser incluído na parte do kernel onde o problema específico está sendo depurado.

Ao contrário, DTrace é executado em sistemas de produção – sistemas que estão executando aplicações importantes ou críticas – e não causa prejuízos ao sistema. Ele atrasa as atividades quando estiver habilitado, porém, depois da execução, ele retorna o sistema ao seu estado anterior à

depuração. Essa também é uma ferramenta ampla e profunda. Ela pode depurar amplamente tudo o que está acontecendo no sistema (tanto nos níveis de usuário e de kernel quanto entre as camadas de usuário e de kernel). DTrace também pode se aprofundar no código, mostrando instruções de CPU individuais ou atividades de sub-rotina do kernel.

A ferramenta **DTrace** é composta de um compilador, um framework, **provedores** de **sondas** escritos dentro desse framework e **consumidores** dessas sondas. Provedores de DTrace criam sondas. Estruturas do kernel registram todas as sondas que os provedores criaram. As sondas são armazenadas em uma estrutura de dados de tabela hash que é dividida por nome e indexada de acordo com identificadores de sonda exclusivos. Quando uma sonda é habilitada, um bit de código na área a ser sondada é reescrito para chamar `dtrace probe(probe identifier)` e depois continuar com a operação original do código. Diferentes provedores criam diferentes tipos de sondas. Por exemplo, uma sonda de chamada do sistema do kernel funciona de forma diferente de uma sonda do processo do usuário, e isso é diferente de uma sonda de E/S.

DTrace possui um compilador que gera um código de bytes executado no kernel. Esse código tem a garantia do compilador de ser “seguro”. Por exemplo, nenhum loop é permitido, e apenas modificações específicas do estado do kernel podem ser feitas. Somente os usuários com os “privilegios” do DTrace (ou usuários “raiz”) têm permissão para usar o DTrace, pois ele pode recuperar dados privados do kernel (e modificar os dados, se for solicitado). O código gerado roda no kernel e habilita as sondas. Ele também habilita os consumidores no modo usuário e habilita as comunicações entre os dois.

Um consumidor DTrace é o código que está interessado em uma sondagem e em seus resultados. Um consumidor solicita que o provedor crie uma ou mais sondas. Quando a sonda é disparada, ela emite dados que são gerenciados pelo kernel. Dentro do kernel, ações chamadas de **blocos de controle habilitadores**, ou **ECBs** (Enabling Control Blocks), são realizadas quando as sondas são disparadas. Uma sonda pode fazer vários ECBs serem executados se mais de um consumidor estiver interessado nessa sonda. Cada ECB contém um predicado (“instrução if”) que pode filtrar esse ECB.

Caso contrário, a lista de ações no ECB é executada. A ação mais comum é capturar algum bit de dados, como o valor de uma variável nesse ponto da execução da sonda. Reunindo tais dados, o DTrace pode montar uma imagem completa da ação de um usuário ou do kernel. Além do mais, as sondas disparando a partir do espaço do usuário e do kernel podem mostrar como uma ação em nível de usuário causou reações em nível do kernel. Esses dados são muito valiosos para o monitoramento de performance e a otimização do código.

Quando o consumidor da sonda termina, seus ECBs são removidos. Se não houver ECBs consumindo uma sonda, a sonda é removida. Isso envolve a reescrita do código para remover a chamada `dtrace_probe` e colocar de volta o código original. Assim, depois que uma sonda é destruída, o sistema é exatamente o mesmo que antes de a sonda ser criada, como se nenhuma sondagem tivesse ocorrido.

O DTrace não permite que as sondas usem muita memória ou capacidade da CPU, o que poderia prejudicar o sistema em execução. Os buffers usados para manter os resultados da sonda são monitorados em busca de limites default e máximo em excesso. O tempo de CPU para a execução da sonda também é monitorado. Se os limites forem ultrapassados, o consumidor é terminado, juntamente com as sondas responsáveis. Os buffers são alocados por CPU para evitar a disputa e a perda de dados.

Um exemplo de código D e sua saída mostra parte de sua utilidade. O programa na [Figura 2.24](#) mostra o código DTrace para permitir que o escalonador sonde e registre a quantidade de tempo de CPU de cada processo rodando com a ID de usuário 101 enquanto essas sondas estão habilitadas (ou seja, enquanto o programa é executado). A saída do programa, mostrando os processos e quanto tempo (em nanosegundos) eles gastam rodando nas CPUs, aparece na [Figura 2.25](#).

```

        sched:::on-cpu
        uid == 101
        {
            self->ts = timestamp;
        }

        sched:::off-cpu
        self->ts
        {
            @time[execname] = sum(timestamp - self->ts);
            self->ts = 0;
        }
    }

```

FIGURA 2.24 Código de DTrace.

```

#dtrace -s sched.d
dtrace: script 'sched.d' matched 6 probes
^C
      gnome-settings-d          142354
      gnome-vfs-daemon          158243
      dsdm                      189804
      wnck-applet                200030
      gnome-panel                 277864
      clock-applet                374916
      mapping-daemon              385475
      xscreensaver                514177
      metacity                     539281
      Xorg                         2579646
      gnome-terminal                5007269
      mixer_applet2                7388447
      java                         10769137

```

FIGURA 2.25 Saída do código D.

Como o DTrace faz parte do sistema operacional de código aberto Solaris 10, ele está sendo acrescentado aos sistemas operacionais quando esses sistemas não possuem acordos de licença em conflito. Por exemplo, DTrace foi acrescentado ao Mac OS X 10.5 e ao FreeBSD e provavelmente será ainda mais divulgado devido às suas capacidades exclusivas. Outros sistemas operacionais, especialmente os derivados do Linux, também estão acrescentando funcionalidade de rastreamento do kernel. Ainda outros sistemas operacionais estão começando a incluir diversas ferramentas de performance e rastreio, promovidas pela pesquisa em diversas instituições, incluindo o projeto Paradyn (<http://www.cs.wisc.edu/paradyn/>).

2.11 Geração do sistema operacional

É possível projetar, codificar e implementar um sistema operacional especificamente para uma máquina em um local. Entretanto, é mais comum que os sistemas operacionais sejam projetados para execução em qualquer uma dentre uma classe de máquinas, em uma grande variedade de locais e com muitas configurações de periféricos. O sistema precisa ser configurado ou gerado para cada instalação de computador específica, um processo às vezes conhecido como **geração do sistema (SYSGEN)**.

O sistema operacional é distribuído em disco ou CD-ROM ou DVD-ROM. Para gerar um sistema, usamos um programa especial. O programa SYSGEN lê de determinado arquivo ou pede ao operador do sistema informações referentes à configuração específica do sistema de hardware ou então sonda o hardware diretamente para determinar quais os componentes que existem. Os tipos de informação a seguir precisam ser determinados.

- Qual CPU deve ser utilizada? Que opções (conjuntos de instruções estendidas, aritmética de ponto flutuante, e assim por diante) estão instaladas? Para sistemas com múltiplas CPUs, cada CPU precisa ser descrita.
- Como o disco será formatado? Haverá quantas seções ou “partições” e o que vai entrar em cada partição?
- Quanta memória está disponível? Alguns sistemas determinarão esse valor por si mesmos, fazendo referência a um local de memória após o outro até que seja gerada uma falha de “endereço ilegal”. Esse procedimento define o último endereço válido e, portanto, a quantidade de memória disponível.
- Que dispositivos estão disponíveis? O sistema precisará saber como endereçar cada dispositivo (o número do dispositivo), o número da interrupção do dispositivo, o tipo e modelo do dispositivo e quaisquer características especiais do dispositivo.
- Que opções do sistema operacional são desejadas ou que valores de parâmetro devem ser usados? Essas opções ou valores poderiam incluir quantos buffers e de quais tamanhos devem ser usados, que tipo de algoritmo de escalonamento de CPU é desejado, que número máximo de processos deve ser admitido, e assim por diante.

Quando essas informações são determinadas, elas podem ser usadas de várias maneiras. Em um extremo, um administrador do sistema pode usá-las para modificar uma cópia do código-fonte do sistema operacional. O sistema operacional, em seguida, é totalmente compilado. Declarações de dados, inicializações e constantes, juntamente com a compilação condicional, produzem uma versão objeto de saída do sistema operacional, adaptada para o sistema descrito.

Em um nível ligeiramente menos personalizado, a descrição do sistema pode ocasionar a criação de tabelas e a seleção de módulos de uma biblioteca pré-compilada. Esses módulos estão interligados para formar o sistema operacional gerado. A seleção permite que a biblioteca contenha os drivers de dispositivo para todos os dispositivos de E/S aceitos, mas somente aqueles necessários são ligados ao sistema operacional. Como o sistema não é recompilado, a geração do sistema é mais rápida, mas o sistema resultante pode ser bastante genérico.

No outro extremo, é possível construir um sistema controlado por tabela. Todo o código sempre faz parte do sistema, e a seleção ocorre durante a execução, e não no momento da compilação ou do linker. A geração do sistema envolve a criação das tabelas apropriadas para descrever o sistema.

As principais diferenças entre essas técnicas são o tamanho e a generalidade do sistema gerado, e a facilidade de modificação à medida que a configuração de hardware muda. Considere o custo de modificar o sistema para dar suporte a um terminal gráfico recém-adquirido ou outra unidade de disco. Naturalmente, lado a lado com esse custo, está a frequência (ou falta de frequência) de tais mudanças.

2.12 Boot do sistema

Após um sistema operacional ser gerado, ele precisa estar disponível para uso pelo hardware. Mas como o hardware sabe onde o kernel se encontra ou como carregar esse kernel? O procedimento para iniciar um computador carregando o kernel é conhecido como **boot do sistema**. Na maioria dos sistemas computadorizados, existe uma pequena parte do código conhecida como **programa de boot** ou **bootstrap loader** que localiza o kernel, carrega-o para a memória principal e inicia sua execução. Alguns sistemas computadorizados, como PCs, utilizam um processo em duas etapas, em que um bootstrap loader simples apanha um programa de boot mais complexo no disco, que por sua vez carrega o kernel.

Quando uma CPU recebe um evento de reset – por exemplo, quando recebe energia ou quando é reinicializada –, o registrador de instrução é carregado com um endereço de memória predefinido, e a execução começa de lá. Nesse local está o programa de boot. Esse programa está na forma de **memória somente de leitura (ROM)**, pois a RAM está em um estado desconhecido no boot do sistema. A ROM é conveniente porque não precisa de inicialização e não pode ser infectada por um vírus de computador.

Esse programa de boot pode realizar diversas tarefas. Normalmente, uma delas é executar diagnósticos que determinam o estado da máquina. Se os diagnósticos passarem, o programa pode continuar com as outras etapas do boot. Ele também pode inicializar todos os aspectos do sistema, desde os registradores da CPU até controladores de dispositivos e o conteúdo da memória principal. Mais cedo ou mais tarde, ele inicia o sistema operacional.

Alguns sistemas – como telefones celulares, PDAs e consoles de jogos – armazenam o sistema operacional inteiro na ROM. O armazenamento do sistema operacional na ROM é adequado para sistemas operacionais pequenos, hardware de suporte simples e operação reforçada. Um problema com essa técnica é que a mudança do código de boot requer a mudança dos chips de hardware da ROM. Alguns sistemas resolvem esse problema usando a **memória somente de leitura apagável programaticamente (EPROM)**, que é somente de leitura, exceto quando recebe explicitamente um comando para se tornar regravável. Todas as formas de ROM também são conhecidas como **firmware**, pois suas características estão em um ponto intermediário entre o hardware e o software. Um problema com o firmware em geral é que a execução do código é mais lenta do que a execução do código na RAM. Alguns sistemas armazenam o sistema operacional em firmware e o copiam para a RAM, para agilizar a execução. Um aspecto final do firmware é que ele é relativamente caro, de modo que apenas pequenas quantidades estão disponíveis.

Para sistemas operacionais grandes (incluindo sistemas operacionais mais genéricos, como Windows, Mac OS X e UNIX) ou para sistemas que mudam com frequência, o bootstrap loader é armazenado em firmware e o sistema operacional está no disco. Nesse caso, o código de boot executa rotinas de diagnóstico e possui um pequeno código que pode ler um único bloco em um local fixo (digamos, o bloco zero) do disco para a memória, executando o código a partir desse **bloco de boot**. O programa armazenado no bloco de boot pode ser sofisticado o bastante para carregar o sistema operacional inteiro para a memória e iniciar sua execução. Mais tipicamente, ele é um código simples (pois cabe em um único bloco do disco) e só conhece o endereço no disco e o tamanho do restante do programa de boot. Todo o boot ligado ao disco e o próprio sistema operacional podem ser facilmente alterados pela escrita de novas versões no disco. Um disco que possui uma partição de boot (veja mais sobre isso na [Seção 12.5.1](#)) é chamado de **disco de boot** ou **disco do sistema**.

Agora que o programa de boot completo foi carregado, ele pode atravessar o sistema de arquivos para encontrar o kernel do sistema operacional, carregá-lo para a memória e iniciar sua execução. É somente nesse ponto que podemos dizer que o sistema está **rodando** (sendo executado).

2.13 Resumo

Os sistemas operacionais fornecem diversos serviços. No nível mais baixo, as chamadas de sistema permitem que um programa em execução faça requisições diretamente do sistema operacional. Em um nível mais alto, o interpretador de comandos ou shell provê um mecanismo para o usuário emitir uma requisição sem escrever um programa. Os comandos podem vir de arquivos durante a execução no modo de lote ou diretamente por um terminal, quando em um modo interativo ou de tempo compartilhado. Os programas do sistema são fornecidos para satisfazer muitas requisições comuns do usuário.

Os tipos de requisições variam de acordo com o nível da requisição. O nível de chamada de sistema precisa prover as funções básicas, como controle de processos e manipulação de arquivos e dispositivos. As requisições de nível mais alto, satisfeitas pelo interpretador de comandos ou pelos programas do sistema, são traduzidas para uma sequência de chamadas de sistema. Os serviços do sistema podem ser classificados em diversas categorias: controle de programa, requisições de status e requisições de E/S. Os erros do programa podem ser considerados requisições de serviço implícitas.

Quando os serviços do sistema são definidos, a estrutura do sistema operacional pode ser desenvolvida. Diversas tabelas são necessárias para registrar as informações que definem o estado do sistema computadorizado e o status das tarefas do sistema.

O projeto de um sistema operacional novo é uma grande tarefa. É importante que os objetivos do sistema sejam bem definidos antes que o projeto seja iniciado. O tipo do sistema desejado é o alicerce para as escolhas entre diversos algoritmos e estratégias necessárias.

Como um sistema operacional é grande, a modularidade é importante. Projetar um sistema como uma sequência de camadas ou usar um microkernel é considerado uma boa técnica. O conceito de máquina virtual usa a técnica em camadas e trata o kernel do sistema operacional e o hardware como se tudo fosse hardware. Até mesmo outros sistemas operacionais podem ser carregados em cima dessa máquina virtual.

No decorrer de todo o ciclo de projeto do sistema operacional, temos de ter o cuidado de separar as decisões de política dos detalhes da implementação (mecanismos). Essa separação permite o máximo de flexibilidade se decisões de política tiverem de ser alteradas mais tarde.

Os sistemas operacionais agora são quase sempre escritos em uma linguagem de implementação de sistemas ou em uma linguagem de alto nível. Esse recurso melhora sua implementação, manutenção e portabilidade. Para criar um sistema operacional para determinada configuração de máquina, temos de realizar a geração do sistema.

Para um sistema computadorizado iniciar sua execução, a CPU precisa inicializar e começar a executar o programa de boot no firmware. O bootstrap pode executar o sistema operacional se o sistema operacional também estiver no firmware ou então pode completar uma sequência em que carrega programas progressivamente mais inteligentes a partir do firmware e disco, até que o próprio sistema operacional esteja carregado na memória e em execução.

Exercícios práticos

- 2.1. Qual é o propósito das chamadas do sistema?
- 2.2. Quais são as cinco principais atividades de um sistema operacional em relação ao gerenciamento de processos?
- 2.3. Quais são as três principais atividades de um sistema operacional em relação ao gerenciamento de memória?
- 2.4. Quais são as três principais atividades de um sistema operacional em relação ao gerenciamento de armazenamento secundário?
- 2.5. Qual é a finalidade do interpretador de comandos? Por que, normalmente, ele é separado do kernel?
- 2.6. Quais chamadas do sistema precisam ser executadas por um interpretador de comandos ou shell a fim de iniciar um novo processo?
- 2.7. Qual é a finalidade dos programas do sistema?
- 2.8. Qual é a principal vantagem da técnica de camadas para o projeto do sistema? Quais são as desvantagens do uso da técnica de camadas?
- 2.9. Relacione cinco serviços fornecidos por um sistema operacional e explique como cada um cria conveniência para os usuários. Em que casos seria impossível que os programas no nível do usuário provessem esses serviços? Explique sua resposta.
- 2.10. Por que alguns sistemas armazenam o sistema operacional no firmware, enquanto outros o armazenam no disco?
- 2.11. Como um sistema poderia ser projetado para permitir uma escolha de sistemas operacionais para o boot do sistema? O que o programa de boot precisaria fazer?

Exercícios

- 2.12. Os serviços e funções fornecidos por um sistema operacional podem ser divididos em duas categorias principais. Descreva rapidamente as duas categorias e discuta como elas diferem.
- 2.13. Descreva três métodos gerais para passar parâmetros para o sistema operacional.
- 2.14. Descreva como você poderia obter um perfil estatístico da quantidade de tempo que um programa gasta executando diferentes seções de seu código. Discuta a importância desse perfil estatístico.
- 2.15. Quais são as cinco principais atividades de um sistema operacional em relação à gerência de processos?
- 2.16. Quais são as vantagens e desvantagens de usar a mesma interface de chamada de sistema para manipular arquivos e dispositivos?
- 2.17. Qual é a finalidade do interpretador de comandos? Por que ele normalmente é separado do kernel? Seria possível para o usuário desenvolver um novo interpretador de comandos usando a interface de chamada de sistema fornecida pelo sistema operacional?
- 2.18. Quais são os dois modelos de comunicação entre processos? Quais são os pontos fortes e fracos das duas técnicas?
- 2.19. Por que a separação do mecanismo e da política é desejável?
- 2.20. Às vezes, é difícil conseguir uma técnica em camadas se dois componentes do sistema operacional forem dependentes um do outro. Identifique um cenário em que não é claro como colocar dois componentes do sistema em camadas que exigem o acoplamento de suas funcionalidades.
- 2.21. Qual é a principal vantagem da técnica de microkernel para o projeto de sistema? Como os programas do usuário e os serviços do sistema interagem em uma arquitetura de microkernel? Quais são as desvantagens de usar a técnica de microkernel?
- 2.22. De que maneiras uma técnica de kernel modular é semelhante à técnica em camadas? De que maneiras ela difere da técnica em camadas?
- 2.23. Qual é a principal vantagem para um projetista de sistema operacional usar uma arquitetura de máquina virtual? Qual é a principal vantagem para um usuário?
- 2.24. Por que um compilador just-in-time é útil para a execução de programas Java?
- 2.25. Qual é o relacionamento entre um sistema operacional guest e um sistema operacional hospedeiro em um sistema como o VMware? Que fatores precisam ser considerados na escolha do sistema operacional hospedeiro?
- 2.26. O sistema operacional experimental Synthesis possui um montador (assembler) incorporado dentro do kernel. Para otimizar o desempenho da chamada de sistema, o kernel monta rotinas dentro do seu espaço para diminuir o caminho que a chamada de sistema precisa percorrer no kernel. Essa técnica é a antítese da técnica em camadas, na qual o caminho pelo kernel é estendido para facilitar a montagem do sistema operacional. Discuta os prós e os contras da técnica do Synthesis para o projeto do kernel e a otimização do desempenho do sistema.

Problemas de programação

2.27. Na [Seção 2.3](#), descrevemos um programa que copia o conteúdo de um arquivo para um arquivo de destino. Esse programa funciona primeiro pedindo do usuário o nome dos arquivos de origem e destino. Escreva esse programa em Java. Não se esqueça de incluir toda a verificação de erro necessária, inclusive verificando se o arquivo de origem existe.

Quando tiver projetado e testado corretamente o programa, se usou um sistema que admite isso, execute o programa usando um utilitário que rastreie chamadas de sistema. Embora o Java não tenha suporte para fazer chamadas do sistema diretamente, será possível observar quais métodos Java correspondem a chamadas do sistema reais. Alguns sistemas Linux fornecem o utilitário `strace`. Em sistemas Mac OS X e Solaris, use `dtrace`. Como os sistemas Windows não oferecem esses recursos, você terá que rastrear por meio da versão Win32 desse programa, usando um depurador.

Projetos de programação

Incluindo uma chamada de sistema no kernel do Linux

Neste projeto, você estudará a interface de chamada de sistema fornecida pelo sistema operacional Linux e aprenderá como os programas do usuário se comunicam com o kernel do sistema operacional por meio dessa interface. Sua tarefa é incorporar uma nova chamada de sistema no kernel, expandindo, assim, a funcionalidade do sistema operacional.

Parte 1: Iniciando

Uma chamada de procedimento no modo usuário é realizada passando argumentos para o procedimento chamado, seja na pilha ou por meio dos registradores, salvando o estado atual e o valor do contador de programa, e saltando para o início do código correspondente ao procedimento chamado. O processo continua a ter os mesmos privilégios de antes.

As chamadas de sistema aparecem como chamadas de procedimento aos programas do usuário, mas resultam em uma mudança no contexto de execução e privilégios. No Linux na arquitetura Intel 386, uma chamada de sistema é realizada armazenando o número da chamada de sistema no registrador EAX, armazenando argumentos para a chamada de sistema em outros registradores de hardware e executando uma instrução de trap (que é a instrução assembly INT 0x80). Depois que o trap é executado, o número da chamada de sistema é usado para indexar em uma tabela de ponteiros de código, para obter o endereço inicial para o código manipulador implementando a chamada de sistema. O processo, então, salta para esse endereço, e os privilégios do processo são passados do modo usuário para kernel. Com os privilégios expandidos, o processo agora pode executar o código do kernel, que pode incluir instruções privilegiadas, que não podem ser executadas no modo usuário. O código do kernel pode, então, realizar os serviços requisitados, como interagir com os dispositivos de E/S, e pode realizar a gerência de processo e outras atividades desse tipo, que não podem ser realizadas no modo usuário.

Os números de chamada de sistema para versões recentes do kernel do Linux são listados em `/usr/src/linux-2.x/include/asm-i386/unistd.h`. (Por exemplo, `_NR_close` corresponde à chamada de sistema `close()`, que é invocada para fechar um descritor de arquivo, e é definida como valor 6.) A lista de ponteiros para manipuladores de chamada de sistema normalmente é armazenada no arquivo `/usr/src/linux-2.x/arch/i386/kernel/entry.s` sob o título `ENTRY(sys_call_table)`. Observe que `sys_close` é armazenado no número de entrada 6 na tabela, para ser coerente com o número de chamada de sistema descrito no arquivo `unistd.h`. (A palavra-chave `.long` indica que a entrada ocupará o mesmo número de bytes que o valor de dados do tipo `long`.)

Parte 2: Montando um novo kernel

Antes de acrescentar uma chamada de sistema ao kernel, você precisa familiarizar-se com a tarefa de montar o binário para um kernel a partir de seu código-fonte e dar boot na máquina com o kernel recém-montado. Essa atividade compreende as tarefas a seguir, algumas das quais dependentes da instalação em particular do sistema operacional Linux.

- Obter o código-fonte do kernel para a distribuição Linux. Se o pacote do código-fonte tiver sido instalado na sua máquina, os arquivos correspondentes podem estar disponíveis sob `/usr/src/linux` ou `/usr/src/linux-2.x` (onde o sufixo corresponde ao número de versão do kernel). Se o pacote não tiver sido instalado, ele pode ser baixado do provedor da sua distribuição Linux ou de <http://www.kernel.org>.
- Aprenda como configurar, compilar e instalar o binário do kernel. Isso variará entre as diferentes distribuições do kernel, mas alguns comandos típicos para a montagem do kernel (depois de entrar no diretório onde o código-fonte do kernel está armazenado) incluem:
 - make xconfig
 - make dep
 - make bzImage

Inclua uma nova entrada no conjunto de kernels inicializáveis admitidos pelo sistema. O sistema operacional Linux normalmente usa utilitários como `lilo` e `grub` para manter uma lista de kernels inicializáveis, da qual o usuário pode escolher durante o boot da máquina. Se o seu sistema admite `lilo`, inclua uma entrada em `lilo.conf`, como:

```
image=/boot/bzImage.mykernel
label=mykernel
root=/dev/hda5
read-only
```

onde `/boot/bzImage.mykernel` é a imagem do kernel e `mykernel` é o label associado ao novo kernel. Essa

etapa permitirá que você escolha o novo kernel durante o processo de boot, de modo que terá a opção de dar boot em um novo kernel ou dar boot no kernel não modificado, se o kernel recém-montado não funcionar corretamente.

Parte 3: Estendendo o fonte do kernel

Você pode agora experimentar a inclusão de um novo arquivo no conjunto de arquivos-fonte usados para compilar o kernel. Normalmente, o código-fonte é armazenado no diretório `/usr/src/linux-2.x/kernel`, embora o local possa ser diferente na sua distribuição Linux. Existem duas opções para incluir a chamada de sistema. A primeira é incluir a chamada de sistema em um arquivo-fonte existente em seu diretório. A segunda é criar um novo arquivo no diretório-fonte e modificar `/usr/src/linux-2.x/kernel/Makefile` para incluir o arquivo recém-criado no processo de compilação. A vantagem da primeira técnica é que, quando você modifica um arquivo existente que já faz parte do processo de compilação, o Makefile não exige modificação.

Parte 4: Incluindo uma chamada de sistema no kernel

Agora que você está familiarizado com as diversas tarefas de base correspondentes à montagem e boot de kernels do Linux, poderá iniciar o processo de incluir uma nova chamada de sistema no kernel do Linux. No projeto, a chamada de sistema terá funcionalidade limitada; ele passará do modo usuário para o modo kernel, imprimirá uma mensagem que está registrada com as mensagens do kernel e passará de volta para o modo usuário. Chamaremos essa chamada de sistema de `helloworld`. Embora ela tenha funcionalidade limitada, ilustra o mecanismo de chamada de sistema e esclarece a interação entre os programas do usuário e o kernel.

- Crie um novo arquivo chamado `helloworld.c` para definir sua chamada de sistema. Inclua os arquivos de cabeçalho `linux/linkage.h` e `linux/kernel.h`. Inclua o seguinte código nesse arquivo:

```
#include <linux/linkage.h>
#include <linux/kernel.h>
asmlinkage int sys_helloworld() {
    printk(KERN_EMERG "hello world!");
    return 1;
}
```

Isso cria uma chamada de sistema com o nome `sys_helloworld()`. Se você decidir incluir essa chamada de sistema em um arquivo existente no diretório de origem, tudo o que é necessário é incluir a função `sys_helloworld()` no arquivo escolhido. No código, `asmlinkage` vem dos dias em que o Linux usava código C++ e C, e é usado para indicar que o código é escrito em C. A função `printk()` é usada para imprimir mensagens em um arquivo de log do kernel e, portanto, só pode ser chamada do kernel. As mensagens do kernel especificadas no parâmetro para `printk()` são registradas em log no arquivo `/var/log/kernel/warnings`. O protótipo de função para a chamada `printk()` é definido em `/usr/include/linux/kernel.h`.

- Defina um novo número de chamada de sistema para `_NR_helloworld` em `/usr/src/linux-2.x/include/asm-i386/unistd.h`. Um programa do usuário pode usar esse número para identificar a chamada de sistema recém-acrescentada. Certifique-se também de incrementar o valor para `_NR_syscalls`, que é armazenado no mesmo arquivo. Essa constante rastreia o número de chamadas de sistema atualmente definidas no kernel.
- Inclua uma entrada `.long sys_helloworld` em `sys_call_table`, definida no arquivo `/usr/src/linux-2.x/arch/i386/kernel/entry.S`. Conforme já dissemos, o número da chamada de sistema é usado para indexação nessa tabela, para encontrar a posição do código manipulador para a chamada de sistema invocada.
- Inclua seu arquivo `helloworld.c` no Makefile (se você criou um novo arquivo para a sua chamada de sistema). Salve uma cópia da sua antiga imagem binária do kernel (para o caso de haver problemas com o seu kernel recém-criado). Você agora pode montar o novo kernel, trocar seu nome para distingui-lo do kernel não modificado e acrescentar uma entrada nos arquivos de configuração (como `lilo.conf`). Depois de completar essas etapas, você poderá dar boot no kernel antigo ou no novo kernel que contém sua chamada de sistema.

Parte 5: Usando a chamada de sistema por um programa do usuário

Quando você dá o boot com o novo kernel, ele aceita a chamada de sistema recém-definida; você agora precisa invocar essa chamada de sistema a partir de um programa do usuário. Normalmente, a biblioteca C padrão admite uma interface para chamadas de sistema definidas para o sistema operacional Linux. Porém, como sua nova chamada de sistema não está vinculada à biblioteca C

padrão, invocar sua chamada de sistema exigirá intervenção manual.

Conforme já observamos, uma chamada de sistema é invocada armazenando o valor apropriado em um registrador de hardware e realizando uma instrução de trap. Infelizmente, essas são operações de baixo nível, que não podem ser realizadas com instruções da linguagem C e, em vez disso, exigem instruções em assembly. Felizmente, o Linux provê macros para instanciar funções wrapper que contêm as instruções assembly apropriadas. Por exemplo, o programa C a seguir utiliza a macro `_syscall0()` para invocar a chamada de sistema recém-definida:

```
#include <linux/errno.h>
#include <sys/syscall.h>
#include <linux/unistd.h>

_syscall0(int, helloworld);

main()
{
    helloworld();
}
```

- A macro `_syscall0` utiliza dois argumentos. O primeiro especifica o tipo do valor retornado pela chamada de sistema; o segundo é o nome da chamada de sistema. O nome é usado para identificar o número da chamada de sistema que é armazenado no registrador de hardware antes que a instrução de trap seja executada. Se a sua chamada de sistema exigir argumentos, então uma macro diferente (como `_syscallN`, onde o sufixo indica o número de argumentos) poderá ser usada para instanciar o código assembly exigido para realizar a chamada de sistema.
- Compile e execute o programa com o kernel recém-montado. Deverá haver uma mensagem “hello world!” no arquivo de log do kernel `/var/log/kernel/warnings` para indicar que a chamada de sistema foi executada.

Como uma nova etapa, considere a expansão da funcionalidade da sua chamada de sistema. Como você passaria um valor inteiro ou uma string de caracteres para a chamada de sistema e a imprimiria no arquivo de log do kernel? Quais são as implicações de passar ponteiros para dados armazenados no espaço de endereço do programa do usuário, em vez de passar um valor inteiro do programa do usuário para o kernel usando registradores de hardware?

Notas bibliográficas

Dijkstra [1968] defendeu a técnica em camadas para o projeto do sistema operacional. Brinch-Hansen [1970] foi um dos primeiros a propor a construção de um sistema operacional como um kernel no qual podem ser construídos sistemas mais completos.

A instrumentação do sistema e o rastreio dinâmico são descritos em Tamches e Miller [1999]. DTrace é discutido em Cantrill e outros [2004]. O código-fonte DTrace está disponível em <http://src.opensolaris.org/source/>. Cheung e Loong [1995] exploram os aspectos de estruturação de sistemas operacionais desde microkernel até sistemas extensíveis.

O MS-DOS, versão 3.1, é descrito em Microsoft [1986]. Windows NT e Windows 2000 são descritos por Solomon [1998] e Solomon e Russinovich [2000]. O Windows 2003 e o Windows XP estão descritos em Russinovich e Solomon [2005]. Hart [2005] aborda com detalhes as programações dos sistemas Windows. O BSD UNIX é descrito em McKusick e outros [1996]. Bovet e Cesati [2006] abordam o kernel do Linux com detalhes. Diversos sistemas UNIX, incluindo Mach, são abordados com detalhes em Vahalia [1996]. O Mac OS X é apresentado em <http://www.apple.com/macosx> e em Singh [2007]. O Solaris é descrito por completo em Mauro e McDougall [2007].

O primeiro sistema operacional a fornecer uma máquina virtual foi o CP/67 em um IBM 360/67. O sistema operacional IBM VM/370, disponível comercialmente, foi derivado do CP/67. Os detalhes com relação ao Mach, um sistema operacional baseado em microkernel, podem ser encontrados em Young e outros [1987]. Kaashoek e outros [1997] apresentam detalhes com relação aos sistemas operacionais de exokernel, onde a arquitetura separa as questões de gerenciamento da proteção, dando assim ao software não confiável a capacidade de exercer controle sobre recursos de hardware e software.

A especificação para a linguagem Java e a máquina virtual Java são apresentadas por Gosling e outros [2005] e por Lindholm e Yellin [1999], respectivamente. O funcionamento interno da máquina virtual Java é descrito totalmente por Vermers [1998]. Golm e outros [2002] destacam o sistema operacional JX; Back e outros [2000] abordam diversas questões no projeto dos sistemas operacionais Java. Outras informações sobre Java estão disponíveis na Web, em <http://java.sun.com>. Os detalhes sobre a implementação do VMware podem ser encontrados em Sugerman e outros [2001]. Informações sobre o Open Virtual Machine Format podem ser encontradas em <http://www.vmware.com/appliances/learn.ovf.html>.

PARTE II

GERÊNCIA DE PROCESSOS

ESBOÇO

[Capítulo 4: Introdução a Gerência de processos](#)

[Capítulo 5: Processos](#)

[Capítulo 6: Threads](#)

[Capítulo 7: Escalonamento de CPU](#)

[Capítulo 8: Sincronismo de processos](#)

[Capítulo 9: Deadlocks](#)

Introdução a Gerência de processos

Um *processo* pode ser imaginado como um programa em execução. Um processo precisará de certos recursos – como tempo de CPU, memória, arquivos e dispositivos de E/S – para realizar sua tarefa. Esses recursos são alocados ao processo quando ele é criado ou enquanto está sendo executado.

Um processo é a unidade de trabalho na maioria dos sistemas. Sistemas consistem em uma coleção de processos: os processos do sistema operacional executam o código do sistema, e os processos do usuário executam o código do usuário. Todos esses processos podem ser executados de forma concorrente.

Embora, tradicionalmente, um processo tenha apenas uma única thread de controle enquanto é executado, a maioria dos sistemas operacionais modernos admite processos com múltiplas threads.

O sistema operacional é responsável pelas seguintes atividades relacionadas com a gerência de processo e thread: a criação e remoção dos processos do usuário e do sistema; o escalonamento dos processos; e a provisão de mecanismos para o tratamento de sincronismo, comunicação e deadlock para os processos.

CAPÍTULO 3

Processos

Os primeiros sistemas computadorizados permitiam que apenas um programa fosse executado de cada vez. Esse programa tinha controle total do sistema e acesso a todos os recursos do sistema. Por outro lado, atualmente, os sistemas computadorizados permitem que vários programas sejam carregados na memória e executados concorrentemente. Essa evolução exigiu controle mais rígido e maior compartimentalização dos diversos programas, e essas necessidades resultaram na noção de **processo**, que é um programa em execução. Um processo é a unidade de trabalho em um sistema moderno de tempo compartilhado.

Quanto mais complexo for o sistema operacional, mais se espera dele em favor de seus usuários. Embora sua preocupação principal seja a execução dos programas do usuário, ele também precisa cuidar das diversas tarefas do sistema que funcionam melhor fora do próprio kernel. Um sistema, portanto, consiste em uma coleção de processos: processos do sistema operacional executando código do sistema e processos do usuário executando código do usuário. Potencialmente, todos esses processos podem ser executados de forma concorrente, com a CPU (ou CPUs) multiplexada(s) entre eles. Com a CPU alternando entre os processos, o sistema operacional pode tornar o computador mais produtivo. Neste capítulo, você entenderá o que são processos e como eles funcionam.

OBJETIVOS DO CAPÍTULO

- Introduzir a noção de processo - um programa em execução que forma a base de toda a computação.
- Descrever os diversos recursos dos processos, incluindo escalonamento, criação, término e comunicação.
- Descrever a comunicação nos sistemas cliente-servidor.

3.1 Conceito de processo

Uma pergunta que surge quando se discute sobre sistemas operacionais envolve como chamar todas as atividades da CPU. Um sistema batch executa *jobs*, enquanto um sistema de tempo compartilhado possui *programas do usuário*, ou *tarefas*. Até mesmo em um sistema monousuário, como o Microsoft Windows original, um usuário pode ser capaz de executar diversos programas ao mesmo tempo: um processador de textos, um navegador Web e um pacote de correio eletrônico. Mesmo que o usuário possa executar apenas um programa de cada vez, o sistema operacional pode precisar dar suporte às suas próprias atividades internas programadas, como a gerência de memória. Em muitos aspectos, todas essas atividades são semelhantes e, por isso, chamamos todas elas de *processos*.

Os termos *tarefa* (job) e *processo* são usados para indicar quase a mesma coisa neste texto. Embora pessoalmente prefiramos o termo *processo*, grande parte da teoria e terminologia de sistema operacional foi desenvolvida durante uma época em que a principal atividade dos sistemas operacionais era o processamento de tarefas. Seria confuso evitar o uso dos termos mais aceitos, que incluem a palavra *tarefa* (como *escalonamento de tarefa*) porque *processo* substituiu *tarefa*.

3.1.1 O processo

Informalmente, como já dissemos, um processo é um programa em execução. Um processo é mais do que um código de programa, que às vezes é conhecido como **seção de texto(text section)**. Ele também inclui a atividade atual, representada pelo valor do **contador de programa** e o conteúdo dos registradores do processador. Em geral, um processo também inclui a **pilha(stack)** de processo, que contém **dados temporários** (como parâmetros de método, endereços de retorno e variáveis locais) e uma **seção de dados(data section)**, que contém variáveis globais. Um processo também pode incluir **uma pilha de heap**, que é a **memória alocada dinamicamente** durante o tempo de execução do processo. A estrutura do processo em memória é mostrada na [Figura 3.1](#).

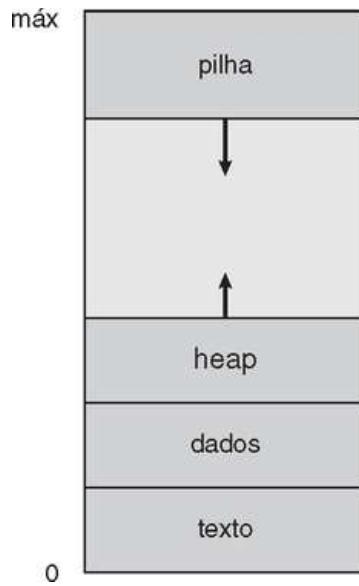


FIGURA 3.1 Processo na memória.

Enfatizamos que **um programa, por si só, não é um processo**; um programa é **uma entidade passiva**, como um arquivo contendo uma lista de instruções armazenadas em disco (em geral denominado **arquivo executável**), enquanto um processo é uma entidade *ativa*, com um contador de programa especificando a próxima instrução a ser executada e um conjunto de recursos associados. Um programa se torna um processo quando um **arquivo executável** é carregado na memória. Duas técnicas comuns para carregar arquivos executáveis são o clique duplo em um ícone representando o **arquivo executável** e a **entrada do nome do arquivo executável na linha de comandos** (como em prog.exe ou a.out).

Embora **um programa possa ser associado a dois processos**, mesmo assim estes são considerados duas sequências de execução separadas. Por exemplo, diversos usuários podem executar diferentes cópias do programa de correio ou, então, o mesmo usuário pode invocar muitas cópias do programa editor. **Cada um é um processo separado**, e embora as seções de texto sejam equivalentes as seções de dados, heap e pilha variam. Também é comum ter um processo que cria muitos processos enquanto é executado. Discutiremos essas questões na [Seção 3.4](#).

3.1.2 Estado do processo

Enquanto um processo é executado, ele muda de **estado**. O estado de um processo é definido em parte pela atividade atual desse processo. Cada processo pode estar em um dos seguintes estados:

- **Novo (New)**. O processo está sendo criado.
- **Executando (Running)**. As instruções estão sendo executadas.
- **Esperando (Waiting)**. O processo está esperando algum evento (como término de E/S ou recebimento de um sinal).
- **Pronto (Ready)**. O processo está esperando para ser atribuído a um processador.
- **Terminado (Terminated)**. O processo terminou sua execução.

Esses nomes são arbitrários e variam de acordo com os sistemas operacionais. Todavia, os estados que representam são encontrados em todos os sistemas. Certos sistemas operacionais também delineiam os estados do processo de modo mais minucioso. É importante observar que apenas um processo pode estar *executando(running)* em qualquer processador em determinado instante. Contudo, muitos processos podem estar *prontos(ready)* e *esperando(waiting)*. O diagrama de estados correspondente a esses estados é apresentado na [Figura 3.2](#).



FIGURA 3.2 Diagrama de estado do processo.

3.1.3 Bloco de controle de processo

Cada processo é representado no sistema operacional por um **bloco de controle de processo (Process Control Block - PCB)**, também conhecido como *bloco de controle de tarefa*. Um PCB é representado na [Figura 3.3](#). Ele contém muitas informações associadas a um processo específico, incluindo:

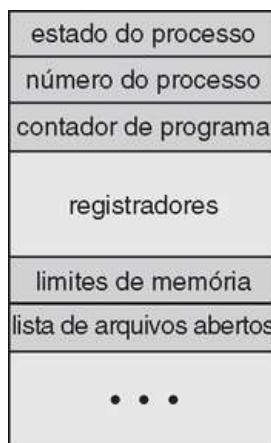


FIGURA 3.3 Bloco de controle de processo (PCB).

- **Estado do processo**. O estado pode ser novo (new), pronto (ready), executando (running), esperando (waiting), interrompido (halted), e assim por diante.
- **Contador de programa (program counter)**. O contador indica o endereço da próxima instrução a ser executada para esse processo.
- **Registradores da CPU**. Os registradores variam em quantidade e tipo, dependendo da arquitetura do computador. Eles incluem acumuladores, registradores de índice, ponteiros de

pilha e registradores de uso geral, além de qualquer informação de código de condição. Junto com o contador de programa, essa informação de status precisa ser salva quando ocorre uma interrupção, para permitir que o processo seja continuado corretamente depois disso (Figura 3.4).

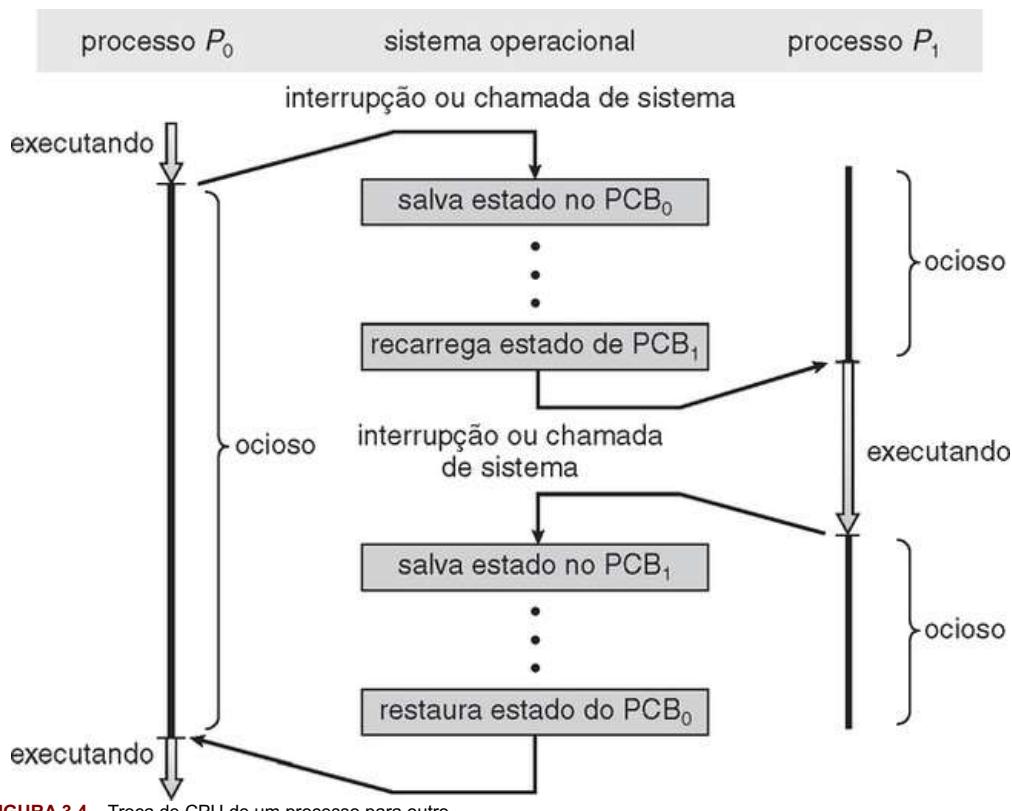


FIGURA 3.4 Troca de CPU de um processo para outro.

- **Informação de escalonamento de CPU.** Essa informação inclui uma prioridade de processo, ponteiros para filas de escalonamento e quaisquer outros parâmetros de escalonamento. (O Capítulo 5 descreve o escalonamento de processos.)
 - **Informação de gerência de memória.** Essa informação pode incluir dados como o valor dos registradores de base e limite, as tabelas de página ou as tabelas de segmento, dependendo do sistema de memória utilizado pelo sistema operacional (Capítulo 8).
 - **Informação contábil.** Essa informação inclui a quantidade de CPU e o tempo de leitura utilizado, limites de tempo, números de conta, números de tarefa ou processo, e assim por diante.
 - **Informação de status de E/S.** Essa informação inclui a lista de dispositivos de E/S alocados ao processo, uma lista de arquivos abertos, e assim por diante.
- Resumindo, o PCB serve como o repositório para quaisquer informações que possam variar de um processo para outro.

3.1.4 Threads

O modelo de processo discutido até aqui implicou que um processo é um programa que executa uma única **thread**. Por exemplo, quando um processo está executando um programa de processamento de textos, uma única thread de instruções está sendo executada. Essa única thread de controle permite que o processo execute apenas uma tarefa de cada vez. O usuário não pode, ao mesmo tempo, digitar caracteres e executar o corretor ortográfico dentro do mesmo processo, por exemplo. Muitos sistemas operacionais modernos estenderam o conceito de processo para permitir que um processo tenha várias threads e, portanto, realize mais de uma tarefa de cada vez. Em um sistema que admite threads, o PCB é expandido para incluir informações para cada thread. Outras mudanças por todo o sistema também são necessárias para dar suporte às threads. O Capítulo 4 explora os detalhes dos processos dotados de múltiplas threads (multithreaded).

3.2 Escalonamento de processos

O objetivo da multiprogramação é ter algum processo executando o tempo todo, para melhorar a utilização da CPU. O objetivo do compartilhamento de tempo é alternar a CPU entre os processos com tanta frequência que os usuários possam interagir com cada programa enquanto ele está sendo executado. Para atender a esses objetivos, o **escalonador de processos(process scheduler)** seleciona um processo disponível (possivelmente a partir de um conjunto de vários processos disponíveis) para execução do programa na CPU. Para um sistema de processador único, nunca haverá mais do que um processo em execução. Se houver mais processos, o restante terá de esperar até que a CPU esteja livre e possa ser reescalonada.

EXEMPLO DE API PADRÃO

O bloco de controle de processo no sistema operacional Linux é representado pela estrutura C `task_struct`. Essa estrutura contém toda a informação necessária para representar um processo, incluindo o estado do processo, informações de escalonamento e gerência de memória, lista de arquivos abertos e ponteiros para o pai do processo e qualquer um de seus filhos. (O *pai* de um processo é o processo que o criou; seus *filhos* são quaisquer processos que ele cria.) Alguns desses campos são:

```
pid_t pid; /* identificador de processo */  
long state; /* estado do processo */  
unsigned int time_slice /* informação de escalonamento */  
struct task_struct *parent; /* pai deste processo */  
struct list_head children; /* filhos deste processo */  
struct files_struct *files; /* lista de arquivos abertos */  
struct mm_struct *mm; /* espaço de endereços deste processo */
```

Por exemplo, o estado de um processo é representado pelo campo `long state` nessa estrutura. Dentro do kernel do Linux, todos os processos ativos são representados usando uma lista duplamente interligada de `task_struct`, e o kernel mantém um ponteiro - `current` - para o processo atualmente executado no sistema. Isso aparece na [Figura 3.5](#).



FIGURA 3.5 Processos ativos no Linux.

Como uma ilustração de como o kernel poderia manipular um dos campos na `task_struct` para um processo especificado, vamos supor que o sistema gostaria de mudar o estado do processo atualmente em execução para o valor `new_state`. Se `current` for um ponteiro para o processo em execução, seu estado será alterado com o seguinte:

```
current->state = new_state;
```

3.2.1 Filas de escalonamento

Quando os processos entram no sistema, eles são colocados em uma **fila de tarefas(job queue)**, que consiste em todos os processos no sistema. Os processos que estão residindo na memória principal e que estão prontos e esperando para serem executados são mantidos em uma lista chamada **fila de prontos(ready queue)**. Em geral, essa fila é armazenada como uma lista interligada. Um cabeçalho da fila de prontos contém ponteiros para o primeiro e último PCB na lista.

Cada PCB inclui um campo de ponteiro, que aponta para o próximo PCB na fila de prontos.

O sistema também inclui outras filas. Quando um processo recebe alocação de CPU, ele é executado por um tempo e por fim termina, é interrompido ou espera pela ocorrência de determinado evento, como o término de uma requisição de E/S. Suponha que o processo faça uma requisição de E/S a um dispositivo compartilhado, como um disco. Como existem muitos processos no sistema, o disco pode estar ocupado com a requisição de E/S de algum outro processo. O processo, portanto, pode ter de esperar pelo disco. A lista de processos esperando por determinado dispositivo de E/S é denominada **fila de dispositivo(device queue)**. Cada dispositivo possui sua própria fila de dispositivo (Figura 3.6).

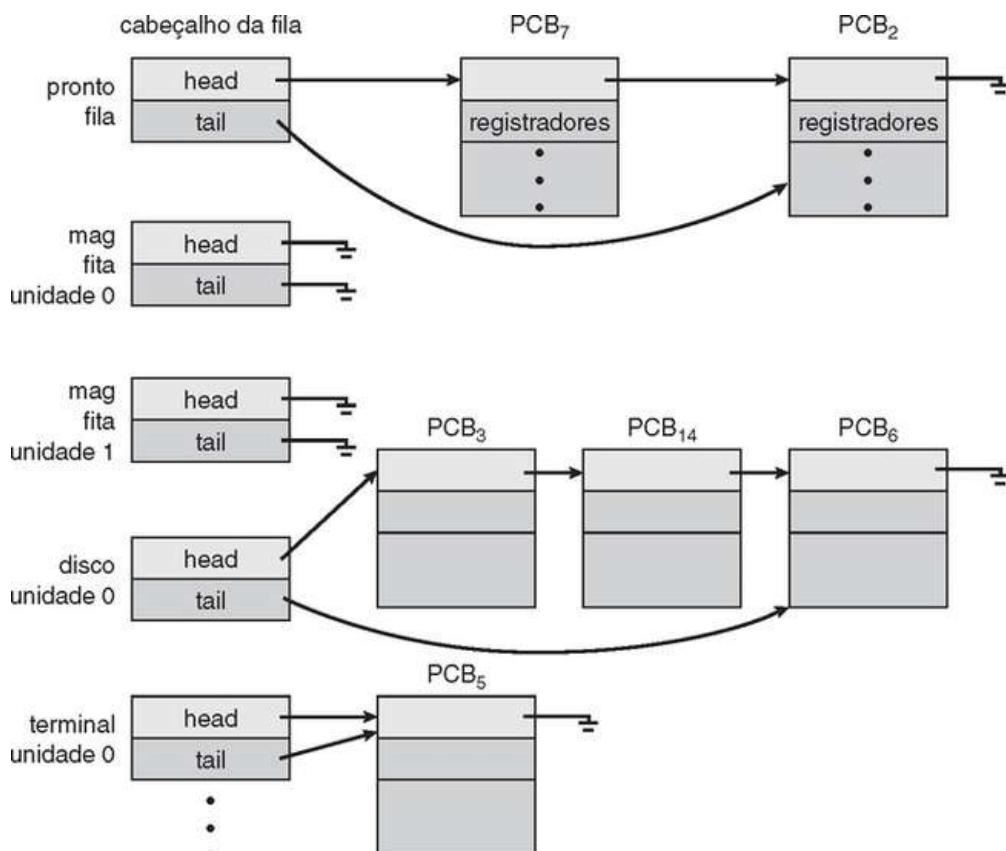


FIGURA 3.6 A fila de prontos e diversas filas de dispositivo de E/S.

Uma representação comum do escalonamento de processos é um **diagrama de fila**, como o que aparece na Figura 3.7. Cada caixa retangular representa uma fila. Dois tipos de filas estão presentes: a fila de prontos e um conjunto de filas de dispositivo. Os círculos representam os recursos que atendem às filas, e as setas indicam o fluxo de processos no sistema.

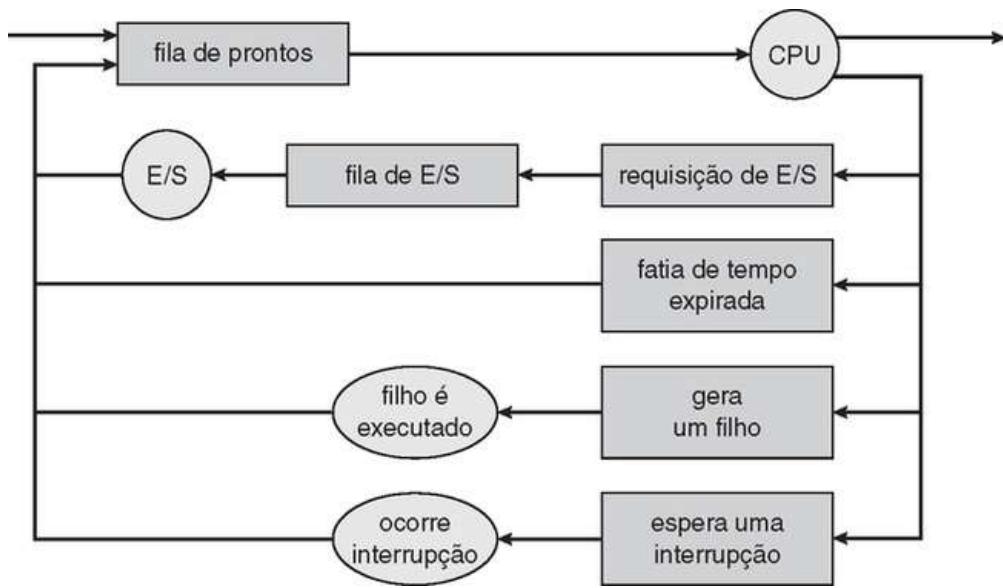


FIGURA 3.7 A representação do diagrama de fila do escalonamento de processos.

Um processo novo é inicialmente colocado na fila de prontos. Ele espera lá até que seja selecionado para execução (**despachado**). Quando o processo recebe tempo de CPU e está executando, pode ocorrer um dentre vários eventos:

- O processo pode emitir uma requisição de E/S e depois ser colocado em uma fila de E/S.
- O processo pode criar um novo subprocesso e esperar pelo término do subprocesso.
- O processo pode ser removido forçadamente da CPU, como resultado de uma interrupção, e ser colocado de volta na fila de prontos.

Nos dois primeiros casos, o processo por fim passa do estado de espera para o estado de pronto e depois é colocado de volta na fila de prontos. Um processo continua esse ciclo até que termine, quando é removido de todas as filas e perde a alocação do seu PCB e de seus recursos.

3.2.2 Escalonadores

Um processo migra entre diversas filas de escalonamento no decorrer do seu tempo de vida. O sistema operacional precisa selecionar, para fins de escalonamento, processos vindos de alguma maneira dessas filas. O processo de seleção é executado pelo **escalonador** apropriado.

Normalmente, em um sistema batch, mais processos são submetidos do que a quantidade que pode ser executada imediatamente. Esses processos são guardados em um dispositivo de armazenamento em massa (em geral, um disco), onde são mantidos para execução posterior. O **escalonador de longo prazo(long term scheduler)**, ou **escalonador de tarefas**, seleciona processos desse banco e os carrega para a memória, para execução. O **escalonador de curto prazo(short term scheduler)**, ou **escalonador de CPU**, seleciona dentro dos processos que estão prontos para executar e aloca a CPU a um deles.

A principal distinção entre esses dois escalonadores está na frequência de execução. O escalonador de curto prazo precisa selecionar um novo processo para a CPU frequentemente. Um processo só pode ser executado por alguns milissegundos antes de esperar por uma requisição de E/S. Em geral, o escalonador de curto prazo é executado pelo menos uma vez a cada 100 milissegundos. Devido ao curto prazo entre as execuções, o escalonador de curto prazo precisa ser veloz. Se forem necessários 10 milissegundos para decidir executar um processo por 100 milissegundos, então $10/(100 + 10) = 9\%$ da CPU em uso (desperdiçada) para o escalonamento do trabalho.

O escalonador de longo prazo é executado com muito menos frequência; minutos podem separar a criação de um novo processo e o seguinte. O escalonador de longo prazo controla o **grau de multiprogramação** (o número de processos na memória). Se o grau de multiprogramação for estável, a taxa média de criação do processo precisará ser igual à taxa de saída média dos processos que deixam o sistema. Assim, o escalonador de longo prazo pode ter de ser invocado apenas quando um processo sai do sistema. Devido ao intervalo mais longo entre as execuções, o escalonador de longo prazo pode ter mais tempo para decidir qual processo deverá ser selecionado para execução.

É importante que o escalonador de longo prazo faça uma seleção cuidadosa. Em geral, a maior parte dos processos pode ser descrita como I/O-bound, uso intensivo de E/S, ou CPU-bound, pouco uso de E/S. Um **processo I/O-bound** é aquele que gasta mais do seu tempo realizando E/S do que gasta realizando cálculos. Um **processo CPU-bound**, ao contrário, gera requisições de E/S com pouca frequência, usando mais do seu tempo realizando cálculos. O escalonador de longo prazo precisa selecionar um bom **mix de processos**, entre processos I/O-bound e processos CPU-bound.

Se todos os processos forem I/O-bound, a fila de prontos quase sempre estará vazia, e o escalonador de curto prazo terá pouca coisa a fazer. Se todos os processos forem CPU-bound, a fila de espera de E/S quase sempre estará vazia, os dispositivos serão pouco usados e novamente o sistema estará desequilibrado. O sistema com o melhor desempenho, assim, terá uma combinação de processos CPU-bound e I/O-bound.

Em alguns sistemas, o escalonador de longo prazo pode estar ausente ou ser mínimo. Por exemplo, os sistemas de tempo compartilhado, como UNIX e Microsoft Windows, normalmente não possuem escalonador de longo prazo, mas colocam cada novo processo na memória, para o escalonador de curto prazo. A estabilidade desses sistemas depende de uma limitação física (como o número de terminais disponíveis) ou da natureza de autoajuste dos usuários humanos. Se o desempenho diminuir até chegar a níveis inaceitáveis em um sistema multiusuário, alguns usuários sairão.

Alguns sistemas operacionais, como os sistemas de tempo compartilhado, podem introduzir um nível de escalonamento intermediário adicional. Esse **escalonador de médio prazo (medium-term scheduler)** é representado na [Figura 3.8](#). A ideia principal por trás de um escalonador de médio prazo é que, às vezes, pode ser vantajoso remover processos da memória (e da disputa ativa pela CPU) e, assim, reduzir o grau de multiprogramação. Mais tarde, o processo pode ser reintroduzido na memória e sua execução pode continuar de onde parou. Esse esquema é chamado de **troca (swapping)**. O processo é descarregado para a área de troca (swapped out) e mais tarde é carregado da área de troca (swapped in) pelo escalonador de médio prazo. A troca pode ser necessária para melhorar a mistura de processos ou porque uma mudança nos requisitos de memória comprometeu a memória disponível, exigindo a liberação da memória. A troca é discutida no [Capítulo 8](#).

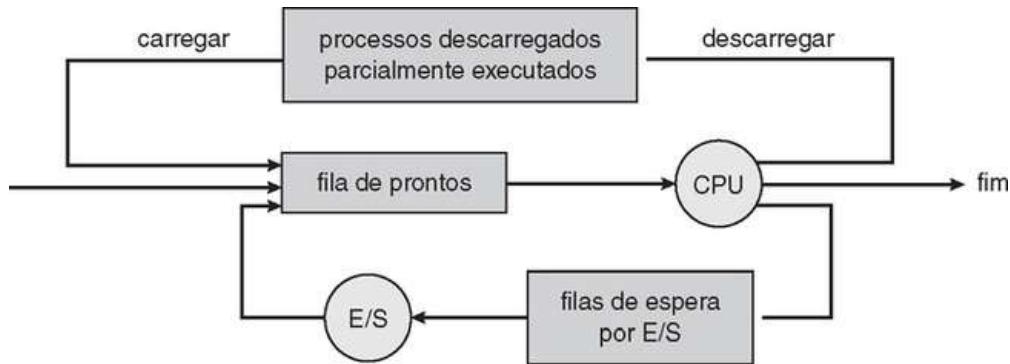


FIGURA 3.8 Acréscimo do escalonamento de meio-termo para o diagrama de enfileiramento.

3.2.3 Troca de contexto

Conforme mencionamos na [Seção 1.2.1](#), as interrupções fazem o sistema operacional passar a CPU da sua tarefa atual e executar uma rotina do kernel. Essas operações acontecem com frequência nos sistemas de uso geral. Quando ocorre uma interrupção, o sistema precisa salvar o **contexto** atual do processo atualmente em execução na CPU, para que possa restaurar esse contexto quando seu processamento terminar, basicamente suspendendo o processo e depois retomando-o. O contexto é representado no PCB do processo; ele inclui o valor dos registradores da CPU, o estado do processo ([Figura 3.2](#)) e informações de gerência de memória. Genericamente, realizamos um **salvamento de estado** do estado atual da CPU, seja no modo kernel ou usuário, e depois uma **restauração de estado** para retomar as operações.

A passagem da CPU para outro processo exige o salvamento do estado atual do processo e a restauração do estado de um processo diferente. Essa tarefa é conhecida como **troca de contexto(context switch)**. Quando ocorre uma troca de contexto, o kernel salva o contexto do processo antigo em seu PCB e carrega o contexto salvo do novo processo. O tempo da troca de contexto é puramente custo adicional, pois o sistema não realiza um trabalho útil durante a troca. Sua velocidade varia de uma máquina para outra, dependendo da velocidade da memória, do número de registradores que precisam ser copiados e da existência de instruções especiais (como uma única instrução para carregar ou armazenar todos os registradores). A velocidade típica é inferior a alguns milissegundos.

Os tempos de troca de contexto dependem bastante do suporte do hardware. Por exemplo, alguns processadores (como o Sun UltraSPARC) fornecem diversos conjuntos de registradores. Uma troca de contexto aqui exige a troca do ponteiro para o conjunto de registradores atual. Se houver mais processos ativos do que os conjuntos de registradores, o sistema terá de usar a cópia de dados de registrador na memória, como antes. Além disso, quanto mais complexo o sistema operacional, mais trabalho deverá ser feito durante uma troca de contexto. Conforme veremos no [Capítulo 8](#), técnicas

avançadas de gerência de memória podem exigir que dados extras sejam trocados em cada contexto. Por exemplo, o espaço de endereços do processo atual precisa ser preservado enquanto o espaço da próxima tarefa é preparado para uso. Como o espaço de endereço é preservado e que quantidade de trabalho é necessária para preservá-lo dependem do método de gerência de memória do sistema operacional.

3.3 Operações sobre processos

Os processos na maioria dos sistemas podem ser executados de forma concorrente e ser criados e removidos dinamicamente. Assim, esses sistemas operacionais precisam prover um mecanismo para a criação e o término de processo. Nesta seção, exploramos os mecanismos envolvidos na criação de processos e ilustramos a criação do processo nos sistemas UNIX e Windows, bem como a criação de processos usando Java.

3.3.1 Criação de processo

Um processo pode gerar diversos novos processos, por meio de uma chamada de sistema (system call ou syscall) “criar processo”, no decorrer da sua execução. O processo que criou novos processos é chamado de processo **pai**, e os novos processos são considerados **filhos** desse processo. Cada um desses novos processos pode, por sua vez, criar outros processos, formando uma **árvore** de processos.

A maioria dos sistemas operacionais (incluindo UNIX e a família Windows de sistemas operacionais) identifica os processos de acordo com um **identificador de processo**(ou **pid**) exclusivo, que normalmente é um valor inteiro. A [Figura 3.9](#) ilustra uma árvore de processos típica para o sistema operacional Solaris, mostrando o nome de cada processo e seu pid. No Solaris, o processo no topo da árvore é o processo `sched`, com pid 0. O processo `sched` cria diversos processos filhos – incluindo `pageout` e `fsflush`. Esses processos são responsáveis por gerenciar a memória e os sistemas de arquivos. O processo `sched` também cria o processo `init`, que serve como processo pai raiz para todos os processos do usuário. Na [Figura 3.9](#), vemos dois filhos de `init` – `inetd` e `dtlogin`, em que `inetd` é responsável por serviços de rede como `telnet` e `ftp`; `dtlogin` é o processo representando uma tela de login do usuário. Quando um usuário efetua o login, `dtlogin` cria uma sessão X-windows (`Xsession`), que, por sua vez, cria o processo `sdt_shel`. Abaixo de `sdt_shel`, um shell de linha de comandos do usuário – o C-shell ou `csh` – é criado. É nessa interface da linha de comandos que o usuário invoca diversos processos filhos, como os comandos `ls` e `cat`. Também vemos um processo `csh` com pid 7778, representando um usuário que efetuou logon no sistema usando `telnet`. Esse usuário iniciou o navegador da Netscape (pid 7785) e o editor `emacs` (pid 8105).

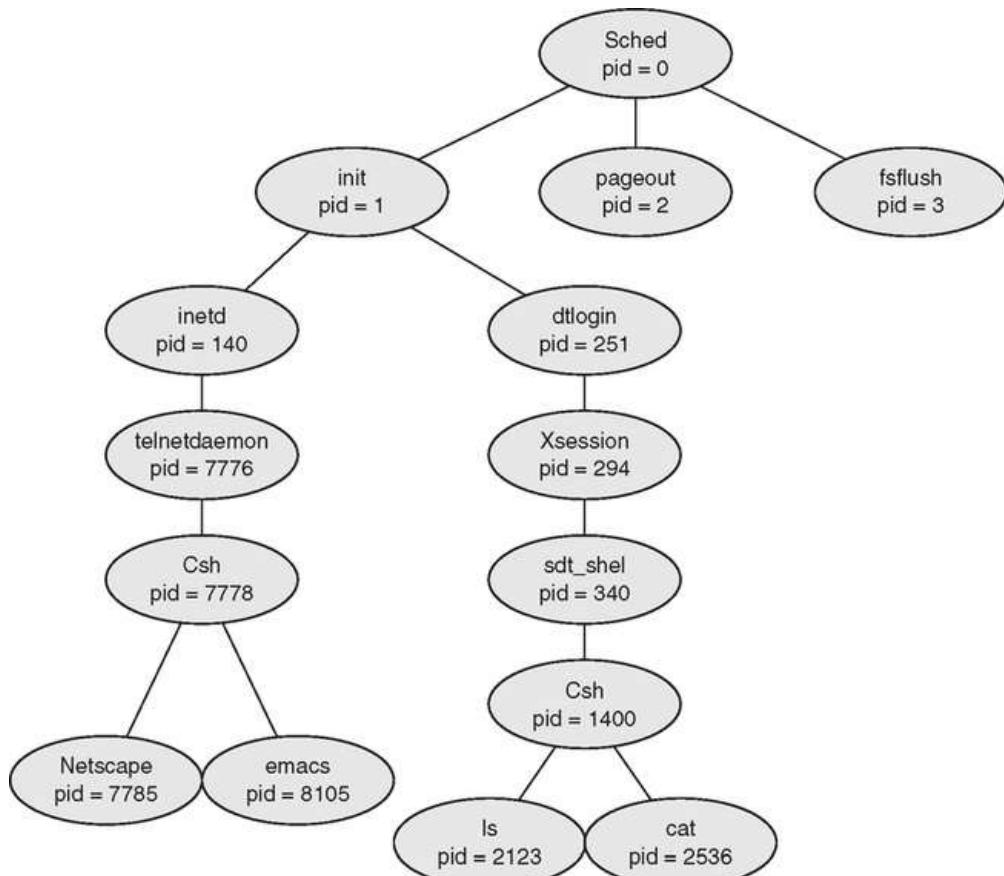


FIGURA 3.9 Uma árvore de processos em um sistema Solaris típico.

No UNIX, podemos obter uma listagem de processos usando o comando `ps`. Por exemplo, o comando `ps -el` listará informações completas para todos os processos atualmente ativos no sistema. É fácil construir uma árvore de processos, semelhante à que aparece na [Figura 3.9](#), rastreando recursivamente os processos pai até chegar ao processo `init`.

Em geral, um processo precisará de certos recursos (tempo de CPU, memória, arquivos, dispositivos de E/S) para realizar sua tarefa. Quando um processo cria um subprocesso, esse subprocesso pode ser capaz de obter seus recursos diretamente do sistema operacional ou, então, pode ser restrito a um subconjunto dos recursos do processo pai. O pai pode ter de repartir seus recursos entre seus filhos ou pode ser capaz de compartilhar alguns recursos (como memória ou arquivos) entre vários dos filhos. A restrição de um processo filho a um subconjunto dos recursos do pai impede que qualquer processo sobrecarregue o sistema, criando muitos subprocessos.

Além dos diversos recursos físicos e lógicos fornecidos, o processo pai pode passar os dados de inicialização (entrada) para o processo filho. Por exemplo, considere um processo cuja função seja exibir o conteúdo de um arquivo – digamos, `img.jpg` – na tela de um terminal. Quando for criado, ele apanhará, como entrada do seu processo pai, o nome do arquivo `img.jpg` e usará esse nome de arquivo, abrirá o arquivo e escreverá o conteúdo. Ele também pode apanhar o nome do dispositivo de saída. Alguns sistemas operacionais passam recursos aos processos filhos. Em tal sistema, o novo processo pode receber dois arquivos abertos, `img.jpg` e o dispositivo terminal, e pode transferir os dados entre os dois.

Quando um processo cria um novo processo, existem duas possibilidades em termos de execução:

1. O pai continua a ser executado simultaneamente com seus filhos.
2. O pai espera até que algum ou todos os seus filhos tenham terminado.

Também existem duas possibilidades em termos do espaço de endereços do novo processo:

1. O processo filho é uma duplicata do processo pai (ele tem o mesmo programa e dados do pai).
2. O processo filho tem um novo programa carregado nele.

Em seguida, vamos ilustrar essas diferenças nos sistemas UNIX e Windows.

3.3.1.1 Criação de processo no UNIX

No UNIX, como visto, cada processo é identificado por seu identificador de processo (process id – PID), que é um inteiro exclusivo. Um novo processo é criado pela chamada de sistema (syscall) `fork()`. O novo processo consiste em uma cópia do espaço de endereços do processo original. O mecanismo permite que o processo pai se comunique facilmente com seu processo filho. Os dois processos (o pai e o filho) continuam a execução na instrução após o `fork()`, com uma diferença: o código de retorno para o `fork()` é zero para o novo processo (filho), enquanto o identificador de processo (diferente de zero) do filho é retornado ao pai.

Normalmente, a chamada de sistema (syscall) `exec()` é usada após uma chamada de sistema `fork()` por um dos dois processos para substituir o espaço de memória do processo por um novo programa. A chamada de sistema `exec()` carrega um arquivo binário na memória (destruindo a imagem do programa em memória que contém a chamada de sistema `exec()`) e inicia sua execução. Desse modo, os dois processos são capazes de se comunicar e depois seguir seus caminhos separados. O pai pode, então, criar mais filhos ou – se não tiver nada mais a fazer enquanto o filho é executado – pode emitir uma chamada de sistema `wait()` para sair da fila de prontos (ready queue) até que a execução do filho termine.

O programa em C mostrado na [Figura 3.10](#) ilustra as chamadas de sistema do UNIX descritas anteriormente. Agora temos dois processos diferentes executando uma cópia do mesmo programa. A única diferença é que o valor do pid (o identificador de processo) para o processo filho é zero; o pai possui um valor inteiro maior do que zero (na verdade, esse é o pid real do processo filho). O processo filho herda os privilégios e os atributos de escalonamento do processo pai, além de certos recursos, como arquivos abertos. O processo filho sobrepõe seu espaço de endereços com o comando do UNIX `/bin/ls` (usado para obter uma listagem do diretório) usando a chamada de sistema (syscall) `execvp()` (`execvp()` é uma versão da chamada de sistema `exec()`). O pai espera até que o processo filho termine com a chamada de sistema `wait()`. Quando o processo filho termina (ao evocar implicitamente ou explicitamente `exit()`), o processo pai retorna da chamada `wait()` e termina usando a chamada de sistema `exit()`. Isso é ilustrado na [Figura 3.11](#).

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
pid_t pid;

/* cria um processo filho */
pid = fork();

if (pid < 0) { /* houve erro */
    fprintf(stderr, "Falha na execução do fork");
    return 1;
}
else if (pid == 0) { /* processo filho */
    execl("/bin/ls","ls",NULL);
}
else { /* processo pai */
    /* pai esperará até que o filho termine */
    wait(NULL);
    printf("Filho terminou a tarefa");
}

return 0;
}

```

FIGURA 3.10 Criando um processo separado usando a chamada de sistema `fork()`.

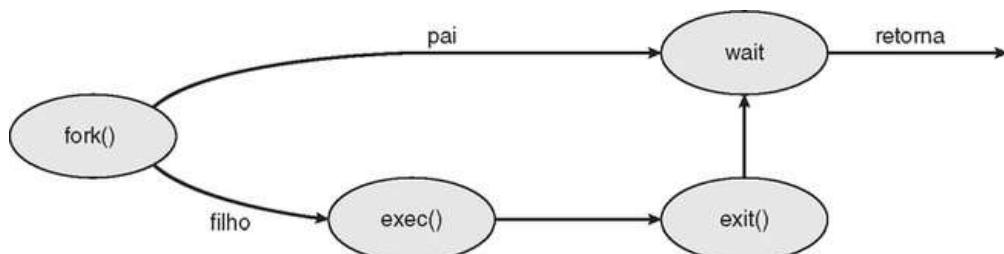


FIGURA 3.11 Criação de processo usando a chamada de sistema `fork()`.

3.3.1.2 Criação de processo no Windows

Como exemplo alternativo, em seguida vamos considerar a criação de processo no Windows. Os processos são criados na API Win32 usando a função `CreateProcess()`, que é semelhante a `fork()` porque um pai cria um novo processo filho. Contudo, enquanto `fork()` tem o processo filho herdando o espaço de endereços de seu pai, `CreateProcess()` requer a carga de um programa especificado no espaço de endereços do processo filho na criação do processo. Além do mais, enquanto `fork()` não recebe parâmetros, `CreateProcess()` não espera menos do que dez parâmetros.

O programa C mostrado na [Figura 3.12](#) ilustra a função `CreateProcess()`, que cria um processo filho que carrega a aplicação `mspaint.exe`. No programa, optamos por muitos dos valores default dos dez parâmetros passados a `CreateProcess()`. Encorajamos os leitores interessados em buscar os detalhes sobre criação e gerência de processos na API Win32 a consultar as notas bibliográficas ao final deste capítulo.

```

#include <stdio.h>
#include <windows.h>

int main(VOID)
{
    STARTUPINFO si;
    PROCESS_INFORMATION pi;

    // aloca memória
    ZeroMemory(&si, sizeof(si));
    si.cb = sizeof(si);
    ZeroMemory(&pi, sizeof(pi));

    // cria processo filho
    if (!CreateProcess(NULL, // usa linha de comando
                      "C:\\\\WINDOWS\\\\system32\\\\mspaint.exe", // linha de comando
                      NULL, // não herda descritor do processo
                      NULL, // não herda descritor de thread
                      FALSE, // desativa herança de descritor
                      0, // nenhum flag de criação
                      NULL, // usa bloco de ambiente do pai
                      NULL, // usa diretório existente do pai
                      &si,
                      &pi))
    {
        fprintf(stderr, "Falha ao criar processo");
        return -1;
    }
    // pai esperará que o filho termine
    WaitForSingleObject(pi.hProcess, INFINITE);
    printf("Filho terminou a tarefa");

    // fecha descritores
    CloseHandle(pi.hProcess);
    CloseHandle(pi.hThread);
}

```

FIGURA 3.12 Criando um processo separado com a API Win32.

Dois parâmetros passados a `CreateProcess()` são instâncias das estruturas `STARTUPINFO` e `PROCESS_INFORMATION`. `STARTUPINFO` especifica muitas propriedades do novo processo, como tamanho e aparência de janela, e descritores (handles) dos arquivos de entrada e saída padrão. A estrutura `PROCESS_INFORMATION` contém um descritor e os identificadores para o processo recém-criado e seu thread. Invocamos a função `ZeroMemory()` para alocar memória para cada uma dessas estruturas antes de prosseguir com `CreateProcess()`.

Os dois primeiros parâmetros passados a `CreateProcess()` são o nome da aplicação e os parâmetros da linha de comandos. Se o nome da aplicação for `NULL` (como neste caso), o parâmetro da linha de comandos especifica a aplicação a carregar. Nesse caso, estamos carregando a aplicação `mspaint.exe` do Microsoft Windows. Além desses dois parâmetros iniciais, usamos os parâmetros default para herdar descritores de processo e thread, além de não especificar nenhum flag de criação. Também usamos o bloco de ambiente existente do pai e diretório de partida. Por fim, fornecemos dois ponteiros para as estruturas `STARTUPINFO` e `PROCESS_INFORMATION`, criados no início do programa. Na [Figura 3.10](#), o processo pai espera que o filho termine invocando a chamada de sistema `wait()`. O equivalente disso em Win32 é `WaitForSingleObject()`, que recebe um descritor do processo filho - `pi.hProcess` - e espera que esse processo termine. Quando o processo filho termina, o controle retorna a partir da função `WaitForSingleObject()` no processo pai.

3.3.1.3 Criação de processos em Java

Quando um programa em Java inicia a execução, uma instância da máquina virtual Java é criada. Na maioria dos sistemas, a JVM aparece como uma aplicação comum rodando como um processo separado no sistema operacional hospedeiro. Cada instância da JVM provê suporte para várias threads de controle, mas a Java não admite um modelo de processo, o que permitiria que a JVM criasse vários processos dentro da mesma máquina virtual. Embora haja uma pesquisa considerável

acontecendo nessa área, o principal motivo para a Java atualmente não admitir um modelo de processo é que é difícil isolar a memória de um processo da memória de outro dentro da mesma máquina virtual.

Contudo, é possível criar um processo externo à JVM, usando a classe `ProcessBuilder`, que permite que um programa Java especifique um processo nativo ao sistema operacional (como `/usr/bin/ls` ou `C:\\WINDOWS\\system32\\mspaint.exe`). Isso é ilustrado na [Figura 3.13](#). A execução desse programa envolve passar o nome do programa que deve ser executado como um processo externo na linha de comandos.

```
import java.io.*;

public class OSProcess
{
    public static void main(String[] args) throws IOException {
        if (args.length != 1) {
            System.err.println("Uso: java OSProcess <comando>");
            System.exit(0);
        }

        // args[0] é o comando executado em um processo separado
        ProcessBuilder pb = new ProcessBuilder(args[0]);
        Process process = pb.start();

        // obtém o fluxo de entrada
        InputStream is = process.getInputStream();
        InputStreamReader isr = new InputStreamReader(is);
        BufferedReader br = new BufferedReader(isr);

        // lê a saída do processo
        String line;
        while ( (line = br.readLine()) != null)
            System.out.println(line);

        br.close();
    }
}
```

FIGURA 3.13 Criando um processo externo com a API Java

Criamos o novo processo invocando o método `start()` da classe `ProcessBuilder`, que retorna uma instância de um objeto `Process`. Esse processo será executado externamente à máquina virtual e não poderá afetar a máquina virtual e vice-versa. A comunicação entre a máquina virtual e o processo externo ocorre por meio do `InputStream` e `OutputStream` do processo externo.

3.3.2 Término de processo

Um processo termina quando acaba de executar sua instrução final e pede ao sistema operacional para removê-lo usando a chamada de sistema (`syscall`) `exit()`. Nesse ponto, o processo pode retornar um valor de status (normalmente, um inteiro) ao seu processo pai (por meio da chamada de sistema `wait()`). Todos os recursos do processo – incluindo a memória física e virtual, arquivos abertos e buffers de E/S – são dealocados pelo sistema operacional.

O término também pode ocorrer em outras circunstâncias. Um processo pode causar o término de outro processo por meio de uma chamada de sistema apropriada (por exemplo, `TerminateProcess()`, no Win32). Normalmente, essa chamada de sistema só pode ser invocada pelo pai do processo que deve ser terminado. Caso contrário, os usuários poderiam encerrar as tarefas uns dos outros de forma arbitrária. Observe que um pai precisa conhecer as identidades de seus filhos. Assim, quando um processo cria um novo processo, a identidade do processo recém-criado é passada ao pai.

Um pai pode terminar a execução de um de seus filhos por diversos motivos, como:

- O filho ultrapassou seu uso de alguns dos recursos alocados a ele. (Para determinar se isso aconteceu, o pai precisa ter um mecanismo para inspecionar o estado de seus filhos.)
- A tarefa atribuída ao filho não é mais necessária.
- O pai está saindo, e o sistema operacional não permite que um filho continue se seu pai terminar. Alguns sistemas, incluindo VMS, não permitem que um filho exista se o pai tiver terminado. Nesses sistemas, se um processo terminar (de modo normal ou anormal), então todos os seus filhos também precisam terminar. Esse fenômeno, conhecido como **término em cascata(cascading termination)**, normalmente é conduzido pelo sistema operacional.

No UNIX, podemos terminar um processo usando a chamada de sistema `exit()`; seu processo pai pode esperar pelo término de um processo filho usando a chamada de sistema `wait()`. A chamada de sistema `wait()` retorna o identificador do processo de um filho terminado, de modo que o pai pode dizer qual dos seus filhos terminou. Contudo, se o pai terminar, todos os seus filhos receberão como seu novo pai o processo `init`. Assim, os filhos ainda terão um pai para coletar seu status e as estatísticas de execução.

3.4 Comunicação entre processos

Os processos em execução concorrentemente no sistema operacional podem ser processos independentes ou processos cooperativos. Um processo é **independente** se não puder afetar ou ser afetado pelos outros processos em execução no sistema. Qualquer processo que não compartilhe dados com qualquer outro processo é independente. Um processo é **cooperativo** se puder afetar ou ser afetado por outros processos em execução no sistema. Naturalmente, qualquer processo que compartilhe dados com outros processos é um processo cooperativo.

Existem vários motivos para prover um ambiente que permita a cooperação de processos:

- **Compartilhamento de informações.** Como diversos usuários podem estar interessados na mesma informação (por exemplo, um arquivo compartilhado), temos de prover um ambiente para permitir o acesso concorrente a tais informações.
- **Agilidade na computação.** Se quisermos que determinada tarefa seja executada mais rapidamente, temos de dividi-la em subtarefas, cada uma sendo executada em paralelo com as outras. Observe que essa agilidade só pode ser obtida se o computador tiver vários elementos de processamento (como CPUs ou canais de E/S).
- **Modularidade.** Podemos querer construir um sistema de uma forma modular, dividindo as funções do sistema em processos ou threads separadas, conforme discutimos no [Capítulo 2](#).
- **Conveniência.** Até mesmo um usuário individual pode trabalhar em muitas tarefas ao mesmo tempo. Por exemplo, um usuário pode estar editando, imprimindo e compilando em paralelo.

Os processos cooperativos precisam de mecanismos de **comunicação entre processos** (*interprocess communication- IPC*) que lhes permitam a troca de dados e informações. Existem dois modelos fundamentais de comunicação entre processos: (1) **memória compartilhada** e (2) **troca de mensagem**. No modelo de memória compartilhada, uma região da memória que é compartilhada pelos processos cooperativos é estabelecida. Os processos podem, então, trocar informações pela leitura ou escrita de dados para a região compartilhada. No modelo troca de mensagem, a comunicação ocorre por meio de mensagens trocadas entre os processos cooperativos. Os dois modelos de comunicação são mostrados na [Figura 3.14](#).

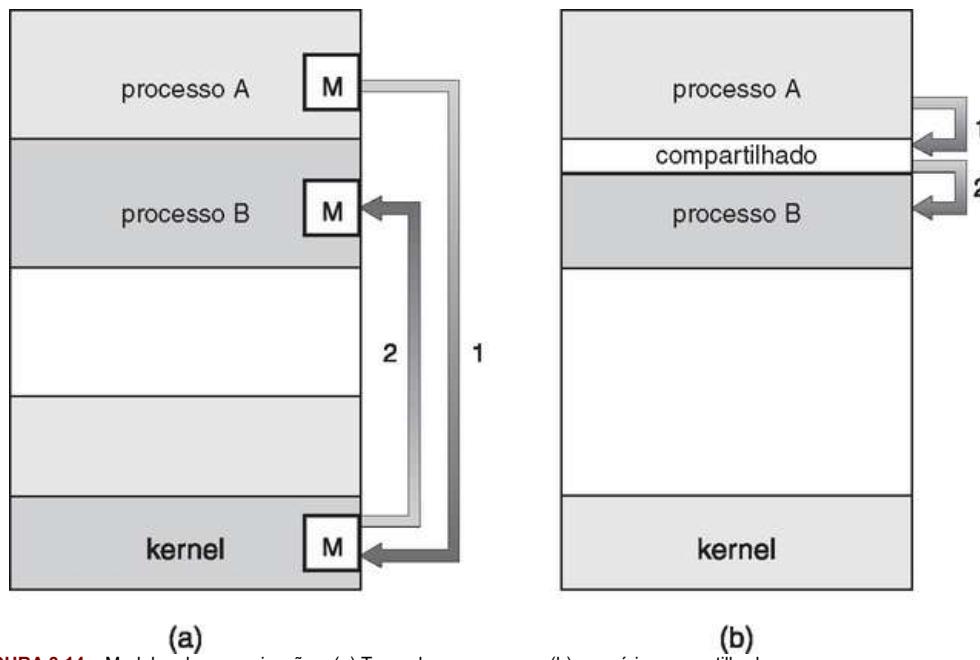


FIGURA 3.14 Modelos de comunicações. (a) Troca de mensagem e (b) memória compartilhada.

Os dois modelos são comuns nos sistemas operacionais, e muitos sistemas implementam ambos. A troca de mensagem é útil para trocar quantidades menores de dados, pois nenhum conflito precisa ser evitado. A troca de mensagem também é mais fácil de implementar do que a memória compartilhada para a comunicação entre computadores. A memória compartilhada permite velocidade máxima e conveniência de comunicação. A memória compartilhada é mais rápida do que a troca de mensagem porque os sistemas de troca de mensagem normalmente são implementados com o uso de chamadas de sistema e, assim, exigem a tarefa mais demorada de intervenção do kernel. Por outro lado, nos sistemas de memória compartilhada as chamadas de sistema só precisam estabelecer regiões de memória compartilhada. Quando a memória compartilhada é estabelecida, todos os acessos são tratados como acessos à memória de rotina, e não é preciso qualquer

assistência do kernel. No restante desta seção, exploramos esses modelos de IPC com mais detalhes.

3.4.1 Sistemas de memória compartilhada

A comunicação entre processos usando memória compartilhada requer a comunicação dos processos para estabelecer uma região de memória compartilhada. Normalmente, uma região de memória compartilhada reside no espaço de endereços do processo criando o segmento de memória compartilhada. Outros processos que queiram se comunicar usando esse segmento de memória compartilhada precisam conectá-la ao seu espaço de endereços. Lembre-se de que, normalmente, o sistema operacional tenta impedir que um processo acesse a memória de outro processo. A memória compartilhada exige que dois ou mais processos concordem em remover essa restrição. Eles podem, então, trocar informações lendo e escrevendo dados nas áreas compartilhadas. O formato dos dados e o local são determinados por esses processos e não estão sob o controle do sistema operacional. Os processos também são responsáveis por garantir que não estarão escrevendo no mesmo local simultaneamente.

Para ilustrar o conceito de processos cooperativos, vamos considerar o problema produtor-consumidor, que é um paradigma comum para os processos cooperativos. Um processo **produtor** produz informações consumidas por um processo **consumidor**. Por exemplo, um compilador pode produzir código assembly, que é consumido por um assemblér (ou montador). O assemblér, por sua vez, pode produzir módulos objeto, que são consumidos pelo carregador. O problema produtor-consumidor também fornece uma metáfora útil para o paradigma cliente-servidor. Em geral, pensamos em um servidor como um produtor e um cliente como um consumidor. Por exemplo, um servidor Web produz (ou seja, fornece) arquivos HTML e imagens que são consumidos (ou seja, lidos) pelo navegador Web cliente requisitando o recurso.

Uma solução para o problema produtor-consumidor usa a memória compartilhada. Para permitir que processos produtor e consumidor sejam executados concorrentemente, precisamos ter à disposição um buffer de itens que possa ser preenchido pelo produtor e esvaziado pelo consumidor. Esse buffer residirá na região da memória que é compartilhada pelos processos produtor e consumidor. Um produtor pode produzir um item enquanto o consumidor está consumindo outro. O produtor e o consumidor precisam estar sincronizados, de modo que o consumidor não tente consumir um item que ainda não foi produzido.

Dois tipos de buffer podem ser usados. O **buffer ilimitado(unbounded buffer)** não coloca um limite prático no tamanho do buffer. O consumidor pode ter de esperar a produção de novos itens, mas o produtor sempre pode produzir novos itens. O **buffer limitado (bounded buffer)** considera que o tamanho do buffer é fixo. Nesse caso, o consumidor precisa esperar o buffer estar vazio, e o produtor precisa esperar o buffer estar cheio.

Vamos ilustrar agora uma solução para o problema produtor-consumidor usando a memória compartilhada. Essas soluções podem implementar a interface Buffer, mostrada na [Figura 3.15](#). O processo produtor chama o método `insert()` ([Figura 3.16](#)) quando deseja inserir um item no buffer, e o consumidor chama o método `remove()` ([Figura 3.17](#)) quando quer consumir um item do buffer.

```
public interface Buffer <E>
{
    // Produtores chamam este método
    public void insert(E item);

    // Consumidores chamam este método
    public E remove();
}
```

FIGURA 3.15 Interface para implementações de buffer.

```

// Produtores chamam este método
public void insert(E item) {
    while (count == BUFFER_SIZE)
        ; // não faz nada -- nenhum buffer livre

    // acrescenta um item ao buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;
}

```

FIGURA 3.16 O método insert().

```

// Consumidores chamam este método
public E remove() {
    E item;

    while (count == 0)
        ; // não faz nada -- nada para consumir

    // remove um item do buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    --count;

    return item;
}

```

FIGURA 3.17 O método remove().

Embora a linguagem Java não forneça suporte para memória compartilhada, podemos projetar uma solução para o problema de buffer limitado em Java, que simula a memória compartilhada permitindo que os processos produtor e consumidor compartilhem uma instância da classe `BoundedBuffer` ([Figura 3.18](#)), que implementa a interface `Buffer`. Esse compartilhamento envolve a passagem de uma referência a uma instância da classe `BoundedBuffer` para os processos produtor e consumidor, que é ilustrado na [Figura 3.19](#).

```

import java.util.*;

public class BoundedBuffer<E> implements Buffer<E>
{
    private static final int BUFFER_SIZE = 5;
    private int count; // número de itens no buffer
    private int in; // aponta para próxima posição livre
    private int out; // aponta para próxima posição cheia
    private E[] buffer;

    public BoundedBuffer() {
        // buffer inicialmente vazio
        count = 0;
        in = 0;
        out = 0;

        buffer = (E[]) new Object[BUFFER_SIZE];
    }

    // produtores chamam esse método
    public void insert(E item) {
        // Figura 3.16
    }

    // consumidores chamam esse método
    public E remove() {
        // Figura 3.17
    }
}

```

FIGURA 3.18 Solução de memória compartilhada para o problema produtor-consumidor.

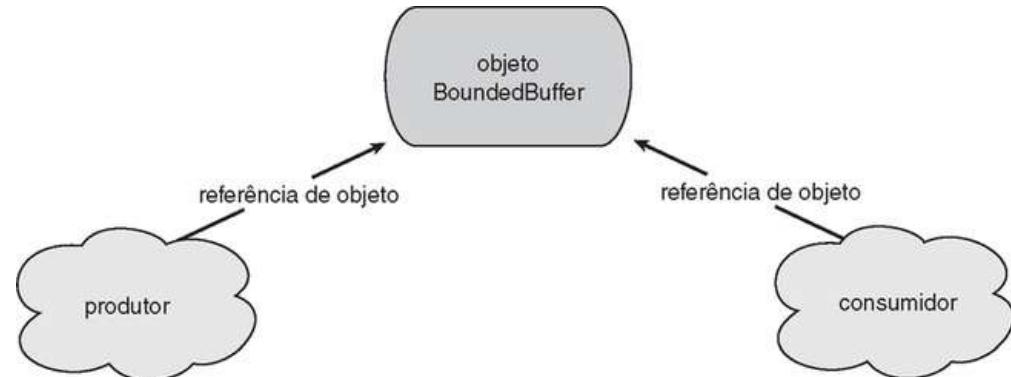


FIGURA 3.19 Simulando a memória compartilhada em Java.

O buffer compartilhado é implementado como um array circular com dois ponteiros lógicos: `in` e `out`. A variável `in` aponta para a próxima posição livre no buffer; `out` aponta para a primeira posição cheia no buffer. A variável `count` é o número de itens atualmente no buffer. O buffer está vazio quando `count == 0` e está cheio quando `count == BUFFER_SIZE`. Observe que tanto o produtor quanto o consumidor serão bloqueados no loop `while` se o buffer não puder ser utilizado por eles. No [Capítulo 6](#), discutiremos como o sincronismo entre os processos cooperativos pode ser implementado de forma eficiente em um ambiente de memória compartilhada.

3.4.2 Sistema de troca de mensagens

Na [Seção 3.4.1](#), mostramos como os processos cooperativos podem se comunicar em um ambiente

de memória compartilhada. O esquema exige que esses processos compartilhem uma região da memória e que o código para acesso e manipulação da memória compartilhada seja escrito explicitamente pelo programador da aplicação. Outra maneira de conseguir o mesmo efeito é deixar que o sistema operacional forneça os meios para que os processos cooperativos se comuniquem entre si por meio de um recurso de troca de mensagens.

A troca de mensagens fornece um mecanismo para permitir que os processos se comuniquem e sincronizem suas ações sem compartilhar o mesmo espaço de endereços e é útil em um ambiente distribuído, no qual os processos em comunicação podem residir em diferentes computadores, conectados com uma rede. Por exemplo, um programa de **bate-papo** usado na World Wide Web poderia ser projetado de modo que os participantes do bate-papo se comuniquem entre si trocando mensagens.

Um recurso de troca de mensagens fornece pelo menos as duas operações `send(mensagem)` e `receive(mensagem)`. As mensagens enviadas por um processo podem ser de tamanho fixo ou variável. Se somente mensagens de tamanho fixo puderem ser enviadas, a implementação em nível de sistema é muito simples. No entanto, essa restrição torna a tarefa de programação mais difícil. Por outro lado, mensagens de tamanho variável exigem uma implementação mais complexa em nível de sistema, mas a tarefa de programação se torna mais simples. Esse é um tipo comum de compensação, encarada durante o projeto do sistema operacional.

Se os processos P e Q desejam se comunicar, eles precisam enviar mensagens e receber mensagens um do outro; é preciso haver um enlace de comunicação entre eles. Esse enlace pode ser implementado de diversas maneiras. Aqui, estamos preocupados não com a implementação física do enlace (como memória compartilhada, barramento do hardware ou rede, que são abordados no [Capítulo 16](#)), mas, sim, com sua implementação lógica. Aqui estão diversos métodos para implementar logicamente um enlace e as operações `send()/receive()`:

- Comunicação direta ou indireta.
- Comunicação síncrona ou assíncrona.
- Buffer automático ou explícito.

Examinaremos, em seguida, os aspectos relativos a cada um desses recursos.

3.4.2.1 Nomes

Os processos que desejam se comunicar precisam ter um modo de se referir uns aos outros. Eles podem usar a comunicação direta ou indireta.

Sob a **comunicação direta**, cada processo que deseja se comunicar precisa nomear explicitamente o destinatário ou emissor da comunicação. Nesse esquema, as primitivas `send()` e `receive()` são definidas como:

- `send(P, mensagem)` – Enviar uma mensagem ao processo P .
- `receive(0, mensagem)` – Receber uma mensagem do processo 0.

Um enlace de comunicação nesse esquema possui as seguintes propriedades:

- Um enlace é estabelecido automaticamente entre cada par de processos que deseja se comunicar. Os processos precisam saber apenas a identidade uns dos outros para se comunicarem.
- Um enlace é associado a exatamente dois processos.
- Entre cada par de processos, existe exatamente um enlace.

Esse esquema exibe *simetria* no endereçamento, ou seja, os processos emissor e receptor precisam citar o nome do outro para haver comunicação. Uma variante desse esquema emprega a *assimetria* no endereçamento. Somente o emissor informa o nome do destinatário, e ele não precisa nomear o emissor. Nesse esquema, as primitivas `send()` e `receive()` são definidas da seguinte forma:

- `send(P, mensagem)` – Enviar uma mensagem ao processo P .
- `receive(id, mensagem)` – Receber uma mensagem de qualquer processo; a variável id é definida com o nome do processo que efetuou a comunicação.

A desvantagem desses dois esquemas (simétrico e assimétrico) é a modularidade limitada das definições de processo resultantes. A mudança do identificador de um processo pode exigir o exame de todas as definições de processo. Todas as referências ao identificador antigo precisam ser localizadas para que possam ser modificadas para o novo identificador. Em geral, todas essas técnicas de **codificação rígida**, nas quais os identificadores precisam ser indicados explicitamente, são menos desejáveis do que aquelas envolvendo um nível de indireção, conforme descrevemos a seguir.

Com a **comunicação indireta**, as mensagens são enviadas e recebidas a partir de **caixas de correio**, ou **portas**. Uma caixa de correio (mailbox) pode ser vista de forma abstrata como um objeto em que as mensagens podem ser incluídas pelos processos e do qual as mensagens podem ser removidas. Cada caixa de correio possui uma identificação exclusiva. Por exemplo, as filas de mensagens POSIX utilizam um valor inteiro para identificar uma caixa de correio. Nesse esquema, um processo pode se comunicar com algum outro processo por meio de diferentes caixas de correio. Contudo, dois processos só podem se comunicar se tiverem uma caixa de correio compartilhada. As primitivas `send()` e `receive()` são definidas da seguinte forma:

- `send(A, mensagem)` – Enviar uma mensagem à caixa de correio A .
- `receive(A, mensagem)` – Receber uma mensagem da caixa de correio A .

Nesse esquema, um enlace de comunicação tem as seguintes propriedades:

- Um enlace é estabelecido entre um par de processos somente se os dois membros do par tiverem uma caixa de correio compartilhada.
- Um enlace pode ser associado a mais de dois processos.
- Entre cada par de processos em comunicação, pode haver diversos enlaces diferentes, com cada um correspondendo a uma caixa de correio.

Agora, suponha que os processos P_1 , P_2 e P_3 compartilhem a caixa de correio A . O processo P_1 envia uma mensagem para A , enquanto P_2 e P_3 executam um `receive()` de A . Qual processo receberá a mensagem enviada por P_1 ? A resposta depende do esquema que escolhermos:

- Permitir que um enlace seja associado a no máximo dois processos.
- Permitir que no máximo um processo por vez execute uma operação `receive()`.
- Permitir que o sistema selecione arbitrariamente qual processo receberá a mensagem (ou seja, P_2 ou P_3 receberá a mensagem, mas não ambos). O sistema também pode definir um algoritmo para selecionar qual processo receberá a mensagem (ou seja, *por revezamento*, no qual os processos, por sua vez, recebem mensagens). O sistema pode identificar o receptor para o emissor.

Uma caixa de correio (mailbox) pode pertencer a um processo ou ao sistema operacional. Se a caixa de correio pertencer a um processo (ou seja, a caixa de correio faz parte do espaço de endereços do processo), então distinguimos entre o proprietário (que só pode receber mensagens por meio dessa caixa de correio) e o usuário (que só pode enviar mensagens à caixa de correio). Como cada caixa de correio possui um proprietário exclusivo, não poderá haver confusão sobre quem deve receber uma mensagem enviada a essa caixa de correio. Quando um processo que possui uma caixa de correio terminar, a caixa de correio desaparecerá. Qualquer processo que, mais tarde, enviar uma mensagem a essa caixa de correio deverá ser notificado de que a caixa de correio não existe mais.

Por outro lado, uma caixa de correio que pertence ao sistema operacional tem uma existência própria. Ela é independente e não está ligada a qualquer processo em particular. O sistema operacional precisa fornecer um mecanismo permitindo que um processo faça o seguinte:

- Criar uma nova caixa de correio.
- Enviar e receber mensagens por meio da caixa de correio.
- Excluir uma caixa de correio.

O processo que cria uma nova caixa de correio, como padrão, é o proprietário da caixa de correio. Inicialmente, o proprietário é o único processo que pode receber mensagens por meio dessa caixa de correio. No entanto, a posse e o privilégio de recebimento podem ser passados para outros processos por meio de chamadas de sistema apropriadas. Naturalmente, essa provisão pode resultar em diversos receptores para cada caixa de correio.

3.4.2.2 Síncronismo

A comunicação entre os processos ocorre por meio de chamadas às primitivas `send()` e `receive()`. Existem diferentes opções de projeto para implementar cada primitiva. A troca de mensagens pode ser **com bloqueio(blocking)** ou **sem bloqueio(nonblocking)** - também conhecidas como **síncrona e assíncrona**.

- **Envio com bloqueio:** o processo que envia é bloqueado até que a mensagem seja recebida pelo processo receptor ou pela caixa de correio.
- **Envio sem bloqueio:** o processo envia a mensagem e continua sua operação.
- **Recebimento com bloqueio:** o receptor é bloqueado até que uma mensagem seja recebida ou esteja disponível.
- **Recebimento sem bloqueio:** o receptor recebe uma mensagem válida ou uma mensagem nula.

É possível haver diferentes combinações de `send()` e `receive()`. Quando `send()` e `receive()` forem com bloqueio, temos um **encontro (rendezvous)** entre o emissor e o receptor. A solução para o problema do produtor-consumidor torna-se trivial quando usamos instruções `end()` e `receive()` com bloqueio. O produtor invoca a chamada `send()` com bloqueio e espera até que a mensagem seja entregue ou ao receptor ou à caixa de correio. De modo semelhante, quando o consumidor invoca `receive()`, ele é bloqueado até que uma mensagem esteja disponível.

Observe que os conceitos de síncrono e assíncrono ocorrem frequentemente nos algoritmos de E/S do sistema operacional, conforme veremos no decorrer deste texto.

3.4.2.3 Buffers

Seja a comunicação direta ou indireta, as mensagens trocadas pelos processos em comunicação residem em uma fila temporária. Basicamente, essas filas podem ser implementadas de três maneiras:

- **Capacidade zero:** a fila tem o tamanho máximo de zero; assim, um enlace não pode ter quaisquer mensagens aguardando nela. Nesse caso, o emissor precisa ser bloqueado até o destinatário receber a mensagem.
- **Capacidade limitada:** a fila possui um tamanho finito n ; assim, no máximo n mensagens pode residir nela. Se a fila não estiver cheia quando uma nova mensagem for enviada, a mensagem é

colocada na fila (ou a mensagem é copiada ou é mantido um ponteiro para a mensagem) e o emissor pode continuar a execução sem esperar. Entretanto, a capacidade do enlace é finita. Se o enlace estiver cheio, o emissor terá de ser bloqueado até haver espaço disponível na fila.

■ **Capacidade ilimitada:** a fila potencialmente possui tamanho infinito; assim, qualquer quantidade de mensagens poderá esperar nela. O emissor nunca é bloqueado.

O caso da capacidade zero às vezes é conhecido como sistema de mensagem sem buffer; os outros casos são conhecidos como sistema com buffer automático.

3.4.2.4 Exemplo: Troca de mensagens com Java

Agora, vamos examinar uma solução para o problema produtor-consumidor utilizando a troca de mensagens. Nossa solução implementará a interface Channel mostrada na [Figura 3.20](#). O produtor e o consumidor se comunicarão indiretamente usando a caixa de correio compartilhada, apresentada na [Figura 3.21](#).

```
public interface Channel<E>
{
    // Envia uma mensagem ao canal
    public void send(E item);

    // Recebe uma mensagem do canal
    public E receive();
}
```

FIGURA 3.20 Interface para a troca de mensagens.

```
import java.util.Vector;

public class MessageQueue<E> implements Channel<E>
{
    private Vector<E> queue;

    public MessageQueue() {
        queue = new Vector<E>();
    }

    // Isso implementa um send sem bloqueio
    public void send(E item) {
        queue.addElement(item);
    }

    // Isso implementa um receive sem bloqueio
    public E receive()
    if (queue.size() == 0)
        return null;
    else
        return queue.remove(0);
}
}
```

FIGURA 3.21 Caixa de correio para a troca de mensagens.

O buffer é implementado usando a classe `java.util.Vector`, significando que ele será um buffer com capacidade ilimitada. Observe também que os métodos `send()` e `receive()` são sem bloqueio.

Quando o produtor gera um item, ele coloca esse item na caixa de correio por meio do método `send()`. O código para o produtor aparece na [Figura 3.22](#).

```
Channel<Date> mailBox;

while (true) {
    Date message = new Date();
    mailBox.send(message); }
```

FIGURA 3.22 O processo produtor.

O consumidor obtém um item da caixa de correio usando o método `receive()`. Como `receive()` é sem bloqueio, o consumidor precisa avaliar o valor do `Object` retornado de `receive()`. Se for `null`, a caixa de correio está vazia. O código para o consumidor aparece na [Figura 3.23](#).

```
Channel<Date> mailBox;

while (true) {
    Date message = mailBox.receive();
    if (message != null)
        // consome a mensagem
}
```

FIGURA 3.23 O processo consumidor.

O [Capítulo 4](#) mostra como implementar o produtor e o consumidor com threads de controle separadas e como permitir que a caixa de correio seja compartilhada entre as threads.

3.5 Exemplos de sistemas de IPC

Nesta seção, vamos explorar dois sistemas diferentes de comunicação entre processos. Primeiro, descrevemos a troca de mensagens no sistema operacional Mach. Depois, discutimos sobre o Windows XP, que utiliza a memória compartilhada como mecanismo para fornecer certos tipos de troca de mensagens.

3.5.1 Um exemplo: Mach

O sistema operacional Mach, desenvolvido na Carnegie Mellon University, é um exemplo de um sistema operacional baseado em mensagens. Talvez você se lembre que apresentamos o Mach no [Capítulo 2](#) como parte do sistema operacional Mac OS X. O kernel do Mach permite a criação e a destruição de várias tarefas, que são semelhantes aos processos, mas possuem várias threads de controle. A maior parte da comunicação no Mach – incluindo a maior parte das chamadas de sistema e todas as informações entre tarefas – é executada por *mensagens*. As mensagens são enviadas e recebidas das caixas de correio, chamadas *portas* no Mach.

Até mesmo as chamadas de sistema são feitas por mensagens. Quando uma tarefa é criada, duas caixas de correio especiais – a caixa de correio *Kernel* e a caixa de correio *Notify* – também são criadas. A caixa de correio *Kernel* é usada pelo kernel para a comunicação com a tarefa. O kernel envia notificação de ocorrências de evento para a porta *Notify*. Somente três chamadas de sistema são necessárias para a transferência de mensagens. A chamada `msg_send()` envia uma mensagem a uma caixa de correio. Uma mensagem é recebida por meio de `msg_receive()`. As remote procedure calls (RPCs) são executadas via `msg_rpc()`, que envia uma mensagem e espera exatamente uma mensagem de retorno do emissor. Desse modo, a RPC modela um procedimento típico de chamada à sub-rotina, mas pode atuar entre sistemas – daí o termo *remoto*.

A chamada de sistema `port_allocate()` cria uma nova caixa de correio e aloca espaço para sua fila de mensagens. O tamanho máximo da fila de mensagens é de oito mensagens como padrão. A tarefa que cria a caixa de correio é o proprietário dessa caixa de correio. O proprietário também tem permissão de receber da caixa de correio. Somente uma tarefa de cada vez pode possuir ou receber de uma caixa de correio, mas esses direitos podem ser enviados para outras tarefas, se for preciso.

A caixa de correio possui uma fila de mensagens inicialmente vazia. À medida que as mensagens são enviadas para a caixa de correio, elas são copiadas para lá. Todas as mensagens têm a mesma prioridade. O Mach garante que diversas mensagens do mesmo emissor sejam enfileiradas na ordem primeiro a entrar, primeiro a sair (FIFO), mas não garante uma ordenação absoluta. Por exemplo, as mensagens de dois emissores podem ser enfileiradas em qualquer ordem.

As mensagens propriamente ditas consistem em um cabeçalho de tamanho fixo, seguido por uma parte de dados de tamanho variável. O cabeçalho indica o tamanho da mensagem e inclui dois nomes de caixa de correio. Um nome de caixa de correio é a caixa de correio à qual a mensagem está sendo enviada. Normalmente, a thread que envia espera uma resposta; portanto, o nome da caixa de correio do emissor é passado para a tarefa que recebe, que poderá usá-lo como um “endereço de retorno”.

A parte variável de uma mensagem é uma lista de itens de dados com tipo. Cada entrada na lista possui tipo, tamanho e valor. O tipo dos objetos especificados na mensagem é importante, pois os objetos definidos pelo sistema operacional – como direitos de propriedade ou de acesso para recebimento, estados de tarefa e segmentos de memória – podem ser enviados nas mensagens.

As próprias operações `send` e `receive` são flexíveis. Por exemplo, quando uma mensagem é enviada para uma caixa de correio, ela pode estar cheia. Se a caixa de correio não estiver cheia, a mensagem é copiada para a caixa de correio, e a thread que envia continua seu trabalho. Se a caixa de correio estiver cheia, a thread que envia tem quatro opções:

1. Esperar indefinidamente até que haja espaço na caixa de correio.
2. Esperar no máximo n milissegundos.
3. Não esperar, mas retornar imediatamente.
4. Colocar uma mensagem temporariamente em cache. Uma mensagem pode ser dada ao sistema operacional para que a mantenha, embora a caixa de correio à qual está sendo enviada esteja cheia. Quando a mensagem puder ser colocada na caixa de correio, uma mensagem será enviada de volta ao emissor; somente uma mensagem desse tipo para uma caixa de correio cheia poderá estar pendente a qualquer momento para determinada thread de envio.

A última opção serve para tarefas do servidor, como um driver de impressora de linha. Após terminar uma requisição, essas tarefas podem ter de enviar uma resposta de única vez para a tarefa que havia requisitado o serviço; mas elas também precisam continuar com outras requisições de serviço, mesmo que a caixa de correio de resposta para um cliente esteja cheia.

A operação `receive` precisa especificar de qual caixa de correio ou conjunto de caixas de correio a mensagem será recebida. O **conjunto de caixas de correio** é uma coleção de caixas de correio, conforme declarado pela tarefa, que podem ser agrupadas e tratadas como se fossem uma caixa de correio para os propósitos da tarefa. As threads de uma tarefa podem receber somente de uma caixa

de correio ou conjunto de caixas de correio para os quais essa tarefa tem acesso de recebimento. Uma chamada de sistema `port_status()` retorna o número de mensagens em determinada caixa de correio. A operação `receive` tenta receber de (1) qualquer caixa de correio em um conjunto de caixas de correio ou (2) uma caixa de correio específica (nomeada). Se nenhuma mensagem estiver esperando para ser recebida, a thread receptora pode esperar no máximo n milissegundos ou não esperar.

O sistema Mach foi projetado para sistemas distribuídos, que discutiremos nos [Capítulos 16 a 18](#) [Capítulo 17](#) [Capítulo 18](#), mas o Mach também é adequado para estações de trabalho individuais, como evidenciado pela sua inclusão no sistema Mach OS X. O maior problema com os sistemas de mensagem tem sido o desempenho fraco, causado pela dupla cópia de mensagens; a mensagem é copiada, primeiro, do emissor para a caixa de correio e depois da caixa de correio para o receptor. O sistema de mensagens do Mach tenta evitar as operações de dupla cópia usando técnicas de gerência de memória virtual ([Capítulo 9](#)). Basicamente, o Mach mapeia o espaço de endereços contendo a mensagem do emissor para o espaço de endereços do receptor. A mensagem em si nunca é copiada. Essa técnica de gerenciamento de mensagens provê um grande aumento de desempenho, mas funciona somente para mensagens dentro do sistema.

3.5.2 Um exemplo: Windows XP

O sistema operacional Windows XP (descrito com mais detalhes no [Capítulo 22](#)) é um exemplo de projeto moderno, que emprega a modularidade para aumentar a funcionalidade e diminuir o tempo necessário para implementar novos recursos. O Windows XP fornece suporte a vários ambientes operacionais, ou *subsistemas*, com os quais os programas de aplicação se comunicam por meio de um mecanismo de troca de mensagens. Os programas de aplicação podem ser considerados clientes do subsistema servidor do Windows XP.

O recurso de troca de mensagens no Windows XP é denominado **chamada de procedimento local**(**Local Procedure Call** - **LPC**). A LPC no Windows XP comunica entre dois processos na mesma máquina. Ela é semelhante ao mecanismo de RPC padrão bastante utilizado, mas é otimizado para o Windows XP e específico a ele. Como o Mach, o Windows XP utiliza um objeto porta para estabelecer e manter uma conexão entre dois processos. Cada cliente que chama um subsistema precisa de um canal de comunicação, fornecido por um objeto porta e nunca herdado. O Windows XP utiliza dois tipos de portas: portas de conexão e portas de comunicação. Eles na realidade são os mesmos, mas recebem nomes diferentes, de acordo com o modo como são usados.

As portas de conexão são chamadas *objetos* e são visíveis a todos os processos; elas dão às aplicações um modo de configurar canais de comunicação. Essa comunicação funciona da seguinte maneira:

- O cliente abre um descritor (handle) para o objeto porta de conexão do subsistema.
- O cliente envia uma requisição de conexão.
- O servidor cria duas portas de comunicação privadas e retorna o descritor para uma delas ao cliente.
- O cliente e o servidor utilizam o descritor de porta correspondente para enviar mensagens ou callbacks e escutar as respostas.

O Windows XP utiliza dois tipos de técnicas de troca de mensagens por uma porta, que o cliente especifica quando estabelece o canal. O mais simples, usado para mensagens pequenas, utiliza a fila de mensagens da porta como armazenamento intermediário e copia a mensagem de um processo para outro. Sob esse método, podem ser enviadas mensagens de até 256 bytes.

Se um cliente precisar enviar uma mensagem maior, ele passará a mensagem por um **objeto seção**, que configura uma região da memória compartilhada. O cliente precisa decidir, quando configurar o canal, se precisará ou não enviar uma mensagem grande. Se o cliente determinar que deseja enviar mensagens grandes, ele pedirá que um objeto seção seja criado. De modo semelhante, se o servidor decidir que as respostas serão grandes, ele criará um objeto seção. Para que o objeto seção possa ser usado, uma pequena mensagem é enviada, contendo um ponteiro e informações de tamanho desse objeto seção. Esse método é mais complicado do que o primeiro, mas evita a cópia de dados. Nos dois casos, um mecanismo de callback pode ser usado quando o cliente ou o servidor não puder responder imediatamente a uma requisição. O mecanismo de callback permite que realizem o tratamento assíncrono de mensagens. A estrutura das chamadas de procedimento locais no Windows XP aparece na [Figura 3.24](#).

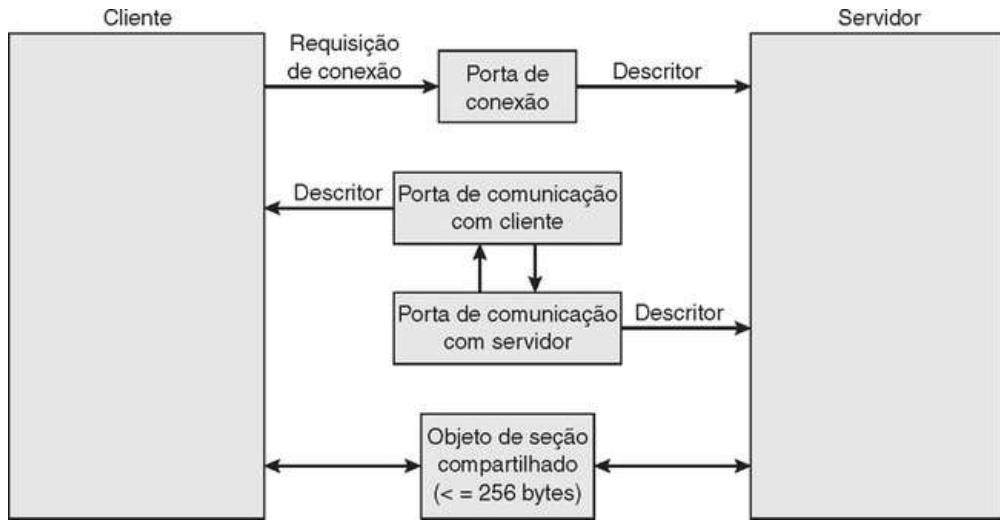


FIGURA 3.24 Chamadas de procedimento local no Windows XP.

É importante observar que a facilidade da LPC no Windows XP não faz parte da API Win32, e por isso não é visível ao programador de aplicação. Em vez disso, as aplicações usando a API Win32 invocam chamadas de procedimento remoto padrão. Quando a RPC está sendo invocada em um processo no mesmo sistema, a RPC é tratada indiretamente por meio de uma chamada de procedimento local. LPCs também são usadas em algumas outras funções que fazem parte da API Win32.

3.6 Comunicação em sistemas cliente-servidor

Na [Seção 3.4](#), descrevemos como os processos podem se comunicar usando memória compartilhada e troca de mensagens. Essas técnicas também podem ser usadas para a comunicação nos sistemas cliente-servidor ([Seção 1.12.2](#)). Nesta seção, vamos explorar três outras estratégias para a comunicação nos sistemas cliente-servidor: sockets, remote procedure calls (RPCs) e remote method invocation (RMI) da Java.

3.6.1 Sockets

Um **socket** é definido como uma extremidade para comunicação. Um par de processos comunicando por uma rede emprega um par de sockets - um para cada processo. Um socket é identificado por um endereço IP concatenado com um número de porta. Em geral, os sockets utilizam uma arquitetura cliente-servidor. O servidor espera por requisições vindas do cliente, escutando em uma porta especificada. Quando uma requisição é recebida, o servidor aceita uma conexão do socket do cliente para completar a conexão. Os servidores implementando serviços específicos (como telnet, FTP e HTTP) escutam portas bem conhecidas (um servidor telnet escuta na porta 23, um servidor FTP escuta na porta 21, e um servidor Web, ou HTTP, escuta na porta 80). Todas as portas abaixo de 1024 são consideradas *bem conhecidas* (*well known*); podemos usá-las para implementar serviços-padrão.

Quando um processo cliente inicia uma requisição para uma conexão, ela é atribuída a uma porta pelo computador host. Essa porta possui um número qualquer, maior do que 1024. Por exemplo, se um cliente no host X com endereço IP 146.86.5.20 deseja estabelecer uma conexão com um servidor Web (que está escutando na porta 80) no endereço 161.25.19.8, o host X pode receber a porta 1625. A conexão consistirá em um par de sockets: (146.86.5.20:1625) no host X e (161.25.19.8:80) no servidor Web. Essa situação é apresentada na [Figura 3.25](#). Os pacotes trafegando entre os hosts são entregues ao processo destino, com base no número da porta de destino.

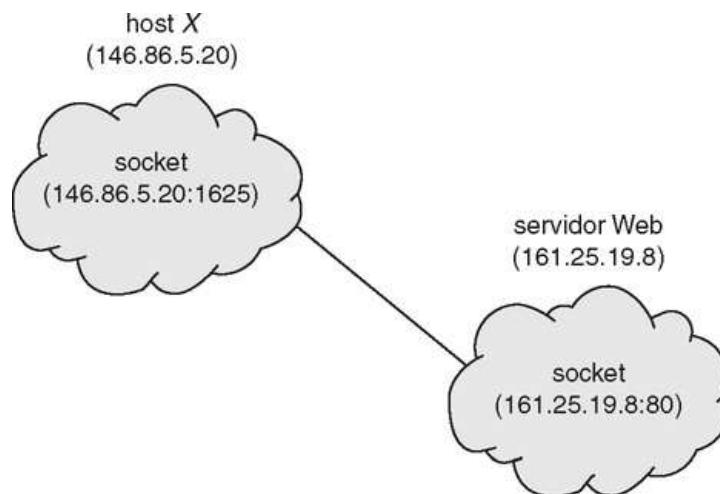


FIGURA 3.25 Comunicação usando sockets.

Todas as conexões precisam ser exclusivas. Portanto, se outro processo também no host X quisesse estabelecer outra conexão com o mesmo servidor Web, ele receberia um número de porta maior do que 1024 e diferente de 1625. Isso garante que todas as conexões consistem em um par de sockets exclusivo.

Para explorar ainda mais a programação por sockets, passamos em seguida a uma ilustração usando Java. A Java fornece uma interface fácil para a programação com sockets e possui uma rica biblioteca de utilitários de rede adicionais.

A Java fornece três tipos de sockets diferentes. Os **sockets orientados a conexão (TCP)** são implementados com a classe `Socket`. Os **sockets sem conexão (UDP)** utilizam a classe `DatagramSocket`. Por fim, a classe `MulticastSocket` é uma subclasse da classe `DatagramSocket`. Um socket multicast permite o envio dos dados a diversos destinatários.

Nosso exemplo descreve um servidor de data que utiliza sockets TCP orientados a conexão. A operação permite aos clientes requisitar a data e hora atuais do servidor. O servidor escuta na porta 6013, embora a porta pudesse ser qualquer número maior do que 1024. Quando uma conexão é recebida, o servidor retorna a data e hora ao cliente.

O servidor de data é mostrado na [Figura 3.26](#). O servidor cria um `ServerSocket` que especifica que ele escutará na porta 6013. O servidor, então, começa a escutar na porta com o método `accept()`. O

servidor é bloqueado no método `accept()`, esperando que um cliente requisite uma conexão. Quando uma requisição de conexão é recebida, `accept()` retorna um socket que o servidor pode utilizar para se comunicar com o cliente.

```
import java.net.*;
import java.io.*;

public class DateServer
{
    public static void main(String[ ] args)  {
        try {
            ServerSocket sock = new ServerSocket(6013);

            // agora escuta conexões
            while (true) {
                Socket client = sock.accept( );

                PrintWriter pout = new
                    PrintWriter(client.getOutputStream( ), true);

                // escreve Date no socket
                pout.println(new java.util.Date( ).toString( ));

                // fecha o socket e continua
                // escutando conexões
                client.close( );
            }
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}
```

FIGURA 3.26 Servidor de data.

Os detalhes de como o servidor se comunica com o socket são: o servidor primeiro estabelece um objeto `PrintWriter` que usará para se comunicar com o cliente. Um objeto `PrintWriter` permite que o servidor escreva no socket usando os métodos `print()` e `println()` de rotina para a saída. O processo servidor envia a data ao cliente, chamando o método `println()`. Quando ele tiver escrito a data no socket, o servidor fechará o socket com o cliente e continuará escutando mais requisições.

Um cliente se comunica com o servidor criando um socket e conectando-se à porta em que o servidor está escutando. Implementamos esse cliente no programa Java que aparece na [Figura 3.27](#). O cliente cria um `Socket` e requisita uma conexão com o servidor no endereço IP 127.0.0.1, na porta 6013. Quando a conexão for feita, o cliente poderá ler do socket usando instruções de E/S de fluxos normais. Depois de receber a data do servidor, o cliente fecha o socket e sai. O endereço IP 127.0.0.1 é um endereço IP especial, conhecido como **loopback**. Quando um computador se referir ao endereço IP 127.0.0.1, ele estará se referindo a si mesmo. Esse mecanismo permite que um cliente e um servidor no mesmo host se comuniquem usando o protocolo TCP/IP. O endereço IP 127.0.0.1 poderia ser substituído pelo endereço IP de outro host executando o servidor de data. Além de usar um endereço IP, um nome de host real (como `www.westminstercollege.edu`) também pode ser usado.

```

import java.net.*;
import java.io.*;

public class DateClient
{
    public static void main(String[] args) {
        try {
            // faz conexão com socket do servidor
            Socket sock = new Socket("127.0.0.1",6013);

            InputStream in = sock.getInputStream();
            BufferedReader bin = new
                BufferedReader(new InputStreamReader(in));

            // lê a data do socket
            String line;
            while ( (line = bin.readLine()) != null)
                System.out.println(line);

            // fecha a conexão do socket
            sock.close();
        }
        catch (IOException ioe) {
            System.err.println(ioe);
        }
    }
}

```

} Cliente de data.

FIGURA 3.27

A comunicação usando sockets - embora comum e eficiente - é considerada uma forma de comunicação de baixo nível entre processos distribuídos. Um motivo é que os sockets só permitem que um fluxo de bytes não estruturado seja trocado entre as threads em comunicação. É responsabilidade da aplicação cliente ou servidora impor uma estrutura sobre os dados. Nas duas subseções seguintes, examinamos dois métodos de comunicação alternativos de nível superior: remote procedure calls (RPCs) e remote method invocation (RMI) da Java.

3.6.2 Remote Procedure Calls

Uma das formas mais comuns de serviço remoto é o paradigma da RPC, que discutimos rapidamente na [Seção 3.5.1](#). A RPC foi projetada como um meio de separar o mecanismo de chamada de procedimento para uso entre sistemas com conexões de rede. Em vários aspectos, ela é semelhante ao mecanismo de IPC descrito na [Seção 3.4](#), e normalmente é montada em cima desse sistema. Contudo, nesse caso, como estamos lidando com um ambiente em que os processos estão executando em sistemas separados, temos de usar o esquema de comunicação baseado em mensagem para fornecer o serviço remoto. Diferentemente das mensagens IPC, as mensagens trocadas na comunicação por RPC são bem estruturadas e, portanto, não são apenas pacotes de dados. Cada mensagem é endereçada a um daemon RPC escutando em uma porta no sistema remoto e contém um identificador da função a ser executada e os parâmetros que devem ser passados a essa função. A função é executada conforme requisitado, e qualquer saída é enviada de volta ao requisitante em uma mensagem separada.

Uma *porta* é um número incluído no início de um pacote de mensagem. Enquanto um sistema possui um endereço de rede, ele pode ter muitas portas dentro desse endereço, para diferenciar os muitos serviços de rede que admite. Se um processo remoto precisa de um serviço, ele endereça uma mensagem à respectiva porta. Por exemplo, se um sistema quiser permitir que outros sistemas sejam capazes de listar seus usuários atuais, ele terá um daemon dando suporte a essa RPC conectada a uma porta - digamos, a porta 3027. Qualquer sistema remoto poderia obter a informação necessária (ou seja, a lista dos usuários atuais) enviando uma mensagem RPC para a porta 3027 no servidor; os dados seriam recebidos em uma mensagem de resposta.

A semântica das RPCs permite que um cliente chame um procedimento em um host remoto da mesma forma como chamaria um procedimento local. O sistema de RPC esconde os detalhes que permitem a comunicação, fornecendo um **stub** no lado do cliente. Normalmente, existe um stub separado para cada procedimento remoto separado. Quando o cliente chama um procedimento remoto, o sistema RPC chama o stub apropriado, passando-lhe os parâmetros fornecidos ao procedimento remoto. Esse stub localiza a porta no servidor e *empacota* os parâmetros. O empacotamento de parâmetros envolve a inclusão dos parâmetros em uma forma que possa ser transmitida por uma rede. O stub transmite, então, uma mensagem ao servidor usando a troca de mensagens. Um stub semelhante, no servidor, recebe essa mensagem e chama o procedimento no servidor. Se for preciso, valores de retorno são passados de volta ao cliente, usando a mesma técnica.

Uma questão que precisa ser tratada refere-se às diferenças na representação de dados nas máquinas cliente e servidor. Considere a representação de inteiros em 32 bits. Alguns sistemas (conhecidos como *big-endian*) armazenam o byte mais significativo primeiro, enquanto outros sistemas (conhecidos como *little-endian*) armazenam o byte menos significativo primeiro. Nenhuma ordem é “melhor” por si, mas a escolha é arbitrária dentro de uma arquitetura de computador. Para resolver diferenças como essa, muitos sistemas de RPC definem uma representação de dados independentemente da máquina. Uma representação desse tipo é conhecida como **representação de dados externos (eXternal Data Representation- XDR)**. No cliente, o empacotamento de parâmetros envolve a conversão dos dados dependentes da máquina para XDR antes de serem enviados ao servidor. No servidor, os dados XDR são convertidos para a representação dependente da máquina adequada ao servidor.

Outra questão importante é a semântica de uma chamada. Embora as chamadas de procedimento local só falhem sob circunstâncias extremas, as RPCs podem falhar ou ser duplicadas e executadas mais de uma vez, como resultado de erros comuns na rede. Um modo de resolver esse problema é fazer o sistema operacional atuar sobre as mensagens *exatamente uma vez(exactly once)*, outro é garantir que ele atue *no máximo uma vez(at most once)*. A maior parte das chamadas de procedimento locais possui a funcionalidade “exatamente uma vez”, mas isso é mais difícil de implementar.

Primeiro, consideramos “no máximo uma vez”. Essa semântica pode ser implementada anexando-se uma estampa de tempo a cada mensagem. O servidor precisa manter um histórico de todas as estampas de tempo das mensagens já processadas ou um histórico grande o suficiente para garantir que mensagens repetidas sejam detectadas. As mensagens que chegam e que possuem estampas de tempo já no histórico são ignoradas. O cliente pode, então, enviar uma mensagem uma ou mais vezes e ter certeza de que ela só foi executada uma vez. (A criação dessas estampas de tempo é discutida na [Seção 18.1](#).)

Para “exatamente uma vez”, precisamos remover o risco de o servidor nunca receber a requisição. Para conseguir isso, o servidor precisa implementar o protocolo “no máximo uma vez”, descrito no parágrafo anterior, e deve também confirmar ao cliente o recebimento e a execução da chamada por RPC. Essas mensagens ACK (confirmação) são comuns nas redes. O cliente precisa reenviar periodicamente cada chamada por RPC até receber o ACK para essa chamada.

Outra questão importante refere-se à comunicação entre um servidor e um cliente. Com chamadas de procedimento padrão, alguma forma de associação ocorre durante o momento do enlace, carga ou execução ([Capítulo 8](#)), de modo que o nome de uma chamada de procedimento seja substituído pelo endereço de memória da chamada de procedimento. O esquema RPC exige uma associação semelhante da porta do cliente e do servidor, mas como um cliente poderá saber os números de porta no servidor? Nenhum dos sistemas possui informações completas sobre o outro, pois eles não compartilham memória.

Duas técnicas para essa questão são comuns. Na primeira, a informação de associação pode ser predeterminada, na forma de endereços de porta fixos. No momento da compilação, uma chamada por RPC possui um número de porta fixo associado a ela. Quando um programa é compilado, o servidor não pode mudar o número de porta do serviço requisitado. Na segunda, a associação pode ser feita de forma dinâmica por um mecanismo de encontro. Normalmente, um sistema operacional fornece um daemon rendevous (também conhecido como **matchmaker**) em uma porta de RPC fixa. Um cliente envia, então, uma mensagem, contendo o nome da RPC, para o daemon de encontro, requisitando o endereço de porta da RPC que precisa executar. O número de porta é retornado, e as chamadas por RPC podem ser enviadas a essa porta até terminar o processo (ou o servidor travar). Esse método exige o custo adicional da requisição inicial, mas é mais flexível do que a primeira técnica. A [Figura 3.28](#) mostra um exemplo de interação.

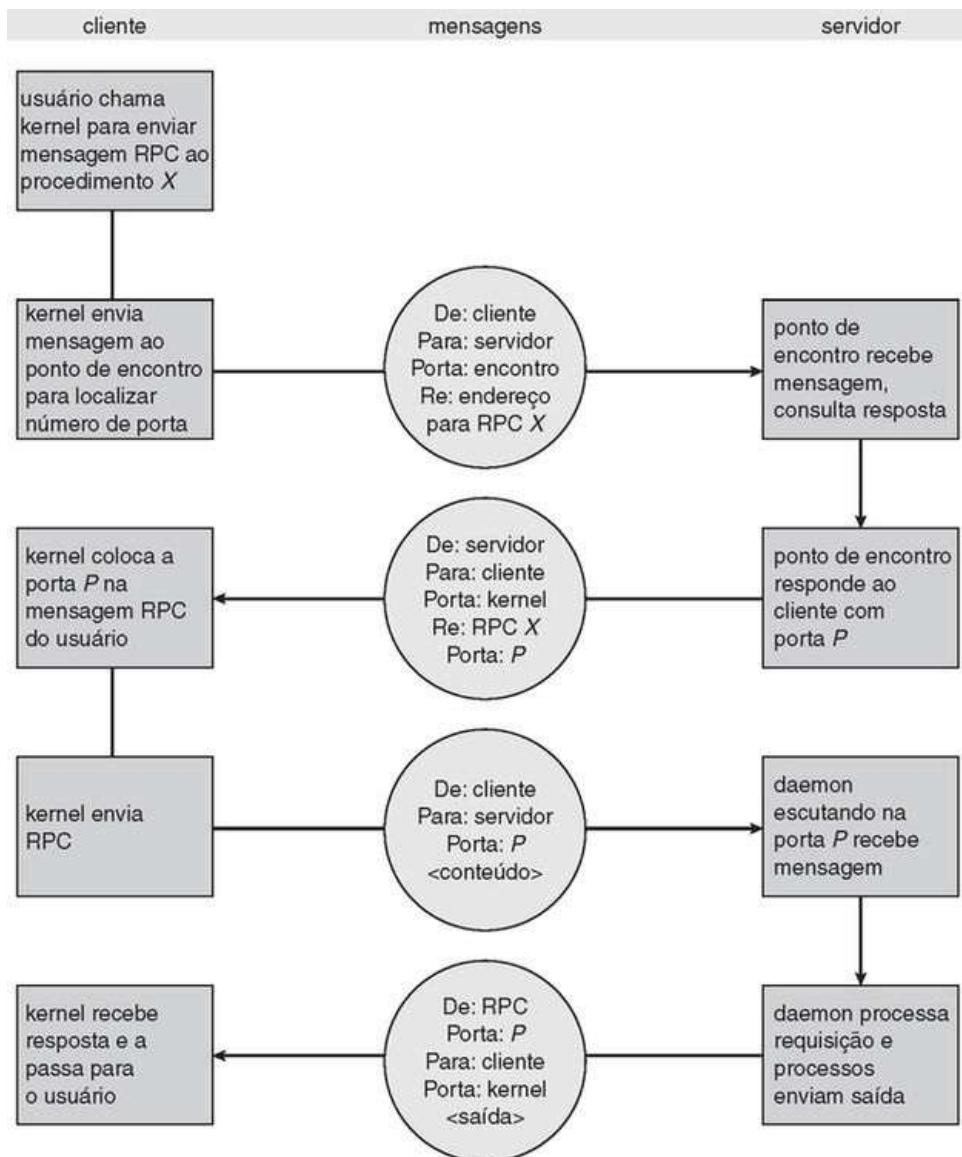


FIGURA 3.28 Execução de uma remote procedure call (RPC).

O esquema RPC é útil na implementação de um sistema de arquivos distribuído ([Capítulo 17](#)). Tal sistema pode ser implementado como um conjunto de daemons e clientes de RPC. As mensagens são endereçadas para a porta DFS em um servidor no qual uma operação de arquivo deverá ocorrer. A mensagem contém a operação de disco a ser realizada. As operações de disco poderiam ser read, write, rename, delete ou status, correspondendo às chamadas de sistema comuns, relacionadas com arquivos. A mensagem de retorno contém quaisquer dados resultantes dessa chamada, executada pelo daemon DFS em favor do cliente. Por exemplo, uma mensagem poderia conter uma requisição para transferir um arquivo inteiro para um cliente ou poderia ser limitada a simples requisições de bloco. Nesse último caso, várias dessas requisições poderiam ser necessárias se um arquivo inteiro tivesse de ser transferido.

3.6.3 Remote Method Invocation

A **Remote Method Invocation** (RMI) é um recurso Java semelhante às RPCs. A RMI permite que uma thread invoque um método em um objeto remoto. Os objetos são considerados remotos se residirem em uma máquina virtual Java (JVM) diferente. Portanto, o objeto remoto pode estar em uma JVM diferente no mesmo computador ou em um host remoto, conectado por uma rede. Essa situação é mostrada na [Figura 3.29](#).

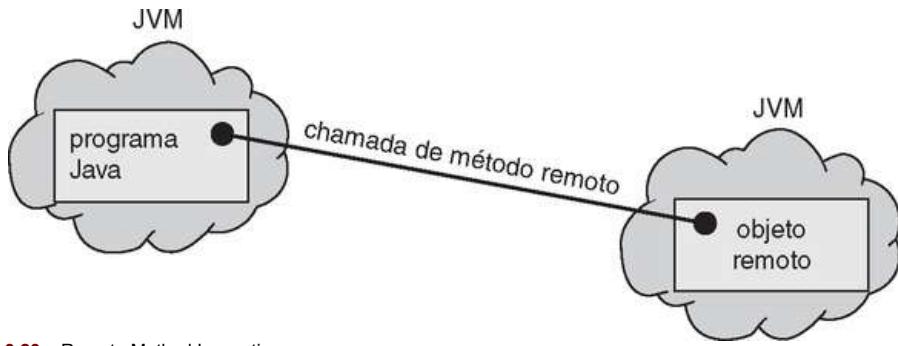


FIGURA 3.29 Remote Method Invocation.

As diferenças fundamentais entre RMI e RPC são duas. Primeiro, as RPCs admitem a programação por procedimentos, na qual somente procedimentos ou funções remotas podem se chamadas. Por outro lado, a RMI é baseada em objeto: ela admite a invocação de métodos em objetos remotos. Segundo, os parâmetros dos procedimentos remotos são estruturas de dados comuns em RPC; com a RMI, é possível passar tipos de dados primitivos (por exemplo, int, boolean), bem como objetos como parâmetros aos métodos remotos. Permitindo que um programa Java invoque métodos em objetos remotos, a RMI permite que os usuários desenvolvam aplicações Java distribuídas por uma rede.

Para tornar os métodos remotos transparentes ao cliente e ao servidor, a RMI implementa o objeto remoto usando stubs e esqueletos. Um **stub** é um substituto para o objeto remoto; ele reside com o cliente. Quando um cliente invoca um método remoto, o stub para o objeto remoto é chamado. Esse stub no cliente é responsável por criar um **pacote** contendo o nome do método a ser invocado no servidor e os parâmetros encaminhados para o método. O stub, então, envia esse pacote ao servidor, onde o esqueleto para o objeto remoto o recebe. O **esqueleto** é responsável por desempacotar os parâmetros e invocar o método desejado no servidor. O esqueleto empacota, então, o valor de retorno (ou exceção, se houver) em um pacote e o retorna ao cliente. O stub desempacota o valor de retorno e o passa para o cliente.

Vejamos mais de perto como funciona esse processo. Suponha que um cliente queira invocar um método em um servidor de objetos remoto com uma assinatura `métodoRemoto(Objeto, Objeto)`, que retorna um valor booleano. O cliente executa a instrução `boolean val = server.métodoRemoto(A, B);`

A chamada a `métodoRemoto()` com os parâmetros A e B invoca o stub para o objeto remoto. O stub empacota os parâmetros A e B e o nome do método que deve ser invocado no servidor, depois envia esse pacote ao servidor. O esqueleto no servidor desempacota os parâmetros e invoca o método `métodoRemoto()`. A implementação real de `métodoRemoto()` reside no servidor. Quando o método é completado, o esqueleto empacota o valor booleano retornado de `métodoRemoto()` e envia esse valor de volta ao cliente. O stub desempacota esse valor de retorno e o passa ao cliente. O processo pode ser visto no diagrama de sequência da UML (Unified Modeling Language) mostrado na [Figura 3.30](#).

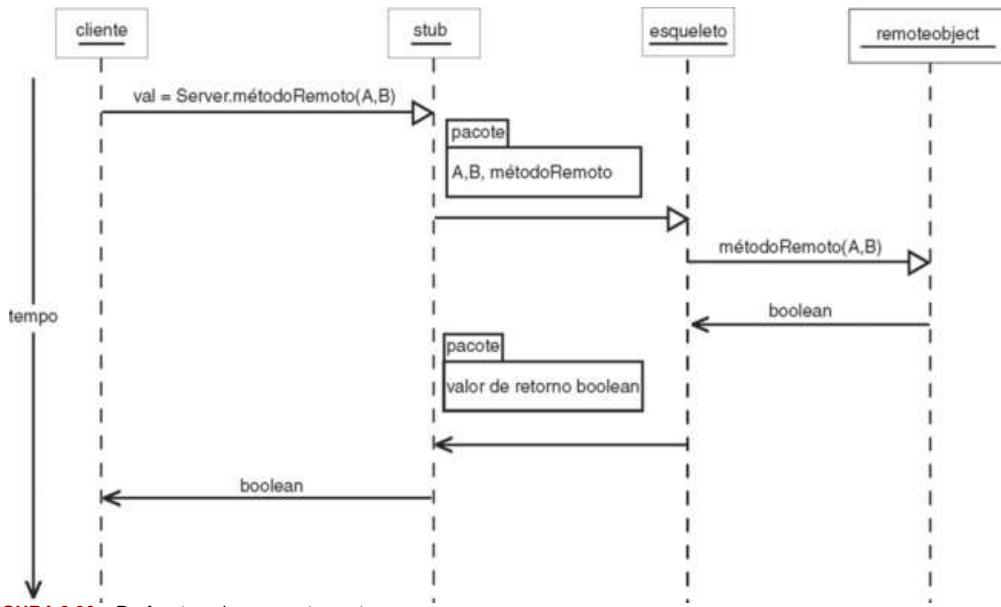


FIGURA 3.30 Parâmetros de empacotamento.

Felizmente, o nível de abstração que a RMI fornece torna os stubs e os esqueletos transparentes,

permitindo aos desenvolvedores Java escrever programas que invocam métodos distribuídos da mesma forma como invocariam métodos locais. Contudo, é fundamental entender algumas regras sobre o comportamento da passagem de parâmetros e valores de retorno:

- Se os parâmetros empacotados forem objetos **locais** (ou **não remotos**), eles serão passados por cópia, usando uma técnica conhecida como **serialização de objetos**, que permite que o estado de um objeto seja escrito em um fluxo de bytes. O único requisito para a serialização de objetos é que um objeto deva implementar a interface `java.io.Serializable`. A maioria dos objetos na API core Java implementa essa interface, permitindo que sejam usados com RMI.
- Objetos remotos são passados por referência. A passagem de um objeto por referência permite que o receptor altere o estado do objeto remoto, além de chamar seus métodos remotos.

Em nosso exemplo, se `A` é um objeto local e `B` um objeto remoto, `A` é serializado e passado por cópia, enquanto `B` é passado por referência. Isso permitirá que o servidor chame os métodos em `B` remotamente.

Em seguida, usando RMI, construimos uma aplicação que retorna a data e a hora correntes de forma similar ao programa baseado em sockets, mostrado na [Seção 3.6.1](#).

3.6.3.1 Objetos remotos

Antes da montagem de uma aplicação distribuída, precisamos definir inicialmente os objetos remotos necessários. Começamos declarando uma interface que especifique os métodos que podem ser invocados remotamente. No exemplo de um servidor de data, o método remoto será chamado `getDate()` e retornará um tipo `java.util.Date` contendo a data atual. Para providenciar objetos remotos, essa interface também precisará estender a interface `java.rmi.Remote`, que identifica os objetos implementando essa interface como sendo remota. Além disso, cada método declarado na interface precisa lançar uma exceção `java.rmi.RemoteException`. Para os objetos remotos, fornecemos a interface `RemoteDate`, mostrada na [Figura 3.31](#).

```
import java.rmi.*;
import java.util.Date;

public interface RemoteDate extends Remote
{
    public Date getDate() throws
    RemoteException;
}
```

FIGURA 3.31 A interface `RemoteDate`.

A classe que define o objeto remoto precisa implementar a interface `RemoteDate` ([Figura 3.32](#)). Além de definir o método `getDate()`, a classe também precisa estender `java.rmi.server.UnicastRemoteObject`. A extensão de `UnicastRemoteObject` permite a criação de um único objeto remoto que escuta as requisições da rede usando o esquema de sockets-padrão da RMI para a comunicação na rede. Essa classe também inclui um método `main()`. O método `main()` cria uma instância do objeto e registradores, com o registro da RMI executando no servidor com o método `rebind()`. Nesse caso, a instância do objeto se registra com o nome “`RMIDateObject`”. Observe também que precisamos criar um construtor-padrão para a classe `RemoteDateImpl`, e ele precisa lançar uma `RemoteException`, caso uma falha na comunicação ou na rede impeça a RMI de exportar o objeto remoto.

```

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.util.Date;

public class RemoteDateImpl extends UnicastRemoteObject
    implements RemoteDate
{
    public RemoteDateImpl( ) throws RemoteException { }

    public Date getDate( ) throws RemoteException {
        return new Date( );
    }

    public static void main(String[ ] args) {
        try {
            RemoteDate dateServer = new RemoteDateImpl( );

            // Associa essa instância do objeto ao nome "RMIDateObject"
            Naming.rebind("RMIDateObject", dateServer);
        }
        catch (Exception e) {
            System.err.println(e);
        }
    }
}

```

FIGURA 3.32 A implementação da interface RemoteDate.

3.6.3.2 Acesso ao objeto remoto

Quando um objeto é registrado no servidor, um cliente (mostrado na [Figura 3.33](#)) pode receber uma referência do proxy a esse objeto remoto a partir do registro da RMI em execução no servidor, usando o método estático `lookup()` na classe `Naming`. A RMI provê um esquema de pesquisa baseado em URL, usando a forma `rmi://servidor/nomeObjeto`, na qual o servidor é o nome IP (ou endereço) do servidor no qual o objeto remoto `nomeObjeto` reside e `nomeObjeto` é o nome do objeto remoto especificado pelo servidor no método `rebind()` (neste caso, `RMIDateObject`). Quando o cliente tiver a referência do substituto para o objeto remoto, ele chamará o método remoto `getDate()`, que retorna a data e hora atuais. Como os métodos remotos – bem como o método `Naming.lookup()` – podem lançar exceções, eles precisam ser colocados em blocos `try-catch`.

```

import java.rmi.*;

public class RMIClient
{
    static final String server = "127.0.0.1";

    public static void main(String args[])
    {
        try {
            String host = "rmi://" + server + "/RMIDateObject";

            RemoteDate dateServer = (RemoteDate)Naming.lookup(host);
            System.out.println(dateServer.getDate());
        }
        catch (Exception e) {
            System.err.println(e);
        }
    }
}

```

FIGURA 3.33 O cliente de RMI.

3.6.3.3 Execução dos programas

Agora, demonstramos as etapas necessárias para executar os programas de exemplo. Para simplificar, estamos considerando que todos os programas estão executando no host local - ou seja, no endereço IP 127.0.0.1. Entretanto, a comunicação ainda é considerada remota, pois os programas cliente e servidor estão executando cada um em sua própria JVM.

1. **Compile todos os arquivos-fonte.** Certifique-se de que o arquivo `RemoteDate.class` esteja no mesmo diretório de `RMIClient`.

2. **Inicie o registro e crie o objeto remoto.** Para iniciar o registro em plataformas UNIX, o usuário pode digitar
`rmiregistry &`
 Para o Windows, o usuário pode digitar
`start rmiregistry`

Esse comando inicia o registro com o qual o objeto remoto se registrará. Em seguida, crie uma instância do objeto remoto com

`java RemoteDateImpl`

Esse objeto remoto se registrará usando o nome `RMIDateObject`.

3. **Referencie o objeto remoto.** Digite a instrução

`java RMIClient`

na linha de comandos para iniciar o cliente. Esse programa apanhará uma referência substituta ao objeto remoto, chamada `RMIDateObject`, e invocará o método remoto `getDate()`.

3.6.3.4 RMI versus Sockets

Compare o programa cliente baseado em socket, mostrado na [Figura 3.27](#), com o cliente usando RMI, mostrado na [Figura 3.33](#). O cliente baseado em socket precisa gerenciar a conexão do socket, incluindo abertura e fechamento do socket, e estabelecer um `InputStream` para ler do socket. O projeto do cliente usando RMI é muito mais simples. Tudo o que ele precisa fazer é apanhar um substituto para o objeto remoto, que permita invocar o método remoto `getDate()` da mesma forma como invocaria um método local comum.

Esse exemplo ilustra a atração por técnicas como RPCs e RMI: elas fornecem aos desenvolvedores de sistemas distribuídos um mecanismo de comunicação que permite o projeto de programas distribuídos sem incorrer em custo adicional do gerenciamento de sockets.

3.7 Resumo

Um processo é um programa em execução. Enquanto um processo é executado, ele muda de estado. O estado de um processo é definido pela atividade atual desse processo. Cada processo pode estar em um dos seguintes estados: novo (new), pronto (ready), executando (running), aguardando (waiting) ou terminado (terminated). Cada processo é representado no sistema operacional por seu próprio bloco de controle de processo (PCB).

Um processo, quando não estiver sendo executado, é colocado em alguma fila de espera. Existem duas classes principais de filas em um sistema operacional: filas de requisição de E/S e a fila de prontos (ready queue). A fila de prontos contém todos os processos que estão prontos para serem executados e estão aguardando pela CPU. Cada processo é representado por um PCB, e os PCBs podem ser vinculados para formar uma fila de prontos. O escalonamento de longo prazo (tarefas) é a seleção de processos para que tenham permissão para disputar a CPU. Normalmente, o escalonamento de longo prazo é bastante influenciado por considerações de alocação de recursos, em especial a gerência de memória. O escalonamento de curto prazo (CPU) é a seleção de um processo a partir da fila de prontos.

Os sistemas operacionais precisam fornecer um mecanismo para os processos pai criarem novos processos filhos. O pai pode esperar que seus filhos terminem antes de prosseguir ou o pai e os filhos podem ser executados concorrentemente. Existem vários motivos para permitir a execução simultânea: compartilhamento de informações, agilidade da computação, modularidade e conveniência.

Os processos em execução no sistema operacional podem ser processos independentes ou processos cooperativos. Os processos cooperativos precisam de mecanismos de comunicação entre processos para se comunicarem entre si. A comunicação é obtida por meio de dois esquemas complementares: memória compartilhada e troca de mensagens. O método de memória compartilhada exige que os processos em comunicação compartilhem algumas variáveis. Os processos deverão trocar informações por meio dessas variáveis compartilhadas. Em um sistema de memória compartilhada, a responsabilidade por fornecer comunicação recai sobre os programadores de aplicações; o sistema operacional só precisa fornecer a memória compartilhada. O método de troca de mensagens permite aos processos trocarem mensagens. A responsabilidade por fornecer a comunicação pode ficar com o próprio sistema operacional. Esses dois esquemas não são mutuamente exclusivos, podendo ser usados ao mesmo tempo dentro de um único sistema operacional.

Três técnicas diferentes para a comunicação nos sistemas cliente-servidor são (1) sockets; (2) remote procedure calls (RPCs); ou (3) a remote method invocation (RMI) da Java. Um socket é definido como uma extremidade para comunicação. Uma conexão entre um par de aplicações consiste em um par de sockets, um em cada ponta do canal de comunicação. As RPCs são outra forma de comunicação distribuída. Uma RPC ocorre quando um processo (ou thread) chama um procedimento em uma aplicação remota. A RMI é a versão Java de uma RPC. A RMI permite que uma thread invoque um método em um objeto remoto da mesma forma como invocaria um método em um objeto local. A principal distinção entre RPCs e RMI é que, na RPC, os dados são passados a um procedimento remoto na forma de uma estrutura de dados comum, enquanto a RMI permite que objetos sejam passados nas chamadas ao método remoto.

Exercícios práticos

- 3.1. O Palm OS não oferece um meio de processamento concorrente. Discuta três complicações importantes que o processamento concorrente acrescenta a um sistema operacional.
- 3.2. O processador Sun UltraSPARC possui vários conjuntos de registradores. Descreva o que acontece quando ocorre uma troca de contexto se o novo contexto já estiver carregado em um dos conjuntos de registradores. O que acontece se o novo contexto estiver na memória, em vez de um conjunto de registradores, e todos os conjuntos de registradores estiverem em uso?
- 3.3. Quando um processo cria um novo processo usando a operação `fork()`, qual dos seguintes estados é compartilhado entre o processo pai e o processo filho?
 - a. Pilha
 - b. Heap
 - c. Segmentos de memória compartilhada
- 3.4. Com relação ao mecanismo RPF, considere a semântica “exatamente uma vez”. O algoritmo para implementar essa semântica é executado corretamente mesmo que a mensagem ACK de volta ao cliente for perdida devido a um problema na rede? Descreva a sequência de mensagens e discuta se “exatamente uma vez” ainda é preservada nesta situação.
- 3.5. Suponha que um sistema distribuído seja suscetível à falha no servidor. Que mecanismos seriam exigidos para garantir a semântica “exatamente uma vez” para a execução de RPCs?

Exercícios

3.6. Descreva as diferenças entre o escalonamento de curto, médio e longo prazos.

3.7. Descreva as ações realizadas por um kernel para a troca de contexto entre os processos.

3.8. Construa uma árvore de processos semelhante à [Figura 3.9](#). Para obter informações de processo para o sistema UNIX ou Linux, use o comando `ps -ael`. Use o comando `man ps` para obter mais informações sobre o comando `ps`. Em sistemas Windows, você terá que usar o gerenciador de tarefas.

3.9. Incluindo o processo pai inicial, quantos processos são criados pelo programa mostrado na [Figura 3.34](#)?

```
#include <stdio.h>
#include <unistd.h>

int main()
{
    /* cria um processo filho */
    fork();

    /* cria outro processo filho */
    fork();

    /* e cria outro */
    fork();

    return 0;
}
```

FIGURA 3.34 Quantos processos são criados?

3.10. Usando o programa na [Figura 3.35](#), identifique os valores de `pid` nas linhas A, B, C e D. (Suponha que os pids reais do pai e do filho sejam 2600 e 2063, respectivamente.)

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int main()
{
    pid_t pid, pid1;

    /* cria um processo filho */
    pid = fork();

    if (pid < 0) { /* houve erro */
        fprintf(stderr, "Falha na criação do processo");
        return 1;
    }
    else if (pid == 0) { /* processo filho */
        pid1 = getpid();
        printf("filho: pid = %d", pid); /* A */
        printf("filho: pid1 = %d", pid1); /* B */
    }
    else { /* processo pai */
        pid1 = getpid();
        printf("pai: pid = %d", pid); /* C */
        printf("pai: pid1 = %d", pid1); /* D */
        wait(NULL);
    }

    return 0;
}

```

FIGURA 3.35 Quais são os valores de pid?

- 3.11. Considere o mecanismo de RPC. Descreva as consequências indesejáveis que poderiam surgir se não houver imposição da semântica “no máximo uma vez” ou “exatamente uma vez”. Descreva possíveis usos para um mecanismo que não tenha nenhuma dessas garantias.
- 3.12. Explique as diferenças fundamentais entre RMI e RPCs.
- 3.13. Usando o programa mostrado na [Figura 3.36](#), explique qual será a saída na Linha A.

```

#include <sys/types.h>
#include <stdio.h>
#include <unistd.h>

int value = 5;

int main()
{
    pid_t pid;

    pid = fork();

    if (pid == 0) { /* processo filho */
        value += 15;
        return 0;
    }
    else if (pid > 0) { /* processo pai */
        wait(NULL);
        printf("PAI: valor = %d", value); /* LINHA A */
        return 0;
    }
}

```

FIGURA 3.36 Qual será a saída na Linha A?

- 3.14. Quais são os benefícios e as desvantagens de cada um dos seguintes? Considere o nível do sistema e o nível do programador.
- Comunicação síncrona e assíncrona
 - Buffering automático e explícito
 - Envio por cópia e envio por referência
 - Mensagens de tamanho fixo e tamanho variável

Problemas de programação

- 3.15. A [Seção 3.6.1](#) descreve os números de porta abaixo de 1024 como bem conhecidos; ou seja, eles oferecem serviços-padrão. A porta 17 é conhecida como o serviço *citação do dia*. Quando um cliente se conecta à porta 17 em um servidor, este responde com uma citação para aquele dia. Modifique o servidor de data mostrado na [Figura 3.26](#) de modo que ele forneça uma citação do dia ao invés da data atual. As citações deverão ser caracteres ASCII imprimíveis e conter menos de 512 caracteres, embora seja permitido usar várias linhas. Como a porta 17 é considerada bem conhecida e, portanto, indisponível, faça seu servidor escutar a porta 6017. O cliente de data mostrado na [Figura 3.27](#) poderá ser usado para ler as citações retornadas pelo seu servidor.
- 3.16. Um *haiku* é um poema de três linhas, no qual a primeira linha contém cinco sílabas, a segunda linha contém sete sílabas e a terceira contém cinco sílabas. Escreva um servidor de haiku que escute na porta 5575. Quando um cliente se conecta a essa porta, o servidor responde com um haiku. O cliente de data mostrado na [Figura 3.27](#) poderá ser usado para ler as linhas retornadas pelo seu servidor de haiku.
- 3.17. Escreva uma aplicação cliente-servidor usando sockets Java, permitindo que um cliente escreva uma mensagem (como uma `String`) em um socket. Um servidor lerá essa mensagem, contará o número de caracteres e dígitos na mensagem e enviará essas duas contagens de volta ao cliente. O servidor escutará na porta 6100. O cliente poderá obter a mensagem em `String` que deverá passar ao servidor ou pela linha de comando ou usando um prompt para o usuário. Uma estratégia para enviar as duas contagens de volta ao cliente é que o servidor construa um objeto contendo
- A mensagem que ele recebe do cliente
 - Uma contagem do número de caracteres na mensagem
 - Uma contagem do número de dígitos na mensagem

Esse objeto pode ser modelado usando a seguinte interface:

```
public interface Message
{
    // define as contagens de caracteres e dígitos
    public void setCounts();

    // retorna o número de caracteres
    public int getCharacterCount();

    // retorna o número de dígitos
    public int getDigitCount();
}
```

O servidor lerá a `String` do socket, construirá um novo objeto que implementa a interface `Message`, contará o número de caracteres e dígitos na `String` e escreverá o conteúdo do objeto `Message` no socket. O cliente enviará uma `String` ao servidor e esperará que o servidor responda com uma `Message` contendo a contagem do número de caracteres e dígitos na mensagem. A comunicação pela conexão de socket exigirá a obtenção do `InputStream` e `OutputStream` para o socket.

Objetos que são escritos ou lidos de um `OutputStream` ou `InputStream` precisam ser serializados e, portanto, deverão implementar a interface `java.io.Serializable`. Essa interface é conhecida como interface **marcadora**, significando que ela não possui realmente método algum que deva ser implementado; basicamente, quaisquer objetos que implementem essa interface podem ser usados com um `InputStream` ou um `OutputStream`. Para esta tarefa, você criará um objeto chamado `MessageImpl` que implementa `java.io.Serializable` e a interface `Message` mostrada anteriormente.

A serialização de um objeto requer a obtenção de um `java.io.ObjectOutputStream` e depois a escrita do objeto usando o método `writeObject()` na classe `ObjectOutputStream`. Assim, a atividade do servidor será organizada mais ou menos da seguinte forma:

- Ler a string do socket
- Construir um novo objeto `MessageImpl` e contar o número de caracteres e dígitos na mensagem
- Obter o `ObjectOutputStream` para o socket e escrever o objeto `MessageImpl` nesse fluxo de saída

A leitura de um objeto serializado requer a obtenção de um `java.io.ObjectInputStream` e a leitura do objeto serializado usando o método `readObject()` na classe `java.io.ObjectInputStream`. Portanto, a atividade do cliente será arrumada mais ou menos assim:

- Escrever a string no socket
- Obter o `ObjectInputStream` do socket e ler o objeto `MessageImpl`. Consulte outros detalhes na API Java.

- 3.18. Escreva uma aplicação RMI que permita que um cliente abra e leia um arquivo residindo em um servidor remoto. A interface para acessar o objeto de arquivo remoto se parece com

```

import java.rmi.*;

public interface RemoteFileObject extends Remote
{
    public abstract void open(String fileName)
        throws RemoteException;

    public abstract String readLine()
        throws RemoteException;

    public abstract void close()
        throws RemoteException;
}

```

Ou seja, o cliente abrirá o arquivo remoto usando o método `open()`, onde o nome do arquivo que está sendo aberto é fornecido como um parâmetro. O arquivo será acessado por meio do método `readLine()`. Esse método é implementado de modo semelhante ao método `readLine()` na classe `java.io.BufferedReader` da API Java. Ou seja, ele lerá e retornará uma linha de texto que termina com um line feed (`\n`), um carriage return (`\r`) ou um carriage return seguido imediatamente por um line feed. Quando o final do arquivo tiver sido alcançado, `readLine()` retornará o valor `null`. Quando o arquivo tiver sido lido, o cliente fechará o arquivo usando o método `close()`. Para simplificar, vamos considerar que o arquivo que está sendo lido é um fluxo de caracteres (texto). O programa cliente só precisa exibir o arquivo no console (`System.out`).

Uma questão a ser discutida refere-se ao tratamento de exceções. O servidor terá que implementar os métodos esboçados na interface `RemoteFileObject` usando métodos de E/S padrão fornecidos na API Java, sendo que a maioria levanta uma `java.io.IOException`. Entretanto, os métodos a serem implementados na interface `RemoteFileObject` são declarados para levantar uma `RemoteException`. Talvez o modo mais fácil de lidar com essa situação no servidor seja colocar as chamadas apropriadas aos métodos de E/S padrão usando `try-catch` para `java.io.IOException`. Se houver essa exceção, capture-a e depois levante-a novamente como uma `RemoteException`. O código pode se parecer com este:

```

try {
    . . .
}
catch (java.io.IOException ioe) {
    throw new RemoteException("Exceção de
E/S",ioe);
}

```

Você pode tratar de uma `java.io.FileNotFoundException` de modo semelhante.

Projetos de programação

Criando uma interface de shell usando Java

Este projeto consiste em modificar um programa Java de modo que ele sirva como uma interface de shell que aceita comandos do usuário e depois executa cada comando em um processo separado, externo à máquina virtual Java.

Visão geral

Uma interface de shell fornece ao usuário um prompt, depois do qual o usuário entra com o próximo comando. O exemplo a seguir ilustra o prompt `jsh>` e o próximo comando do usuário: `cat Prog.java`. Esse comando exibe o arquivo `Prog.java` no terminal usando o comando `cat` do UNIX.

```
jsh> cat Prog.java
```

Talvez a técnica mais fácil para implementar uma interface de shell seja fazer o programa primeiro ler o que o usuário insere na linha de comandos (aqui, `cat Prog.java`) e depois criar um processo externo separado que executa o comando. Criamos um processo separado usando o objeto `ProcessBuilder()`, conforme ilustrado na [Figura 3.13](#). Em nosso exemplo, esse processo separado é externo à JVM, e inicia a execução quando seu método `run()` é invocado.

Um programa Java que fornece as operações básicas de um shell da linha de comandos é fornecido na [Figura 3.37](#). O método `main()` apresenta o prompt `jsh>` (de java shell) e espera para ler a entrada do usuário. O programa termina quando o usuário pressiona `<Control><C>`.

```
import java.io.*;

public class SimpleShell
{
    public static void main(String[ ] args) throws java.
        io.IOException {
        String commandLine;
        BufferedReader console = new BufferedReader
            (new InputStreamReader(System.in));

        // Saímos com <control><C>
        while (true) {
            // Lê o que o usuário digitou
            System.out.print("jsh>");
            commandLine = console.readLine( );

            // se o usuário digitou Return, entra em loop
            // novamente
            if (commandLine.equals(""))
                continue;

            /** As etapas são:
            (1) desmembrar a entrada para obter o comando e quaisquer parâmetros
            (2) criar um objeto ProcessBuilder
            (3) iniciar o processo
            (4) obter o fluxo de saída
            (5) enviar o conteúdo retornado pelo comando */
        }
    }
}
```

FIGURA 3.37 Esboço de shell simples.

Esse projeto é organizado em três partes: (1) criar o processo externo e executar o comando nesse processo; (2) modificar o shell para permitir a mudança de diretórios; e (3) incluir um recurso de histórico.

Parte 1: Criando um processo externo

A primeira parte desse projeto é modificar o método `main()` na [Figura 3.37](#), de modo que um processo externo seja criado e execute o comando especificado pelo usuário. Inicialmente, o comando precisa ser desmembrado em parâmetros separados e passado ao construtor para o objeto `ProcessBuilder`. Por exemplo, se o usuário entrar com o comando

jsh> cat Prog.java

os parâmetros serão (1) `cat` e (2) `Prog.java`, e esses parâmetros precisam ser passados ao construtor `ProcessBuilder`. Talvez a estratégia mais fácil seja usar o construtor com a seguinte assinatura:

```
public ProcessBuilder (List<String> command)
```

Uma `java.util.ArrayList` – que implementa a interface `java.util.List` – pode ser usada neste caso, onde o primeiro elemento da lista é `cat` e o segundo elemento é `Prog.java`. Essa é uma estratégia especialmente útil, pois o número de argumentos passados aos comandos UNIX pode variar (o comando `cat` aceita um argumento; o comando `cp` aceita dois, e assim por diante).

Se o usuário inserir um comando inválido, o método `start()` na classe `ProcessBuilder` gerará uma `java.io.IOException`. Se isso ocorrer, seu programa deverá enviar uma mensagem de erro apropriada e continuar esperando outro comando do usuário.

Parte 2: Mudando diretórios

A próxima tarefa é modificar o programa na [Figura 3.37](#) de modo que ele mude de diretório. Nos sistemas UNIX, encontramos o conceito do *diretório de trabalho atual*, que é o diretório em que você se encontra atualmente. O comando `cd` permite que um usuário mude o diretório atual. Sua interface de shell precisa dar suporte a esse comando. Por exemplo, se o diretório atual for `/usr/tom` e o usuário digitar `cd music`, o diretório atual se torna `/usr/tom/music`. Comandos subsequentes se relacionam com esse diretório atual. Por exemplo, a entrada de `ls` mostrará todos os arquivos em `/usr/tom/music`.

A classe `ProcessBuilder` fornece o seguinte método para alterar o diretório de trabalho:

```
public ProcessBuilder directory  
(File directory)
```

Quando o método `start()` de um processo subsequente for invocado, o novo processo o usará como diretório de trabalho atual. Por exemplo, se um processo com um diretório de trabalho atual `/usr/tom` invocar o comando `cd music`, os processos subsequentes deverão definir seus diretórios de trabalho para `/usr/tom/music` antes de iniciar a execução. É importante observar que o seu programa precisa primeiro certificar-se de que o novo caminho que está sendo especificado é um diretório válido. Se não, seu programa deverá enviar uma mensagem de erro apropriada.

Se o usuário digitar o comando `cd`, mude o diretório de trabalho atual para o diretório `home` do usuário. O diretório `home` para o usuário atual pode ser obtido pela chamada do método estático `getProperty()` na classe `System`, da seguinte maneira:

```
System.getProperty("user.dir");
```

Parte 3: Incluindo um recurso de histórico

Muitos shells do UNIX fornecem um recurso de *histórico(history)* que permite que os usuários vejam o histórico dos comandos que eles entraram e executem novamente um comando a partir desse histórico. O histórico inclui todos os comandos que foram digitados pelo usuário desde que o shell foi iniciado. Por exemplo, se o usuário entrasse com o comando `history` e visse como saída:

```
0 pwd  
1 ls -l  
2 cat Prog.java
```

o histórico listaria `pwd` como primeiro comando digitado, `ls -l` como segundo comando, e assim por diante. Modifique seu programa de shell de modo que os comandos sejam inseridos em um histórico. (Dica: A `java.util.ArrayList` fornece uma estrutura de dados útil para armazenar esses comandos.)

Seu programa precisa permitir que os usuários executem novamente os comandos a partir de seu histórico, realizando as duas técnicas a seguir:

1. Quando o usuário digitar o comando `history`, você imprimirá o conteúdo do histórico de comandos que foram digitados no shell, junto com os números de comando.
2. Quando o usuário digitar `!!`, execute o comando anterior no histórico. Se não houver comando anterior, envie uma mensagem de erro apropriada.
3. Quando o usuário digitar `!<valor inteiro i>`, execute o *i*-ésimo comando no histórico. Por exemplo, digitar `!4` executará novamente o quarto comando no histórico de comandos. Não se esqueça de realizar a verificação de erro apropriada para garantir que o valor inteiro seja um número válido no histórico de comandos.

Notas bibliográficas

A comunicação entre processos no sistema RC 4000 é discutida por [Brinch-Hansen \[1970\]](#). [Schlichting e Schneider \[1982\]](#) discutem sobre as primitivas de troca de mensagens assíncrona. O recurso de IPC implementado no nível do usuário é descrito por [Bershad e outros \[1990\]](#).

Os detalhes da comunicação entre processos nos sistemas UNIX são apresentados por [Gray \[1997\]](#). [Barrera \[1991\]](#) e [Vahalia \[1996\]](#) apresentam a comunicação entre processos no sistema Mach. [Russinovich e Solomon \[2005\]](#), [Solomon e Russinovich \[2000\]](#) e [Stevens \[1999\]](#) esboçam a comunicação entre processos no Windows 2003, Windows 2000 e no UNIX, respectivamente. [Hart \[2005\]](#) aborda a programação de sistemas Windows com detalhes. A implementação de RPCs é discutida por [Birrell e Nelson \[1984\]](#). Um projeto de um mecanismo de RPC confiável é descrito por [Shrivastava e Panzieri \[1982\]](#), e [Tay e Ananda \[1990\]](#) apresentam um estudo sobre RPCs. [Stankovic \[1982\]](#) e [Staunstrup \[1982\]](#) discutem a respeito da comunicação por chamadas de procedimento *versus* troca de mensagens. [Harold \[2005\]](#) fornece uma explicação sobre programação com sockets em Java, e [Grosso \[2002\]](#) aborda a RMI da Java.

CAPÍTULO 4

Threads

O modelo de processo apresentado no [Capítulo 3](#) considerava que um processo era um programa em execução com uma única thread de controle. A maioria dos sistemas operacionais modernos agora fornece recursos que permitem que um processo tenha diversas threads de controle. Este capítulo apresenta muitos conceitos associados aos sistemas computadorizados dotados de múltiplas threads (multithreaded), incluindo APIs para Pthreads, Win32 e bibliotecas threads em Java. Examinamos muitas questões relacionadas com a programação multithreads e seu efeito sobre o projeto dos sistemas operacionais. Finalmente, exploramos como os sistemas operacionais modernos Windows XP e Linux admitem threads no nível do kernel.

OBJETIVOS DO CAPÍTULO

- Introduzir a noção de thread - uma unidade fundamental de utilização da CPU, que forma a base dos sistemas computadorizados multithreaded.
- Discutir as APIs para as bibliotecas de threads Pthreads, Win32 e Java.
- Examinar questões relacionadas com a programação multithreading.

4.1 Visão geral

Uma thread é uma unidade básica de utilização de CPU; ela compreende um ID de thread, um contador de programa, um conjunto de registradores e uma pilha. Além disso, compartilha com outras threads pertencentes ao mesmo processo sua seção de código, seção de dados e outros recursos do sistema operacional, como arquivos abertos e sinais. Um processo tradicional (ou pesado) possui uma única thread de controle. Se o processo tiver múltiplas threads de controle, ele poderá fazer mais de uma tarefa ao mesmo tempo. A Figura 4.1 ilustra a diferença entre um processo tradicional **dotado de uma única thread (single-threaded)** e um processo **dotado de múltiplas threads (multithreaded)**.

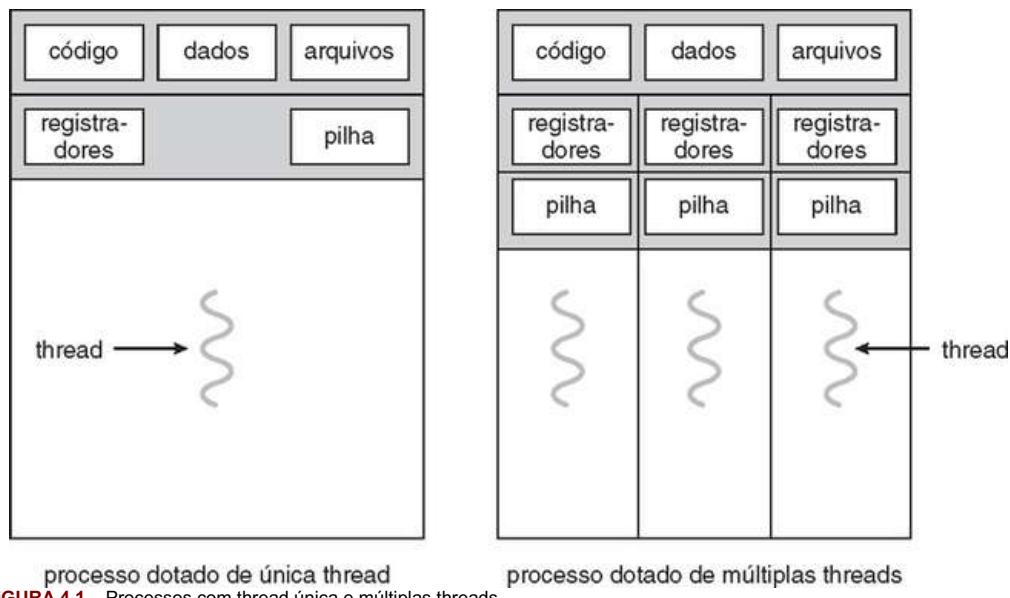


FIGURA 4.1 Processos com thread única e múltiplas threads.

4.1.1 Usos

Muitos pacotes de software executados nos PCs desktop modernos são dotados de múltiplas threads. Uma aplicação normalmente é implementada como um processo separado, com várias threads de controle. Um navegador Web pode ter uma thread exibindo imagens ou texto enquanto outra thread recebe dados da rede, por exemplo. Um processador de textos pode ter uma thread para exibir gráficos, outra thread para ler os toques de tecla do usuário e uma terceira thread para realizar a verificação ortográfica e gramatical em segundo plano.

Em certas situações, uma única aplicação pode ter de realizar diversas tarefas semelhantes. Por exemplo, um servidor Web aceita requisições do cliente para páginas Web, imagens, sons, e assim por diante. Um servidor Web ocupado pode ter vários clientes (talvez milhares deles) acessando-o concorrentemente. Se o servidor Web fosse executado como um processo tradicional, dotado de única thread, ele só poderia atender a um cliente de cada vez e este teria de esperar muito para que sua requisição fosse atendida.

Uma solução é fazer o servidor ser executado como um único processo que aceita requisições. Quando o servidor recebe uma requisição, ele cria um processo separado para atender a essa requisição. Na verdade, esse método de criação de processo já era comum antes de as threads se tornarem populares. Porém, a criação de processos é demorada e exige muitos recursos. Se o novo processo tiver de realizar as mesmas tarefas do processo existente, por que incorrer em todo esse custo adicional? Em geral, é mais eficaz que um processo contendo múltiplas threads. Se o processo servidor Web for de múltiplas threads, ele criará uma thread separada, que escutará as requisições do cliente. Quando uma requisição for feita, em vez de criar outro processo, ele criará outra thread para atender a requisição e continuará escutando requisições adicionais. Isso é ilustrado na Figura 4.2.

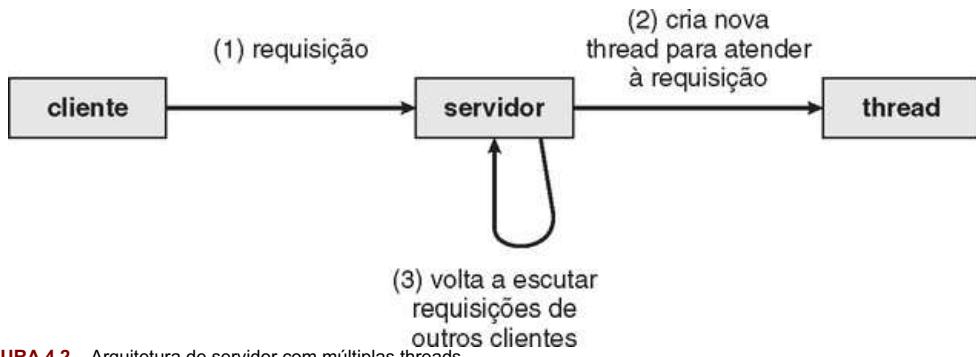


FIGURA 4.2 Arquitetura de servidor com múltiplas threads.

As threads também desempenham um papel vital nos sistemas com remote procedure calls (RPC). Lembre-se, do Capítulo 3, de que as RPCs permitem a comunicação entre processos, fornecendo um mecanismo de comunicação similar às chamadas comuns de função ou procedimento. Os servidores de RPC costumam ser multithreads. Quando um servidor recebe uma mensagem, ele atende a essa mensagem usando uma thread separada. Isso permite que o servidor trate de várias requisições simultâneas. Os sistemas de RMI da Java funcionam de modo semelhante.

Por fim, muitos kernels de sistema operacional agora são multithreads; diversos threads operam no kernel, e cada thread realiza uma tarefa específica, como o gerenciamento de dispositivos ou o tratamento de interrupções. Por exemplo, o Solaris cria um conjunto de threads no kernel especificamente para o tratamento de interrupções; o Linux utiliza uma thread no kernel para gerenciar a quantidade de memória livre nos sistemas.

4.1.2 Benefícios

Os benefícios da programação multithread podem ser divididos em quatro categorias principais:

1. **Responsividade.** O uso de multithreads em uma aplicação interativa pode permitir que um programa continue funcionando mesmo que parte dele esteja bloqueada ou realizando uma operação longa, aumentando assim a responsividade ao usuário. Por exemplo, um navegador Web multithreads pode permitir a interação do usuário em uma thread enquanto uma imagem é carregada em outra thread.
2. **Compartilhamento de recursos.** Os processos só podem compartilhar recursos por meio de técnicas como memória compartilhada ou troca de mensagens. Essas técnicas precisam ser arranjadas explicitamente pelo programador. Porém, como padrão, as threads compartilham memória e os recursos do processo ao qual pertencem. O benefício do compartilhamento de código e de dados é que isso permite que uma aplicação tenha várias threads de atividades diferentes dentro do mesmo espaço de endereços.
3. **Economia.** A alocação de memória e recursos para a criação de processos é dispendiosa. Como as threads compartilham recursos do processo ao qual pertencem, é mais econômico criar e trocar o contexto das threads. A avaliação empírica da diferença no custo adicional pode ser difícil, mas, em geral, é muito mais demorado criar e gerenciar processos do que threads. No Solaris, por exemplo, a criação de um processo é cerca de trinta vezes mais lenta do que a criação de uma thread, e a troca de contexto é cerca de cinco vezes mais lenta.
4. **Escalabilidade.** Os benefícios do uso de multithreads podem ser muito maiores em uma arquitetura multiprocessada, na qual as threads podem ser executadas em paralelo nos diferentes processadores. Um processo dotado de única thread só pode ser executado em um processador, não importa quantas estejam à disposição. O uso de múltiplas threads em uma máquina de múltiplas CPUs aumenta o paralelismo. Exploramos essa questão com mais profundidade na próxima seção.

4.1.3 Programação multicore

Uma tendência recente no projeto de sistemas tem sido colocar vários núcleos de computação (cores) em um único chip. Cada um desses núcleos se parece, para o sistema operacional, como um processador separado (Seção 1.3.2). A programação multithreaded oferece um mecanismo para o uso mais eficaz dos múltiplos núcleos e para melhorar a concorrência. Considere uma aplicação com quatro threads. Em um sistema com um único núcleo de computação, a concorrência significa que a execução das threads será intercalada com o tempo (Figura 4.3), pois o núcleo de processamento só pode executar uma thread por vez. No entanto, em um sistema com múltiplos núcleos, a concorrência significa que as threads podem ser executadas em paralelo, pois o sistema pode atribuir uma thread separada a cada núcleo (Figura 4.4).

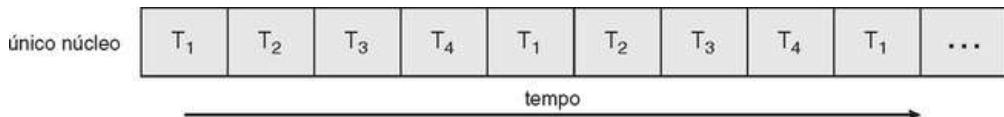


FIGURA 4.3 Execução concorrente em um sistema de único núcleo (single-core).

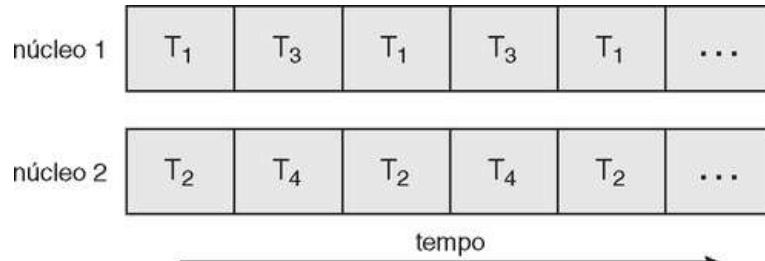


FIGURA 4.4 Execução paralela em um sistema de múltiplos núcleos (multicore).

A tendência em direção aos sistemas multicore pressionou os projetistas de sistemas, bem como os programadores de aplicação, para que façam melhor uso dos múltiplos núcleos de computação. Os projetistas de sistemas operacionais precisam escrever algoritmos de escalonamento que utilizam os múltiplos núcleos de processamento a fim de permitir a execução paralela, mostrada na [Figura 4.4](#). Para programadores de aplicação, o desafio é modificar os programas existentes, bem como o projeto de novos programas que são multithreaded, para que tirem proveito dos sistemas multicore. Em geral, cinco áreas apresentam desafios na programação de sistemas multicore:

1. **Divisão de atividades.** Envolve o exame das aplicações para localizar áreas que podem ser divididas em tarefas separadas, concorrentes e, dessa forma, serem executadas em paralelo nos núcleos individuais.
2. **Equilíbrio.** Ao identificar tarefas que possam ser executadas em paralelo, os programadores também precisam garantir que as tarefas realizem um trabalho igual com o mesmo valor. Em alguns casos, determinada tarefa pode não contribuir com tanto valor para o processo geral quanto outras tarefas; o uso de um núcleo de computação separado para executar essa tarefa pode não compensar o custo dessa implementação.
3. **Separação de dados.** Assim como as aplicações são divididas em tarefas separadas, os dados acessados e manipulados pelas tarefas também precisam ser divididos em núcleos separados.
4. **Dependência de dados.** Os dados acessados pelas tarefas precisam ser examinados em busca de dependências entre duas ou mais tarefas. Em casos onde uma tarefa depende dos dados de outra, os programadores precisam garantir que a execução das tarefas seja sincronizada, levando em consideração a dependência de dados. Examinamos essas estratégias no [Capítulo 6](#).
5. **Teste e depuração.** Quando um programa está sendo executado em paralelo, por múltiplos núcleos, existem muitos caminhos de execução diferentes. O teste e a depuração desses programas concorrentes são inherentemente mais difíceis do que o teste e a depuração de aplicações de única thread.

Devido a esses desafios, muitos desenvolvedores de software argumentam que o advento de sistemas multicore exigirá uma técnica totalmente nova para o projeto de sistemas de software no futuro.

4.2 Modelos de múltiplas threads (multithreading)

Nossa discussão até aqui tratou das threads em um sentido genérico. No entanto, o suporte para as threads pode ser fornecido no nível do usuário para **threads de usuário** ou, pelo kernel, para **threads de kernel**. As threads de usuário são admitidas acima do kernel e gerenciadas sem o suporte do kernel, enquanto as threads de kernel são admitidas e gerenciadas diretamente pelo sistema operacional. Quase todos os sistemas operacionais contemporâneos – incluindo Windows XP, Windows Vista, Linux, Mac OS X, Solaris e Tru64 UNIX (originalmente, Digital UNIX) – admitem threads de kernel.

Por fim, é preciso que haja um relacionamento entre as threads de cada usuário e as de kernel. Nesta seção, examinamos três formas comuns de estabelecer esse relacionamento.

4.2.1 Modelo muitos para um

O modelo muitos para um (Figura 4.5) associa muitas threads no nível do usuário a uma thread de kernel. O gerenciamento de threads é feito pela biblioteca threads no espaço do usuário, de modo que é eficiente; mas o processo inteiro será bloqueado se uma thread fizer uma chamada de sistema bloqueante. Além disso, como somente uma thread pode acessar o kernel por vez, várias threads não podem ser executadas em paralelo em multiprocessadores. **Green threads** – uma biblioteca threads disponível para o Solaris – utiliza esse modelo, assim como **GNU Portable Threads**.

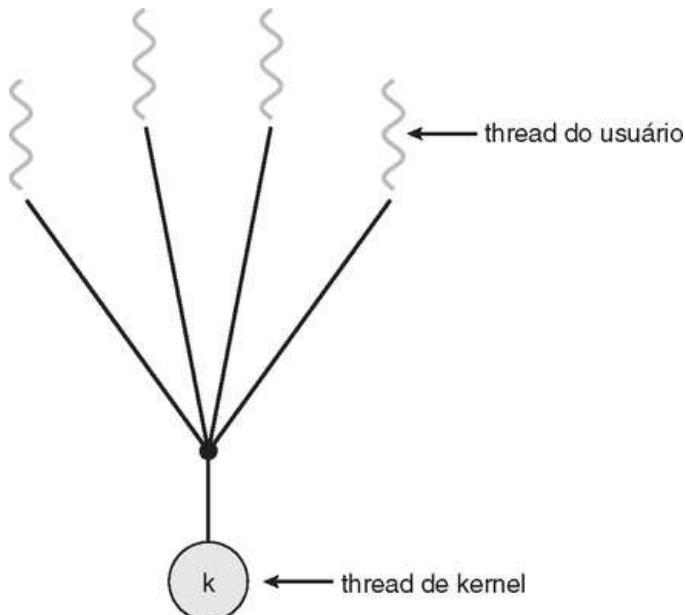


FIGURA 4.5 Modelo muitos para um.

4.2.2 Modelo um para um

O modelo um para um (Figura 4.6) associa a thread de cada usuário a uma thread de kernel. Ele provê maior concorrência do que o modelo muitos para um, permitindo que outra thread seja executada quando uma thread faz uma chamada de sistema bloqueante; ele também permite que várias threads sejam executadas em paralelo em multiprocessadores. A única desvantagem desse modelo é que a criação de uma thread de usuário requer a criação de uma thread de kernel correspondente. Como o custo adicional da criação de threads do kernel pode prejudicar o desempenho de uma aplicação, a maioria das implementações desse modelo restringe o número de threads admitidos pelo sistema. O Linux, juntamente com a família de sistemas operacionais Windows, implementa o modelo um para um.

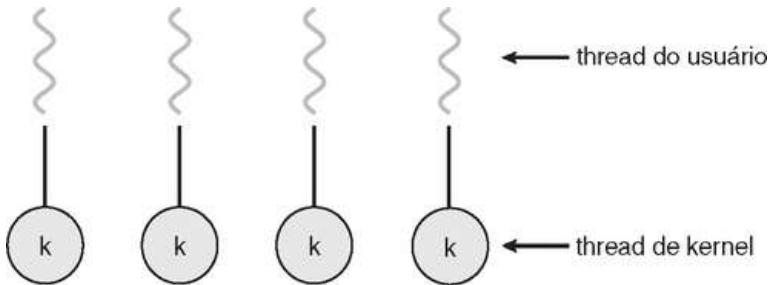


FIGURA 4.6 Modelo um para um.

4.2.3 Modelo muitos para muitos

O modelo muitos para muitos (Figura 4.7) multiplexa muitas threads no nível do usuário para um número menor ou igual de threads de kernel. O número de threads de kernel pode ser específico para determinada aplicação ou para determinada máquina (uma aplicação pode receber mais threads de kernel em um sistema multiprocessado do que em um monoprocessado). Enquanto o modelo muitos para um permite que o desenvolvedor crie quantas threads de usuário desejar, a verdadeira concorrência não é obtida porque o kernel só pode escalonar uma thread de cada vez. O modelo um para um permite maior concorrência, mas o desenvolvedor precisa ter o cuidado de não criar muitas threads dentro de uma aplicação (e, em alguns casos, pode estar limitado no número de threads que pode criar). O modelo muitos para muitos não sofre de nenhuma dessas limitações: os desenvolvedores podem criar quantas threads forem necessárias, e as threads de kernel correspondentes podem ser executadas em paralelo em um sistema multiprocessado. Além disso, quando uma thread realiza uma chamada de sistema bloqueante, o kernel pode escalonar outra thread.

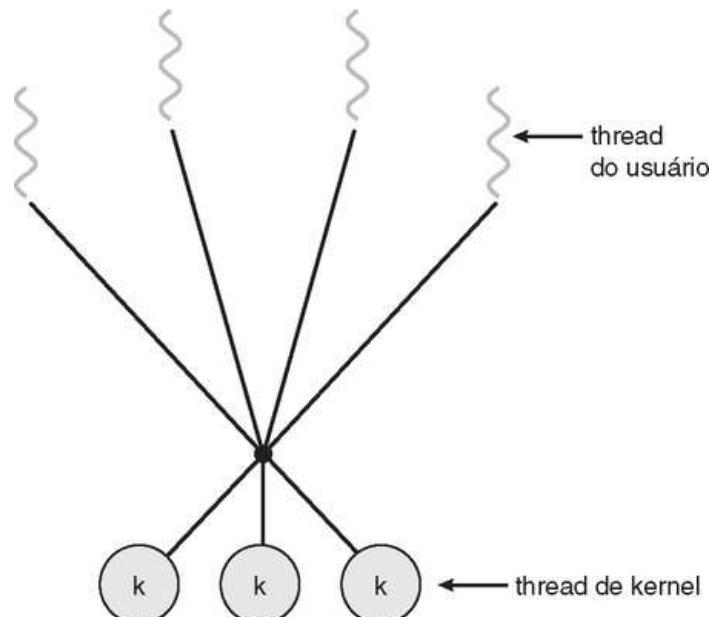


FIGURA 4.7 Modelo muitos para muitos.

Uma variação popular do modelo muitos para muitos ainda multiplexa muitas threads no nível do usuário para um número menor ou igual de threads de kernel, mas também permite que uma thread no nível do usuário esteja ligada a uma thread de kernel. Essa variação, às vezes chamada *modelo de dois níveis (two-level model)* (Figura 4.8), é aceita por sistemas operacionais como o IRIX, o HP-UX e o Tru64 UNIX. O sistema operacional Solaris admitia o modelo de dois níveis nas versões anteriores ao Solaris 9. Contudo, a partir do Solaris 9, esse sistema utiliza o modelo um para um.

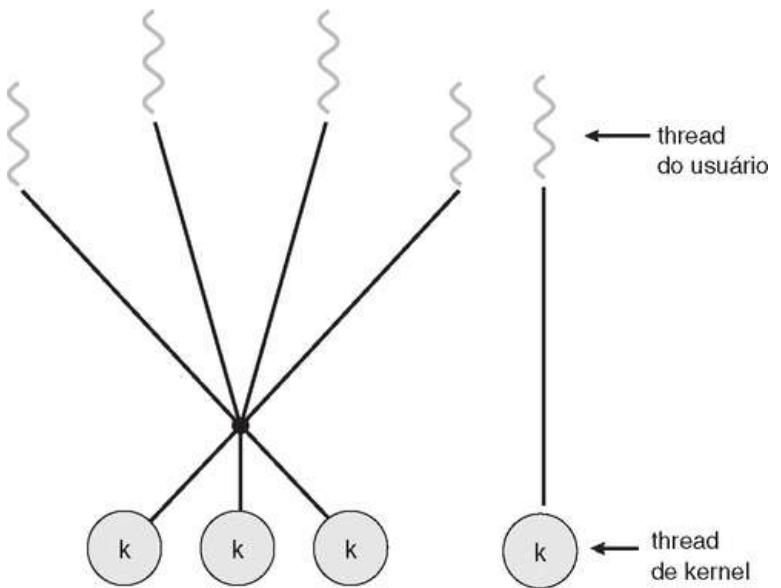


FIGURA 4.8 Modelo de dois níveis (two-level model)

4.3 Bibliotecas de threads

Uma **biblioteca de threads** fornece ao programador uma API para a criação e o gerenciamento de threads. Existem duas formas principais de implementar uma biblioteca de threads. A primeira técnica é fornecer uma biblioteca inteiramente no espaço do usuário, sem suporte do kernel. Todo o código e as estruturas de dados para a biblioteca existem no espaço do usuário. Isso significa que a chamada de uma função na biblioteca resulta em uma chamada de função local no espaço do usuário, e não em uma chamada de sistema.

A segunda técnica é implementar uma biblioteca no nível do kernel, com o suporte direto do sistema operacional. Nesse caso, o código e as estruturas de dados para a biblioteca existem no espaço do kernel. A chamada de uma função na API para a biblioteca em geral resulta em uma chamada de sistema ao kernel.

Três bibliotecas de threads principais estão em uso atualmente: (1) POSIX Pthreads; (2) Win32; e (3) Java. O Pthreads, uma extensão do padrão POSIX para threads, pode ser fornecido como biblioteca no nível do usuário ou do kernel. A biblioteca de threads Win32 é uma biblioteca no nível do kernel disponível no sistema Windows. A API de threads Java permite a criação e a gerência de threads diretamente nos programas em Java. Contudo, como na maioria das instâncias a Java Virtual Machine (JVM) está executando em cima de um sistema operacional hospedeiro, a API de threads Java normalmente é implementada usando uma biblioteca threads disponível no sistema hospedeiro. Isso significa que, nos sistemas Windows, threads Java normalmente são implementadas usando a API Win32; sistemas UNIX e Linux normalmente utilizam Pthreads.

No restante desta seção, descrevemos a criação básica de threads usando as bibliotecas de threads Pthreads e Win32. Abordamos as threads Java com mais detalhes na [Seção 4.4](#). Como exemplo ilustrativo dessas bibliotecas de threads, criamos um programa multithreads que realiza o somatório de um inteiro não negativo em uma thread separada, usando a função bem conhecida de somatório:

$$soma = \sum_{i=0}^N i$$

Por exemplo, se N fosse 5, essa função representaria o somatório de inteiros de 0 a 5, que é 15. Cada um dos três programas será executado com os limites superiores do somatório inseridos na linha de comandos; assim, se o usuário digitar 8, o resultado exibido será o somatório dos valores inteiros de 0 a 8.

4.3.1 Pthreads

O **Pthreads** refere-se ao padrão POSIX (IEEE 1003.1c) que define uma API para a criação e sincronismo de thread. Essa é uma *especificação* para o comportamento da thread, e não uma *implementação*. Os projetistas de sistema operacional podem implementar a especificação como desejarem. Diversos sistemas implementam a especificação Pthreads, incluindo Solaris, Linux, Mac OS X e Tru64 UNIX. Também existem implementações *shareware* em domínio público para os diversos sistemas operacionais Windows.

O programa em C mostrado na [Figura 4.9](#) demonstra a API Pthreads básica para a construção de um programa multithreads que calcula o somatório de um inteiro não negativo em uma thread separada. Em um programa Pthreads, threads separadas iniciam a execução em uma função específica. Na [Figura 4.9](#), essa é a função `runner()`. Quando esse programa é iniciado, uma única thread de controle é iniciada em `main()`. Após alguma inicialização, `main()` cria uma segunda thread que inicia o controle na função `runner()`. As duas threads compartilham a soma de dados global.

```

#include <pthread.h>
#include <stdio.h>

int sum; /* esses dados são compartilhados pela(s) thread(s) */
void *runner(void *param); /* a thread */

int main(int argc, char *argv[])
{
    pthread_t tid; /* o identificador da thread */
    pthread_attr_t attr; /* conjunto de atributos da thread */

    if (argc != 2) {
        fprintf(stderr,"uso: a.out <valor inteiro>\n");
        return -1;
    }
    if (atoi(argv[1]) < 0) {
        fprintf(stderr,"%d precisa ser >= 0\n",atoi(argv[1]));
        return -1;
    }

    /* apanha os atributos padrão */
    pthread_attr_init(&attr);
    /* cria a thread */
    pthread_create(&tid,&attr,runner,argv[1]);
    /* espera que a thread termine */
    pthread_join(tid,NULL);

    printf("sum = %d\n",sum);
}

/* A thread iniciará o controle nesta função */
void *runner(void *param)
{
    int i, upper = atoi(param);
    sum = 0;

    for (i = 1; i <= upper; i++)
        sum += i;

    pthread_exit(0);
}

```

FIGURA 4.9 Programa C multithreads usando a API Pthreads.

Analisemos detalhadamente esse programa. Todos os programas Pthreads precisam incluir o arquivo de cabeçalho `pthread.h`. A instrução `pthread_t tid` declara o identificador para a thread que criaremos. Cada thread possui um conjunto de atributos, incluindo informações de tamanho de pilha e escalonamento. A declaração `pthread_attr_t attr` representa os atributos para a thread. Definiremos os atributos na chamada de função `pthread_attr_init(&attr)`. Como não definimos quaisquer atributos explicitamente, usaremos os atributos-padrão fornecidos. (Na [Seção 5.4.2](#), discutiremos sobre alguns dos atributos de escalonamento fornecidos pela API Pthreads.) Uma thread separada é criada com a chamada de função `pthread_create()`. Além de passar o identificador e os atributos da thread para a thread, também passamos o nome da função onde a nova thread começará sua execução – nesse caso, a função `runner()`. Por último, passamos o parâmetro inteiro fornecido na linha de comandos, `argv[1]`.

Nesse ponto, o programa possui duas threads: a thread inicial (ou mãe) em `main()` e a thread de soma (ou filha) realizando o somatório na função `runner()`. Após a criação da thread soma, a thread mãe esperará até que a thread de soma termine, chamando a função `pthread_join()`. A thread de soma terminará quando chamar a função `pthread_exit()`. Quando a thread de soma tiver retornado, a thread mãe informará o valor dos dados compartilhados, `sum`.

4.3.2 Threads Win32

A técnica para criar threads usando a biblioteca threads Win32 é semelhante à técnica Pthreads de várias maneiras. Ilustramos a API de threads Win32 no programa C mostrado na [Figura 4.10](#). Observe que precisamos incluir o arquivo de cabeçalho `windows.h` quando usamos a API Win32.

```

#include <windows.h>
#include <stdio.h>
DWORD Soma; /* dados compartilhados pela(s) thread(s) */

/* a thread executa nesta função separada */
DWORD WINAPI Somatorio(LPVOID Param)
{
    DWORD Upper = *(DWORD*)Param;
    for (DWORD i = 0; i <= Upper; i++)
        Soma += i;
    return 0;
}

int main(int argc, char *argv[])
{
    DWORD ThreadId;
    HANDLE ThreadHandle;
    int Param;
    /* realiza alguma verificação de erro básica */
    if (argc != 2) {
        fprintf(stderr,"Um parâmetro inteiro é exigido\n");
        return -1;
    }
    Param = atoi(argv[1]);
    if (Param < 0) {
        fprintf(stderr,"Um inteiro >= 0 é exigido\n");
        return -1;
    }

    // cria a thread
    ThreadHandle = CreateThread(
        NULL, // atributos de segurança padrão
        0, // tamanho de pilha padrão
        Somatorio, // função de thread
        &Param, // parâmetro para a função de thread
        0, // flags de criação padrão
        &ThreadId); // retorna o identificador da thread

    if (ThreadHandle != NULL) {
        // agora espera que a thread termine
        WaitForSingleObject(ThreadHandle,INFINITE);

        // fecha o descritor da thread
        CloseHandle(ThreadHandle);

        printf("soma = %d\n",Soma);
    }
}

```

FIGURA 4.10 Programa C multithreads usando a API Win32.

Assim como na versão Pthreads mostrada na [Figura 4.9](#), os dados compartilhados pelas threads separadas – nesse caso, `Soma` – são declarados globalmente (o tipo de dados `DWORD` é um inteiro de 32 bits sem sinal). Também definimos a função `Somatorio()`, que deve ser executada em uma thread separada. Essa função recebe um ponteiro para um `void`, que a Win32 define como `LPVOID`. A thread que realiza essa função define o dado global `Soma` como o valor do somatório de 0 até o parâmetro passado a `Somatorio()`.

As threads são criadas na API Win32 usando a função `CreateThread()` e – assim como em Pthreads – um conjunto de atributos para a thread é passado a essa função. Esses atributos incluem informações de segurança, o tamanho da pilha e um flag que pode ser marcado para indicar se a thread deve ser iniciada em um estado suspenso. Nesse programa, usamos os valores default para esses atributos (que não definem inicialmente a thread para um estado suspenso e, em vez disso, tornam-na elegível para ser executada pelo escalonador da CPU). Quando a thread de somatório é criada, o pai precisa esperar que ela termine antes de enviar o valor de `Soma`, poitthread de somatório. Lembre-se de que o programa Pthread ([Figura 4.9](#)) fazia a thread mãe esperar pela thread de somatório usando a instrução `pthread_join()`. Realizamos o equivalente disso na API Win32 usando a função `WaitForSingleObject()`, que faz a thread de criação ser bloqueada até que a thread de somatório termine. (Veremos os objetos de sincronismo com mais detalhes no [Capítulo 6](#).)

4.4 Threads em Java

As threads são um modelo fundamental da execução do programa em um programa Java, e a linguagem Java e seu API fornecem um rico conjunto de aspectos da criação e gerenciamento de threads. Todos os programas Java incluem pelo menos uma única thread de controle que inicia a execução no método `main()` do programa.

4.4.1 Criando threads Java

Existem duas técnicas para a criação de threads em um programa Java. Uma é criar uma nova classe derivada da classe `Thread` e redefinir o método `run()` da classe `Thread`. No entanto, a técnica mais comum é definir uma classe que implemente a interface `Runnable`. A interface `Runnable` é definida da seguinte forma:

```
public interface Runnable
{
    public abstract void run();
}
```

Quando uma classe implementa `Runnable`, ela precisa definir um método `run()`. O código que implementa o método `run()` é o que é executado como uma thread separada.

A [Figura 4.11](#) mostra a versão Java de um programa multithreads que determina o somatório de um inteiro não negativo. A classe `Somatorio` implementa a interface `Runnable`. A criação de threads é realizada pela criação de uma instância de objeto da classe `Thread` e pela passagem de um objeto `Runnable` ao construtor.

```

class Soma
{
    private int soma;
    public int getSoma( ) {
        return soma;
    }
    public void setSoma(int soma) {
        this.soma = soma;
    }
}

class Somatorio implements Runnable
{
    private int upper;
    private Soma valorSoma;

    public Somatorio(int upper, Soma valorSoma) {
        this.upper = upper;
        this.valorSoma = valorSoma;
    }

    public void run( ) {
        int soma = 0;
        for (int i = 0; i <= upper; i++)
            soma += i;
        valorSoma.setValue(soma);
    }
}

public class Driver
{
    public static void main(String[ ] args) {
        if (args.length > 0) {
            if (Integer.parseInt(args[0]) <0)
                System.err.println(args[0] + "precisa ser >= 0.");
            else {
                // cria o objeto a ser compartilhado
                Soma objetoSoma= new Soma( );
                int upper = Integer.parseInt(args[0]);
                Thread thrd = new Thread(new Somatorio(upper, objetoSoma));
                thrd.start( );
                try {
                    thrd.join( );
                    System.out.println
                        ("A soma de "+upper+" é "+objetoSoma.getSoma());
                } catch (InterruptedException ie) { }
            }
        }
        else
            System.err.println("Uso: Soma <Valor inteiro >");
    }
}

```

FIGURA 4.11 Programa Java para o somatório de um inteiro não negativo.

A criação de um objeto `Thread` não cria especificamente a nova thread; em vez disso, é o método `start()` que realmente cria a nova thread. Chamar o método `start()` para o novo objeto resulta em duas coisas:

1. Aloca memória e inicializa uma nova thread na JVM.
2. Chama o método `run()`, tornando a thread elegível para ser executada pela JVM. (Observe que nunca chamamos o método `run()` diretamente. Em vez disso, chamamos o método `start()` e ele chama o método `run()` em nosso favor.)

Quando o programa de somatório é executado, duas threads são criadas pela JVM. A primeira é a thread mãe, que inicia sua execução no método `main()`. A segunda thread é criada quando o método `start()` no objeto `Thread` é invocado. Essa thread filha inicia a execução no método `run()` da classe `Somatorio`. Depois de enviar o valor do somatório, essa thread termina quando sai de seu método `run()`.

O compartilhamento de dados entre as threads ocorre facilmente em Win32 e Pthreads, pois os dados compartilhados são declarados globalmente. Como uma linguagem orientada a objeto pura, a Java não possui a noção de dados globais; se duas ou mais threads tiverem que compartilhar dados em um programa Java, o compartilhamento ocorrerá pela passagem de referências ao objeto compartilhado para as threads apropriadas. No programa Java mostrado na [Figura 4.11](#), a thread principal e a thread de somatório compartilham a instância de objeto da classe `Sum`. Esse objeto compartilhado é referenciado por meio dos métodos `getSoma()` e `setSoma()` apropriados. (Você poderia questionar por que não usamos um objeto `java.lang.Integer` em vez de criar uma nova classe `Soma`. O motivo é que a classe `java.lang.Integer` é **imutável**, ou seja, quando seu valor inteiro é definido, ele não pode mudar.)

Lembre-se de que as threads pai nas bibliotecas Pthreads e Win32 utilizam `pthread_join()` e `WaitForSingleObject()`, respectivamente, para esperar que as threads de somatório terminem antes

de prosseguir. O método `join()` em Java fornece uma funcionalidade semelhante. Observe que `join()` pode gerar uma `InterruptedException`, que escolhemos ignorar por enquanto. Discutiremos o tratamento dessa exceção no [Capítulo 6](#).

Na realidade, a Java identifica dois tipos diferentes de threads: (1) **daemon** (pronuncia-se “dêmon”); e (2) **threads não daemon**. A diferença fundamental entre os dois tipos é a regra simples de que a JVM encerra quando todas as threads não daemon tiverem terminado. De outras maneiras, os dois tipos de thread são idênticos. Quando a JVM é iniciada, ela cria várias threads daemon internas para realizar tarefas como a **coleta de lixo**. Uma thread daemon é criada pela chamada do método `setDaemon(true)` da classe `Thread` e passagem do valor `true` ao método. Por exemplo, poderíamos ter definido a thread no programa mostrado na [Figura 4.11](#) como um daemon, acrescentando a seguinte linha após a criação – porém, antes da partida – da thread.

```
thrd.setDaemon(true);
```

Para o restante deste texto, vamos nos referir apenas a threads não daemon, a menos que especificado de outra forma.

4.4.2 A JVM e o sistema operacional hospedeiro

Como já discutido, a JVM normalmente é implementada em cima de um sistema operacional hospedeiro (ver [Figura 2.20](#)). Essa configuração permite que a JVM esconda os detalhes da implementação do sistema operacional subjacente e forneça um ambiente coerente e abstrato, permitindo que os programas Java operem em qualquer plataforma que admita uma JVM. A especificação para a JVM não indica como as threads Java devem ser associadas ao sistema operacional subjacente, deixando essa decisão para a implementação específica da JVM. Por exemplo, o sistema operacional Windows XP utiliza o modelo um para um; portanto, cada thread Java para uma JVM rodando nesse sistema é associada a uma thread de kernel. Em sistemas operacionais que utilizam o modelo muitos para muitos (como o Tru64 UNIX), uma thread Java é associada de acordo com o modelo muitos para muitos. O Solaris inicialmente implementou a JVM usando o modelo muitos para um (a biblioteca green threads já mencionada). Outras versões da JVM foram implementadas usando o modelo muitos para muitos. A partir do Solaris 9, as threads Java foram associadas por meio do modelo um para um. Além disso, pode haver um relacionamento entre a biblioteca de threads Java e a biblioteca de threads no sistema operacional hospedeiro. Por exemplo, as implementações de uma JVM para a família Windows de sistemas operacionais poderiam usar a API Win32 ao criar threads Java; sistemas Linux e Solaris poderiam usar a API Pthreads.

4.4.3 Estados de threads Java

Uma thread Java pode estar em um destes seis estados na JVM:

1. **Novo (NEW)**. Uma thread está nesse estado quando um objeto para a thread é criado com a instrução `new`, mas a thread ainda não foi iniciada.
2. **Executável (RUNNABLE)**. A chamada do método `start()` aloca memória para a nova thread na JVM e chama o método `run()` para o objeto de thread. Quando o método `run()` de uma thread é chamado, essa thread passa do estado novo para o estado executável. Uma thread no estado executável é elegível para ser executada pela JVM. Observe que a Java não distingue entre uma thread elegível para execução e uma thread que está sendo executada. Uma thread em execução ainda está no estado executável.
3. **Bloqueado (BLOCKED)**. Uma thread está nesse estado enquanto espera para adquirir um bloqueio (`lock`) – uma ferramenta usada para sincronismo de threads. Veremos essas ferramentas no [Capítulo 6](#).
4. **Esperando (WAITING)**. Uma thread nesse estado está esperando uma ação por outra thread. Por exemplo, uma thread chamando o método `join()` entra nesse estado enquanto espera o término da thread em que está se associando.
5. **Esperando com tempo (TIMED_WAITING)**. Esse estado é semelhante ao estado esperando, exceto que uma thread especifica uma quantidade de tempo máxima que ela esperará. Por exemplo, o método `join()` possui um parâmetro opcional que a thread aguardando pode utilizar para especificar quanto tempo ela esperará até que a outra thread termine. A espera com tempo impede que uma thread permaneça no estado esperando indefinidamente.
6. **Morto (TERMINATED)**. Uma thread passa para o estado morto quando o método `run()` termina.

A [Figura 4.12](#) ilustra os diferentes estados de thread e rotula várias transições possíveis. É importante observar que esses estados se relacionam com a máquina virtual Java e não são necessariamente associados ao estado da thread rodando no sistema operacional hospedeiro.

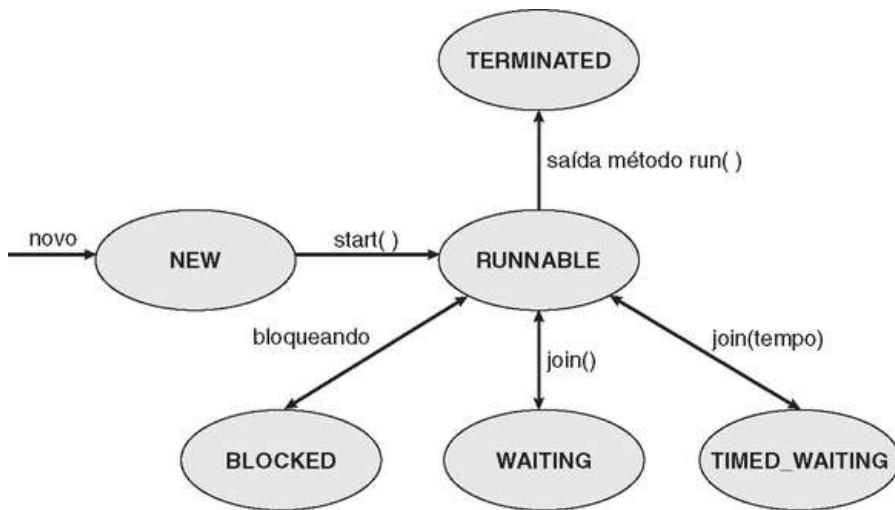


FIGURA 4.12 Estados da thread em Java.

A API Java para a classe `Thread` fornece vários métodos para determinar o estado de uma thread. O método `isAlive()` retorna `true` se uma thread tiver sido iniciada mas ainda não tiver alcançado o estado Morto; caso contrário, ele retorna `false`. O método `getState()` retorna o estado de uma thread como um tipo de dado enumerado, como um dos valores acima. O código-fonte disponível com este texto oferece um programa de exemplo usando o método `getState()`.

4.4.4 Uma solução com multithreads para o problema produtor-consumidor

Concluímos nossa discussão sobre threads Java com uma solução completa, multithreads, para o problema produtor-consumidor, que utiliza a troca de mensagens. A classe `Factory` da Figura 4.13 primeiro cria uma caixa de correio para a colocação de mensagens em buffers, usando a classe `MessageQueue` desenvolvida no Capítulo 3. Depois, ela cria threads produtor e consumidor separadas (Figuras 4.14 e 4.15, respectivamente) e passa para cada thread uma referência à caixa de correio compartilhada. A thread produtor alterna entre dormir por um tempo, produzir um item e entrar com esse item na caixa de correio. O consumidor alterna entre dormir e depois apanhar um item da caixa de correio e consumi-lo. Como o método `receive()` da classe `MessageQueue` é não bloqueante (*nonblocking*), o consumidor precisa verificar se a mensagem recebida é nula.

```

import java.util.Date;

public class Factory
{
    public static void main(String args[])
    {
        // cria a fila de mensagens
        Channel<Date> queue = new MessageQueue<Date>();

        // Cria as threads produtor e consumidor e passa
        // a cada thread uma referência ao objeto MessageQueue.
        Thread producer = new Thread(new Producer(queue));
        Thread consumer = new Thread(new Consumer(queue));

        // inicia as threads
        producer.start();
        consumer.start();
    }
}

```

FIGURA 4.13 A classe `Factory`.

```

import java.util.Date;

class Producer implements Runnable
{
    private Channel<Date> queue;

    public Producer(Channel<Date> queue) {
        this.queue = queue;
    }

    public void run( ) {
        Date message;

        while (true) {
            // dorme um pouco
            SleepUtilities.nap( );

            // produz um item e o inclui no buffer
            message = new Date( );
            System.out.println("Produtor produziu " + message);
            queue.send(message);
        }
    }
}

```

FIGURA 4.14 Thread Producer.

```

import java.util.Date;

class Consumer implements Runnable
{
    private Channel<Date> queue;

    public Consumer(Channel<Date> queue) {
        this.queue = queue;
    }

    public void run( ) {
        Date message;

        while (true) {
            // dorme um pouco
            SleepUtilities.nap( );

            // consome um item do buffer
            message = queue.receive( );

            if (message != null)
                System.out.println("Consumidor consumiu " + message);
        }
    }
}

```

FIGURA 4.15 Thread Consumer.

4.5 Aspectos do uso de threads

Nesta seção, vamos discutir alguns dos aspectos a serem considerados com programas dotados de múltiplas threads. Em vários casos destacamos esses aspectos com programas Java.

4.5.1 As chamadas de sistema fork() e exec()

No [Capítulo 3](#), descrevemos como a chamada de sistema `fork()` é utilizada para criar um processo separado, duplicado. A semântica das chamadas de sistema `fork()` e `exec()` muda em um programa dotado de múltiplas threads.

Se uma thread em um programa chamar `fork()`, o novo processo duplica todas as threads ou o novo processo possui uma única thread? Alguns sistemas UNIX escolheram ter duas versões de `fork()`, uma que duplica todas as threads e outra que duplica apenas a thread que invocou a chamada de sistema `fork()`.

A chamada de sistema `exec()` normalmente atua da mesma maneira descrita no [Capítulo 3](#), ou seja, se uma thread invocar a chamada de sistema `exec()`, o programa especificado no parâmetro de `exec()` substituirá o processo inteiro, incluindo todas as threads.

Qual das duas versões de `fork()` será utilizada depende da aplicação. Se a `exec()` for chamada imediatamente após a criação, então a duplicação de todas as threads será desnecessária, pois o programa especificado nos parâmetros da `exec()` substituirá o processo. Nesse caso, é apropriado duplicar somente a thread que chama. Todavia, se o processo separado não chamar `exec()` após a criação, o processo separado deverá duplicar todas as threads.

4.5.2 Cancelamento

O **cancelamento da thread** é a tarefa de terminar uma thread antes de ela ser sido concluída. Por exemplo, se várias threads estiverem pesquisando concorrentemente um banco de dados e uma thread retornar o resultado, as threads restantes poderão ser canceladas. Outra situação poderia ocorrer quando um usuário pressiona um botão em um navegador Web que interrompe a carga do restante da página. Em geral, uma página Web é carregada usando várias threads – cada imagem é carregada por uma thread separada. Quando um usuário pressiona o botão *Parar*, todas as threads carregando a página são canceladas.

Uma thread que precisa ser cancelada é denominada **thread-alvo (target thread)**. O cancelamento de uma thread-alvo pode ocorrer em dois cenários diferentes:

1. **Cancelamento assíncrono.** Uma thread termina imediatamente a thread-alvo.
2. **Cancelamento adiado.** A thread-alvo pode verificar periodicamente se deve terminar, o que dá a oportunidade de terminar de forma controlada.

A dificuldade com o cancelamento ocorre em situações em que foram alocados recursos a uma thread cancelada ou em que uma thread é cancelada enquanto está no meio da atualização dos dados que está compartilhando com outras threads. Isso se torna especialmente problemático com o cancelamento assíncrono. Em geral, o sistema operacional retomará os recursos de sistema de uma thread cancelada, mas não todos os recursos. Portanto, o cancelamento de uma thread de forma assíncrona pode não liberar os recursos necessários no nível de sistema.

Por outro lado, com o cancelamento adiado, uma thread indica que uma thread-alvo deve ser cancelada, mas o cancelamento só ocorre depois de a thread-alvo verificar um sinalizador, para determinar se deve ser cancelada ou não. Isso permite que a thread seja cancelada em um ponto em que possa ser cancelada com segurança. Os pthreads referem-se a esses pontos como **pontos de cancelamento (cancellation points)**.

As threads em Java podem ser terminadas de forma assíncrona por meio do método `stop()` da classe `Thread`. No entanto, esse método foi **desaprovado**. Os métodos desaprovados ainda estão implementados na API atual; contudo, seu uso é recomendável. A [Seção 7.3.2](#) explica por que `stop()` foi desaprovado.

É possível também cancelar uma thread Java usando o cancelamento adiado. Como descrito, o cancelamento adiado funciona por meio de uma thread-alvo verificando periodicamente se deve ser terminada. Em Java, a verificação envolve o uso do método `interrupt()`. A API Java define o método `interrupt()` para a classe `Thread`. Quando o método `interrupt()` é chamado, o **status de interrupção** da thread-alvo é definido. Uma thread pode verificar periodicamente seu status de interrupção chamando o método `interrupted()` ou o método `isInterrupted()`, ambos retornando `true` (verdadeiro) se o status da interrupção da thread-alvo estiver marcado. (É importante observar que o método `interrupted()` apagará o status de interrupção da thread-alvo, enquanto o método `isInterrupted()` preservará o status da interrupção.) A [Figura 4.16](#) ilustra como o cancelamento adiado funciona quando o método `isInterrupted()` é utilizado.

```

class InterruptibleThread implements Runnable
{
    /**
     * Esta thread continuará executando até
     * que seja interrompida.
     */
    public void run( ) {
        while (true) {
            /**
             * realiza algum trabalho por um tempo
             * . . .
             */

            if (Thread.currentThread( ).isInterrupted( )) {
                System.out.println("Eu fui interrompida!");
                break;
            }
        }
        // limpa e termina
    }
}

```

FIGURA 4.16 Cancelamento adiado usando o método `isInterrupted()`.

Uma instância de `InterruptibleThread` pode ser interrompida usando o seguinte código:

```

Thread thrd = new Thread(new
    InterruptibleThread( ));
thrd.start( );
. . .
thrd.interrupt( );

```

Nesse exemplo, a thread-alvo pode verificar periodicamente seu status de interrupção por meio do método `isInterrupted()` e – se estiver marcado – fazer a limpeza antes de terminar. Como a classe `InterruptibleThread` não estende `Thread`, ela não pode chamar diretamente os métodos de instância na classe `Thread`. Para chamar os métodos de instância em `Thread`, um programa primeiro precisa chamar o método estático `currentThread()`, que retorna um objeto `Thread` representando a thread sendo executada no momento. Esse valor de retorno pode, então, ser usado para acessar métodos de instância na classe `Thread`.

É importante reconhecer que a interrupção de uma thread por meio do método `interrupt()` só define o status de interrupção de uma thread; fica a critério da thread-alvo verificar com frequência esse status de interrupção. Tradicionalmente, a Java não desperta uma thread bloqueada em uma operação de E/S usando o pacote `java.io`. Nenhuma thread bloqueada realizando E/S nesse pacote poderá verificar seu status de interrupção até que a chamada para E/S esteja concluída. Entretanto, o pacote `java.nio` introduzido na Java 1.4 provê facilidades para a interrupção de uma thread que esteja bloqueada realizando E/S.

4.5.3 Tratamento de sinais

Um **sinal** é usado nos sistemas UNIX para notificar a um processo a ocorrência de um evento. Um sinal pode ser recebido de forma síncrona ou assíncrona, dependendo da origem e do motivo para o evento ser sinalizado. Todos os sinais, sejam síncronos ou assíncronos, seguem o mesmo padrão:

1. Um sinal é gerado pela ocorrência de um evento em particular.
2. Um sinal gerado é entregue a um processo.
3. Uma vez entregue, o sinal precisa ser tratado.

Exemplos de sinal síncrono incluem um acesso ilegal à memória ou a divisão por 0. Se um programa em execução realizar uma dessas ações, um sinal será gerado. Os sinais síncronos são entregues ao mesmo processo que realizou a operação que causou o sinal (esse é o motivo para serem considerados síncronos).

Quando um sinal é gerado por um evento externo a um processo em execução, esse processo

recebe o sinal assincronamente. Alguns exemplos desses sinais incluem o término de um processo com toques de tecla específicos (como `<control> <C>`) e o término do tempo de um temporizador. Normalmente, um sinal assíncrono é enviado a outro processo.

Todo sinal pode ser *tratado* por um dentre dois tipos de tratadores possíveis:

1. Um tratador de sinal padrão.
2. Um tratador de sinal definido pelo usuário.

Cada sinal possui um **tratador de sinal (signal Handler) padrão** que o kernel executa ao tratar desse sinal. Essa ação-padrão pode ser modificada por uma função **tratadora de sinal definida pelo usuário**, que é chamada para tratar do sinal. Os sinais podem ser tratados de diferentes maneiras. Alguns podem ser ignorados (como a mudança do tamanho de uma janela); outros podem ser tratados pelo término do programa (como um acesso ilegal à memória).

O tratamento de sinais em programas com única thread é simples; os sinais são sempre entregues a um processo. Entretanto, a entrega de sinais é mais complicada em programas multithreads, em que um processo pode ter várias threads. Onde, então, um sinal deve ser entregue? Em geral, existem as seguintes opções:

1. Entregar o sinal à thread à qual o sinal se aplica.
2. Entregar o sinal a cada thread no processo.
3. Entregar o sinal a determinadas threads no processo.
4. Atribuir uma thread específica para receber todos os sinais para o processo.

O método de entrega de um sinal depende do tipo de sinal gerado. Por exemplo, os sinais síncronos precisam ser entregues à thread que causa o sinal, e não a outras threads no processo. Todavia, a situação com sinais assíncronos não é tão clara. Alguns sinais assíncronos – como um sinal que termina um processo (`control<C>`, por exemplo) – devem ser enviados a todas as threads.

A maioria das versões multithreads do UNIX permite que uma thread especifique quais sinais aceitará e quais bloqueará. Portanto, em alguns casos, um sinal assíncrono pode ser entregue somente às threads que não o estão bloqueando. Porém, como os sinais precisam ser tratados apenas uma vez, um sinal é entregue apenas à primeira thread encontrada que não o esteja bloqueando. A função-padrão do UNIX para a entrega de um sinal é `kill(pid_t pid, int signal)`; aqui, especificamos o processo (`pid`) ao qual um sinal em particular deve ser entregue. Porém, Pthreads POSIX também fornecem a função `pthread_kill(pthread_t tid, int signal)`, que permite que um sinal seja entregue a uma thread especificada (`tid`).

Embora o Windows não forneça suporte explícito para sinais, eles podem ser simulados por meio de **chamadas de procedimento assíncronas (Asynchronous Procedure Calls - APCs)**. O recurso de APC permite que uma thread de usuário especifique uma função que deve ser chamada quando a thread de usuário receber notificação de um evento em particular. Conforme indicado por seu nome, uma APC é um equivalente aproximado de um sinal assíncrono no UNIX. Contudo, enquanto o UNIX precisa disputar com a forma de tratar com sinais em um ambiente multithreads, a facilidade da APC é mais direta, pois uma APC é entregue a uma thread em particular, em vez de a um processo.

4.5.4 Bancos de threads

Na [Seção 4.1](#), mencionamos o uso de multithreading em um servidor Web. Nessa situação, sempre que o servidor recebe uma requisição, ele cria uma thread separada para atender a requisição. Embora a criação de uma thread separada certamente seja superior à criação de um processo separado, um servidor multithreads, apesar disso, possui problemas em potencial. O primeiro refere-se à quantidade de tempo necessária para criar a thread antes de atender a requisição, junto com o fato de que essa thread será descartada quando tiver concluído seu trabalho. A segunda questão é mais problemática: se permitirmos que todas as requisições concorrentes sejam atendidas em uma nova thread, não teremos colocado um limite sobre o número de threads ativas concorrentes no sistema. As threads ilimitadas poderiam esgotar os recursos do sistema, como o tempo de CPU ou a memória. Uma solução para esse problema é usar **bancos de threads**.

A ideia geral por trás de um banco de threads é criar uma série de threads na partida do processo e colocá-las em um *banco*, no qual ficarem esperando para atuar. Quando um servidor recebe uma requisição, ele acorda uma thread do seu banco – se houver uma disponível – e passa a requisição do serviço. Quando a thread conclui seu serviço, ela retorna ao banco e espera por mais trabalho. Se o banco não tem uma thread disponível, o servidor espera até haver uma livre.

Os bancos de threads fornecem os seguintes benefícios principais:

1. O atendimento a uma requisição com uma thread existente normalmente é mais rápido do que esperar para criar uma thread.
2. Um banco de threads limita o número de threads existentes a qualquer momento. Isso é importante principalmente em sistemas que não podem admitir uma grande quantidade de threads concorrentes.

A quantidade de threads no banco pode ser definida heuristicamente com base em fatores como o número de CPUs no sistema, a quantidade de memória física e o número esperado de requisições de cliente concorrentes. Arquiteturas mais sofisticadas de banco de threads podem ajustar de forma

dinâmica o número de threads no banco de acordo com os padrões de uso. Essas arquiteturas fornecem o benefício adicional de ter um banco menor - consumindo, assim, menos memória - quando a carga no sistema for baixa.

O pacote `java.util.concurrent` inclui uma API para bancos de threads, junto com outras ferramentas para programação concorrente. A API Java fornece diversas variedades de arquiteturas de bancos de threads; vamos enfocar os três modelos a seguir, que estão disponíveis como métodos `static` na classe `java.util.concurrent.Executors`.

1. Executor de única thread - `newSingleThreadExecutor()` - cria um banco de tamanho 1.
2. Executor de thread fixo - `newFixedThreadPool(int size)` - cria um banco de threads com um número especificado de threads.
3. Executor de thread em cache - `newCachedThreadPool()` - cria um banco de threads ilimitado, reutilizando threads em muitas instâncias.

Na API Java, os bancos de threads são estruturados em torno da interface `Executor`, que aparece da seguinte maneira:

```
public interface Executor
{
    void execute(Comando executável);
}
```

As classes implementando essa interface precisam definir o método `execute()`, que recebe um objeto `Runnable`, como a seguir:

```
public class Task implements Runnable
{
    public void run() {
        System.out.println("Eu estou trabalhando
em uma tarefa.");
        . . .
    }
}
```

Para desenvolvedores Java, isso significa que o código que roda como uma thread separada usando a técnica a seguir, conforme ilustramos inicialmente na [Seção 4.4](#):

```
Thread worker = new Thread(new Task());
worker.start();
```

também pode ser executado como um `Executor`:

```
Executor service = new Executor;
service.execute(new Task());
```

Um banco de threads é criado usando-se um dos métodos de fábrica na classe `Executors`:

- `static ExecutorService newSingleThreadExecutor()`.
- `static ExecutorService newFixedThreadPool(int nThreads)`.
- `static ExecutorService newCachedThreadPool()`.

Cada um desses métodos de fábrica cria e retorna uma instância de objeto que implementa a interface `ExecutorService`. `ExecutorService` estende a interface `Executor`, permitindo que chamemos o método `execute()` nesse objeto. Contudo, `ExecutorService` também fornece métodos adicionais para gerenciar o término do banco de threads.

O exemplo mostrado na [Figura 4.17](#) cria um banco de threads em cache e submete tarefas para serem executadas por uma thread no banco. Quando o método `shutdown()` é chamado, o banco de threads rejeita tarefas adicionais e termina quando todas as tarefas existentes tiverem terminado a execução. No [Capítulo 6](#), fornecemos um exercício de programação para criar e implementar um

banco de threads.

```
import java.util.concurrent.*;

public class TPExample
{
    public static void main(String[ ] args) {
        int numTasks = Integer.parseInt(args[0].trim( ));

        // Cria o banco de threads
        ExecutorService pool = Executors.newCachedThreadPool( );

        // Executa cada tarefa usando uma thread no banco
        for (int i = 0; i < numTasks; i++)
            pool.execute(new Task( ));

        // Fecha o banco. Isso fecha o banco somente
        // depois que todas as threads tiverem terminado.
        pool.shutdown( );
    }
}
```

FIGURA 4.17 Criando um banco de threads em Java.

4.5.5 Dados específicos da thread

As threads pertencentes ao mesmo processo compartilham os dados do processo. Na realidade, esse compartilhamento de dados fornece um dos benefícios da programação multithreads. Contudo, em algumas circunstâncias, cada thread poderia precisar de sua própria cópia de determinados dados. Chamaremos esses dados de **dados específicos da thread**. Por exemplo, em um sistema de processamento de transações, poderíamos atender a cada transação em uma thread separada. Além do mais, cada transação pode receber um identificador exclusivo. Para associar cada thread ao seu identificador exclusivo, poderíamos usar dados específicos da thread.

A maior parte das bibliotecas de threads - incluindo Win32 e Pthreads - fornece alguma forma de suporte para dados específicos da thread. A Java também provê suporte. À primeira vista, pode parecer que a Java não precisa de dados específicos da thread, visto que tudo que é preciso para dar a cada thread seus próprios dados privados é criar threads subclassificando a classe Thread e declarar dados de instância em sua classe. Essa técnica funciona bem enquanto as threads são construídas dessa maneira. Entretanto, quando o desenvolvedor não tem controle sobre o processo de criação da thread - por exemplo, quando um banco de threads estiver sendo usado -, uma técnica alternativa é necessária.

A API Java fornece a classe ThreadLocal para declarar dados específicos da thread. Os dados de ThreadLocal podem ser inicializados com o método initialValue() ou com o método set(), e uma thread pode perguntar sobre o valor dos dados de ThreadLocal usando o método get(). Normalmente, os dados de ThreadLocal são declarados como static. Considere a classe Service mostrada na [Figura 4.18](#), que declara errorCode como dados de ThreadLocal. O método transaction() nessa classe pode ser invocado por qualquer quantidade de threads. Se houver uma exceção, atribuímos a exceção a errorCode usando o método set() da classe ThreadLocal. Agora considere um cenário em que duas threads - digamos, thread 1 e thread 2 - invocam transaction(). Suponha que a thread 1 gere a exceção A e a thread 2 gere a exceção B. Os valores de errorCode para a thread 1 e a thread 2 serão A e B, respectivamente. A [Figura 4.19](#) ilustra como uma thread pode perguntar sobre o valor de errorCode() depois de invocar transaction().

```

class Service
{
    private static ThreadLocal errorCode =
        new ThreadLocal( );

    public static void transaction( ) {
        try {
            /**
             * alguma operação onde pode ocorrer erro
             *
            */
        }
        catch (Exception e) {
            errorCode.set(e);
        }
    }

    /**
     * Apanha código de erro para esta transação
     */
    public static Object getErrorCode( ) {
        return errorCode.get();
    }
}

```

FIGURA 4.18 Usando a classe ThreadLocal.

```

class Worker implements Runnable
{
    private static Service provider;

    public void run( ) {
        provider.transaction( );
        System.out.println(provider.getErrorCode( ));
    }
}

```

FIGURA 4.19 Perguntando o valor dos dados de ThreadLocal.

4.5.6 Ativações do escalonador (Scheduler Activations)

Uma questão final a ser considerada com programas multithreads refere-se à comunicação entre o kernel e a biblioteca threads, que pode ser exigida pelos modelos muitos para muitos e de dois níveis, discutidos na [Seção 4.2.3](#). Essa coordenação permite que a quantidade de threads de kernel seja ajustada de maneira dinâmica para ajudar a garantir o melhor desempenho.

Muitos sistemas implementando o modelo muitos para muitos ou de dois níveis incluem uma estrutura de dados intermediária entre as threads de usuário e do kernel. Essa estrutura de dados, conhecida como um processo leve, ou LightWeight Process (LWP), é mostrada na [Figura 4.20](#). Para a biblioteca threads de usuário, o LWP parece ser um *processador virtual* em que a aplicação pode agendar uma thread de usuário para executar. Cada LWP é ligado a uma thread de kernel, e são as threads de kernel que o sistema operacional escalona para executar nos processadores físicos. Se uma thread de kernel for bloqueada (por exemplo, enquanto esperar o término de uma operação de E/S), o LWP também é bloqueado. Mais adiante na cadeia, a thread no nível do usuário ligado ao LWP também é bloqueada.

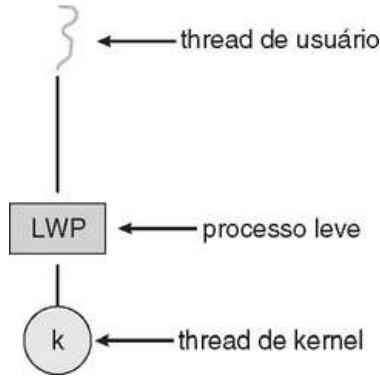


FIGURA 4.20 Processo leve (LWP).

Uma aplicação pode exigir qualquer quantidade de LWPs para ser executada de modo eficiente. Considere uma aplicação ligada à CPU executando em um sistema de processador único. Nesse cenário, somente uma thread pode executar ao mesmo tempo, de modo que um LWP é suficiente. No entanto, uma aplicação com uso intenso de E/S pode exigir vários LWPs para ser executada. Normalmente, um LWP é exigido para cada chamada de sistema simultânea com bloqueio. Suponha, por exemplo, que cinco requisições de leitura de arquivo diferentes ocorram concorrentemente. Cinco LWPs são necessários, pois todos poderiam estar esperando pelo término da E/S no kernel. Se um processo tem apenas quatro LWPs, então a quinta requisição precisa esperar que um dos LWPs retorne do kernel.

Um esquema para a comunicação entre a biblioteca threads de usuário e o kernel é conhecido como **ativação do escalonador (schedule activation)**. Isso funciona da seguinte forma: o kernel provê a uma aplicação um conjunto de processadores virtuais (LWPs), e a aplicação pode escalarar threads de usuário para um processador virtual disponível. Além do mais, o kernel precisa informar sobre determinados eventos a uma aplicação. Esse procedimento é conhecido como **upcall**. Upcalls são tratados pela biblioteca threads com um **tratador de upcall**, e os tratadores de upcall são executados em um processador virtual. Um evento que dispara um upcall ocorre quando a thread de uma aplicação está para ser bloqueada. Nesse cenário, o kernel faz um upcall para a aplicação, informando que uma thread está para ser bloqueada e identificando a thread específica. O kernel, então, aloca um novo processador virtual para a aplicação. A aplicação executa um tratador de upcall nesse novo processador virtual, que salva o estado da thread em bloqueio e abre mão do processador virtual em que a thread com bloqueio está sendo executada. O tratador de upcall escala outra thread elegível para ser executada nesse novo processador virtual. Quando ocorre o evento esperado pela thread com bloqueio, o kernel faz outro upcall para a biblioteca threads, informando que a thread anteriormente bloqueada agora está elegível para execução. O tratador de upcall para esse evento também exige um processador virtual, e o kernel pode alocar um novo processador virtual ou se apossar de uma das threads de usuário e executar o tratador de upcall em seu processador virtual. Depois de marcar a thread não bloqueada como elegível para execução, a aplicação escala uma thread elegível para executar em um processador virtual disponível.

4.6 Exemplos em sistemas operacionais

Nesta seção vamos explorar como as threads são implementadas nos sistemas Windows XP e Linux.

4.6.1 Threads no Windows XP

O Windows XP implementa a API Win32, que é a principal API para a família de sistemas operacionais da Microsoft (Windows 95, 98, NT, 2000 e XP). Na verdade, grande parte do mencionado nesta seção se aplica a essa família de sistemas operacionais.

Uma aplicação para Windows XP é executada como um processo separado, e cada processo pode conter uma ou mais threads. A API Win32 para criação de threads é tratada na [Seção 4.3.2](#). O Windows XP utiliza o mapeamento um para um, descrito na [Seção 4.2.2](#), no qual cada thread no nível do usuário é associada a uma thread de kernel. Entretanto, o Windows XP também provê suporte para uma biblioteca **fiber**, que fornece a funcionalidade do modelo muitos para muitos ([Seção 4.2.3](#)). Ao usar a biblioteca threads, qualquer thread pertencente a um processo pode acessar o espaço de endereços virtuais do processo.

Os componentes gerais de uma thread incluem:

- Uma ID de thread, identificando a thread de forma exclusiva.
- Um conjunto de registradores representando o status do processador.
- Uma pilha do usuário, usada quando a thread está executando no modo usuário, e uma pilha do kernel, usada quando a thread está executando no modo kernel.
- Uma área de armazenamento privada, usada por diversas bibliotecas em tempo de execução e bibliotecas de vínculo dinâmico (Dynamic Link Libraries - DLLs).

O conjunto de registradores, pilhas e área de armazenamento privado é conhecido como **contexto** da thread. As principais estruturas de dados de uma thread são:

- ETHREAD (bloco de thread do executivo).
- KTHREAD (bloco de thread de kernel).
- TEB (bloco de ambiente da thread).

Os principais componentes do bloco ETHREAD incluem um ponteiro para o processo ao qual a thread pertence e o endereço da rotina em que a thread inicia o controle. O bloco ETHREAD também contém um ponteiro para o KTHREAD correspondente.

O bloco KTHREAD inclui informações de escalonamento e sincronismo para a thread. Além disso, o KTHREAD inclui a pilha do kernel (usada quando a thread está executando no modo kernel) e um ponteiro para o TEB.

O ETHREAD e o KTHREAD existem no espaço do kernel; isso significa que apenas o kernel pode acessá-los. O TEB é uma estrutura de dados no espaço do usuário, acessada quando a thread está executando no modo usuário. Entre outros campos, o TEB contém um identificador de thread, uma pilha do modo usuário e um array para dados específicos da thread (que o Windows XP chama de **armazenamento local à thread**). A estrutura da thread do Windows XP é ilustrada na [Figura 4.21](#).

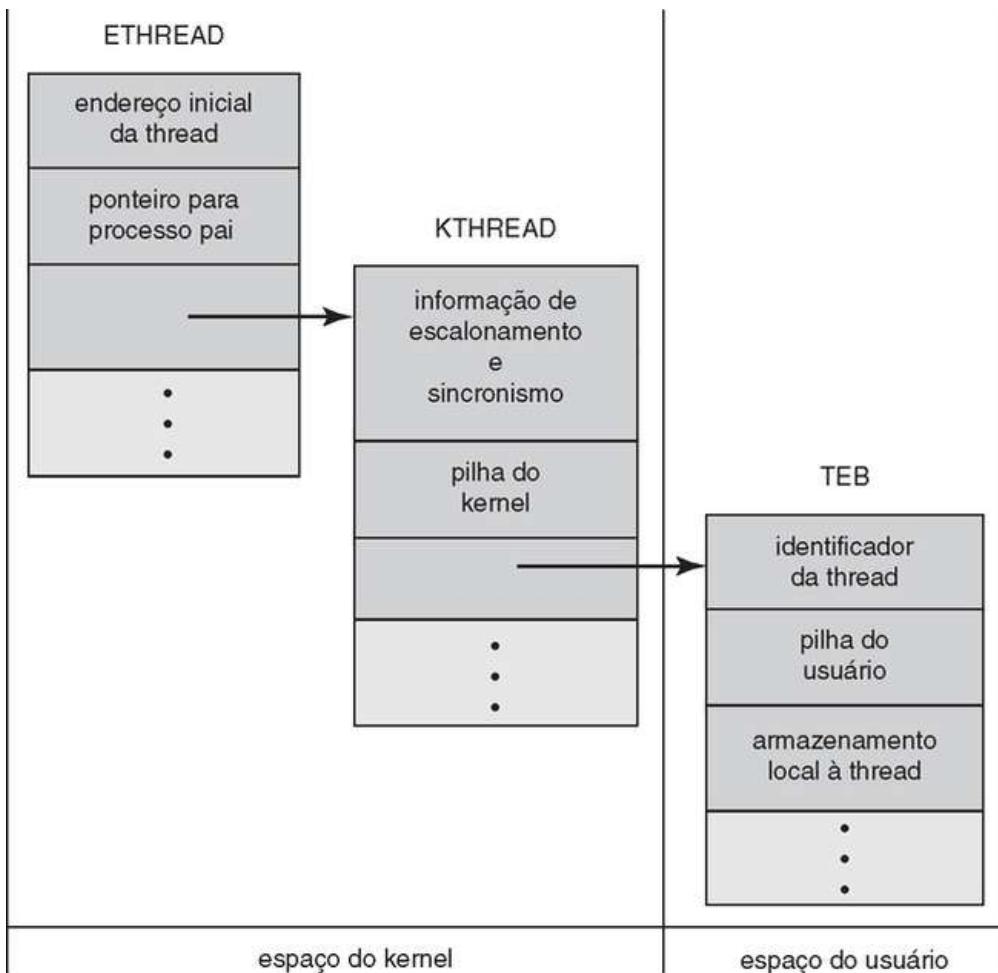


FIGURA 4.21 Estruturas de dados de uma thread no Windows XP.

4.6.2 Threads no Linux

O Linux provê uma chamada de sistema `fork()` com a funcionalidade tradicional da duplicação de um processo, como descrito no [Capítulo 3](#). O Linux também provê a capacidade de criar threads usando chamada de sistema `clone()`. Contudo, o Linux não distingue entre processos e threads. De fato, o Linux geralmente usa o termo *tarefa* – em vez de *processo* ou *thread* – quando se refere a um fluxo de controle dentro de um programa. Quando `clone()` é chamado, ele recebe um conjunto de flags que determinam quanto compartilhamento deverá ocorrer entre as tarefas pai e filha. Alguns desses flags são listados a seguir:

flag	significado
<code>CLONE_FS</code>	Informação do sistema de arquivos é compartilhada.
<code>CLONE_VM</code>	O mesmo espaço de memória é compartilhado.
<code>CLONE_SIGHAND</code>	Manipuladores de sinal são compartilhados.
<code>CLONE_FILES</code>	O conjunto de arquivos abertos é compartilhado.

Por exemplo, se `clone()` receber os flags `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND` e `CLONE_FILES`, as tarefas pai e filha compartilharão a mesma informação do sistema de arquivos (como o diretório de trabalho atual), o mesmo espaço de memória, os mesmos manipuladores de sinal e o mesmo conjunto de arquivos abertos. Usar `clone()` dessa maneira é equivalente a criar uma thread conforme descrevemos neste capítulo, pois a tarefa pai compartilha a maioria de seus recursos com sua tarefa filha. Contudo, se nenhum dos flags for definido quando `clone()` for invocado, nenhum compartilhamento ocorre, resultando em uma funcionalidade semelhante à que é fornecida pela chamada de sistema `fork()`.

O nível de compartilhamento variável é possível devido ao modo como um processo é representado no kernel do Linux. Existe uma estrutura de dados exclusiva do kernel (especificamente, `struct task_struct`) para cada processo no sistema. Essa estrutura dos dados, em vez de armazenar os dados para a tarefa, contém ponteiros para outras estruturas de dados, onde os dados são armazenados – por exemplo, estruturas de dados que representam a lista de arquivos abertos, informações de tratamento de sinal e memória virtual. Quando `fork()` é invocado, uma nova tarefa é criada junto com uma *cópia* de todas as estruturas de dados associadas do processo pai. Uma nova tarefa também é criada quando é feita a chamada de sistema `clone()`. Entretanto, em

vez de copiar todas as estruturas de dados, a nova tarefa *aponta* para as estruturas de dados da tarefa pai, dependendo do conjunto de sinalizadores (flags) passado para `clone()`.

Várias distribuições do kernel do Linux agora incluem a biblioteca de threads NPTL. A NTPL (que significa Native POSIX Thread Library - biblioteca nativa de threads POSIX) oferece um modelo de threads compatíveis com POSIX para sistemas Linux, juntamente com vários outros recursos, como melhor suporte para sistemas SMP, além de tirar proveito do suporte para NUMA. Além disso, o custo inicial para a criação de uma thread é mais baixo com NPTL do que com threads Linux tradicionais. Finalmente, com NPTL, o sistema tem o potencial de dar suporte a centenas de milhares de threads. Esse suporte torna-se mais importante com o crescimento de sistemas multicore e outros sistemas SMP.

4.7 Resumo

Uma thread é um fluxo de controle dentro de um processo. Um processo multithreads contém diversos fluxos de controle diferentes dentro do mesmo espaço de endereços. Os benefícios do uso de multithreads incluem maior responsividade ao usuário, compartilhamento de recursos dentro do processo, economia e questões de escalabilidade, como o uso mais eficiente de múltiplos núcleos.

As threads no nível do usuário são threads visíveis ao programador e desconhecidas do kernel. O kernel do sistema operacional aceita e gerencia as threads no nível do kernel. Em geral, as threads no nível do usuário são mais rápidas de criar e gerenciar do que as threads de kernel, portanto não é necessário nenhuma intervenção do kernel. Três tipos diferentes de modelos relacionam as threads de usuário e de kernel. O modelo muitos para um associa muitas threads de usuário a uma única thread de kernel. O modelo um para um associa cada thread de usuário a uma thread de kernel correspondente. O modelo muitos para muitos multiplexa muitas threads de usuário a um número menor ou igual de threads de kernel.

A maior parte dos sistemas operacionais modernos provê suporte do kernel para as threads; entre eles estão Windows 98, XP e Vista, além do Solaris e do Linux.

Bibliotecas de threads fornecem ao programador de aplicação uma API para criar e gerenciar threads. Três bibliotecas de threads principais são as mais utilizadas: Pthreads POSIX, threads Win32 para sistemas Windows e threads Java.

Os programas multithreads apresentam muitos desafios para o programador, incluindo a semântica das chamadas de sistema `fork()` e `exec()`. Outras questões incluem o cancelamento da thread, o tratamento de sinais e os dados específicos da thread. A API Java resolve muitos desses problemas com threads.

Exercícios práticos

- 4.1. Prepare dois exemplos de programação nos quais o uso de multithreading ofereça melhor desempenho do que uma solução de única thread.
- 4.2. Quais são as duas diferenças entre as threads em nível de usuário e as threads em nível de kernel? Sob quais circunstâncias um tipo é melhor do que o outro?
- 4.3. Descreva as ações tomadas por um kernel para a troca de contexto entre as threads em nível de kernel.
- 4.4. Quais recursos são usados quando uma *thread* é criada? Qual a diferença entre eles e aqueles usados quando um *processo* é criado?
- 4.5. Suponha que um sistema operacional faça um mapeamento entre as threads em nível de usuário e o kernel, usando o modelo muitos para muitos, e que o mapeamento seja feito por meio de LWPs. Além do mais, o sistema permite que os desenvolvedores criem threads em tempo real para uso em sistemas de tempo real. É necessário vincular uma thread em tempo real a um processo leve? Explique.
- 4.6. Um programa Pthread que executa a função de somatório foi apresentado na [Seção 4.3.1](#). Reescreva esse programa em Java.

Exercícios

- 4.7. Forneça dois exemplos de programação em que o uso de multithreading *não* provê melhor desempenho do que uma solução com uma única thread.
- 4.8. Descreva as ações tomadas por uma biblioteca de threads para a troca de contexto entre threads no nível do usuário.
- 4.9. Sob quais circunstâncias uma solução multithreads usando múltiplas threads de kernel provê desempenho melhor do que uma solução com única thread em um sistema de único processador?
- 4.10. Quais dos seguintes componentes de estado do programa são compartilhados pelas threads em um processo com multithreading?
 - a. Valores de registrador
 - b. Memória heap
 - c. Variáveis globais
 - d. Memória de pilha
- 4.11. Uma solução com multithreading usando múltiplas threads em nível de usuário pode conseguir desempenho melhor em um sistema de multiprocessadores do que em um sistema de único processador? Explique.
- 4.12. Conforme descrevemos na [Seção 4.6.2](#), o Linux não distingue entre processos e threads. Em vez disso, o Linux trata ambos da mesma maneira, permitindo que uma tarefa seja mais semelhante a um processo ou a uma thread, dependendo do conjunto de flags passados à chamada de sistema `clone()`. Porém, muitos sistemas operacionais – como Windows XP e Solaris – tratam processos e threads de formas diferentes. Normalmente, tais sistemas utilizam uma notação na qual a estrutura de dados para um processo contém ponteiros para as threads separadas pertencentes ao processo. Compare essas duas técnicas para modelagem de processos e threads dentro do kernel.
- 4.13. O programa mostrado na [Figura 4.22](#) usa a API Pthreads. Qual seria a saída do programa em LINHA F e LINHA P?

```

#include <pthread.h>
#include <stdio.h>

int valor = 0;
void *soma(void *param); /* a thread */

int main(int argc, char *argv[ ])
{
    int pid;
    pthread_t tid;
    pthread_attr_t attr;

    pid = fork( );

    if (pid == 0) { /* processo filho */
        pthread_attr_init(&attr);
        pthread_create(&tid,&attr,soma,NULL);
        pthread_join(tid,NULL);
        printf("FILHO: valor = %d",valor); /* LINHA F */
    }
    else if (pid > 0) { /* processo pai */
        wait(NULL);
        printf("PAI: valor = %d", valor); /* LINHA P */
    }
}

void *soma(void *param) {
    valor = 5;
    pthread_exit(0);
}

```

FIGURA 4.22 Programa em C para o Exercício 4.13.

- 4.14. Considere um sistema de multiprocessador e um programa com multithreading escrito usando o modelo de threading muitos para muitos. Considere que a quantidade de threads em nível de usuário no programa seja maior que o número de processadores no sistema. Discuta as implicações de desempenho nos cenários a seguir.

- a. A quantidade de threads de kernel alocadas ao programa é menor que a quantidade de processadores.
- b. A quantidade de threads de kernel alocadas ao programa é igual à quantidade de processadores.
- c. A quantidade de threads de kernel alocadas ao programa é maior que a quantidade de processadores, mas é menor que a quantidade de threads em nível de usuário.

Problemas de programação

- 4.15. O Exercício 3.17, no Capítulo 3, envolve a criação de um programa cliente-servidor onde o cliente envia uma mensagem ao servidor e o servidor responde com um contador contendo o número de caracteres e dígitos na mensagem. Porém, esse servidor é de única thread, significando que o servidor não pode responder a outros clientes até que o cliente atual feche sua conexão. Modifique a solução do Exercício 3.17 de modo que o servidor atenda a cada cliente em uma requisição separada.
- 4.16. Escreva um programa com multithreading (em Java, Pthreads ou Win32) que gere números primos. Esse programa deverá atuar da seguinte maneira: o usuário executará o programa e informará um número na linha de comandos. O programa, então, criará uma thread separada, que gerará todos os números primos menores ou iguais ao número digitado pelo usuário.
- 4.17. Modifique o servidor de data baseado em socket (Figura 3.26), do Capítulo 3, de modo que o servidor atenda à requisição de cada cliente em uma thread separada.
- 4.18. Modifique o servidor de data baseado em socket (Figura 3.26), do Capítulo 3, de modo que o servidor atenda à requisição de cada cliente usando um banco de threads.
- 4.19. A sequência de Fibonacci é a série de números 0, 1, 1, 2, 3, 5, 8, ... Formalmente, ela pode ser expressa como:

$$\begin{aligned}fib_0 &= 0 \\fib_1 &= 1 \\fib_n &= fib_{n-1} + fib_{n-2}\end{aligned}$$

Escreva um programa com multithreading que gere a sequência de Fibonacci usando a biblioteca de threads Java, Pthreads ou Win32. Esse programa deverá funcionar da seguinte maneira: o usuário digitará na linha de comandos a quantidade de números de Fibonacci que o programa deve gerar; o programa, então, criará uma thread separada que gerará os números de Fibonacci, colocando a sequência nos dados que são compartilhados pelas threads (um array provavelmente é a estrutura de dados mais conveniente). Quando a thread terminar sua execução, a thread mãe exibirá a sequência gerada pela thread filha. Como a thread mãe não pode começar a exibir a sequência de Fibonacci antes que a thread filha termine, será preciso que a thread mãe espere que a filha termine, usando as técnicas descritas na Seção 4.3.

- 4.20. Escreva um programa de classificação multithreading em Java que funcione da seguinte maneira: uma coleção de itens é dividida em duas listas de mesmo tamanho. Cada lista é passada, então, a uma thread separada (uma *thread de classificação*), que ordena a lista usando algum algoritmo (ou algoritmos) de classificação à sua escolha. As duas listas classificadas são passadas para uma terceira thread (uma *thread de mesclagem*), que reúne as duas listas separadas em uma única lista classificada. Quando as duas listas tiverem sido reunidas, a lista classificada completa é enviada para a saída. Se estivéssemos classificando valores inteiros, este programa seria estruturado conforme a Figura 4.23.

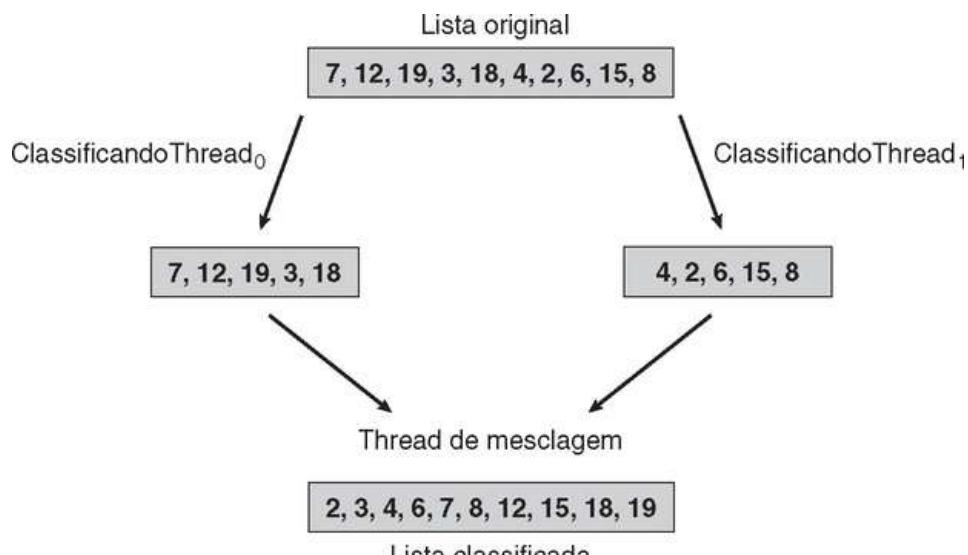


FIGURA 4.23 Classificação.

Talvez o modo mais fácil de projetar uma thread de classificação seja passar ao construtor um array contendo `java.lang.Object`, onde cada `Object` deverá implementar a interface `java.util.Comparable`. Muitos objetos na API Java implementam a interface `Comparable`. Para os propósitos deste projeto, recomendamos o uso de objetos `Integer`. Para garantir que as duas threads de classificação tenham concluído sua execução, a thread principal precisará usar o método `join()` nas duas threads de classificação antes de passar as duas listas classificadas para a thread de mesclagem. De modo semelhante, a thread principal precisará usar `join()` na thread de mesclagem antes de enviar a lista classificada completa para a saída. Para ver uma discussão sobre algoritmos de classificação, consulte a bibliografia.

- 4.21. Escreva um programa em Java que liste todos os usuários na máquina virtual Java. Antes de prosseguir, você precisará de alguma informação básica. Todas as threads na JVM pertencem a um grupo de threads, e um grupo de threads é identificado na API Java pela classe `ThreadGroup`. Grupos de threads são organizados como uma estrutura de árvore, onde a raiz da árvore é o grupo de threads do sistema. O grupo de threads do sistema contém threads criadas automaticamente pela JVM, principalmente para gerenciar referências a objetos. Abaixo do grupo de threads do sistema está o grupo de threads principais. O grupo de threads principais contém a thread inicial em um programa Java que inicia a execução no método `main()`. Esse também é o grupo de threads padrão, significando que - a menos que seja especificado de outra

forma – todas as threads que você cria pertencem a esse grupo. É possível criar grupos de threads adicionais e atribuir threads recém-criadas a esses grupos. Além do mais, ao criar um grupo de threads, você poderá especificar seu pai. Por exemplo, as instruções a seguir criam três novos grupos de threads: alfa, beta e teta:

```
ThreadGroup alfa = new ThreadGroup("alfa");
ThreadGroup beta = new ThreadGroup("beta");
ThreadGroup teta = new ThreadGroup(alfa,
    "teta");
```

Os grupos alfa e beta pertencem ao grupo de threads padrão – ou principal. Porém, o construtor para o grupo de threads teta indica que seu grupo de threads pai é alfa. Assim, temos uma hierarquia do grupo de threads representada na [Figura 4.24](#).

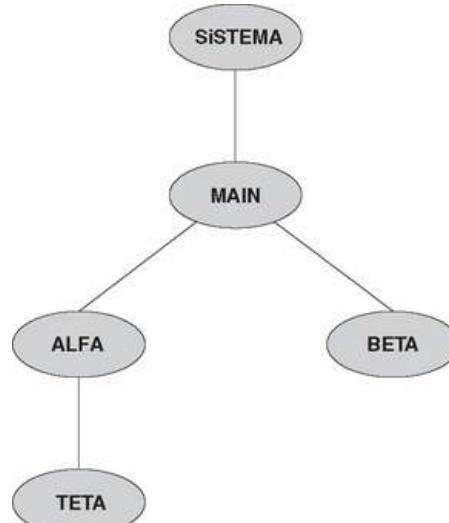


FIGURA 4.24 Hierarquia do grupo de threads.

Observe que todos os grupos de threads possuem um pai, com a exceção óbvia do grupo de threads do sistema, cujo pai é null.

A escrita de um programa que liste todas as threads na máquina virtual Java envolverá uma leitura cuidadosa da API Java – particularmente, as classes `java.lang.ThreadGroup` e `java.lang.Thread`. Uma estratégia para construir esse programa é primeiro identificar todos os grupos de threads na JVM e depois identificar todas as threads dentro de cada grupo. Para determinar todos os grupos de threads, primeiro obtenha o `ThreadGroup` da thread atual e depois sobe na hierarquia de árvore do grupo de threads até a sua raiz. Em seguida, apanhe todos os grupos de threads abaixo do grupo de threads raiz; o método overloaded `enumerate()` na classe `ThreadGroup` é especialmente útil para essa tarefa. Em seguida, identifique as threads que pertencem a cada grupo. Novamente, o método `enumerate()` será bastante útil. Uma coisa a ser observada no uso do método `enumerate()` é que ele espera um array como parâmetro. Isso significa que você precisará determinar o tamanho do array antes de chamar `enumerate()`. Outros métodos na API `ThreadGroup` deverão ser úteis para determinar como dimensionar arrays. Faça a sua saída indicar cada grupo de threads e todas as threads dentro de cada grupo. Por exemplo, com base na [Figura 4.24](#), sua saída aparecerá da seguinte forma:

- system: todas as threads no grupo do sistema
- main: todas as threads no grupo principal
- alpha: todas as threads no grupo alfa
- beta: todas as threads no grupo beta
- theta: todas as threads no grupo teta

Ao enviar a saída de cada thread, liste os seguintes campos:

- a. O nome da thread
- b. O identificador da thread
- c. O estado da thread
- d. Se a thread é um daemon ou não

4.22. Modifique o problema anterior de modo que a saída apareça em um formato de tabela, usando uma interface gráfica. A coluna mais à esquerda deverá representar o nome do grupo de threads e as colunas sucessivas representam os quatro campos de cada thread. Além disso, enquanto o programa no problema anterior lista todas as threads na JVM apenas uma vez, este programa deverá atualizar periodicamente a listagem de threads e os grupos de threads, por meio de um parâmetro de atualização na linha de comandos na camada do programa.

Represente esse parâmetro em milissegundos. Por exemplo, se o seu programa se chama `ListaThreads`, para chamar o programa de modo que ele atualize a lista dez vezes por segundo, digite:

```
java ListaThreads 100
```

Projetos de programação

Os projetos a seguir tratam de dois tópicos distintos – serviços de nomes e multiplicação de matriz.

Projeto 1: Projeto de serviço de nomes

Um serviço de nomes, como DNS (Domain Name System) pode ser usado para traduzir nomes IP em endereços IP. Por exemplo, quando alguém acessa o hospedeiro `www.westminstercollege.edu`, um serviço de nomes é usado para determinar o endereço IP que está relacionado com o nome IP `www.westminstercollege.edu`. Este projeto consiste em escrever em Java um serviço de nomes com multithreading, usando sockets (ver [Seção 3.6.1](#)).

A API `java.net` oferece o seguinte mecanismo para traduzir nomes IP:

```
InetAddress hostAddress =  
    InetAddress.getByName("www.westminstercollege.edu");  
String IPaddress = hostAddress.getHostAddress();
```

onde `getByName()` gera uma `UnknownHostException` se não conseguir traduzir o nome do hospedeiro.

O servidor

O servidor escutará na porta 6052, aguardando conexões do cliente. Quando for feita uma conexão do cliente, o servidor atenderá a conexão em uma thread separada e continuará escrutando outras conexões do cliente. Quando um cliente fizer uma conexão com o servidor, o cliente escreverá o nome IP que deseja que o servidor traduza - como `www.westminstercollege.edu` - no socket. A thread do servidor lerá esse nome IP do socket e traduzirá seu endereço IP ou, se não puder localizar o endereço do hospedeiro, interceptará uma `UnknownHostException`. O servidor escreverá o endereço IP de volta para o cliente ou, no caso de uma `UnknownHostException`, escreverá a mensagem "Não foi possível traduzir < nome do hospedeiro >". Quando o servidor tiver escrito para o cliente, ele fechará sua conexão socket.

O cliente

Inicialmente, escreva apenas a aplicação do servidor e conecte-se a ela via telnet. Por exemplo, supondo que o servidor esteja sendo executado no hospedeiro local, uma sessão telnet se parecerá com o seguinte (as respostas do cliente aparecem em negrito):

```
telnet localhost 6052  
Connected to localhost.  
Escape character is '^]'.  
www.westminstercollege.edu  
146.86.1.17  
Connection closed by foreign host.
```

Fazendo o telnet atuar inicialmente como um cliente, você poderá depurar mais facilmente quaisquer problemas que possa ter com o seu servidor. Quando estiver convencido de que seu servidor está funcionando corretamente, você poderá escrever uma aplicação cliente. O cliente receberá, como parâmetro, o nome IP que deve ser traduzido. Então ele lerá a resposta enviada de volta pelo servidor. Como um exemplo, se o cliente se chama `NSCliente`, ele é chamado da seguinte forma:

```
java NSCliente www.westminstercollege.edu
```

O servidor responderá com o endereço IP correspondente ou com a mensagem "Não foi possível traduzir <nome do hospedeiro>". Quando o cliente tiver informado o endereço IP ou a mensagem de erro, ele fechará sua conexão socket.

Projeto 2: Multiplicação de matriz

Dadas duas matrizes, A e B , onde A é uma matriz com M linhas e K colunas, e a matriz B contém K linhas e N colunas, a matriz produto de A e B é C , onde C contém M linhas e N colunas. A entrada na matriz C para a linha i coluna j (C_{ij}) é a soma dos produtos dos elementos para a linha i na matriz A e coluna j na matriz B , ou seja,

$$C_{ij} = \sum_{n=1}^K A_{i,n} \times B_{n,j}$$

Por exemplo, se A fosse uma matriz 3 por 2 e B fosse uma matriz 2 por 3, o elemento $C_{3,1}$ seria a soma de $A_{3,1} \times B_{1,1}$ e $A_{3,2} \times B_{2,1}$.

Para este projeto, calcule cada elemento C_{ij} em uma thread de *trabalho* separada. Isso envolverá a criação de $M \times N$ threads de trabalho. A thread principal - ou mãe - inicializará as matrizes A e B e alocaará memória suficiente para a matriz C , que manterá o produto das matrizes A e B . Essas matrizes serão declaradas como dados globais, para que cada thread de trabalho tenha acesso a A , B e C .

As matrizes A e B podem ser inicializadas estaticamente, como vemos a seguir:

```
#define M 3
#define K 2
#define N 3

int A [M] [K] = { {1,4}, {2,5}, {3,6} };
int B [K] [N] = { {8,7,6}, {5,4,3} };
int C [M] [N];
```

Como alternativa, elas podem ser preenchidas lendo-se os valores de um arquivo.

Passando parâmetros a cada thread

A thread mãe criará $M \times N$ threads de trabalho, passando a cada uma os valores de linha i e coluna j que ela deve usar no cálculo do produto da matriz. Isso exige a passagem de dois parâmetros a cada thread. A técnica mais fácil com Pthreads e Win32 é criar uma estrutura de dados usando uma struct. Os membros dessa estrutura são i e j , e a estrutura se assemelha a:

```
/* estrutura para passar dados às threads */
struct v
{
    int i; /* linha */
    int j; /* coluna */
};
```

Os programas Pthreads e Win32 criarião as threads de trabalho usando uma estratégia semelhante à que mostramos a seguir:

```
/* Temos que criar M * N threads de trabalho */
for (i = 0; i < M; i++) {
    for (j = 0; j < N; j++) {
        struct v *data = (struct v *)
            malloc(sizeof(struct v));
        data->i = i;
        data->j = j;
        /* Agora cria a thread passando dados como parâmetro */
    }
}
```

O ponteiro `data` será passado à função `pthread_create()` (Pthreads) ou à função `CreateThread()` (Win32), que por sua vez o passará como um parâmetro para a função que deve ser executada como uma thread separada.

O compartilhamento de dados entre threads Java é diferente do compartilhamento entre threads em Pthreads ou Win32. Uma técnica é que a thread principal crie e inicialize as matrizes A , B e C . Essa thread principal, então, criará as threads de trabalho, passando as três matrizes – junto com a linha i e a coluna j – para o construtor de cada thread de trabalho. Assim, o esboço de uma thread de trabalho aparece na [Figura 4.25](#).

```
public class WorkerThread implements Runnable
{
    private int row;
    private int col;
    private int[][] A;
    private int[][] B;
    private int[][] C;

    public WorkerThread(int row, int col, int[][] A,
                        int[][] B, int[][] C) {
        this.row = row;
        this.col = col;
        this.A = A;
        this.B = B;
        this.C = C;
    }

    public void run() {
        /* calcula o produto da matriz em C[row][col] */
    }
}
```

FIGURA 4.25 Thread de trabalho em Java.

Esperando que as threads terminem

Quando todas as threads de trabalho tiverem sido concluídas, a thread principal exibirá o produto contido na matriz C . Isso exige que a thread principal espere até que todas as threads de trabalho terminem para poder exibir o valor da matriz produto. Diversas estratégias diferentes podem ser usadas para permitir que uma thread espere que outras threads terminem. A [Seção 4.3](#) descreve como esperar que uma thread filha termine usando as bibliotecas de threads Win32, Pthreads e Java. Win32 fornece a função `WaitForSingleObject()`, enquanto Pthreads e Java utilizam `pthread_join()` e `join()`, respectivamente. Contudo, nesses exemplos de programação, a thread mãe espera que uma única thread filha termine; a conclusão deste exercício exigirá esperar por múltiplas threads.

Na [Seção 4.3.2](#), descrevemos a função `WaitForSingleObject()`, que é usada para esperar que uma única thread termine. No entanto, a API Win32 também fornece a função `WaitForMultipleObjects()`, que é usada quando se espera que múltiplas threads terminem. `WaitForMultipleObjects()` recebe quatro parâmetros:

1. O número de objetos a esperar.
2. Um ponteiro para o array de objetos.
3. Um flag indicando se todos os objetos foram sinalizados.
4. Uma duração de tempo limite (ou `INFINITE`).

Por exemplo, se `THandles` for um array de objetos `HANDLE` de thread de tamanho N , então a thread mãe poderá esperar que todas as suas threads filhas terminem com a instrução:

```
WaitForMultipleObjects(N, THandles, TRUE, INFINITE);
```

Uma estratégia simples para esperar por várias threads usando `pthread_join()` da Pthreads ou `join()` da Java é manter a operação de junção dentro de um loop `for` simples. Por exemplo, você poderia juntar dez threads usando o código Pthreads representado na [Figura 4.26](#). O código equivalente usando Java aparece na [Figura 4.27](#).

```
#define NUM_THREADS 10

/* um array de threads a serem juntadas */
pthread_t workers[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++)
    pthread_join(workers[i], NULL);
```

FIGURA 4.26 Código Pthreads para juntar dez threads.

```
final static int NUM_THREADS = 10;

/* um array de threads a serem juntadas */
Thread[ ] workers = new Thread[NUM_THREADS];

for (int i = 0; i < NUM_THREADS; i++) {
    try {
        workers[i].join( );
    } catch (InterruptedException ie) { }
}
```

FIGURA 4.27 Código Java para juntar dez threads.

Notas bibliográficas

As threads tiveram uma longa evolução, começando como “concorrência barata” nas linguagens de programação e passando para “processos leves”, com exemplos iniciais que incluíam o sistema Thoth ([Cheriton e outros \[1979\]](#)) e o sistema Pilot ([Redell e outros \[1980\]](#)). [Binding \[1985\]](#) descreveu a passagem das threads para o kernel do UNIX. Mach ([Accetta e outros \[1986\]](#), [Tevanian e outros \[1987a\]](#)) e V ([Cheriton \[1988\]](#)) fizeram bastante uso das threads, e por fim quase todos os principais sistemas operacionais as implementaram de uma forma ou de outra.

As questões de desempenho de thread foram discutidas por [Anderson e outros \[1989\]](#), que continuaram seu trabalho em [Anderson e outros \[1991\]](#), avaliando o desempenho das threads em nível de usuário com suporte do kernel. [Bershad e outros \[1990\]](#) descreveram a combinação de threads com RPC. [Engelschall \[2000\]](#) discute uma técnica para dar suporte às threads em nível de usuário. Uma análise do tamanho ideal para o banco de threads pode ser encontrada em [Ling e outros \[2000\]](#). As ativações do escalonador foram apresentadas inicialmente em [Anderson e outros \[1991\]](#), e [Williams \[2002\]](#) discute as ativações do escalonador no sistema NetBSD. Outros mecanismos pelos quais a biblioteca threads em nível de usuário e o kernel cooperam entre si são discutidos em [Marsh e outros \[1991\]](#), [Govindan e Anderson \[1991\]](#), [Draves e outros \[1991\]](#) e [Black \[1990\]](#). [Zabatta e Young \[1998\]](#) compararam as threads do Windows NT e do Solaris em um multiprocessador simétrico. [Pinilla e Gill \[2003\]](#) compararam o desempenho da thread Java nos sistemas Linux, Windows e Solaris.

[Vahalia \[1996\]](#) aborda o uso de threads em diversas versões do UNIX. [McDougall e Mauro \[2007\]](#) descrevem os desenvolvimentos recentes no uso de threads pelo kernel do Solaris. [Russinovich e Solomon \[2005\]](#) discutem o uso de threads na família de sistemas operacionais Windows. [Bovet e Cesati \[2006\]](#) e [Love \[2004\]](#) explicam como o Linux trata da questão das threads e [Singh \[2007\]](#) aborda as threads no Mac OS X.

As informações sobre a programação Pthreads são dadas em [Lewis e Berg \[1998\]](#) e [Butenhof \[1997\]](#). [Oaks e Wong \[2004\]](#), [Lewis e Berg \[2000\]](#) e [Holub \[2000\]](#) discutem sobre multithreading em Java. [Goetz e outros \[2006\]](#) apresentam uma discussão detalhada da programação concorrente em Java. [Beveridge e Wiener \[1997\]](#), e [Cohen e Woodring \[1997\]](#) discutem o uso de multithreads com sistemas Win32. [Goodrich e Tamassia \[2006\]](#) apresentam estruturas de dados e algoritmos de classificação em Java.

CAPÍTULO 5

Escalonamento de CPU

O escalonamento de CPU é a base dos sistemas operacionais multiprogramados. Alternando a CPU entre os processos, o sistema operacional pode tornar o computador mais produtivo. Neste capítulo, apresentamos os conceitos básicos do escalonamento de CPU e diversos algoritmos de escalonamento de CPU. Também consideramos o problema da seleção de um algoritmo para determinado sistema.

No [Capítulo 4](#), introduzimos as threads no modelo de processo. Nos sistemas operacionais que admitem isso, são as threads no nível do kernel – não os processos – que estão sendo escalonadas pelo sistema operacional. Contudo, os termos **escalonamento de processo** e **escalonamento de threads** normalmente são usados para indicar a mesma coisa. Neste capítulo, usamos *escalonamento de processo* quando discutimos sobre os conceitos gerais de escalonamento, e *escalonamento de thread* para nos referir a ideias específicas da thread.

OBJETIVOS DO CAPÍTULO

- Introduzir o escalonamento de CPU, que é a base para os sistemas operacionais multiprogramados.
- Descrever diversos algoritmos de escalonamento de CPU.
- Discutir os critérios de avaliação para selecionar um algoritmo de escalonamento de CPU para um sistema em particular.

5.1 Conceitos básicos

Em um sistema de processador único, somente um processo pode ser executado de cada vez; quaisquer outros precisam esperar até a CPU estar livre e poder ser reescalonada. O objetivo da multiprogramação é ter algum processo em execução durante todos os momentos, para maximizar a utilização da CPU. A ideia é relativamente simples. Um processo é executado até precisar esperar, em geral, pelo término de alguma requisição de E/S. Em um sistema computadorizado simples, a CPU fica ociosa. Todo esse tempo aguardando é desperdiçado; nenhum trabalho útil é realizado. Com a multiprogramação, tentamos usar esse tempo de forma produtiva. Diversos processos são mantidos na memória ao mesmo tempo. Quando um processo precisa esperar, o sistema operacional afasta a CPU desse processo e dá a CPU a outro processo. Esse padrão continua. Sempre que um processo precisa esperar, outro processo pode assumir o uso da CPU.

O escalonamento desse tipo é uma função fundamental do sistema operacional. Quase todos os recursos do computador são escalonados antes do uso. Naturalmente, a CPU é um dos principais recursos do computador. Assim, seu escalonamento é vital para o projeto do sistema operacional.

5.1.1 Ciclo de burst CPU-E/S

O sucesso do escalonamento da CPU depende da propriedade observada nos processos: a execução de um processo consiste em um **ciclo** de execução da CPU e espera por E/S. Os processos alternam entre esses dois estados. A execução do processo começa com um **burst (surto) de CPU**, que é seguido por um **burst de E/S**, que, por sua vez, é seguido por outro burst de CPU, depois outro burst de E/S, e assim por diante. Por fim, o burst de CPU final termina com uma requisição do sistema para terminar a execução (Figura 5.1).

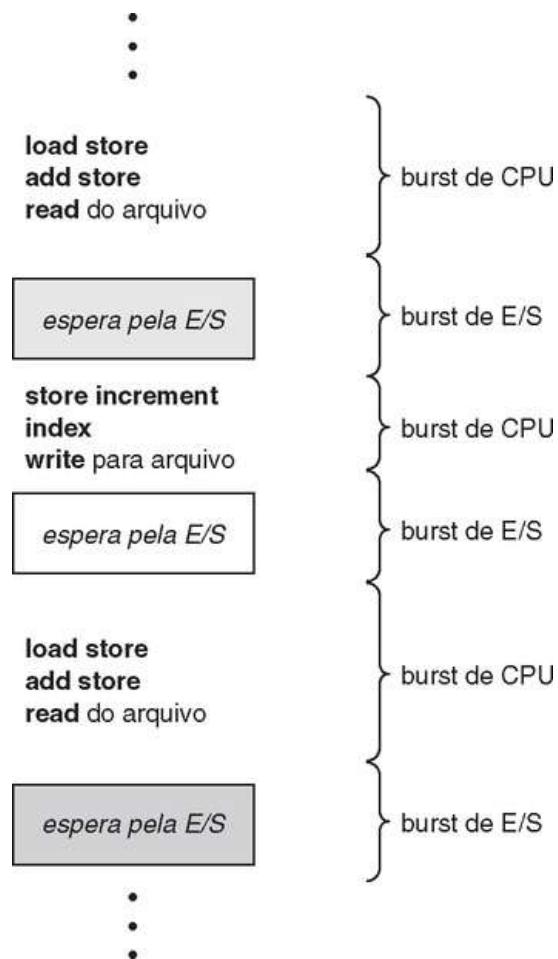


FIGURA 5.1 Alternando a sequência de bursts CPU-E/S.

As durações dos burst de CPU foram medidas extensivamente. Embora variem bastante de um processo para outro e de um computador para outro, costumam ter uma curva de frequência semelhante à que aparece na Figura 5.2. Em geral, a curva é caracterizada como exponencial ou hiperexponencial, com uma grande quantidade de bursts de CPU curtos e uma pequena quantidade

de bursts de CPU longos. Um programa E/S-bound possui muitos bursts de CPU curtos. Um programa CPU-bound poderia ter alguns bursts de CPU longos. Essa distribuição pode ser importante na seleção de um algoritmo de escalonamento de CPU apropriado.

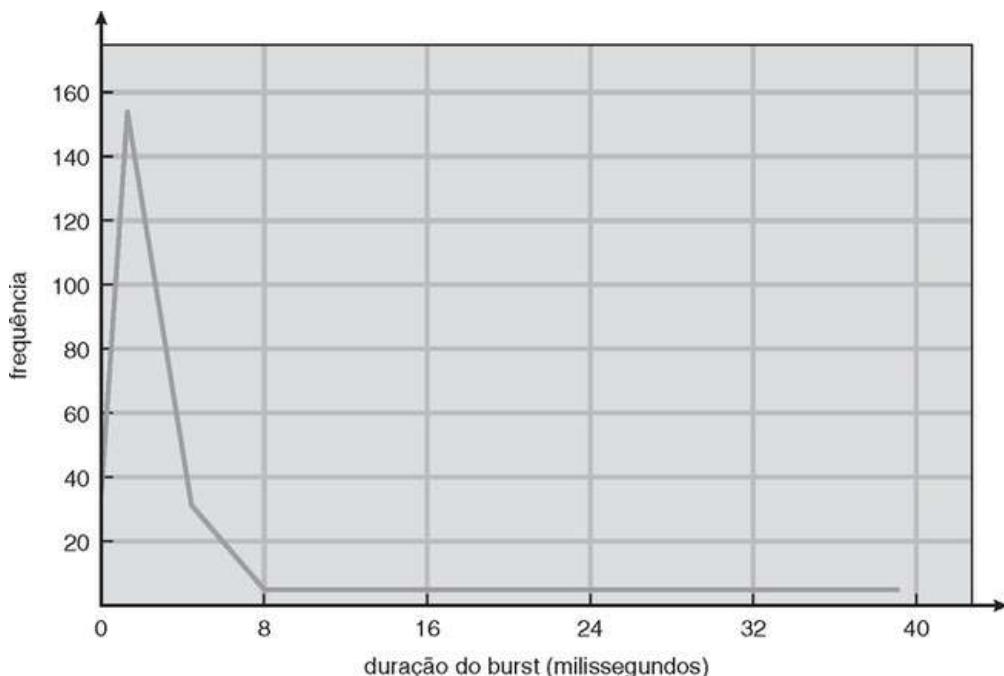


FIGURA 5.2 Histograma dos tempos de burst da CPU.

5.1.2 Escalonador de CPU

Sempre que a CPU se torna ociosa, o sistema operacional precisa selecionar um dos processos na fila de prontos para ser executado. O processo de seleção é executado pelo **escalonador de curto prazo** (ou escalonador de CPU). O escalonador seleciona entre os processos na memória que estão prontos para execução e aloca a CPU a um deles.

Observe que a fila de prontos (ready queue) não é necessariamente uma fila FIFO (primeiro a entrar, primeiro a sair). Conforme veremos quando considerarmos os diversos algoritmos de escalonamento, uma fila de prontos pode ser implementada como uma fila FIFO, uma fila de prioridade, uma árvore ou uma lista interligada fora de ordem. Entretanto, em conceito, todos os processos na fila de prontos estão alinhados e esperando uma chance para execução na CPU. Os registros nas filas são os blocos de controle de processo (PCBs) dos processos.

5.1.3 Escalonamento preemptivo

As decisões de escalonamento de CPU podem ocorrer sob as quatro circunstâncias a seguir:

1. Quando um processo passa do estado executando (running) para o estado esperando (waiting). Por exemplo, como resultado de uma requisição de E/S ou uma chamada wait pelo término de um dos processos filho.
2. Quando um processo passa do estado executando (running) para o estado pronto (ready). Por exemplo, quando ocorre uma interrupção.
3. Quando um processo passa do estado esperando (waiting) para o estado pronto (ready). Por exemplo, no término da E/S.
4. Quando um processo termina.

Para as situações 1 e 4, não há escolha em termos de escalonamento. Um processo novo (se houver um na fila de prontos) precisa ser selecionado para execução. Contudo, existe uma escolha para as situações 2 e 3.

Quando o escalonamento ocorre somente sob as circunstâncias 1 e 4, dizemos que o esquema de escalonamento é **não preemptivo** ou **cooperativo**; caso contrário, ele é **preemptivo**. Sob o escalonamento não preemptivo, quando a CPU tiver sido alocada a um processo, ele retém a CPU até que ela seja liberada pelo término ou pela troca para o estado esperando. Esse método de escalonamento foi usado pelo Microsoft Windows 3.x; o Windows 95 introduziu o escalonamento preemptivo, e todas as versões subsequentes dos sistemas operacionais Windows têm usado o escalonamento preemptivo. O sistema operacional Mac OS X para o Macintosh também utiliza o escalonamento preemptivo; as versões anteriores do sistema operacional Macintosh contavam com

o escalonamento cooperativo. O escalonamento cooperativo é o único método que pode ser usado em certas plataformas de hardware, pois não exige o hardware especial (por exemplo, um temporizador) necessário para o escalonamento preemptivo.

Infelizmente, o escalonamento preemptivo incorre em um custo associado ao acesso a dados compartilhados. Considere o caso de dois processos que compartilham dados. Enquanto um está atualizando os dados, ele é preemptado de modo que o segundo processo possa executar. O segundo processo tenta, então, ler os dados, que estão em um estado incoerente. Nessas situações, precisamos de novos mecanismos para coordenar o acesso aos dados compartilhados; discutiremos esse assunto no [Capítulo 6](#).

A preempção também afeta o projeto do kernel do sistema operacional. Durante o processamento de uma chamada de sistema, o kernel pode estar ocupado com uma atividade em favor de um processo. Essas atividades podem envolver a mudança de dados importantes do kernel (por exemplo, filas de E/S). O que acontece se o processo for preemptado no meio dessas mudanças e o kernel (ou o driver de dispositivo) precisar ler ou modificar a mesma estrutura? Acontece o caos. Certos sistemas operacionais, incluindo a maioria das versões do UNIX, tratam desse problema esperando que uma chamada de sistema seja concluída ou que ocorra uma operação com um bloco de E/S antes de realizar uma troca de contexto. Esse esquema garante que a estrutura do kernel será simples, pois ele não preempta de um processo enquanto as estruturas de dados do kernel estiverem em um estado incoerente. Infelizmente, esse modelo de execução do kernel é muito pobre para o suporte da computação em tempo real e o multiprocessamento. Esses problemas (e suas soluções) são descritos nas [Seções 5.5 e 19.5](#).

Como as interrupções podem, por definição, ocorrer a qualquer momento, e como elas nem sempre podem ser ignoradas pelo kernel, as seções do código afetadas pelas interrupções precisam ser protegidas contra uso simultâneo. O sistema operacional precisa aceitar interrupções em quase todo o tempo; caso contrário, a entrada poderia ser perdida ou a saída modificada. Para que essas seções do código não sejam acessadas concorrentemente por vários processos, eles desativam as interrupções na entrada e as ativam na saída. É importante observar que as seções de código que desativam as interrupções não ocorrem com muita frequência e contêm poucas instruções.

5.1.4 Despachante

Outro componente envolvido na função de escalonamento de CPU é o **despachante (dispatcher)**. O despachante é o módulo que dá o controle da CPU ao processo selecionado pelo escalonador de curto prazo. Essa função envolve o seguinte:

- Trocar o contexto.
- Trocar para o modo usuário.
- Desviar para o local apropriado no programa do usuário para reiniciar esse programa.

O despachante deverá ser o mais rápido possível, pois ele é chamado durante cada troca de processo. O tempo gasto para o despachante interromper um processo e iniciar a execução de outro é conhecido como **latência de despacho (dispatch latency)**.

5.2 Critérios de escalonamento

Diferentes algoritmos de escalonamento de CPU possuem diferentes propriedades, e a escolha de um determinado algoritmo pode favorecer uma classe dos processos em detrimento de outra. Na escolha de qual algoritmo será usado em determinada situação, temos de considerar as propriedades dos diversos algoritmos. Muitos critérios foram sugeridos para a comparação de algoritmos de escalonamento da CPU. A escolha das características que serão usadas para a comparação pode fazer diferença substancial na escolha do algoritmo considerado melhor. Os critérios incluem os seguintes:

- **Utilização de CPU.** Queremos manter a CPU ocupada o máximo de tempo possível. Em conceito, a utilização da CPU pode variar de 0% a 100%. Em um sistema real, ela deverá variar de 40% (para um sistema pouco carregado) até 90% (para um sistema muito utilizado).
- **Throughput (vazão).** Se a CPU estiver ocupada executando processos, então o trabalho está sendo feito. Uma medida do trabalho é o número de processos concluídos por unidade de tempo, algo chamado de *vazão*. Para processos longos, essa taxa pode ser um processo por hora; para transações curtas, ela pode ser 10 processos por segundo.
- **Turnaround time (tempo de retorno).** Do ponto de vista de um processo específico, o critério importante é o tempo necessário para executar esse processo. O intervalo desde o momento da submissão de um processo até o momento do término é o *turnaround time*. O turnaround é a soma dos períodos gastos esperando para entrar na memória, esperando na fila de prontos (ready queue), executando na CPU e realizando E/S.
- **Tempo de espera.** O algoritmo de escalonamento de CPU não afeta a quantidade de tempo durante a qual um processo é executado ou realiza E/S; ele afeta apenas a quantidade de tempo que um processo gasta esperando na fila de prontos. O *tempo de espera* é a soma dos períodos gastos aguardando na fila de espera.
- **Tempo de resposta.** Em um sistema interativo, o turnaround time pode não ser o melhor critério. Em geral, um processo pode produzir alguma saída rapidamente e pode continuar calculando novos resultados enquanto os resultados anteriores estão sendo mostrados ao usuário. Assim, outra medida é o tempo desde a submissão de uma requisição até a primeira resposta ser produzida. Essa medida, chamada *tempo de resposta*, é o tempo gasto para começar a responder, e não o tempo gasto para a saída da resposta. O turnaround time é limitado pela velocidade do dispositivo de saída.

É importante maximizar a utilização e o throughput da CPU, reduzindo o turnaround time, o tempo de espera e o tempo de resposta. Na maioria dos casos, otimizamos o valor da média. Contudo, sob algumas circunstâncias, é importante otimizar os valores mínimo ou máximo, em vez da média. Por exemplo, para garantir que todos os usuários recebam um bom atendimento, podemos tentar reduzir o tempo máximo de resposta.

Investigadores sugeriram que, para sistemas interativos (como sistemas de tempo compartilhado), é mais importante minimizar a *variância* no tempo de resposta do que minimizar o tempo de resposta médio. Um sistema com tempo de resposta razoável e *previsível* pode ser considerado mais desejável do que um sistema que seja mais rápido na média, porém bem variável. Entretanto, pouco trabalho tem sido feito nos algoritmos de escalonamento de CPU para reduzir a variância.

À medida que discutirmos os diversos algoritmos de escalonamento de CPU na próxima seção, ilustraremos sua operação. Uma ilustração precisa envolver muitos processos, cada um sendo uma sequência de centenas de bursts de CPU e bursts de E/S. No entanto, para simplificar, consideraremos apenas um burst de CPU (em milissegundos) por processo em nossos exemplos. Nossa medida de comparação é o tempo de espera médio. Mecanismos de avaliação mais elaborados são discutidos na [Seção 5.8](#).

5.3 Algoritmos de escalonamento

O escalonamento de CPU trata do problema de decidir qual dos processos na fila de prontos (ready queue) deve ser entregue à CPU. Existem muitos algoritmos de escalonamento de CPU diferentes. Nesta seção, vamos descrever vários deles.

5.3.1 Escalonamento First Come, First Served – FCFS

Certamente o mais simples dos algoritmos de escalonamento de CPU é o algoritmo de escalonamento **primeiro a chegar, primeiro a ser atendido (first come, first served - FCFS)**. Com esse esquema, o processo que requisita a CPU em primeiro lugar recebe a CPU primeiro. A implementação da política FCFS é gerenciada com facilidade por meio de uma fila FIFO. Quando um processo entra na fila de prontos (ready queue), seu bloco de controle de processo é associado ao final da fila. Quando a CPU está livre, ela é alocada ao processo na cabeça da fila. O processo em execução, em seguida, é removido da fila. O código para o escalonamento FCFS é simples de escrever e entender.

Entretanto, o tempo de espera médio sob a política FCFS normalmente é muito longo. Considere o seguinte conjunto de processos que chegam no instante 0, com a quantidade de tempo de burst de CPU indicada em milissegundos:

Processo	Tempo de burst
P_1	24
P_2	3
P_3	3

Se os processos chegarem na ordem P_1, P_2, P_3 e forem atendidos na ordem FCFS, obteremos o resultado mostrado no seguinte **diagrama de Gantt**, um gráfico de barras que ilustra um escalonamento em particular, incluindo os tempos de início e fim de cada um dos participantes:



O tempo de espera é de 0 milissegundo para o processo P_1 , 24 milissegundos para o processo P_2 e 27 milissegundos para o processo P_3 . Assim, o tempo de espera médio é de $(0 + 24 + 27)/3 = 17$ milissegundos. Porém, se os processos chegarem na ordem P_2, P_3, P_1 , os resultados serão mostrados no seguinte diagrama de Gant:



O tempo de espera médio agora é $(6 + 0 + 3)/3 = 3$ milissegundos. Essa redução é substancial. Assim, o tempo de espera médio sob uma política FCFS geralmente não é mínimo e pode variar muito se os tempos de burst de CPU dos processos forem bastante variáveis.

Além disso, considere o desempenho do escalonamento FCFS em uma situação dinâmica. Suponha que tenhamos um processo CPU-bound e muitos processos E/S-bound. À medida que os processos fluem pelo sistema, o seguinte cenário pode acontecer. O processo CPU-bound obterá e manterá a CPU. Durante esse tempo, todos os outros processos terminarão sua E/S e passarão para a fila de prontos, esperando pela CPU. Enquanto os processos esperam na fila de prontos, os dispositivos de E/S ficam ociosos. Por fim, o processo CPU-bound termina seu burst de CPU e passa para um dispositivo de E/S. Todos os processos E/S-bound que possuem bursts de CPU pequenos são executados rapidamente e voltam para as filas de E/S. Nesse ponto, a CPU fica ociosa. O processo CPU-bound, então, voltará para a fila de prontos e receberá a CPU. Mais uma vez, todos os processos de E/S acabam esperando na fila de prontos até que o processo CPU-bound seja concluído. Existe um **efeito comboio**, pois todos os outros processos aguardam até que um grande processo libere a CPU. Esse efeito resulta em menor utilização de CPU e dispositivo do que seria possível se processos mais curtos tivessem permissão para entrar primeiro.

Observe que o algoritmo de escalonamento FCFS é não preemptivo. Quando a CPU é alocada a um processo, este mantém a CPU até liberá-la, pelo término ou por uma requisição de E/S. O algoritmo

FCFS é particularmente problemático para sistemas de tempo compartilhado, nos quais é importante que cada usuário tenha uma fatia da CPU em intervalos regulares. Seria desastroso permitir que um processo mantenha a CPU por um período estendido.

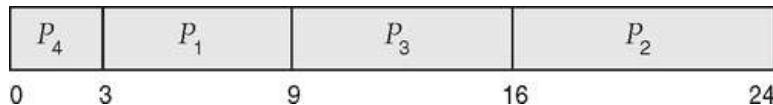
5.3.2 Escalonamento Shortest-Job-First – SJF

Uma técnica diferente para escalonamento de CPU é o **algoritmo de escalonamento shortest-job-first (SJF) (menor tarefa primeiro)**. Esse algoritmo associa a cada processo o tamanho do próximo burst de CPU do processo. Quando a CPU estiver disponível, ela será alocada ao processo que possui o menor próximo burst de CPU. Se os próximos bursts de CPU de dois processos forem iguais, o escalonamento FCFS será usado no desempate. Observe que um termo mais apropriado para esse método de escalonamento seria *algoritmo do próximo menor burst de CPU (shortest-next-CPU-burst algorithm)*, pois o escalonamento depende do tamanho do próximo burst de CPU de um processo, e não do tamanho total. Usamos o termo SJF porque é o termo mais utilizado para referir-se a esse tipo de escalonamento.

Como exemplo de escalonamento SJF, considere o seguinte conjunto de processos, com o tamanho do tempo de burst de CPU indicado em milissegundos:

Processo	Tempo de burst
P_1	6
P_2	8
P_3	7
P_4	3

Usando o escalonamento SJF, escalonaríamos esses processos de acordo com o seguinte diagrama de Gantt:



O tempo de espera é de 3 milissegundos para o processo P_1 , 16 milissegundos para o processo P_2 , 9 milissegundos para o processo P_3 e 0 milissegundo para o processo P_4 . Assim, o tempo de espera médio é $(3 + 16 + 9 + 0)/4 = 7$ milissegundos. Por comparação, se estivéssemos usando o esquema de escalonamento FCFS, o tempo de espera médio seria 10,25 milissegundos.

O algoritmo de escalonamento SJF provavelmente é o *ideal*, pois provê o menor tempo de espera médio para determinado conjunto de processos. Mover um processo curto antes de um longo diminui o tempo de espera do processo curto mais do que aumenta o tempo de espera do processo longo. Consequentemente, o tempo de espera *médio* diminui.

A dificuldade real com o algoritmo SJF é saber a extensão da próxima requisição de CPU. Para o escalonamento de longo prazo (tarefa) em um sistema batch, podemos usar como tamanho o limite de tempo do processo especificado por um usuário ao submeter a tarefa. Assim, os usuários são motivados a estimar o limite de tempo do processo com precisão, pois um valor mais baixo pode significar uma resposta mais rápida. (Um valor muito baixo causará um erro de limite de tempo ultrapassado e exigirá nova submissão.) O escalonamento SJF é usado com frequência no escalonamento de longo prazo.

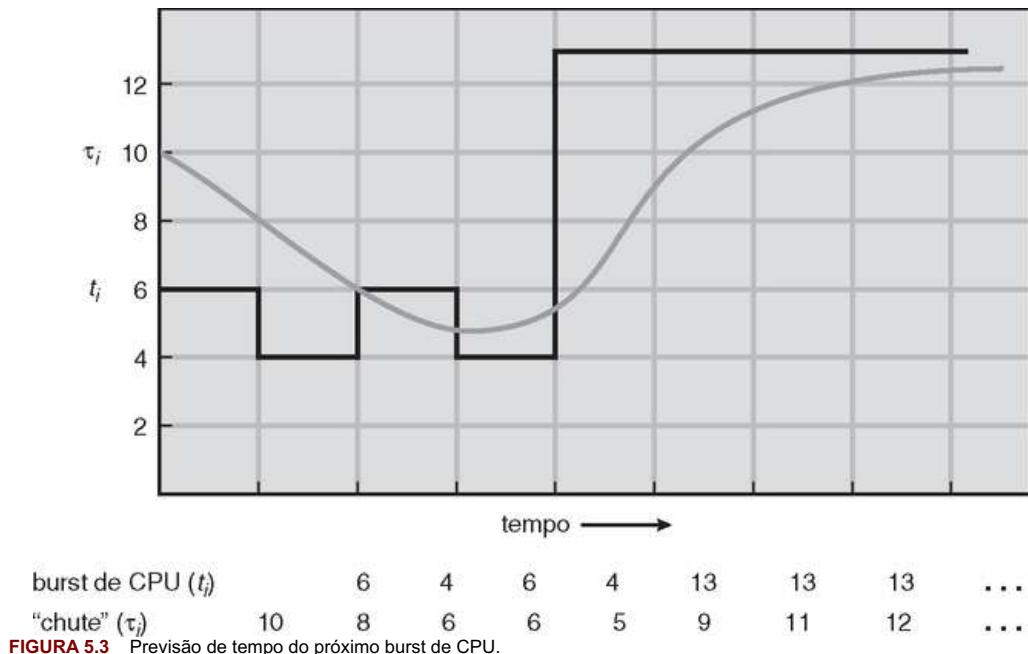
Infelizmente, o algoritmo SJF não pode ser implementado no nível do escalonamento de CPU de curto prazo. Com o escalonamento de curto prazo, não há como saber a extensão do próximo burst de CPU. Uma técnica utilizada é tentar aproximar o escalonamento SJF. Podemos não *saber* a extensão do próximo burst de CPU, mas podemos *prever* seu valor. Esperamos que o próximo burst de CPU seja semelhante em tamanho aos anteriores. Calculando uma aproximação do tamanho do próximo burst de CPU, podemos selecionar o processo com o menor burst de CPU previsto.

O próximo burst de CPU, em geral, é previsto como uma **média exponencial** dos períodos medidos dos bursts de CPU anteriores. Podemos definir a média exponencial com a seguinte fórmula. Seja t_n o tempo do enésimo burst de CPU e seja τ_{n+1} nosso valor previsto para o próximo burst de CPU. Então, para α , $0 \leq \alpha \leq 1$, defina

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

O valor de t_n contém nossa informação mais recente; τ_n armazena o histórico passado. O parâmetro α controla o peso relativo do histórico recente e passado em nossa previsão. Se $\alpha = 0$,

então $\tau_{n+1} = \tau_n$, e o histórico recente não tem efeito (as condições atuais são medidas como sendo transientes); se $\alpha = 1$, então $\tau_{n+1} = t_n$, e somente o burst de CPU mais recente importa (o histórico é considerado antigo e irrelevante). Normalmente, $\alpha = 1/2$, de modo que o histórico recente e o histórico passado têm pesos iguais. O τ_0 inicial pode ser definido como uma constante ou como uma média geral do sistema. A [Figura 5.3](#) mostra uma média exponencial com $\alpha = 1/2$ e $\tau_0 = 10$.



Para entender o comportamento da média exponencial, podemos expandir a fórmula para τ_{n+1} substituindo por τ_n , para encontrar

$$\begin{aligned}\tau_{n+1} &= \alpha_n(1-\alpha)\alpha_{t-1} \dots + (1-\alpha)^j\alpha_{t-j} \\ &\quad + \dots + (1-\alpha)^{n+1}\tau_0.\end{aligned}$$

Como tanto α quanto $(1 - \alpha)$ são menores ou iguais a 1, cada termo sucessivo possui peso menor do que seu predecessor.

O algoritmo SJF pode ser preemptivo ou não preemptivo. A opção surge quando um novo processo chega na fila de prontos durante a execução de um processo anterior. O próximo burst de CPU do novo processo que chega pode ser menor do que aquilo que resta do processo atualmente em execução. Um algoritmo SJF preemptivo retira o processo em execução, enquanto um algoritmo SJF não preemptivo permitirá que o processo em execução termine seu burst de CPU. O escalonamento SJF preemptivo às vezes é chamado de **escalonamento shortest-remaining-time-first (SRTF) (menor tempo restante primeiro)**.

Como exemplo, considere os quatro processos a seguir, com o período de burst de CPU indicado em milissegundos:

Processo	Tempo de chegada	Tempo de burst
P_1	0	8
P_2	1	4
P_3	2	9
P_4	3	5

Se os processos chegarem na fila de prontos nos momentos indicados e precisarem dos tempos de burst indicados, então o escalonamento SJF preemptivo será conforme representado no seguinte diagrama de Gantt:

P_1	P_2	P_4	P_1	P_3
0	1	5	10	17

26

O processo P_1 é iniciado no momento 0, pois é o único processo na fila. O processo P_2 chega no momento 1. O tempo restante para o processo P_1 (7 milissegundos) é maior do que o tempo exigido pelo processo P_2 (4 milissegundos), de modo que o processo P_1 é retirado, e o processo P_2 é escalonado. O tempo de espera médio para esse exemplo é $[(10 - 1) + (1 - 1) + (17 - 2) + (5 - 3)]/4 = 26/4 = 6,5$ milissegundos. O escalonamento SJF não preemptivo resultaria em um tempo de espera médio de 7,75 milissegundos.

5.3.3 Escalonamento por prioridade

O algoritmo SJF é um caso especial do **algoritmo de escalonamento por prioridade**. Uma prioridade é associada a cada processo, e a CPU é alocada ao processo com a maior prioridade. Processos com a mesma prioridade são escalonados na ordem FCFS. Um algoritmo SJF é um algoritmo por prioridade no qual a prioridade (p) é o inverso do próximo burst de CPU. Quanto maior o burst de CPU, menor a prioridade, e vice-versa.

Observe que discutimos sobre o escalonamento em termos de prioridade *alta* e prioridade *baixa*. As prioridades são indicadas por algum intervalo de números, como 0 a 7 ou 0 a 4.095. Contudo, não existe um acordo geral sobre se 0 é a maior ou a menor prioridade. Alguns sistemas utilizam números baixos para representar a prioridade baixa; outros usam números baixos para a prioridade alta. Essa diferença pode causar confusão. Neste texto, consideramos que números baixos representam prioridade alta.

Como exemplo, observe o seguinte conjunto de processos, considerados como tendo chegado no momento 0, na ordem P_1, P_2, \dots, P_5 , com o período de burst de CPU indicado em milissegundos:

Processo	Tempo de burst	Prioridade
P_1	10	3
P_2	1	1
P_3	2	4
P_4	1	5
P_5	5	2

Usando o escalonamento por prioridade, esses processos seriam escalonados de acordo com o seguinte diagrama de Gantt:

P_2	P_5	P_1	P_3	P_4
0	1	6	16	18 19

O tempo de espera médio é de 8,2 milissegundos.

As prioridades podem ser definidas interna ou externamente. As prioridades definidas internamente utilizam alguma quantidade ou quantidades mensuráveis para calcular a prioridade de um processo. Por exemplo, limites de tempo, requisitos de memória, o número de arquivos abertos e a razão entre o burst de E/S médio e o burst de CPU médio têm sido usados no cálculo das prioridades. As prioridades externas são definidas por critérios fora do sistema operacional, como a importância do processo, o tipo e a quantidade de fundos pagos pelo uso do computador, o departamento patrocinando o trabalho e outros fatores, normalmente políticos.

O escalonamento por prioridade pode ser preemptivo ou não preemptivo. Quando um processo chega na fila de prontos (ready queue), sua prioridade é comparada com a prioridade do processo em execução. Um algoritmo de escalonamento por prioridade preemptivo se apropriará da CPU se a prioridade do processo recém-chegado for mais alta do que a prioridade do processo em execução. Um algoritmo de escalonamento por prioridade não preemptivo colocará o novo processo no início da fila de prontos.

Um problema importante com os algoritmos de escalonamento por prioridade é o **bloqueio indefinido** ou **starvation**. Um processo pronto para ser executado, mas aguardando pela CPU, pode ser considerado bloqueado. Um algoritmo de escalonamento por prioridade pode deixar alguns processos de baixa prioridade esperando indefinidamente. Em um sistema computadorizado bastante sobrecarregado, constantes processos de prioridade mais alta podem impedir que um processo de baixa prioridade sequer tenha atenção da CPU. Em geral, poderá acontecer uma destas duas situações: ou o processo por fim será executado (às duas horas da manhã de domingo, quando o sistema não estiver sobrecarregado) ou o sistema computadorizado, por fim, falhará e perderá todos os processos de baixa prioridade não terminados. (Dizem que, quando desativaram o IBM 7094 no MIT em 1973, encontraram um processo de baixa prioridade que tinha sido submetido em

1967 e ainda não tinha sido executado.)

Uma solução para o problema de bloqueio indefinido dos processos de baixa prioridade é o **envelhecimento (aging)**. O envelhecimento é uma técnica de aumentar gradualmente a prioridade dos processos que estão esperando no sistema por um longo tempo. Por exemplo, se as prioridades variarem de 127 (baixa) até 0 (alta), poderemos aumentar em 1 a prioridade do processo aguardando a cada 15 minutos. Por fim, até mesmo um processo com prioridade inicial de 127 teria a maior prioridade no sistema e seria executado. De fato, não seriam necessárias mais do que 32 horas para um processo de prioridade 127 envelhecer até um processo de prioridade 0.

5.3.4 Escalonamento round-robin

O **algoritmo de escalonamento round-robin (RR)** (ou **revezamento**) foi criado para sistemas de tempo compartilhado. Ele é semelhante ao escalonamento FCFS, mas a preempção é acrescentada para permitir que o sistema alterne entre os processos. Uma pequena unidade de tempo, chamada **quantum de tempo** ou fatia de tempo, é definida. Um quantum de tempo costuma variar de 10 a 100 milissegundos de extensão. A fila de prontos é tratada como uma fila circular. O escalonador de CPU percorre a fila de prontos, alocando a CPU a cada processo por um intervalo de tempo de até 1 quantum de tempo.

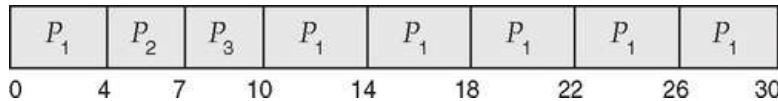
Para implementar o escalonamento RR, mantemos a fila de prontos como uma fila de processos FIFO. Novos processos são acrescentados ao final da fila de prontos. O escalonador de CPU apanha o primeiro processo da fila de prontos, define um temporizador para interromper após 1 quantum de tempo e despacha o processo.

Em seguida, acontecerá uma das duas situações seguintes. O processo pode ter um burst de CPU de menos de 1 quantum de tempo. Nesse caso, o próprio processo liberará a CPU voluntariamente. O escalonador prosseguirá, então, para o próximo processo na fila de prontos. Caso contrário, se o burst de CPU do processo em execução for maior do que 1 quantum de tempo, o temporizador disparará e causará uma interrupção no sistema operacional. Uma troca de contexto será executada e o processo será colocado no **final** da fila de prontos. Por fim, o escalonador de CPU selecionará o próximo processo na fila de prontos.

O tempo de espera médio sob a política RR normalmente é longo. Considere o seguinte conjunto de processos que chegam no momento 0, com a extensão do tempo de burst de CPU indicada em milissegundos:

Processo	Tempo de burst
P_1	24
P_2	3
P_3	3

Se usarmos um quantum de 4 milissegundos, então o processo P_1 receberá os primeiros 4 milissegundos. Como ele precisa de outros 20 milissegundos, é interrompido depois do primeiro quantum de tempo, e a CPU é dada ao próximo processo na fila, o processo P_2 . Como o processo P_2 não precisa de 4 milissegundos, ele termina antes do término do seu quantum de tempo. A CPU, então, é dada ao próximo processo, P_3 . Quando cada processo tiver recebido 1 quantum de tempo, a CPU retornará ao processo P_1 , para um quantum de tempo adicional. O escalonamento RR resultante é o seguinte:



Vamos calcular o tempo de espera médio para o escalonamento acima. P_1 espera por 6 milissegundos ($10 - 4$), P_2 espera por 4 milissegundos e P_3 espera por 7 milissegundos. Então, o tempo de espera médio é $17/3 = 5,66$ milissegundos.

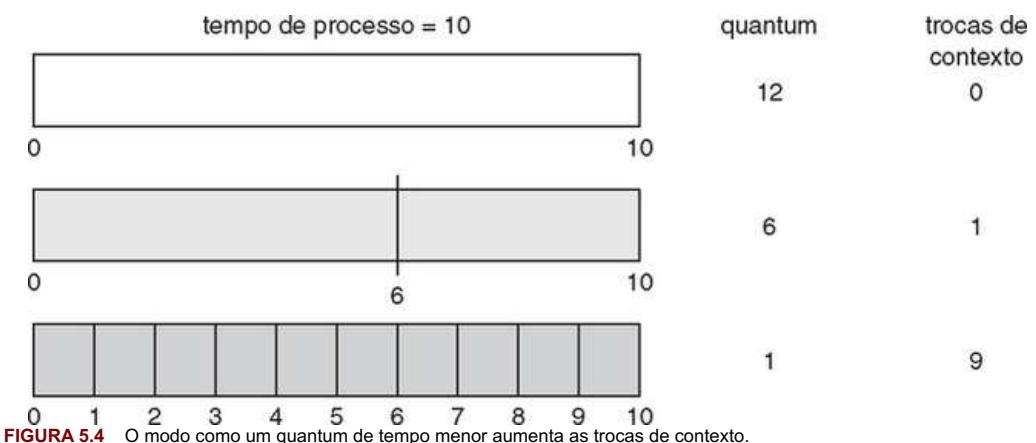
No algoritmo de escalonamento RR, nenhum processo recebe a CPU por mais de 1 quantum de tempo seguido (a menos que seja o único processo executável). Se o burst de CPU de um processo ultrapassar 1 quantum de tempo, esse processo será *preemptado* e colocado de volta na fila de prontos. O algoritmo de escalonamento RR, portanto, é preemptivo.

Se houver n processos na fila de prontos e o quantum de tempo for q , então cada processo receberá $1/n$ do tempo de CPU em pedaços de, no máximo, q unidades de tempo. Cada processo não pode esperar mais do que $(n - 1) \times q$ unidades de tempo até o seu próximo quantum de tempo. Por exemplo, com cinco processos e um quantum de tempo de 20 milissegundos, cada processo receberá até 20 milissegundos a cada 100 milissegundos.

O desempenho do algoritmo RR depende bastante do tamanho do quantum de tempo. Em um extremo, se o quantum de tempo for extremamente grande, a política RR será igual à política FCFS. Por outro lado, se o quantum de tempo for extremamente pequeno (digamos, 1 milissegundo), a

técnica RR será chamada **processor sharing (compartilhamento de processador)** e (teoricamente) criará a aparência de que cada um dos n processos possui seu próprio processador executando em $1/n$ da velocidade do processador real. Essa técnica foi usada no hardware da Control Data Corporation (CDC) para implementar 10 processadores periféricos com somente um conjunto de hardware e 10 conjuntos de registradores. O hardware executa uma instrução para um conjunto de registradores, depois passa para a seguinte. Esse ciclo continua, resultando em 10 processadores lentos, em vez de um rápido. (Na verdade, como o processador era muito mais rápido do que a memória e cada instrução referenciava a memória, os processadores não eram muito mais lentos do que 10 processadores reais teriam sido.)

No software, porém, também precisamos considerar o efeito da troca de contexto sobre o desempenho do escalonamento RR. Suponha, por exemplo, que temos apenas um processo de 10 unidades de tempo. Se o quantum for 12 unidades de tempo, o processo termina em menos de 1 quantum de tempo, sem custo adicional. No entanto, se o quantum for 6 unidades de tempo, o processo exigirá 2 quanta, resultando em uma troca de contexto. Se o quantum de tempo for 1 unidade de tempo, então haverá 9 trocas de contexto, atrasando, assim, a execução do processo (Figura 5.4).



Assim, queremos que o quantum de tempo seja grande com relação ao tempo de troca de contexto. Se o tempo de troca de contexto for aproximadamente 10% do quantum de tempo, então cerca de 10% do tempo de CPU serão gastos na troca de contexto. Na prática, a maioria dos sistemas modernos possui valores de quantum de tempo variando de 10 a 100 milissegundos. O tempo exigido para uma troca de contexto normalmente é de menos de 10 microssegundos; assim, o tempo para troca de contexto é uma fração muito pequena do quantum de tempo.

O turnaround time também depende do tamanho do quantum de tempo. Como podemos ver na Figura 5.5, o turnaround time médio de um conjunto de processos não melhora necessariamente com o aumento de tamanho do quantum de tempo. Em geral, o turnaround time médio pode ser melhorado se a maioria dos processos terminar seu próximo burst de CPU em um único quantum de tempo. Por exemplo, dados três processos de 10 unidades de tempo cada e um quantum de 1 unidade de tempo, o turnaround time médio é 29. Contudo, se o quantum de tempo for 10, o turnaround time médio cai para 20. Se o tempo de troca de contexto for incluído, o turnaround time médio aumentará ainda mais para um quantum de tempo menor, pois mais trocas de contexto serão necessárias.

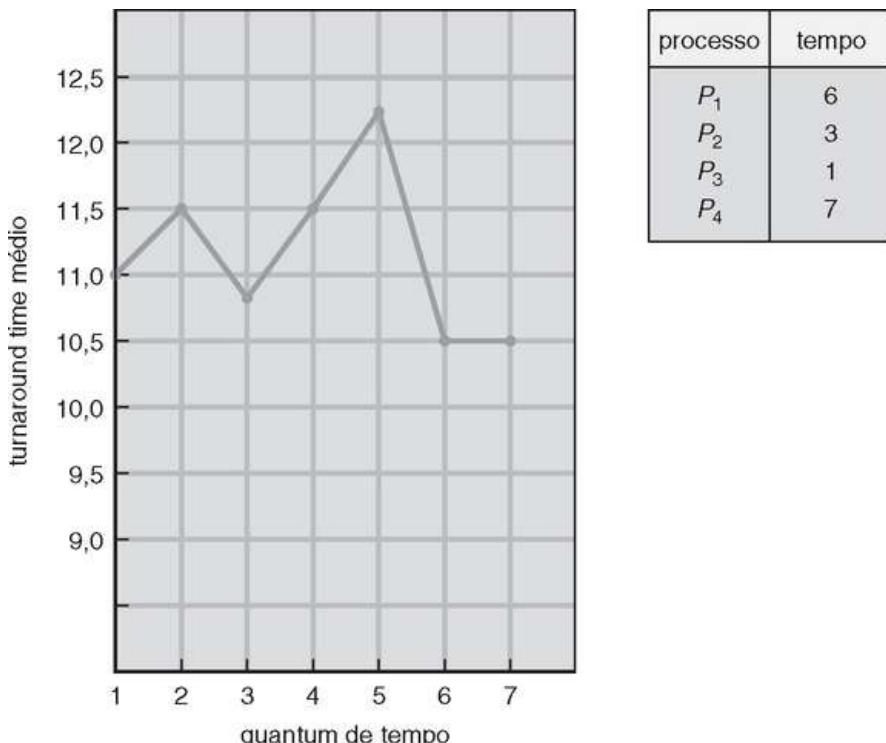


FIGURA 5.5 O modo como o turnaround time varia com o quantum de tempo.

Embora o quantum de tempo deva ser grande em comparação com o tempo de troca de contexto, ele não deve ser muito grande. Se o quantum de tempo for muito grande, como já dissemos, o escalonamento RR degenera para a política FCFS. Uma regra prática é que 80% dos bursts de CPU devem ser menores do que o quantum de tempo.

5.3.5 Escalonamento Multilevel Queue

Outra classe de algoritmos de escalonamento foi criada para situações em que os processos são classificados em diferentes grupos. Por exemplo, uma divisão comum é feita entre processos de **primeiro plano** (interativos) e processos em **segundo plano** (batch). Esses dois tipos de processos possuem requisitos diferentes para o tempo de resposta e, por isso, podem ter necessidades de escalonamento diferentes. Além disso, os processos de primeiro plano podem ter prioridade (definida externamente) acima dos processos de segundo plano.

Um **algoritmo multilevel queue scheduling (fila multinível de escalonamento)** divide a fila de prontos (ready queue) em várias filas separadas (Figura 5.6). Cada processo é atribuído a uma fila, em geral com base em alguma propriedade do processo, como tamanho de memória, prioridade do processo ou tipo de processo. Cada fila possui seu próprio algoritmo de escalonamento. Por exemplo, filas separadas poderiam ser usadas para processos de primeiro e segundo planos. A fila de primeiro plano poderia ser escalonada por um algoritmo RR, enquanto a fila de segundo plano seria escalonada por um algoritmo FCFS.

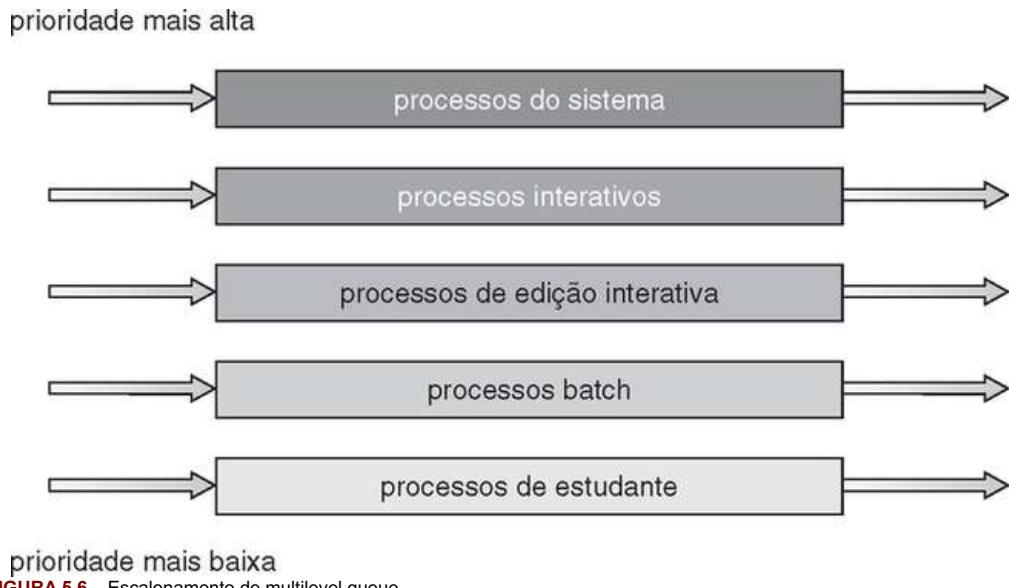


FIGURA 5.6 Escalonamento de multilevel queue.

Além disso, é preciso haver escalonamento entre as filas, o que é implementado como um escalonamento preemptivo de prioridade fixa. Por exemplo, a fila de primeiro plano pode ter prioridade absoluta sobre a fila de segundo plano.

Vejamos um exemplo de algoritmo de escalonamento de fila multinível com cinco filas, listadas a seguir na ordem de prioridade:

1. Processos do sistema.
2. Processos interativos.
3. Processos de edição interativa.
4. Processos batch.
5. Processos de estudante.

Cada fila possui prioridade absoluta acima das filas de menor prioridade. Nenhum processo na fila batch, por exemplo, poderia ser executado a menos que as filas para os processos do sistema, processos interativos e processos de edição interativa estivessem todas vazias. Se um processo de edição interativa entrasse na fila de prontos enquanto um processo batch estivesse sendo executado, o processo batch teria sido preemptado.

Outra possibilidade é dividir a fatia de tempo entre as filas. Cada fila recebe determinada parte do tempo de CPU, que pode então ser escalonado entre os diversos processos em sua fila. Por exemplo, no exemplo da fila de primeiro e segundo planos, a fila de primeiro plano pode receber 80% do tempo de CPU para o escalonamento RR entre seus processos, enquanto a fila de segundo plano recebe 20% da CPU para dar a seus processos com base na política FCFS.

5.3.6 Escalonamento Multilevel Feedback Queue

Em geral, quando um algoritmo de escalonamento de fila multinível é usado, os processos recebem uma fila permanentemente na entrada do sistema. Por exemplo, se houver filas separadas para processos de primeiro e segundo planos, eles não se movem de uma fila para outra, pois não podem mudar sua natureza de primeiro ou segundo plano. Essa configuração tem a vantagem do baixo custo adicional de escalonamento, mas é inflexível.

O **escalonamento multilevel feedback queue (fila multinível com feedback)**, ao contrário, permite que um processo se move entre as filas. A ideia é separar os processos de acordo com as características de burst de CPU. Se um processo utilizar muito tempo de CPU, ele será movido para uma fila de menor prioridade. Esse esquema deixa os processos I/O-bound e interativos nas filas de maior prioridade. Além disso, um processo que espera muito tempo em uma fila de menor prioridade pode ser movido para uma fila de maior prioridade. Essa forma de envelhecimento evita o starvation.

Por exemplo, considere um escalonador multilevel feedback queue com três filas, numeradas de 0 a 2 (Figura 5.7). O escalonador primeiro executa todos os processos na fila 0. Somente quando a fila 0 estiver vazia, ele executará os processos na fila 1. De modo semelhante, os processos na fila 2 só serão executados se as filas 0 e 1 estiverem vazias. Um processo que chega na fila 1 tomará o lugar de um processo na fila 2. Um processo na fila 1, por sua vez, dará lugar a um processo chegando na fila 0.

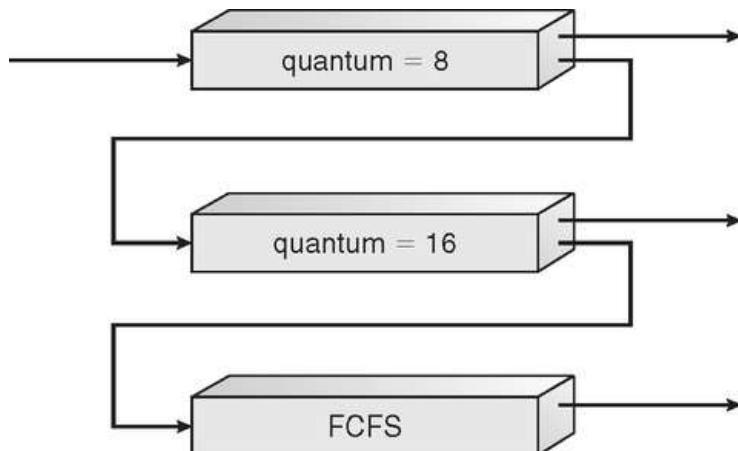


FIGURA 5.7 Multilevel feedback queues.

Um processo entrando na fila de prontos é colocado na fila 0. Um processo na fila 0 recebe um quantum de tempo de 8 milissegundos. Se ele não terminar dentro desse tempo, será movido para o fim da fila 1. Se a fila 0 estiver vazia, o processo no início da fila 1 receberá um quantum de 16 milissegundos. Se ele não terminar, será substituído e colocado na fila 2. Os processos na fila 2 são executados com base na política FCFS, mas apenas quando as filas 0 e 1 estiverem vazias.

Esse algoritmo de escalonamento fornece a mais alta prioridade a qualquer processo com um burst de CPU de 8 milissegundos ou menos. Tal processo rapidamente chegará à CPU, terminará seu burst de CPU e seguirá para seu próximo burst de E/S. Os processos que precisam de mais de 8, porém menos de 24 milissegundos, também são atendidos rapidamente, embora com prioridade menor do que os processos mais curtos. Os processos longos caem automaticamente para a fila 2 e são atendidos segundo a ordem FCFS, com quaisquer ciclos de CPU restantes das filas 0 e 1.

Em geral, um escalonador multilevel feedback queue é definido pelos seguintes parâmetros:

- Número de filas.
- Algoritmo de escalonamento para cada fila.
- Método usado para determinar quando atualizar um processo para uma fila de maior prioridade.
- Método usado para determinar quando rebaixar um processo para uma fila de menor prioridade.
- Método usado para determinar em que fila um processo entrará quando esse processo precisar de atendimento.

A definição de um escalonador multilevel feedback queue o torna o algoritmo de escalonamento de CPU mais genérico. Ele pode ser configurado para corresponder a um sistema específico em projeto. Infelizmente, ele também é o algoritmo mais complexo, pois a definição do melhor escalonador exige algum meio de selecionar valores para todos os parâmetros.

5.4 Escalonamento de thread

No [Capítulo 4](#), apresentamos as threads ao modelo de processo, distinguindo entre threads *no nível do usuário* e *no nível do kernel*. Nos sistemas operacionais que os admitem, são as threads no nível do kernel – e não os processos – que estão sendo escalonados pelo sistema operacional. As threads no nível do usuário são gerenciadas por uma biblioteca threads, e o kernel não as conhece. Para executar em uma CPU, as threads no nível do usuário precisam ser mapeadas a uma thread no nível do kernel, embora esse mapeamento possa ser indireto e utilizar um processo leve (LWP). Nesta seção, vamos explorar as questões de escalonamento entre as threads no nível do usuário e no nível do kernel, fornecendo exemplos específicos de escalonamento para Pthreads.

5.4.1 Escopo de disputa

Uma distinção entre as threads no nível do usuário e no nível do kernel está em como são agendadas. Em sistemas implementando os modelos muitos para um ([Seção 4.2.1](#)) e muitos para muitos ([Seção 4.2.3](#)), a biblioteca threads escalona as threads no nível do usuário para executarem em um LWP disponível, um esquema conhecido como **escopo de disputa do processo (Process Contention Scope - PCS)**, pois a disputa pela CPU ocorre entre as threads pertencentes ao mesmo processo. Quando dizemos que a biblioteca threads *escalona* as threads do usuário nos LWPs disponíveis, não queremos dizer que a thread está realmente sendo executada em uma CPU; isso exigiria que o sistema operacional escalonasse a thread do kernel em uma CPU física. Para decidir qual thread do kernel será escalonada em uma CPU, o número utiliza o **escopo de disputa do sistema (System-Contention Scope - SCS)**. A disputa pela CPU com escalonamento SCS ocorre entre todas as threads no sistema. Os sistemas usando o modelo um para um ([Seção 4.2.2](#)), como Windows XP, Solaris e Linux, escalonam as threads usando apenas o SCS.

Normalmente, o PCS é feito de acordo com a prioridade – o escalonador seleciona a thread executável com a maior prioridade para execução. As prioridades da thread do usuário são definidas pelo programador e não são ajustadas pela biblioteca threads, embora algumas bibliotecas threads possam permitir que o programador mude sua prioridade. É importante observar que o PCS normalmente interromperá a thread sendo executada em favor de uma thread com maior prioridade; porém, não há garantias de fatia de tempo ([Seção 5.3.4](#)) entre as threads de mesma prioridade.

5.4.2 Escalonamento Pthread

Na [Seção 4.3.1](#), apresentamos um exemplo de programa POSIX Pthread e também uma introdução sobre a criação de threads. Agora destacaremos a API POSIX Pthread que nos permite especificar PCS ou SCS durante a criação de threads. Pthreads identifica os seguintes valores de escopo de disputa:

- `PTHREAD_SCOPE_PROCESS` escalona threads usando o escalonamento PCS.
- `PTHREAD_SCOPE_SYSTEM` escalona threads usando o escalonamento SCS.

Nos sistemas implementando o modelo muitos para muitos, a política `PTHREAD_SCOPE_PROCESS` escalona as threads no nível do usuário para os LWPs disponíveis. A quantidade de LWPs é mantida pela biblioteca threads, talvez usando ativações do escalonador ([Seção 4.5.6](#)). A política de escalonamento `PTHREAD_SCOPE_SYSTEM` criará e associará um LWP para cada thread no nível do usuário nos sistemas muitos para muitos, efetivamente mapeando as threads por meio da política um para um.

O IPC Pthread fornece as duas funções a seguir para obter – e definir – a política de escopo de disputa:

- `pthread_attr_setscope(pthread_attr_t *attr, int scope)`.
- `pthread_attr_getscope(pthread_attr_t *attr, int *scope)`.

O primeiro parâmetro para as duas funções contém um ponteiro para o conjunto de atributos para a thread. O segundo parâmetro para a função `pthread_attr_setscope()` recebe o valor `PTHREAD_SCOPE_SYSTEM` ou `PTHREAD_SCOPE_PROCESS`, indicando como o escopo de disputa deve ser definido. No caso de `pthread_attr_getscope()`, esse segundo parâmetro contém um ponteiro para um valor `int` que é definido como o valor atual do escopo de disputa. Se houver um erro, essas funções retornam valores diferentes de zero.

Na [Figura 5.8](#), ilustramos uma API de escalonamento Pthread que determina o escopo de disputa existente e o define como `PTHREAD_SCOPE_PROCESS`. Depois, ela cria cinco threads separadas que serão executadas usando a política de escalonamento SCS. Observe que, em alguns sistemas, somente determinados valores de escopo de disputa são permitidos. Por exemplo, sistemas Linux e Mac OS X permitem apenas `PTHREAD_SCOPE_SYSTEM`.

```

#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[ ])
{
    int i, scope;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* apanha os atributos padrão */
    pthread_attr_init(&attr);

    /* primeira verificação no escopo atual */
    if (pthread_attr_getscope(&attr, &scope) != 0
        fprintf(stderr, "Impossível obter escopo de escalonamento/n");
    else {
        if (scope == PTHREAD_SCOPE_PROCESS)
            printf("PTHREAD_SCOPE_PROCESS");
        else if (scope == PTHREAD_SCOPE_SYSTEM)
            printf ("PTHREAD_SCOPE_SYSTEM");
        else
            fprintf (stderr, "Valor de escopo inválido./n");
    }

    /* define o algoritmo de escalonamento como PCS ou SCS */
    pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

    /* cria as threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i],&attr,runner,NULL);

    /* agora associa em cada thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}

/* Cada thread iniciará o controle nesta função */
void *runner(void *param)
{
    /* faz algum trabalho... */

    pthread_exit(0);
}

```

FIGURA 5.8 API de escalonamento Pthread.

5.5 Escalonamento em múltiplos processadores

Nossa discussão até aqui focalizou os problemas do escalonamento da CPU em um sistema com um único processador. Se várias CPUs estiverem disponíveis, o **compartilhamento de carga** torna-se possível, portanto o problema do escalonamento será relativamente mais complexo. Muitas possibilidades foram experimentadas; e, como vimos com o escalonamento de CPU com processador único, não existe uma solução melhor. Nesta seção, vamos discutir vários aspectos do escalonamento com multiprocessadores. Vamos nos concentrar em sistemas em que os processadores são idênticos - **homogêneos** - em termos de sua funcionalidade; nesse caso, podemos utilizar qualquer processador disponível para executar quaisquer processos na fila. (Observe, no entanto, que até mesmo dentro de multiprocessadores homogêneos, às vezes existem limitações no escalonamento. Considere um sistema com um dispositivo de E/S ligado a um barramento privado de um processador. Os processos que desejam utilizar esse dispositivo precisam ser escalonados para execução nesse processador.)

5.5.1 Técnicas de escalonamento com multiprocessadores

Uma técnica para o escalonamento de CPU em um sistema com multiprocessadores tem todas as decisões de escalonamento, processamento de E/S e outras atividades do sistema tratadas por um processador único - o servidor mestre. Os outros processadores executam somente o código do usuário. Esse **multiprocessamento assimétrico** é simples porque somente um processador acessa as estruturas de dados do sistema, reduzindo a necessidade de compartilhamento de dados.

Uma segunda técnica utiliza o **multiprocessamento simétrico (SMP)**, no qual cada processador é autoescalonado. Todos os processos podem estar em uma fila de prontos comum ou cada processador pode ter sua própria fila particular de processos prontos. Independentemente, o escalonamento continua com o escalonador para cada processador que examina a fila de prontos comum e seleciona um processo para executar. Como veremos no [Capítulo 6](#), se tivermos vários processadores tentando acessar e atualizar uma estrutura de dados comum, cada escalonador terá de ser programado com cuidado: temos de garantir que dois processadores não escolham o mesmo processo e que os processos não se percam da fila. Praticamente todos os sistemas operacionais modernos admitem SMP, incluindo Windows XP, Windows 2000, Solaris, Linux e Mac OS X. No restante da seção discutiremos as questões sobre os sistemas SMP.

5.5.2 Afinidade de processador

Considere o que acontece com a memória cache quando um processo estiver sendo executado em um processador específico. O modelo de dados acessado mais recentemente pelo processo preenche o cache para o processador; como resultado, acessos sucessivos à memória pelo processo normalmente são satisfeitos na memória cache. Agora, considere o que acontece se o processo migrar para outro processador. O conteúdo da memória cache precisa ser invalidado para o primeiro processador, e o cache para o segundo processador precisa ser repreenchido. Devido ao alto custo da invalidação e repreenchimento dos caches, a maioria dos sistemas SMP tenta evitar a migração de processos de um processador para outro e, em vez disso, tenta manter um processo em execução no mesmo processador. Isso é conhecido como **afinidade de processador** - ou seja, um processo tem afinidade com o processador em que está executando atualmente.

A afinidade de processador tem várias formas. Quando um sistema operacional tem uma política de tentar manter um processo em execução no mesmo processador - mas não garantindo que fará isso -, temos uma situação conhecida como **afinidade flexível**. Aqui, é possível que um processo migre entre os processadores. Alguns sistemas - como o Linux - também fornecem chamadas de sistema que dão suporte à **afinidade rígida**, permitindo que um processo especifique que ele não deve migrar para outros processadores. O Solaris permite que os processos sejam atribuídos a **conjuntos de processadores**, limitando quais processos podem ser executados em quais CPUs. Ele também implementa a afinidade flexível.

A arquitetura da memória principal de um sistema pode afetar as questões de afinidade de processador. A [Figura 5.9](#) ilustra uma arquitetura que possui acesso não uniforme à memória (NUMA - Non-Uniform Memory Access), no qual uma CPU tem acesso mais rápido a algumas partes da memória principal do que a outras partes. Em geral, isso ocorre em sistemas contendo placas de CPU e memória combinadas. As CPUs em uma placa podem acessar a memória nessa placa com menos atraso do que podem acessar a memória em outras placas no sistema. Se o escalonador de CPU do sistema operacional e os algoritmos de disposição de memória trabalharem juntos, então um processo que recebeu afinidade com uma CPU em particular poderá ter memória alocada na placa onde a CPU reside. Este exemplo também mostra que os sistemas operacionais não costumam ser tão claramente definidos e implementados conforme é descrito na literatura sobre sistema operacional. Em vez disso, as "linhas sólidas" entre as seções de um sistema operacional normalmente são apenas "linhas pontilhadas", com algoritmos criando conexões que visam a

otimização do desempenho e da confiabilidade.

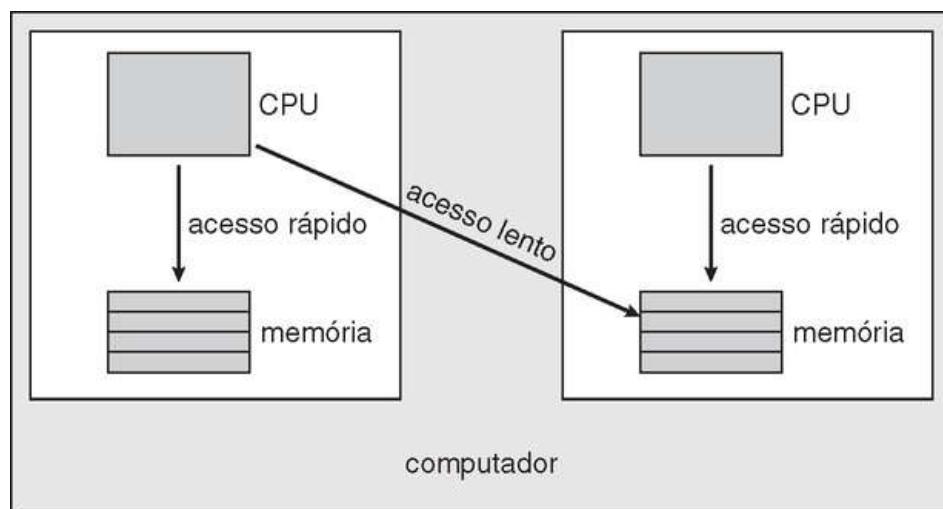


FIGURA 5.9 NUMA e escalonamento de CPU.

5.5.3 Balanceamento de carga

Em sistemas SMP, é importante manter a carga de trabalho balanceada entre todos os processadores, para utilizar por completo os benefícios de se ter mais de um processador. Caso contrário, um ou mais processadores podem ficar ociosos enquanto outros processadores possuem altas cargas de trabalho, junto com listas de processos esperando pela CPU. O **balanceamento de carga** tenta manter a carga de trabalho distribuída uniformemente entre todos os processadores em um sistema SMP. É importante observar que o balanceamento de carga normalmente só é necessário em sistemas em que cada processador tem sua própria fila privada de processos elegíveis para execução. Em sistemas com uma fila de execução comum, o balanceamento de carga normalmente é desnecessário, pois quando um processador fica ocioso, ele extrai imediatamente um processo executável da fila de execução comum. Contudo, também é importante observar que, na maioria dos sistemas operacionais contemporâneos admitindo SMP, cada processador tem uma fila privada de processos elegíveis.

Existem duas técnicas gerais para o balanceamento de carga: **migração push** e **migração pull**. Com a migração push, uma tarefa específica verifica periodicamente a carga em cada processador e - se encontrar um desequilíbrio - distribui a carga de modo uniforme movendo (ou empurrando) processos dos processadores sobrecarregados para os ociosos ou menos ocupados. A migração pull ocorre quando um processador ocioso puxa uma tarefa esperando de um processador ocupado. As migrações push e pull não precisam ser mutuamente exclusivas e, na verdade, normalmente são implementadas em paralelo nos sistemas de balanceamento de carga. Por exemplo, o escalonador do Linux ([Seção 5.6.3](#)) e o escalonador ULE disponível para sistemas FreeBSD implementam essas técnicas. O Linux executa seu algoritmo de balanceamento de carga a cada 200 milissegundos (migração push) ou sempre que a fila de execução para um processador estiver vazia (migração pull).

É interessante que o balanceamento de carga normalmente se opõe aos benefícios da afinidade de processador, discutida na [Seção 5.5.2](#). Ou seja, o benefício de manter um processo em execução no mesmo processador é que o processo pode tirar proveito de seus dados estarem na memória cache desse processador. Empurrando ou puxando um processo de um processador para outro invalidamos esse benefício. Como geralmente acontece na engenharia de sistemas, não existe uma regra absoluta com relação a qual política é a melhor. Assim, em alguns sistemas, um processador ocioso sempre puxa um processo de um processador não ocioso; e, em outros sistemas, os processos só são movidos se o desequilíbrio exceder determinado patamar.

5.5.4 Processadores multicore

Tradicionalmente, os sistemas SMP têm permitido que diversas threads sejam executadas simultaneamente, oferecendo diversos processadores físicos. Porém, uma tendência recente no hardware de computador tem sido colocar vários núcleos (cores) de processador no mesmo chip físico, resultando em um **processador multicore** (ou de múltiplos núcleos). Cada núcleo possui um conjunto de registradores para manter seu estado arquitetônico e, assim, aparece para o sistema operacional como se fosse um processador físico separado. Os sistemas SMP que utilizam

processadores multicore são mais rápidos e consomem menos energia do que os sistemas em que cada processador possui seu próprio chip físico.

Entretanto, os processadores multicore podem tornar os aspectos de escalonamento mais complicados. Vejamos como isso poderia acontecer. Pesquisadores descobriram que, quando um processador acessa a memória, ele gasta um tempo significativo esperando até que os dados estejam disponíveis. Essa situação, conhecida como **stall de memória**, pode ocorrer por diversos motivos, como a falta no cache (acessar dados que não estão na memória cache). A [Figura 5.10](#) ilustra um stall de memória. Neste cenário, o processador pode gastar até 50% do seu tempo esperando até que os dados estejam disponíveis na memória. Para remediar essa situação, muitos projetos de hardware recentes implementaram núcleos de processador multithreaded nos quais duas (ou mais) threads de hardware são atribuídas a cada núcleo. Desse modo, se uma thread fica parada enquanto espera pela memória, o núcleo pode alternar para outra thread. A [Figura 5.11](#) ilustra um núcleo de processador dual-threaded no qual a execução da thread_0 e a execução da thread_1 são intercaladas. Do ponto de vista de um sistema operacional, cada thread de hardware aparece como se fosse um processador lógico que está disponível para executar uma thread de software. Assim, em um sistema dual-threaded, dual-core, quatro processadores lógicos são apresentados ao sistema operacional. A CPU UltraSPARC T1 possui oito núcleos por chip e quatro threads de hardware por núcleo; do ponto de vista do sistema operacional, parece existir 32 processadores lógicos.



FIGURA 5.10 Stall de memória.

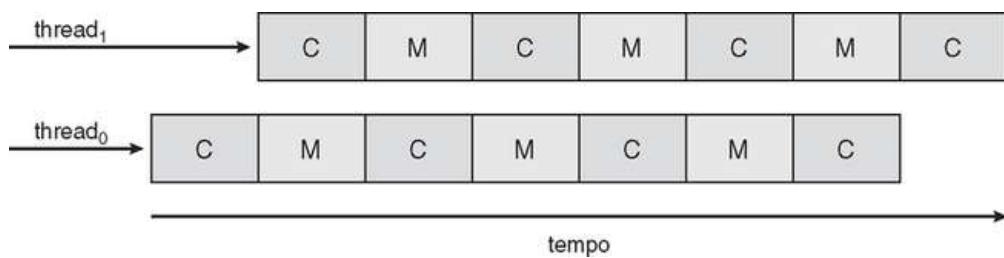


FIGURA 5.11 Sistema multicore com múltiplas threads.

Em geral, existem duas formas de executar múltiplas threads em um processador: multithreading **coarse-grained** e **fine-grained**. Com o multithreading coarse-grained, uma thread é executada em um processador até que ocorra um evento de alta latência, como um stall de memória. Devido ao atraso causado por esse evento, o processador precisa alternar para outra thread para iniciar a execução. Porém, o custo dessa troca entre as threads é alto, pois a pipeline de instruções precisa ser esvaziada antes que a outra thread possa iniciar a execução no núcleo do processador. Quando essa nova thread iniciar sua execução, ela começa a preencher a pipeline com suas próprias instruções. O multithreading fine-grained (ou intercalado) alterna entre as threads em um nível de granularidade muito menor - normalmente, no limite de um ciclo de instrução. Porém, o projeto arquitetônico dos sistemas fine-grained inclui uma lógica para a comutação de threads. Como resultado, o custo da comutação entre as threads é pequeno.

Note que um processador multicore multithreaded na realidade exige dois níveis de escalonamento diferentes. Em um nível estão as decisões de escalonamento que devem ser tomadas pelo sistema operacional quando ele escolhe qual thread de software executar em cada thread de hardware (processador lógico). Para este nível de escalonamento, o sistema operacional pode escolher qualquer algoritmo de escalonamento, como aqueles descritos na [Seção 5.3](#). Um segundo nível de escalonamento especifica como cada núcleo decide qual thread de hardware executar. Existem várias estratégias a serem adotadas nesta situação. O UltraSPARC T1, que mencionamos anteriormente, utiliza um algoritmo de revezamento (round-robin) simples para escalonar as quatro threads de hardware para cada núcleo. Outro exemplo, o Intel Itanium, é um processador dual-core (dois núcleos) com duas threads gerenciadas pelo hardware por núcleo. Associado a cada thread de

hardware existe um valor dinâmico de *urgência*, variando de 0 a 7, com 0 representando a mais baixa urgência, e 7 a mais alta. O Itanium identifica cinco eventos diferentes que podem disparar uma comutação de threads. Quando ocorre um desses eventos, a lógica de comutação de threads compara a urgência das duas threads e seleciona aquela com o valor mais alto para executar no núcleo do processador.

5.5.5 Virtualização e escalonamento

Um sistema com virtualização, mesmo sendo um sistema de única CPU, constantemente atua como um sistema multiprocessador. O software de virtualização apresenta uma ou mais CPUs virtuais a cada uma das máquinas virtuais rodando no sistema e depois escalona o uso das CPUs físicas entre as máquinas virtuais. As variações significativas entre tecnologias de virtualização tornam difícil resumir os efeitos da virtualização no escalonamento ([Seção 2.8](#)). Em geral, no entanto, a maioria dos ambientes virtualizados possui um sistema operacional hospedeiro e muitos sistemas operacionais guest. O sistema operacional hospedeiro cria e gerencia as máquinas virtuais, e cada máquina virtual possui um sistema operacional guest instalado e aplicações sendo executadas dentro desse guest. Cada sistema operacional guest pode ser ajustado para casos de uso, aplicações e usuários específicos, incluindo a operação em tempo compartilhado e até mesmo em tempo real.

Qualquer algoritmo de escalonamento do sistema operacional guest considera que determinada quantidade de progresso em determinada quantidade de tempo será afetada negativamente pela virtualização. Considere um sistema operacional de tempo compartilhado que tenta alocar 100 milissegundos a cada fatia de tempo, para dar aos usuários um tempo de resposta razoável. Em uma máquina virtual, esse sistema operacional está à mercê do sistema de virtualização quanto a quais recursos da CPU ele realmente receberá. Determinada fatia de tempo de 100 milissegundos pode levar muito mais do que 100 milissegundos de tempo de CPU virtual. Dependendo da ocupação do sistema, a fatia de tempo pode levar um segundo ou mais, resultando em tempos de resposta muito fracos para os usuários conectados a essa máquina virtual. O efeito sobre um sistema operacional de tempo real seria ainda mais catastrófico.

O efeito resultante dessas camadas de escalonamento é que os sistemas operacionais virtualizados recebem individualmente apenas uma parte dos ciclos de CPU disponíveis, embora acreditem que estão recebendo todos os ciclos e que, de fato, estão escalonando todos esses ciclos. Normalmente, os relógios nas máquinas virtuais ficam incorretos porque os temporizadores levam mais tempo para serem disparados do que levariam em CPUs dedicadas. A virtualização, portanto, pode desfazer os bons esforços do algoritmo de escalonamento dos sistemas operacionais nas máquinas virtuais.

5.6 Exemplos de sistema operacional

A seguir, vejamos uma descrição das políticas de escalonamento dos sistemas operacionais Solaris, Windows XP e Linux. É importante lembrar que estamos descrevendo o escalonamento das threads de kernel com Solaris e Windows XP. Lembre-se de que o Linux não distingue entre processos e threads; assim, usamos o termo *tarefa* quando discutimos sobre o escalonador do Linux.

5.6.1 Exemplo: escalonamento no Solaris

O Solaris utiliza o escalonamento de thread com base em prioridade, onde cada thread pertence a uma dentre seis classes:

1. Tempo compartilhado (TS).
2. Interativo (IA).
3. Tempo real (RT).
4. Sistema (SYS).
5. Fazia justa (FSS).
6. Prioridade fixa (FP).

Dentro de cada classe existem diferentes prioridades e diferentes algoritmos de escalonamento.

A classe de escalonamento-padrão para um processo é a de tempo compartilhado. A política de escalonamento para a classe de tempo compartilhado altera as prioridades dinamicamente e atribui fatias de tempo de diferentes tamanhos usando uma multilevel feedback queue. Como padrão, existe um relacionamento inverso entre as prioridades e as fatias de tempo: quanto maior a prioridade, menor a fatia de tempo; e quanto menor a prioridade, maior a fatia de tempo. Os processos interativos normalmente possuem uma prioridade mais alta; processos CPU-bound, menor prioridade. Essa política de escalonamento fornece um bom tempo de resposta para os processos interativos e um bom throughput para os processos CPU-bound. A classe interativa utiliza a mesma política de escalonamento da classe de tempo compartilhado, mas dá às aplicações de janelas – como aquelas criadas pelos gerenciadores de janelas KDE e GNOME – uma prioridade maior, para aumentar o desempenho.

A [Figura 5.12](#) ilustra a tabela de despacho para o escalonamento de threads interativas e de tempo compartilhado. Essas duas classes de escalonamento incluem 60 níveis de prioridade, mas, para resumir, apresentaremos apenas algumas. A tabela de despacho mostrada na [Figura 5.12](#) contém os seguintes campos:

Prioridade	Quantum de tempo	Quantum de tempo esgotado	Retorno do sono
0	200	0	50
5	200	0	50
10	160	0	51
15	160	5	51
20	120	10	52
25	120	15	52
30	80	20	53
35	80	25	54
40	40	30	55
45	40	35	56
50	40	40	58
55	40	45	58
59	20	49	59

FIGURA 5.12 Tabela de despacho do Solaris para threads interativas e de tempo compartilhado.

■ **Prioridade.** A prioridade dependente da classe para as classes de tempo compartilhado e interativa. Um número mais alto indica uma prioridade mais alta.

■ **Quantum de tempo.** O quantum de tempo para a prioridade associada. Isso ilustra o relacionamento inverso entre as prioridades e os valores de quantum de tempo: a prioridade mais baixa (prioridade 0) possui o maior quantum de tempo (200 milissegundos), a prioridade mais alta (prioridade 59) possui o menor quantum de tempo (20 milissegundos).

■ **Quantum de tempo esgotado.** A nova prioridade de uma thread que usou seu quantum de

tempo inteiro sem bloquear. Essas threads são consideradas de uso intenso de CPU. Como vemos na tabela, tais threads possuem suas prioridades reduzidas.

■ **Retorno do sono.** A prioridade de uma thread que está retornando do sono (como ao aguardar pela E/S). Como a tabela ilustra, quando a E/S está disponível para thread aguardando, sua prioridade é aumentada entre 50 e 59, fornecendo suporte para a política de escalonamento fornecer um bom tempo de resposta para processos interativos.

As threads na classe de tempo real recebem a maior prioridade. Essa atribuição permite a um processo de tempo real ter uma resposta garantida do sistema dentro de um período associado. Um processo de tempo real será executado antes de um processo em qualquer outra classe. Contudo, em geral, poucos processos pertencem à classe de tempo real.

O Solaris utiliza a classe do sistema para executar as threads do kernel, como o escalonador e o daemon de paginação. Uma vez estabelecida, a prioridade de um processo do sistema não muda. A classe do sistema é reservada para uso do kernel (os processos do usuário executando no kernel não estão na classe dos sistemas).

O Solaris 9 introduziu duas novas classes de escalonamento: **prioridade fixa (fixed priority)** e **fatia justa (fair share)**. As threads na classe de prioridade fixa possuem o mesmo intervalo de prioridade daqueles na classe de tempo compartilhado; porém, suas prioridades não são ajustadas dinamicamente. A classe de escalonamento por fatia justa utiliza **fatias** de tempo da CPU no lugar de prioridades, para tomar suas decisões de escalonamento. Fatias da CPU indicam direito aos recursos disponíveis da CPU e são alocadas a um conjunto de processos (conhecido como **projeto**).

Cada classe de escalonamento inclui um conjunto de prioridades. Todavia, o escalonador converte as prioridades específicas da classe em prioridades globais, e seleciona executar a thread com a mais alta prioridade global. A thread selecionada é executada na CPU até (1) ser bloqueada; (2) usar sua fatia de tempo; ou (3) ser preemptada por uma thread de maior prioridade. Se houver várias threads com a mesma prioridade, o escalonador utiliza uma fila de revezamento. A [Figura 5.13](#) ilustra como as seis classes de escalonamento estão relacionadas umas às outras e como elas são comparadas a prioridades globais. Observe que o kernel mantém 10 threads para atender as interrupções. Essas threads não pertencem a qualquer classe de escalonamento e são executadas na prioridade mais alta (160-169). Como mencionado, o Solaris tradicionalmente tem usado o modelo muitos para muitos ([Seção 4.2.3](#)), mas o Solaris 9 passou para o modelo um para um ([Seção 4.2.2](#)).

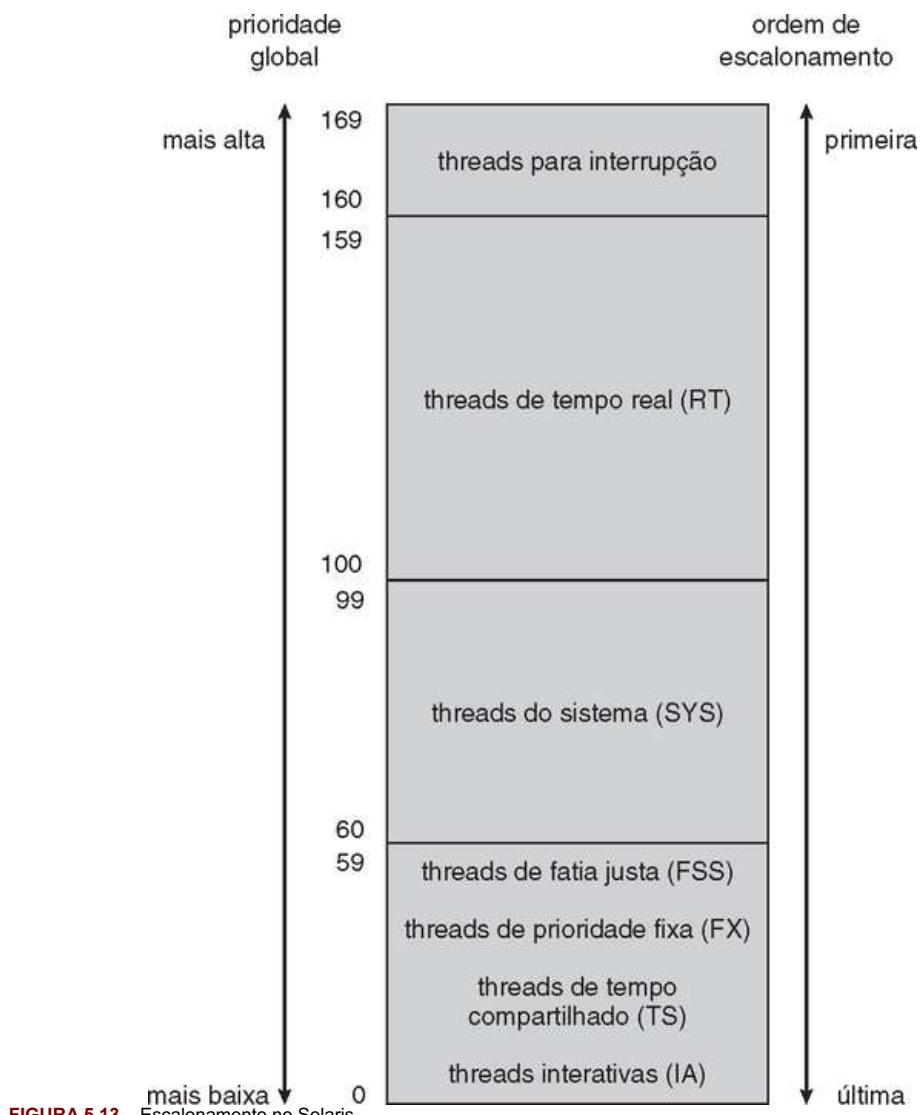


FIGURA 5.13 Escalonamento no Solaris.

5.6.2 Exemplo: escalonamento no Windows XP

O Windows XP escalona as threads usando um algoritmo de escalonamento com base em prioridade, preemptivo. O escalonador do Windows XP garante que a thread de maior prioridade sempre será executada. A parte do kernel do Windows XP que trata do escalonamento é chamada *despachante*. Uma thread selecionada para execução pelo despachante será executada até ser interrompida por uma thread de maior prioridade, até terminar, até seu quantum de tempo terminar ou até invocar uma chamada de sistema bloqueante, como na E/S. Se uma thread de tempo real de maior prioridade ficar pronta enquanto uma thread de menor prioridade estiver sendo executada, a thread de menor prioridade será interrompida. Essa preempção dá a uma thread de tempo real acesso preferencial à CPU quando a thread precisar de tal acesso.

O despachante usa um esquema de prioridade (numerado de 0 a 31) de 32 níveis para determinar a ordem de execução da thread, na qual um número maior indica uma prioridade mais elevada. As prioridades são divididas em duas classes. A **classe variável** contém threads com prioridade de 1 a 15, e a **classe de tempo real** contém threads com prioridades variando de 16 a 31. (Há também uma thread sendo executada na prioridade 0, que é usada para gerência de memória.) O despachante utiliza uma fila para cada prioridade de escalonamento e atravessa o conjunto de filas da mais alta para a mais baixa, até encontrar uma thread que esteja pronta para ser executada. Se nenhuma thread pronta for encontrada, o despachante executará uma thread especial, chamada **thread ociosa (idle)**.

Existe um relacionamento entre as prioridades numéricas do kernel do Windows XP e a API Win32. A API Win32 identifica várias classes de prioridade às quais um processo pode pertencer. Elas incluem:

- `REALTIME_PRIORITY_CLASS`.
- `HIGH_PRIORITY_CLASS`.
- `ABOVE_NORMAL_PRIORITY_CLASS`.

- NORMAL_PRIORITY_CLASS.
- BELOW_NORMAL_PRIORITY_CLASS.
- IDLE_PRIORITY_CLASS.

As prioridades são variáveis em todas as classes, exceto REALTIME_PRIORITY_CLASS, significando que a prioridade de uma thread pertencente a uma dessas classes pode mudar.

Uma thread dentro de uma certa classe de prioridade tem uma prioridade relativa. Os valores para prioridade relativa incluem:

- TIME_CRITICAL.
- HIGHEST.
- ABOVE_NORMAL.
- NORMAL.
- BELOW_NORMAL.
- LOWEST.
- IDLE.

A prioridade de cada thread baseia-se na classe de prioridade a que pertence e na sua prioridade relativa dentro dessa classe. Esse relacionamento pode ser visto na [Figura 5.14](#). Os valores das classes de prioridade aparecem na fileira de cima. A coluna da esquerda contém os valores para as prioridades relativas. Por exemplo, se a prioridade relativa de uma thread na ABOVE_NORMAL_PRIORITY_CLASS for NORMAL, a prioridade numérica desse thread será 10.

	tempo real	alta	acima do normal	normal	abaixo do normal	prioridade ociosa
tempo crítico	31	15	15	15	15	15
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
mais baixa	22	11	8	6	4	2
ocioso	16	1	1	1	1	1

FIGURA 5.14 Prioridades no Windows XP.

Além do mais, cada thread possui uma prioridade básica, representando um valor no intervalo de prioridades para a classe à qual a thread pertence. Como padrão, a prioridade básica é o valor da prioridade relativa NORMAL para essa classe específica. As prioridades básicas para cada classe de prioridade são:

- REALTIME_PRIORITY_CLASS - 24.
- HIGH_PRIORITY_CLASS - 13.
- ABOVE_NORMAL_PRIORITY_CLASS - 10.
- NORMAL_PRIORITY_CLASS - 8.
- BELOW_NORMAL_PRIORITY_CLASS - 6.
- IDLE_PRIORITY_CLASS - 4.

Os processos normalmente são membros da NORMAL_PRIORITY_CLASS. Um processo pertence a essa classe, a menos que o pai do processo tenha sido da IDLE_PRIORITY_CLASS ou a menos que outra classe tenha sido especificada quando o processo foi criado. A prioridade inicial de uma thread normalmente é a prioridade básica do processo ao qual a thread pertence.

Quando o quantum de tempo de uma thread se esgota, essa thread é interrompida; se a thread estiver na classe de prioridade variável, sua prioridade será reduzida. Contudo, a prioridade nunca é reduzida para menos do que a prioridade básica. A redução da prioridade costuma limitar o consumo de CPU das threads voltadas para cálculo. Quando uma thread de prioridade variável é liberada de sua operação de espera, o despachante aumenta a prioridade. A quantidade de aumento depende daquilo que a thread estava esperando; por exemplo, uma thread que estava esperando uma E/S do teclado receberia um aumento grande, por exemplo, enquanto uma thread esperando por uma operação de disco receberia um aumento moderado. Essa estratégia costuma fornecer bons tempos de resposta para threads interativas, que estão usando o mouse e janelas. Ela também permite que threads I/O-bound mantenham os dispositivos de I/O ocupados enquanto permite que threads voltadas para cálculo utilizem ciclos de CPU de reserva em segundo plano. Essa estratégia é usada por vários sistemas operacionais de tempo compartilhado, incluindo o UNIX. Além disso, a janela com a qual o usuário está interagindo atualmente também recebe um aumento de prioridade para melhorar seu tempo de resposta.

Quando um usuário está executando um programa interativo, o sistema precisa prover um desempenho especialmente bom. Por esse motivo, o Windows XP possui uma regra de escalonamento especial para os processos na NORMAL_PRIORITY_CLASS. O Windows XP distingue entre o *processo de primeiro plano*, que está atualmente selecionado na tela, e os *processos de*

segundo plano, que não estão selecionados. Quando um processo passa para o primeiro plano, o Windows XP aumenta o quantum de escalonamento em algum fator (normalmente, 3). Esse aumento dá ao processo de primeiro plano três vezes mais tempo para executar antes de ocorrer uma preempção de tempo compartilhado.

5.6.3 Exemplo: escalonamento no Linux

Antes da versão 2.5, o kernel do Linux executava uma variação do algoritmo de escalonamento tradicional do UNIX. Dois problemas com o escalonador UNIX tradicional são que ele não fornece suporte adequado para sistemas SMP e não se expande bem à medida que o número de tarefas no sistema cresce. Com a versão 2.5, o escalonador foi melhorado, e o kernel agora fornece um algoritmo de escalonamento que é executado em tempo constante - conhecido como $O(1)$ - independentemente do número de tarefas no sistema. O mais novo escalonador também fornece melhor suporte para SMP, incluindo afinidade de processador e balanceamento de carga, além de fornecer justiça e suporte para tarefas interativas.

O escalonador do Linux é um algoritmo preemptivo, baseado em prioridade, com dois intervalos de prioridade separados: um intervalo de **tempo real**, de 0 a 99, e um valor **nice**, variando de 100 a 140. Esses dois intervalos são mapeados em um esquema de prioridade global, no qual valores numericamente mais baixos indicam prioridades mais altas.

Diferentemente dos escalonadores para muitos outros sistemas, incluindo Solaris ([Seção 5.6.1](#)) e Windows XP ([Seção 5.6.2](#)), o Linux atribui, a tarefas de prioridade mais alta, quotas de tempo maiores, e a tarefas de prioridade mais baixa, quotas de tempo menores. O relacionamento entre as prioridades e a extensão da fatia de tempo aparece na [Figura 5.15](#).



FIGURA 5.15 Relacionamento entre prioridades e tamanho da fatia de tempo.

Uma tarefa executável é considerada elegível para execução na CPU desde que tenha tempo restante em sua fatia de tempo. Quando uma tarefa tiver esgotado sua fatia de tempo, ela será considerada expirada, e não elegível para execução novamente, até que todas as outras tarefas também tenham esgotado sua quota de tempo. O kernel mantém uma lista de todas as tarefas executáveis em uma estrutura de dados **runqueue**. Devido a esse suporte para SMP, cada processador mantém sua própria runqueue e escalona-se independentemente. Cada runqueue contém dois arrays de prioridade: **ativo** e **expirado**. O array ativo inclui todas as tarefas com tempo restante em suas fatias de tempo, e o array expirado inclui todas as tarefas expiradas. Cada um desses arrays de prioridade contém uma lista de tarefas indexadas de acordo com a prioridade ([Figura 5.16](#)). O escalonador escolhe a tarefa com a prioridade mais alta a partir do array ativo para execução na CPU. Em máquinas multiprocessadas, isso significa que cada processador está escalonando a tarefa de prioridade mais alta a partir de sua própria estrutura runqueue. Quando todas as tarefas tiverem esgotado suas fatias de tempo (ou seja, o array ativo estiver vazio), os dois arrays de prioridade serão trocados: o array expirado torna-se o array ativo, e vice-versa.

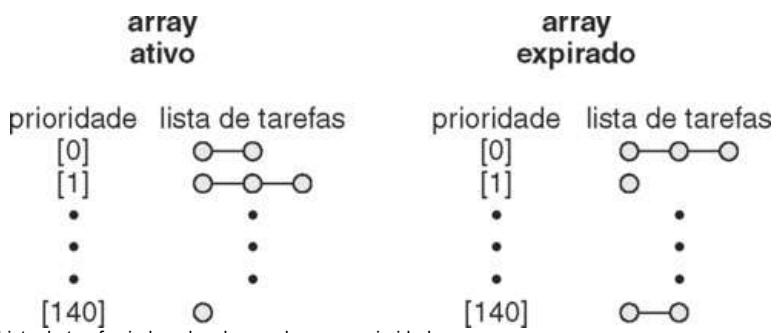


FIGURA 5.16 Lista de tarefas indexadas de acordo com a prioridade.

O Linux implementa escalonamento em tempo real, conforme definido pelo POSIX.1b, que é descrito na [Seção 5.4.2](#). As tarefas de tempo real recebem prioridades estáticas. Todas as outras tarefas possuem prioridades dinâmicas que são baseadas em seus valores *nice*, mais ou menos o valor 5. A interatividade de uma tarefa determina se o valor 5 será somado ou subtraído do valor *nice*. A interatividade de uma tarefa é determinada pelo tempo em que ela esteve dormindo enquanto esperava pela E/S. As tarefas que são mais interativas normalmente possuem tempos de espera maiores e, portanto, provavelmente terão ajustes mais próximos de -5, pois o escalonador favorece as tarefas interativas. O resultado de tais ajustes será prioridades mais altas para essas tarefas. Reciprocamente, tarefas com tempos de espera mais curtos normalmente utilizam mais CPU e, portanto, terão suas prioridades reduzidas.

A prioridade dinâmica de uma tarefa é recalculada quando a tarefa tiver esgotado sua quota de tempo e tiver que ser movida para o array expirado. Assim, quando os dois arrays forem trocados, todas as tarefas no novo array ativo terão recebido novas prioridades e fatias de tempo correspondentes.

5.7 Escalonamento em Java

Vamos considerar o escalonamento em Java. A especificação para a JVM possui uma política de escalonamento livremente definida, que indica que cada thread possui uma prioridade e que threads com prioridade mais alta terão preferência às threads com prioridades mais baixas. Entretanto, ao contrário do caso com escalonamento baseado em prioridade estrita, é possível que uma thread com prioridade mais baixa tenha uma oportunidade de ser executada no lugar de uma thread com prioridade mais alta. Visto que a especificação não diz que uma política de escalonamento precisa ser preemptiva, é possível que uma thread com uma prioridade mais baixa continue a ser executada mesmo quando uma thread com prioridade mais alta se tornar executável.

Além do mais, a especificação para a JVM não indica se as threads têm o tempo repartido por meio de um escalonador de revezamento ([Seção 5.3.4](#)) ou não - essa decisão fica a critério da implementação específica da JVM. Se as threads tiverem o tempo repartido, então uma thread executável será executada até ocorrer um dos seguintes eventos:

1. Seu quantum de tempo se esgota.
2. Ele bloqueia para E/S.
3. Ele sai do seu método `run()`.

Em sistemas que aceitam preempção, uma thread executando em uma CPU também pode ser interrompida por uma thread de prioridade mais alta.

Para que todas as threads tenham uma quantidade igual de tempo de CPU em um sistema que não realiza a repartição de tempo, uma thread pode abandonar o controle da CPU com o método `yield()`. Chamando o método `yield()`, uma thread *sugere* que deseja abrir mão do controle da CPU, permitindo que outra thread tenha a oportunidade de ser executada. Esse abandono de controle é chamado **multitarefa cooperativa**. O uso do método `yield()` aparece como

```
public void run() {
    while (true) {
        // realiza uma tarefa com uso intenso de CPU
        . . .
        // agora abandona o controle da CPU
        Thread.yield();
    }
}
```

5.7.1 Prioridades de thread

Todas as threads em Java recebem uma prioridade que é um inteiro positivo dentro de determinado intervalo. As threads recebem uma prioridade-padrão quando são criadas. A menos que sejam alteradas explicitamente pelo programa, elas mantêm a mesma prioridade durante todo o seu tempo de vida; a JVM não altera prioridades dinamicamente. A classe `Thread` da Java identifica as seguintes prioridades da thread:

Prioridade	Comentário
<code>Thread.MIN_PRIORITY</code>	A prioridade mínima da thread.
<code>Thread.MAX_PRIORITY</code>	A prioridade máxima da thread.
<code>Thread.NORM_PRIORITY</code>	A prioridade-padrão da thread.

`MIN_PRIORITY` possui o valor 1; `MAX_PRIORITY`, o valor 10; e `NORM_PRIORITY`, o valor 5. Cada thread em Java possui uma prioridade que se encontra em algum lugar dentro desse intervalo. (Existe de fato uma prioridade 0, mas é reservada para as threads criadas pela JVM; os desenvolvedores não podem designar uma thread com prioridade 0.) A prioridade-padrão para todas as threads é `NORM_PRIORITY`.

Quando uma thread é criada, ela recebe a mesma prioridade da thread que a criou. A prioridade de uma thread também pode ser definida explicitamente com o método `setPriority()`. A prioridade pode ser definida antes de uma thread ser iniciada ou enquanto uma thread estiver ativa. A classe `HighThread` ([Figura 5.17](#)) ilustra o método `setPriority()` ao aumentar a prioridade da thread para 3.

```

public class HighThread implements Runnable
{
    public void run() {
        Thread.currentThread().setPriority(Thread.NORM_PRIORITY + 3);
        // restante do método run()
        . . .
    }
}

```

FIGURA 5.17 Definindo uma prioridade por meio de `setPriority()`.

Como a JVM normalmente é implementada em cima de um sistema operacional hospedeiro, a prioridade de uma thread Java está relacionada com a prioridade da thread do kernel ao qual é mapeada. Como é de se esperar, esse relacionamento varia de um sistema para outro. Assim, mudar a prioridade de uma thread Java por meio da chamada de sistema `setPriority()` pode ter efeitos diferentes, dependendo do sistema operacional hospedeiro.

Por exemplo, considere os sistemas Solaris. Conforme descrevemos na [Seção 5.6.1](#), as threads no Solaris recebem uma prioridade entre 0 e 59. Nesses sistemas, as dez prioridades atribuídas a threads Java precisam ser associadas de alguma forma às 60 prioridades possíveis das threads do kernel. A prioridade da thread do kernel à qual uma thread Java é mapeada baseia-se em uma combinação de prioridades em nível de Java e na tabela de despacho mostrada na [Figura 5.13](#).

Sistemas Win32 identificam sete níveis de prioridade. Como resultado, diferentes prioridades de thread Java podem ser mapeadas para a mesma prioridade de uma thread do kernel. Por exemplo, `Thread.NORM_PRIORITY +1` e `Thread.NORM_PRIORITY +2` são mapeadas para a mesma prioridade do kernel; alterar a prioridade das threads Java no sistema Win32 pode não ter efeito sobre o modo como tais threads são escalonadas. O relacionamento entre as prioridades das threads Java e Win32 aparece na [Figura 5.18](#).

Prioridade Java	Prioridade Win32
1 (<code>MIN_PRIORITY</code>)	LOWEST
2	LOWEST
3	BELOW_NORMAL
4	BELOW_NORMAL
5 (<code>NORM_PRIORITY</code>)	NORMAL
6	ABOVE_NORMAL
7	ABOVE_NORMAL
8	HIGHEST
9	HIGHEST
10 (<code>MAX_PRIORITY</code>)	TIME_CRITICAL

FIGURA 5.18 Relacionamento entre as prioridades das threads Java e Win32.

É importante observar que a especificação para a JVM não identifica como uma JVM precisa implementar prioridades de threads. Os projetistas de JVM podem implementar a priorização da forma como quiserem. Na verdade, uma implementação de JVM pode decidir ignorar totalmente as chamadas a `setPriority()`.

5.7.2 Escalonamento de threads Java no Solaris

Vejamos mais de perto o escalonamento de threads Java nos sistemas Solaris. O relacionamento entre as prioridades de uma thread Java e sua thread de kernel associada possui uma história interessante. No Solaris, cada thread Java recebe uma thread de usuário exclusiva. Além do mais, as threads de usuário são mapeadas para threads do kernel através de um processo leve, ou LWP, conforme ilustrado na [Figura 5.19](#). O Solaris utiliza as chamadas de sistema `priocntl()` e `thr_setprio()` para mudar as prioridades dos LWPs e das threads do usuário, respectivamente. O kernel está ciente das mudanças de prioridade feitas no LWP com `priocntl()`, mas as mudanças feitas na prioridade de uma thread do usuário com `thr_setprio()` não são refletidas no kernel.

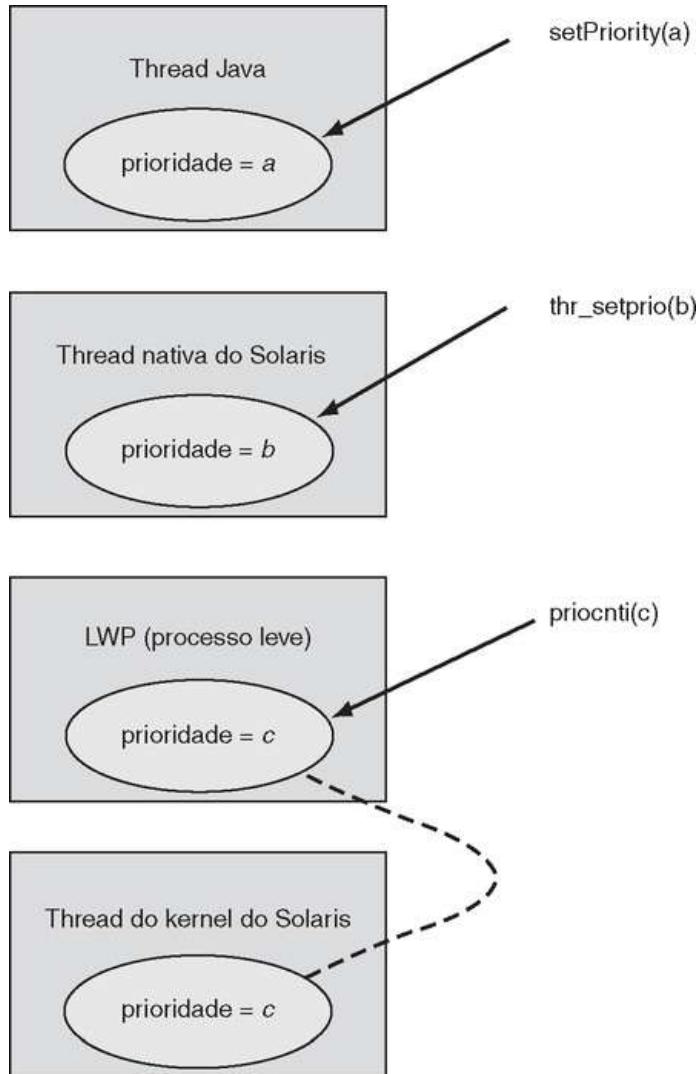


FIGURA 5.19 Threads do usuário são mapeadas para threads do Kernel por meio de um processo leve.

Antes de sua versão 9, o Solaris usava o modelo muitos para muitos para mapear threads do usuário e do kernel ([Seção 4.2.3](#)). A dificuldade com esse modelo foi que, embora fosse possível mudar a prioridade do LWP à qual uma thread Java era mapeada, esse mapeamento era dinâmico, e a thread Java poderia passar para outro LWP sem a ciência da JVM. Essa questão foi resolvida quando o Solaris 9 adotou o modelo um para um ([Seção 4.2.2](#)), garantindo assim que uma thread Java recebesse o mesmo LWP por toda a sua vida.

Quando o método `setPriority()` era chamado antes da versão 1.4.2 da JVM no Solaris, a JVM mudava apenas a prioridade da thread do usuário com a chamada de sistema `thr_setprio()`; a prioridade do LWP subjacente não era mudada, pois a JVM não invocava `priocntl()`. (O motivo para isso foi que o Solaris usava o modelo de threads muitos para muitos naquela época. Mudar a prioridade do LWP fazia pouco sentido, pois uma thread Java poderia migrar entre diferentes LWPs sem o conhecimento do kernel.) Contudo, quando apareceu a versão 1.4.2 da JVM, o Solaris já tinha adotado o modelo um para um. Como resultado, quando uma thread Java invocava `setPriority()`, a JVM chamava tanto `thr_setprio()` quanto `priocntl()` para alterar a prioridade da thread do usuário e do LWP, respectivamente.

A versão 1.5 da JVM no Solaris resolveu questões referentes às prioridades relativas das threads Java e threads executadas nos programas C e C++ nativos. No Solaris, como padrão, um programa C ou C++ inicialmente é executado na prioridade mais alta em sua classe de escalonamento. Contudo, a prioridade-padrão dada a uma thread Java em `Thread.NORM_PRIORITY` está no meio do intervalo de prioridades para sua classe de escalonamento. Como resultado, quando o Solaris executasse simultaneamente um programa C e Java com prioridades de escalonamento-padrão, o sistema operacional normalmente favoreceria o programa C. A versão 1.5 da JVM no Solaris atribui prioridades Java de `Thread.NORM_PRIORITY` até `Thread.MAX_PRIORITY`, a mais alta prioridade na classe de escalonamento. As prioridades Java `Thread.MIN_PRIORITY` até `Thread.NORM_PRIORITY -1` recebem prioridades correspondentemente mais baixas. A vantagem desse esquema é que os programas Java agora são executados com prioridades iguais às dos programas C e C++. A desvantagem é que alterar a prioridade de uma thread Java de `setPriority(Thread.NORM_PRIORITY)` para

`setPriority(Thread.MAX_PRIORITY)` não tem efeito.

5.7.3 Recursos de escalonamento em Java 1.5

Com essa política de escalonamento definida de modo tão solto, os desenvolvedores Java normalmente têm sido desencorajados de usar recursos das API relacionados com o escalonamento, pois o comportamento das chamadas de método pode variar muito de um sistema operacional para outro. No entanto, a partir da Java 1.5, foram acrescentados recursos à API Java para dar suporte a uma política de escalonamento mais determinística. Os acréscimos à API giram em torno do uso de um banco de threads.

Na [Seção 4.5.4](#), explicamos sobre os bancos de threads usando Java. Naquela seção, discutimos três estruturas diferentes de banco de threads: (1) um banco de única thread; (2) um banco de threads com um número fixo de threads; e (3) um banco de threads em cache. Uma quarta estrutura, que executa threads depois de um atraso ou periodicamente, também está disponível. Essa estrutura é semelhante àquela mostrada na [Figura 4.17](#). No entanto, ela é diferente por usar o método de fábrica `newScheduledThreadPool()` na classe `Executors`, que retorna um objeto `ScheduledExecutorService`. Usando esse objeto, invocamos um de quatro métodos possíveis para escalonar uma tarefa `Runnable` ou depois de um atraso fixo, em uma taxa fixa (periódica), ou periodicamente com um atraso inicial. O programa mostrado na [Figura 5.20](#) cria uma tarefa `Runnable` que verifica uma vez por segundo para ver se existem entradas gravadas em um log. Se houver, as entradas são gravadas em um banco de dados. O programa utiliza um banco de threads escalonado com tamanho 1 (só precisamos de uma thread) e depois escalona a tarefa usando o método `scheduleAtFixedRate()` de modo que a tarefa começa a executar imediatamente (atraso 0) e depois é executada uma vez por segundo.

```
import java.util.concurrent.*;

class Task implements Runnable
{
    public void run() {
        /**
         * verifica se existe alguma entrada que
         * precisa ser gravada de um log para um
         * banco de dados.
         */
        System.out.println("Verificando entradas de log ...");
    }
}

public class SPExample
{
    public static void main(String[] args) {
        // cria o grupo de threads escalonadas
        ScheduledExecutorService scheduler =
            Executors.newScheduledThreadPool(1);

        // cria a tarefa
        Runnable task = new Task();

        // escalona a tarefa de modo que seja executada uma vez por segundo
        scheduler.scheduleAtFixedRate(task,0,1,TimeUnit.SECONDS);
    }
}
```

FIGURA 5.20 Criação de um banco de threads escalonado em Java.

5.8 Avaliação de algoritmo

Como selecionamos um algoritmo de escalonamento de CPU para determinado sistema? Como vimos na [Seção 5.3](#), existem muitos algoritmos de escalonamento, cada um com seus próprios parâmetros. Como resultado, a seleção de um algoritmo pode ser difícil.

O primeiro problema é definir os critérios a serem usados na seleção de um algoritmo. Como vimos na [Seção 5.2](#), os critérios normalmente são definidos em termos de utilização de CPU, tempo de resposta ou throughput. Para selecionar um algoritmo, primeiro temos de definir a importância relativa dessas medições. Nossos critérios podem incluir várias medições, como:

- Maximizar a utilização de CPU sob a restrição de o tempo de resposta máximo ser de 1 segundo.
- Maximizar o throughput de modo que o turnaround seja (na média) linearmente proporcional ao tempo de execução total.

Quando os critérios de seleção tiverem sido definidos, desejaremos avaliar os diversos algoritmos em consideração. Descrevemos os diversos métodos de avaliação que podemos usar.

5.8.1 Modelagem determinística

Uma classe importante dos métodos de avaliação é a **avaliação analítica**. A avaliação analítica usa o algoritmo indicado e a carga de trabalho do sistema para produzir uma fórmula ou número que avalia o desempenho do algoritmo para essa carga de trabalho.

Um tipo de avaliação analítica é a **modelagem determinística**. Esse método apanha uma carga de trabalho específica predeterminada e define o desempenho de cada algoritmo para essa carga de trabalho. Por exemplo, suponha que tenhamos a carga de trabalho mostrada a seguir. Todos os cinco processos chegam no momento 0, na ordem indicada, com o tempo de burst de CPU indicado em milissegundos:

Processo	Tempo de burst
P_1	10
P_2	29
P_3	3
P_4	7
P_5	12

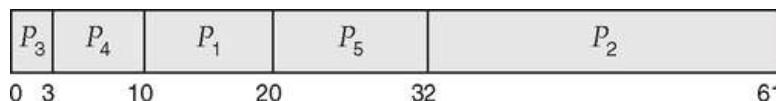
Considere os algoritmos de escalonamento FCFS, SJF e RR (quantum = 10 milissegundos) para esse conjunto de processos. Qual algoritmo daria o menor tempo de espera médio?

Para o algoritmo FCFS, executaríamos os processos como



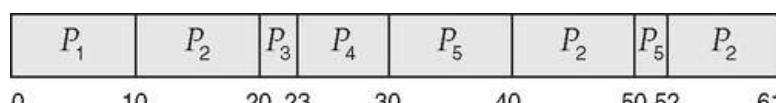
O tempo de espera é de 0 milissegundo para o processo P_1 , 10 milissegundos para o processo P_2 , 39 milissegundos para o processo P_3 , 42 milissegundos para o processo P_4 e 49 milissegundos para o processo P_5 . Assim, o tempo de espera médio é $(0 + 10 + 39 + 42 + 49)/5 = 28$ milissegundos.

Com o escalonamento SJF não preemptivo, executamos os processos como



O tempo de espera é de 10 milissegundos para o processo P_1 , 32 milissegundos para o processo P_2 , 0 milissegundo para o processo P_3 , 3 milissegundos para o processo P_4 e 20 milissegundos para o processo P_5 . Assim, o tempo de espera médio é $(10 + 32 + 0 + 3 + 20)/5 = 13$ milissegundos.

Com o algoritmo RR, executamos os processos como



O tempo de espera é de 0 milissegundo para o processo P_1 , 32 milissegundos para o processo P_2 , 20 milissegundos para o processo P_3 , 23 milissegundos para o processo P_4 e 40 milissegundos para o processo P_5 . Assim, o tempo de espera médio é $(0 + 32 + 20 + 23 + 40)/5 = 23$ milissegundos.

Podemos ver que, nesse caso, o tempo de espera médio obtido com a política SJF resulta em menos de metade do escalonamento FCFS; o algoritmo RR nos dá um valor intermediário.

A modelagem determinística é simples e rápida. Ela nos dá números exatos, permitindo que comparemos os algoritmos. Todavia, ela requer números exatos para entrada, e suas respostas se aplicam apenas a esses casos. Os principais usos da modelagem determinística são para descrever algoritmos de escalonamento e fornecer exemplos. Nos casos em que podemos executar o mesmo programa várias vezes e podemos medir seus requisitos de processamento do programa com exatidão, podemos usar a modelagem determinística para selecionar um algoritmo de escalonamento. Além do mais, por um conjunto de exemplos, a modelagem determinística pode indicar tendências que sejam, então, analisadas e provadas separadamente. Por exemplo, pode ser mostrado que, para o ambiente descrito (todos os processos e seus tempos disponíveis no momento 0), a política SJF sempre resultará no menor tempo de espera.

5.8.2 Modelos de enfileiramento

Em muitos sistemas, os processos executados variam de um dia para o outro, de modo que não existe um conjunto estático de processos (ou tempos) para uso na modelagem determinística. Entretanto, o que pode ser determinado é a distribuição dos bursts de CPU e E/S. Essas distribuições podem ser medidas e depois aproximadas ou estimadas. O resultado é uma fórmula matemática, descrevendo a probabilidade de determinado burst de CPU. Em geral, essa distribuição é exponencial e descrita por sua média. De modo semelhante, a distribuição de tempos quando os processos chegam no sistema (a distribuição do tempo de chegada) precisa ser dada. A partir dessas duas distribuições, é possível calcular, para a maioria dos algoritmos, a média da throughput, da utilização, do tempo de espera e assim por diante.

O sistema computadorizado é descrito como uma rede de servidores. Cada servidor possui uma fila de processos em espera. A CPU é um servidor com sua fila de prontos, assim como o sistema de E/S com suas filas de dispositivo. Conhecendo as taxas de chegada e as taxas de serviço, podemos calcular a utilização, o tamanho médio da fila, o tempo médio de espera e assim por diante. Essa área de estudo é chamada **análise da rede de enfileiramento**.

Como exemplo, seja n o tamanho médio da fila (excluindo o processo sendo atendido), W o tempo médio de espera na fila, e λ a taxa média de chegada de novos processos na fila (por exemplo, três processos por segundo). Em seguida, esperamos que, durante o tempo W que um processo espera, $\lambda \times W$ novos processos cheguem na fila. Se o sistema estiver em um estado uniforme, então o número de processos saindo da fila precisa ser igual ao número de processos que chegam. Assim,

$$n = \lambda \times W.$$

Essa equação, conhecida como **fórmula de Little**, é particularmente útil, pois é válida para qualquer algoritmo de escalonamento e distribuição de chegada.

Podemos usar a fórmula de Little para calcular uma de três variáveis, se soubermos as outras duas. Por exemplo, se soubermos que 7 processos chegam a cada segundo (na média) e que normalmente existem 14 processos na fila, então podemos calcular o tempo de espera médio por processo como 2 segundos.

A análise de enfileiramento pode ser útil na comparação de algoritmos de escalonamento, mas também possui limitações. No momento, as classes de algoritmos e distribuições que podem ser tratadas são bastante limitadas. Pode ser difícil lidar com a matemática complicada dos algoritmos e distribuições. Assim, as distribuições de chegada e atendimento normalmente são definidas de maneiras matematicamente tratáveis – porém não realistas. Também é necessário fazer uma série de suposições independentes, que podem não ser precisas. Como resultado dessas dificuldades, os modelos de enfileiramento são apenas aproximações dos sistemas reais, e a precisão dos resultados calculados pode ser questionável.

5.8.3 Simulações

Para obter uma avaliação mais precisa dos algoritmos de escalonamento, podemos usar **simulações**. O uso de simulações envolve a programação de um modelo do sistema computadorizado. As estruturas de dados do software representam os principais componentes do sistema. O simulador possui uma variável representando um relógio; à medida que o valor dessa variável é aumentado, o simulador modifica o estado do sistema para refletir as atividades dos

dispositivos, dos processos e do escalonador. Enquanto a simulação é executada, as estatísticas que indicam o desempenho do algoritmo são colhidas e impressas.

Os dados para controlar a simulação podem ser gerados de diversas maneiras. O método mais comum utiliza um gerador de números aleatórios, programado para gerar processos, tempos de burst de CPU, chegadas, saídas, e assim por diante, de acordo com as distribuições de probabilidade. As distribuições podem ser definidas matematicamente (uniforme, exponencial, Poisson) ou empiricamente. Se a distribuição tiver de ser definida empiricamente, as medições do sistema real em estudo são tomadas. Os resultados definem a distribuição dos eventos no sistema real; essa distribuição pode, então, ser usada para controlar a simulação.

Contudo, uma simulação controlada por distribuição pode ser pouco precisa, devido aos relacionamentos entre os sucessivos eventos no sistema real. A distribuição de frequência indica apenas quantos ocorrem em cada evento; ela não indica nada sobre a ordem de sua ocorrência. Para corrigir esse problema, podemos usar **fitas de rastreamento (trace tapes)**. Criamos uma fita de rastreamento monitorando o sistema real e registrando a sequência de eventos reais (Figura 5.21). Depois, usamos essa sequência para controlar a simulação. As fitas de rastreamento fornecem uma excelente maneira de comparar dois algoritmos com o mesmo conjunto de entradas reais. Esse método pode produzir resultados precisos para suas entradas.

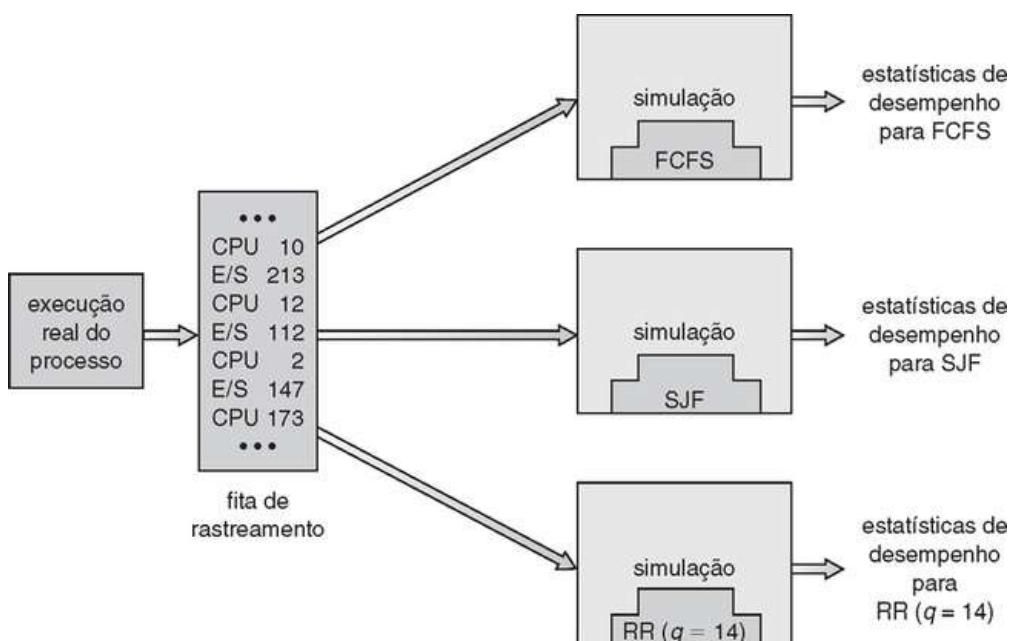


FIGURA 5.21 Avaliação dos escalonadores de CPU pela simulação.

As simulações podem ser caras, exigindo horas de tempo do computador. Uma simulação mais detalhada pode fornecer resultados mais precisos, mas também exige mais tempo do computador. Além disso, as fitas de rastreamento podem exigir grande quantidade de espaço para armazenamento. Finalmente, o projeto, a codificação e a depuração do simulador podem ser uma tarefa importante.

5.8.4 Implementação

Até mesmo uma simulação possui precisão limitada. A única maneira precisa para avaliar um algoritmo de escalonamento é codificá-lo, colocá-lo no sistema operacional e ver como funciona. Essa técnica coloca o algoritmo real no sistema real, para ser avaliado sob condições de operação reais.

A principal dificuldade dessa técnica é o alto custo. O custo aparece não apenas na codificação do algoritmo e na modificação do sistema operacional para dar suporte a ele (e suas estruturas de dados exigidas), mas também na reação dos usuários a um sistema operacional alterado constantemente. A maioria dos usuários não está interessada na montagem de um sistema operacional melhor; eles querem que seus processos sejam executados e usar seus resultados. Um sistema operacional em constante mudança não ajuda os usuários a realizarem seu trabalho.

A outra dificuldade é que o ambiente em que o algoritmo é utilizado mudará. O ambiente mudará não apenas no modo normal, à medida que novos programas são escritos e os tipos de problemas mudam, mas também como resultado do desempenho do escalonador. Se processos curtos recebem prioridade, então os usuários podem dividir processos maiores em conjuntos de processos menores. Se os processos interativos recebem prioridade em relação a processos não interativos, então os

usuários podem passar para o uso interativo.

Por exemplo, os pesquisadores projetaram um sistema que classificava os processos interativos e não interativos automaticamente, examinando a quantidade de E/S do terminal. Se um processo não recebia ou enviava algo para o terminal em um intervalo de 1 segundo, o processo era considerado não interativo, sendo movido para uma fila de menor prioridade. Em resposta a essa política, um programador modificava seus programas para escrever um caractere qualquer no terminal em intervalos regulares de menos de 1 segundo. O sistema dava aos seus programas uma prioridade alta, embora a saída no terminal não tivesse significado algum.

Os algoritmos de escalonamento mais flexíveis são aqueles que podem ser alterados pelos gerentes do sistema ou pelos usuários, para serem ajustados a uma aplicação específica ou conjunto de aplicações. Uma estação de trabalho que realiza aplicações gráficas de última geração, por exemplo, pode ter necessidades de escalonamento diferentes daquelas de um servidor Web ou servidor de arquivos. Alguns sistemas operacionais - em especial várias versões do UNIX - permitem ao gerente do sistema ajustar os parâmetros de escalonamento para determinada configuração do sistema. Por exemplo, o Solaris fornece o comando `dispadmin` para permitir que o administrador do sistema modifique os parâmetros das classes de escalonamento descritas na [Seção 5.6.1](#).

Outra técnica é usar APIs que modificam a prioridade de um processo ou thread. API Java, POSIX e Win32 fornecem essas funções. A desvantagem dessa técnica é que o ajuste do desempenho de um sistema ou aplicação não resulta em melhor desempenho em situações mais genéricas.

5.9 Resumo

O escalonamento de CPU é a tarefa de selecionar um processo em espera na fila de prontos e alocar a CPU para ele. A CPU é alocada ao processo selecionado pelo despachante.

O escalonamento FCFS (first come, first served) é o algoritmo de escalonamento mais simples, mas pode fazer os processos curtos esperarem pelos processos muito longos. O escalonamento SJF (Shortest Job First) provavelmente é o ideal, fornecendo a menor média de tempo de espera. A implementação do escalonamento SJF é difícil porque também é difícil prever a duração do próximo burst de CPU. O algoritmo SJF é um caso especial do algoritmo geral de escalonamento por prioridade, que aloca a CPU ao processo de mais alta prioridade. Tanto o escalonamento por prioridade quanto o SJF podem sofrer de starvation. O envelhecimento é uma técnica para impedir a starvation.

O escalonamento Round-Robin (RR) é mais apropriado para um sistema de tempo compartilhado (interativo). O escalonamento RR aloca a CPU ao primeiro processo na fila de prontos para q unidades de tempo, onde q é o quantum de tempo. Depois de q unidades de tempo, se o processo não tiver abandonado a CPU ele é preemptado, e o processo é colocado no final da fila de prontos (ready queue). O problema principal é a seleção do quantum de tempo. Se o quantum for muito grande, o escalonamento RR degenera para o escalonamento FCFS; se for muito pequeno, o custo adicional do escalonamento, na forma de tempo para troca de contexto, se torna excessivo.

O algoritmo FCFS é não preemptivo; o algoritmo RR é preemptivo. Os algoritmos SJF e de prioridade podem ser preemptivos ou não.

Os algoritmos multilevel queue permitem que diferentes algoritmos sejam utilizados para diversas classes de processos. O modo mais comum inclui uma fila interativa de primeiro plano, que utiliza o escalonamento RR, e a fila batch no segundo plano, que utiliza o escalonamento FCFS. A multilevel feedback queue permite aos processos mudarem de uma fila para outra.

Muitos sistemas computadorizados contemporâneos admitem vários processadores; cada processador realiza seu escalonamento de forma independente. Em geral, cada processador mantém sua própria fila privativa de processos (ou threads), todos disponíveis para execução. Questões relacionadas com o escalonamento de multiprocessadores incluem afinidade de processador, balanceamento de carga e processamento multicore, além do escalonamento em sistemas de virtualização.

Os sistemas operacionais que admitem threads no nível do kernel precisam escalonar threads - e não processos - para serem executados. Isso acontece com o Solaris e o Windows XP. Os dois sistemas escalonam threads usando algoritmos de escalonamento preemptivos, baseados em prioridade, incluindo o suporte para threads de tempo real. O escalonador de processos do Linux também usa um algoritmo baseado em prioridade com suporte de tempo real. Os algoritmos de escalonamento para esses três sistemas operacionais normalmente favorecem os processos interativos em detrimento dos processos batch e CPU-bound. A especificação para escalonamento em Java é definida de forma solta: threads com prioridade mais alta terão preferência em relação a threads com prioridade mais baixa.

A grande variedade de algoritmos de escalonamento exige que tenhamos métodos para selecionar entre os algoritmos. Os métodos analíticos utilizam a análise matemática para determinar o desempenho de um algoritmo. Os métodos por simulação determinam o desempenho imitando o algoritmo de escalonamento em uma amostra "representativa" de processos e calculando o desempenho resultante. Contudo, a simulação no máximo pode fornecer uma aproximação do desempenho real do sistema; a única técnica confiável para avaliar um algoritmo de escalonamento é implementar o algoritmo em um sistema real e monitorar seu desempenho em um ambiente no "mundo real".

Exercícios práticos

- 5.1. Um algoritmo de escalonamento de CPU determina uma ordem para a execução de seus processos escalonados. Com n processos a serem escalonados em um processador, quantos escalonamentos diferentes são possíveis? Mostre uma fórmula em termos de n .
- 5.2. Explique a diferença entre escalonamento preemptivo e não preemptivo.
- 5.3. Suponha que os processos a seguir cheguem para execução nos três tempos indicados. Cada processo será executado por um período listado. Ao responder as perguntas, use o escalonamento não preemptivo e tome todas as decisões baseado nas informações que possui no momento em que a decisão deve ser feita.

Processo	Tempo de chegada	Tempo de burst
P_1	0,0	8
P_2	0,4	4
P_3	1,0	1

- a. Qual é o tempo de turnaround médio para estes processos com o algoritmo de escalonamento FCFS?
- b. Qual é o tempo de turnaround médio para estes processos com o algoritmo de escalonamento SJF?
- c. O algoritmo SJF supostamente melhora o desempenho, mas observe que escolhemos executar o processo P_1 no tempo 0 porque não sabíamos que dois processos mais curtos chegariam logo. Calcule qual será o tempo de turnaround médio se a CPU ficar ociosa para a primeira unidade e depois o escalonamento SJF for usado. Lembre-se que os processos P_1 e P_2 estão esperando durante esse tempo ocioso, de modo que seu tempo de espera poderá aumentar. Este algoritmo poderia ser conhecido como escalonamento por conhecimento do futuro.
- 5.4. Qual é a vantagem de haver diferentes tamanhos de quantum de tempo em diferentes níveis de um sistema de enfileiramento multilevel queue?
- 5.5. Muitos algoritmos de escalonamento de CPU são parametrizados. Por exemplo, o algoritmo RR requer um parâmetro para indicar a fatia de tempo. Multilevel feedback queues exigem parâmetros para definir o número de filas, o algoritmo de escalonamento para cada fila, os critérios usados para mover processos entre as filas, e assim por diante. Esses algoritmos, portanto, são, na realidade, conjuntos de algoritmos (por exemplo, o conjunto de algoritmos RR para todas as fatias de tempo, e assim por diante). Um conjunto de algoritmos pode incluir outro (por exemplo, o algoritmo FCFS é o algoritmo RR com um quantum de tempo infinito). Que relação (se houver alguma) existe entre os seguintes pares de conjuntos de algoritmos?
- a. Prioridade e SJF
 - b. Multilevel feedback queues a e FCFS
 - c. Prioridade e FCFS
 - d. RR e SJF
- 5.6. Suponha que um algoritmo de escalonamento (no nível do escalonamento de CPU em curto prazo) favoreça aqueles processos que usaram menos tempo de processador no passado recente. Por que esse algoritmo favorecerá os programas voltados para E/S e ainda assim não causará starvation permanente nos programas voltados para CPU?
- 5.7. Faça a distinção entre escalonamento PCS e SCS.
- 5.8. Suponha que um sistema operacional faça um mapeamento entre as threads em nível de usuário e de kernel usando o modelo muitos para muitos e que esse mapeamento seja feito por meio de LWPs. Além do mais, o sistema permite que os desenvolvedores de programa criem threads em tempo real. É necessário vincular uma thread em tempo real a um LWP?

Exercícios

- 5.9. Por que é importante o escalonador distinguir programas voltados para E/S dos programas voltados para CPU?
- 5.10. Discuta como os seguintes pares de critérios de escalonamento entram em conflito em certos ambientes.
- Utilização de CPU e tempo de resposta
 - Tempo de turnaround médio e tempo de espera máximo
 - Utilização de dispositivo de E/S e utilização de CPU
- 5.11. Considere a fórmula de média exponencial usada para prever o tamanho do próximo burst de CPU ([Seção 5.3.2](#)). Quais são as implicações de atribuir os valores a seguir aos parâmetros usados pelo algoritmo?
- $a = 0$ e $\tau_0 = 100$ milissegundos
 - $a = 0,99$ e $\tau_0 = 10$ milissegundos
- 5.12. Considere o seguinte conjunto de processos, com o tamanho do tempo de burst de CPU dado em milissegundos:

Processo	Tempo de chegada	Tempo de burst
P_1	10	3
P_2	1	1
P_3	2	3
P_4	1	4
P_5	5	2

Considere que os processos chegaram na ordem P_1, P_2, P_3, P_4, P_5 , todos no momento 0.

- Desenhe quatro gráficos de Gantt que ilustrem a execução desses processos usando os seguintes algoritmos de escalonamento: FCFS, SJF, prioridade não preemptiva (um número de prioridade menor significa uma prioridade mais alta) e o RR (quantum = 1).
 - Qual é o turnaround time de cada processo para cada um dos algoritmos de escalonamento do item a?
 - Qual é o tempo de espera de cada processo para cada um dos algoritmos de escalonamento?
 - Qual dos algoritmos resulta no menor tempo de espera médio (em relação a todos os processos)?
- 5.13. Qual dos seguintes algoritmos de escalonamento poderia resultar em starvation?
- Primeiro a chegar, primeiro a ser atendido (FCFS)
 - Menor tarefa primeiro (SJF)
 - Revezamento (round-robin)
 - Prioridade
- 5.14. Considere uma variante do algoritmo de escalonamento RR, na qual as entradas na fila de prontos são ponteiros para os PCBs.
- Qual seria o efeito de colocar dois ponteiros para o mesmo processo na fila de prontos?
 - Quais seriam duas vantagens importantes e duas desvantagens desse esquema?
 - Como você modificaria o algoritmo RR básico para conseguir o mesmo efeito sem os ponteiros duplicados?
- 5.15. Considere um sistema executando dez tarefas voltadas para E/S e uma tarefa voltada para CPU. Considere que as tarefas voltadas para E/S emitem uma operação de E/S uma vez a cada milissegundo de computação da CPU e que cada operação de E/S leve 10 milissegundos para terminar. Considere também que o overhead de troca de contexto seja de 0,1 milissegundo e que todos os processos sejam tarefas de longa duração. Qual é a utilização de CPU para um escalonador round-robin quando:
- A quota de tempo é de 1 milissegundo?
 - A quota de tempo é de 10 milissegundos?
- 5.16. Considere um sistema implementando escalonamento por fila multinível. Que estratégia um usuário de computador pode empregar para maximizar a quantidade de tempo de CPU alocada ao processo do usuário?
- 5.17. Considere o algoritmo de escalonamento por prioridade preemptiva, baseado em prioridades alteradas dinamicamente. Números de prioridade maiores implicam prioridade mais alta. Quando um processo está esperando pela CPU (na fila de prontos, mas não em execução), sua prioridade muda a uma taxa α ; quando está executando, sua prioridade muda a uma taxa β . Todos os processos recebem uma prioridade 0 quando entram na fila de prontos. Os parâmetros α e β podem ser definidos para dar muitos algoritmos de escalonamento diferentes.
- Qual é o algoritmo que resulta de $\beta > \alpha > 0$?
 - Qual é o algoritmo que resulta de $\alpha < \beta < 0$?
- 5.18. Explique as diferenças no grau em que os algoritmos de escalonamento a seguir são discriminados em favor de processos curtos:
- FCFS
 - RR

- c. Multilevel feedback queues
- 5.19. Usando o algoritmo de escalonamento do Windows XP, qual é a prioridade numérica de uma thread para os cenários a seguir?
- Uma thread na REALTIME_PRIORITY_CLASS com uma prioridade relativa HIGHEST.
 - Uma thread na NORMAL_PRIORITY_CLASS com uma prioridade relativa NORMAL.
 - Uma thread na HIGH_PRIORITY_CLASS com uma prioridade relativa ABOVE_NORMAL.
- 5.20. Considere o algoritmo de escalonamento no sistema operacional Solaris para as threads de tempo compartilhado:
- Qual é o quantum de tempo (em milissegundos) para uma thread com prioridade 10? E com prioridade 55?
 - Considere que uma thread com prioridade 35 tenha usado seu quantum de tempo inteiro sem bloqueio. Que nova prioridade o escalonador atribuirá a essa thread?
 - Considere que uma thread com prioridade 35 é bloqueada para E/S antes de seu quantum de tempo ter se esgotado. Que nova prioridade o escalonador atribuirá a essa thread?
- 5.21. O escalonador UNIX tradicional impõe um relacionamento reverso entre números de prioridade e prioridades: quanto mais alto o número, menor a prioridade. O escalonador recalcula as prioridades de processo uma vez por segundo usando a seguinte função:

$$\text{Prioridade} = (\text{uso de CPU recente}/2) + \text{base}$$
 onde base = 60 e *uso de CPU recente* refere-se a um valor indicando com que frequência um processo usou a CPU desde que as prioridades foram recalculadas pela última vez.
 Considere que o uso de CPU recente para o processo P_1 seja 40, para o processo P_2 seja 18 e para o processo P_3 seja 10. Quais serão as novas prioridades para esses três processos quando as prioridades forem recalculadas? Com base nessa informação, o escalonador tradicional do UNIX aumenta ou diminui a prioridade relativa de um processo voltado para CPU?
- 5.22. Conforme discutimos na [Seção 5.7](#), a especificação para a JVM pode permitir que as implementações ignorem chamadas para `setPriority()`. Um argumento em favor de ignorar `setPriority()` é que aumentar ou diminuir a prioridade de uma thread tem pouco efeito depois que a thread começa a executar em uma thread nativa do sistema operacional, pois o escalonador do sistema operacional modifica a prioridade da thread de kernel à qual a thread Java é mapeada com base na necessidade de uso de CPU ou E/S. Discuta os prós e os contras desse argumento.

Problemas de programação

5.23. No Problema de Programação 4.21, você escreveu um programa que listava da thread na JVM. Modifique esse programa de modo a listar também a prioridade de cada thread.

Notas bibliográficas

As filas com feedback foram implementadas originalmente no sistema CTSS descrito em [Corbato e outros \[1962\]](#). Esse sistema de enfileiramento com feedback foi analisado por [Schrage \[1967\]](#). O algoritmo de escalonamento preemptivo por prioridade do [Exercício 5.17](#) foi sugerido por [Kleinrock \[1975\]](#).

[Anderson e outros \[1989\]](#), [Lewis e Berg \[1998\]](#) e [Philbin e outros \[1966\]](#) abordaram o escalonamento de thread. O escalonamento multicore é discutido por [McNairy e Bhatia \[2005\]](#) e [Kongetira e oturos \[2005\]](#).

Técnicas de escalonamento que levam em consideração informações referentes a tempos de execução de processo das execuções anteriores foram descritas em [Fisher \[1981\]](#), [Hall e outros \[1996\]](#) e [Lowney e outros \[1993\]](#).

Os escalonadores justos são abordados por [Henry \[1984\]](#), [Woodside \[1986\]](#) e [Kay e Lauder \[1988\]](#).

As políticas de escalonamento usadas no sistema operacional UNIX V são descritas por [Bach \[1987\]](#); aquelas para o UNIX FreeBSD 5.2 são apresentadas por [McKusick e Neville-Neil \[2005\]](#); e aquelas para o sistema operacional Mach, por [Black \[1990\]](#). [Love \[2005\]](#) aborda o escalonamento no Linux. Os detalhes do escalonador ULE podem ser encontrados em [Roberson \[2003\]](#). O escalonamento no Solaris é descrito por [Mauro e McDougall \[2007\]](#). [Solomon \[1998\]](#), [Solomon e Russinovich \[2000\]](#) e [Russinovich e Solomon \[2005\]](#) discutem o escalonamento interno no Windows. [Butenhof \[1997\]](#) e [Lewis e Berg \[1998\]](#) descrevem o escalonamento em sistemas Pthreads. [Siddha e outros \[2007\]](#) discutem os desafios de escalonamento em sistemas multicore. O escalonamento de threads em Java é abordado por [Oaks e Wong \[2004\]](#) e [Goetz e outros \[2006\]](#).

CAPÍTULO 6

Sincronismo de processos

Um processo cooperativo é aquele que pode afetar ou ser afetado por outros processos em execução no sistema. Os processos cooperativos podem compartilhar diretamente um espaço de endereços lógico (ou seja, código e dados) ou ter permissão para compartilhar dados apenas por meio de arquivos ou mensagens. O primeiro caso é obtido com o uso de threads, discutido no [Capítulo 4](#). No entanto, o acesso concorrente aos dados compartilhados pode resultar em incoerência de dados. Neste capítulo, vamos discutir sobre os diversos mecanismos para garantir a execução ordenada de processos cooperativos que compartilham um espaço de endereços lógico, permitindo, assim, a manutenção da consistência dos dados.

OBJETIVOS DO CAPÍTULO

- Introduzir o problema de seção crítica, cujas soluções podem ser usadas para garantir a consistência dos dados compartilhados.
- Apresentar diversas soluções de software para o problema da seção crítica.
- Apresentar diversas soluções de hardware para o problema da seção crítica.
- Introduzir o conceito de transação atômica e descrever mecanismos para garantir a atomicidade.

6.1 Fundamentos

No [Capítulo 3](#), desenvolvemos um modelo de um sistema consistindo em processos ou threads sequenciais cooperativas, todas executando de forma assíncrona e possivelmente compartilhando dados. Ilustramos esse modelo com um problema de produtor-consumidor, que representa os sistemas operacionais. Especificamente, na [Seção 3.4.1](#), descrevemos como um buffer limitado poderia ser usado para permitir que os processos compartilhem a memória.

Vamos retornar à nossa consideração do buffer limitado. Aqui, consideraremos que o código para o produtor é o seguinte:

```
while (count == BUFFER_SIZE)
    ; // não faz nada

// acrescenta um item ao buffer
buffer[in] = item;
in = (in + 1) % BUFFER_SIZE;
++count;
```

O código para o consumidor é

```
while (count == 0)
    ; // não faz nada

// remove um item do buffer
item = buffer[out];
out = (out + 1) % BUFFER_SIZE;
--count;
```

Embora as rotinas do produtor e do consumidor estejam corretas separadamente, elas não funcionam de forma correta quando executadas ao mesmo tempo. Como ilustração, suponha que o valor da variável `count` atualmente seja 5 e que as threads produtor e consumidor executem as instruções “`++count`” e “`-count`” de maneira concorrente. Após a execução dessas duas instruções, o valor da variável `count` poderá ser 4, 5 ou 6! O único resultado correto é `count == 5`, que só é gerado se o produtor e o consumidor forem executados em sequência.

Podemos mostrar que o valor de `count` pode estar incorreto da seguinte maneira. Observe que a instrução “`count++`” pode ser implementada em linguagem de máquina (em uma máquina típica) como

```
registrador1 = count
registrador1 = registrador1 + 1
count = registrador1
```

onde `registrador1` é um registrador da CPU local. Da mesma forma, a instrução “`-count`” é implementada da seguinte maneira:

```
registrador2 = count
registrador2 = registrador2 - 1
count = registrador2
```

onde novamente `registrador2` é um registrador da CPU local. Embora `registrador1` e `registrador2` possam ser o mesmo registrador físico (um acumulador, digamos), lembre que o seu conteúdo será

salvo e restaurado pelo tratador de interrupção ([Seção 1.2.3](#)).

A execução concorrente das instruções “`++count`” e “`-count`” equivale a uma execução sequencial na qual as instruções de nível mais baixo apresentadas anteriormente são intercaladas em alguma ordem arbitrária (mas a ordem dentro de cada instrução de alto nível é preservada). Uma intercalação possível é

$T_0: \text{produtor}$	executa	$\text{registrador}_1 = \text{count}$	$\{\text{registrador}_1 = 5\}$
$T_1: \text{produtor}$	executa	$\text{registrador}_1 = \text{registrador}_1 + 1$	$\{\text{registrador}_1 = 6\}$
$T_2: \text{consumidor}$	executa	$\text{registrador}_2 = \text{count}$	$\{\text{registrador}_2 = 5\}$
$T_3: \text{consumidor}$	executa	$\text{registrador}_2 = \text{registrador}_2 - 1$	$\{\text{registrador}_2 = 4\}$
$T_4: \text{produtor}$	executa	$\text{count} = \text{registrador}_1$	$\{\text{count} = 6\}$
$T_5: \text{consumidor}$	executa	$\text{count} = \text{registrador}_2$	$\{\text{count} = 4\}$

Observe que chegamos ao estado incorreto “`count == 4`”, indicando que quatro buffers estão cheios, quando, na verdade, cinco buffers estão cheios. Se invertêssemos a ordem das instruções em T_4 e T_5 , chegariamós ao estado incorreto “`count == 6`”.

Chegaríamos a esse estado incorreto porque permitimos que os dois processos manipulassem a variável `count` de forma concorrente. Uma situação como essa – em que vários processos acessam e manipulam os mesmos dados concorrentemente e o resultado da execução depende da ordem específica em que ocorre o acesso – é chamada de **condição de corrida (race condition)**. Para nos protegermos contra uma condição de corrida, precisamos garantir que somente uma thread de cada vez poderá manipular a variável `count`. Para termos essa garantia, é preciso que os processos estejam sincronizados de alguma maneira.

Situações como a que acabamos de descrever ocorrem com frequência nos sistemas operacionais, pois diferentes partes do sistema manipulam recursos. Além do mais, com o crescimento dos sistemas multicore, existe uma ênfase cada vez maior no desenvolvimento de aplicações com multithreading, onde diversas threads – que possivelmente estão compartilhando dados – estão sendo executadas em paralelo em diferentes núcleos de processamento. Claramente, queremos que quaisquer mudanças resultantes dessas atividades não interfiram umas com as outras. Devido à importância dessa questão, uma parte importante deste capítulo trata do **sincronismo de processos e coordenação** entre os processos em cooperação.

6.2 O problema da seção crítica

Considere um sistema consistindo em n processos $\{P_0, P_1, \dots, P_{n-1}\}$. Cada processo possui um segmento de código, chamado **seção crítica (critical section)**, onde o processo pode alterar variáveis comuns, atualizando uma tabela, gravando um arquivo, e assim por diante. O recurso importante do sistema é que, quando um processo está executando em sua seção crítica, nenhum outro processo pode ter permissão para executar em sua seção crítica, isto é, dois processos não podem executar em suas seções críticas ao mesmo tempo. O *problema da seção crítica* é projetar um protocolo que os processos possam utilizar para a cooperação. Cada processo precisa solicitar permissão para entrar em sua seção crítica. A seção do código que implementa essa solicitação é a **seção de entrada**. A seção crítica pode ser seguida por uma **seção de saída**. O código restante é a **seção restante**. A estrutura geral de um processo típico P_i aparece na [Figura 6.1](#). A seção de entrada e a seção de saída são delimitadas em caixas para destacar esses segmentos de código importantes.

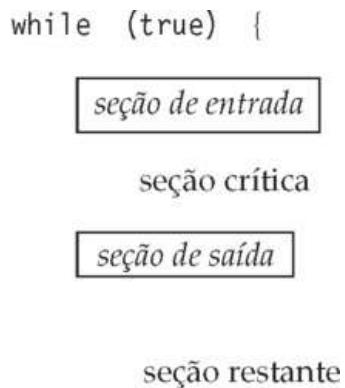


FIGURA 6.1 Estrutura geral de um processo P_i típico.

Uma solução para o problema da seção crítica precisa satisfazer os três requisitos a seguir:

1. **Exclusão mútua.** Se o processo P_i está executando em sua seção crítica, então nenhum outro poderá executar em suas seções críticas.
2. **Progresso.** Se nenhum processo estiver executando em sua seção crítica e alguns processos quiserem entrar em suas seções críticas, então somente os processos que não estão executando em suas seções restantes poderão participar da decisão sobre qual entrará em sua seção crítica em seguida, e essa seleção não pode ser adiada indefinidamente.
3. **Espera limitada.** Existe um limite no número de vezes que outros processos têm permissão para entrar em suas seções críticas após um processo ter feito uma requisição para entrar em sua seção crítica e antes de essa requisição ser atendida.

Consideramos que cada processo esteja sendo executado em uma velocidade diferente de zero. No entanto, não podemos fazer qualquer suposição a respeito da *velocidade relativa* dos n processos.

Em um determinado ponto no tempo, muitos processos do modo kernel podem estar ativos no sistema operacional. Como resultado, o código que implementa um sistema operacional (*código do kernel*) está sujeito a várias condições de corrida possíveis. Considere, como exemplo, uma estrutura de dados do kernel que mantém uma lista de todos os arquivos abertos no sistema. Essa lista precisa ser modificada quando um novo arquivo é aberto ou fechado (acrescentando o arquivo à lista ou removendo-o da lista). Se dois processos tivessem que abrir arquivos simultaneamente, as atualizações separadas nessa lista poderiam resultar em uma condição de corrida. Outras estruturas de dados do kernel que são possíveis de possíveis condições de corrida incluem estruturas para manter a alocação de memória, para manter listas de processos e para tratamento de interrupção. Fica a critério dos desenvolvedores de kernel garantir que o sistema operacional esteja livre de tais condições de corrida.

Duas técnicas gerais são utilizadas para lidar com seções críticas nos sistemas operacionais: (1) **kernels preemptivos**; e (2) **kernels não preemptivos**. Um kernel preemptivo permite que um processo seja interrompido enquanto estiver sendo executado no modo kernel. Um kernel não preemptivo não permite que um processo executando no modo kernel seja interrompido; um processo no modo kernel será executado até que saia do modo kernel, seja bloqueado ou passe voluntariamente o controle da CPU. Obviamente, um kernel não preemptivo é basicamente livre de condições de corrida nas estruturas de dados do kernel, pois somente um processo está ativo no kernel de cada vez. Não podemos dizer o mesmo sobre os kernels preemptivos, de modo que eles

precisam ser projetados cuidadosamente para garantir que os dados de kernel compartilhados sejam livres de condições e corrida. Os kernels preemptivos são especialmente difíceis de se projetar para arquiteturas SMP, pois nesses ambientes é possível que dois processos no modo kernel sejam executados simultaneamente em diferentes processadores.

Por que, então, alguém favoreceria um kernel preemptivo ao invés de um não preemptivo? Um kernel preemptivo é mais apropriado para programação em tempo real, pois permitirá que um processo em tempo real interrompa um processo atualmente em execução no kernel. Além do mais, um kernel preemptivo pode ser mais responsivo, pois há menos risco de que um processo no modo kernel seja executado por um período de tempo arbitrariamente longo antes de abrir mão do processador para os processos que estão aguardando. Naturalmente, esse efeito pode ser aliviado pelo projeto de um código de kernel que não se comporta dessa maneira. Mais adiante, neste capítulo, iremos explorar como os diversos sistemas operacionais gerenciam a preempção dentro do kernel.

6.3 Solução de Peterson

Em seguida, ilustramos uma solução clássica baseada em software para o problema da seção crítica conhecida como **solução de Peterson**. Devido ao modo como as arquiteturas de computador modernas realizam instruções básicas em linguagem de máquina, como `load` e `store`, não existem garantias de que a solução de Peterson funcionará corretamente em tais arquiteturas. Porém, apresentamos a solução porque ela fornece uma boa descrição algorítmica da solução do problema da seção crítica e ilustra algumas das complexidades envolvidas no projeto de software voltado para os requisitos de exclusão mútua, progresso e espera limitada.

A solução de Peterson é restrita a dois processos que alternam a execução entre suas seções críticas e as seções restantes. Os processos são numerados como P_0 e P_1 . Por conveniência, ao apresentar P_i , usamos P_j para indicar o outro processo, ou seja, j é igual a $1 - i$.

A solução de Peterson exige que dois processos compartilhem dois itens de dados:

```
int turn;
boolean flag[2];
```

A variável `turn` indica de quem é a vez de entrar em sua seção crítica. Ou seja, se `turn == i`, então o processo P_i tem permissão para executar em sua seção crítica. O array `flag` é usado para indicar se um processo *está pronto* para entrar em sua seção crítica. Por exemplo, se `flag[i]` for `true`, esse valor indica que P_i está pronto para entrar em sua seção crítica. Com uma explicação dessas estruturas de dados, agora estamos prontos para descrever o algoritmo mostrado na [Figura 6.2](#).

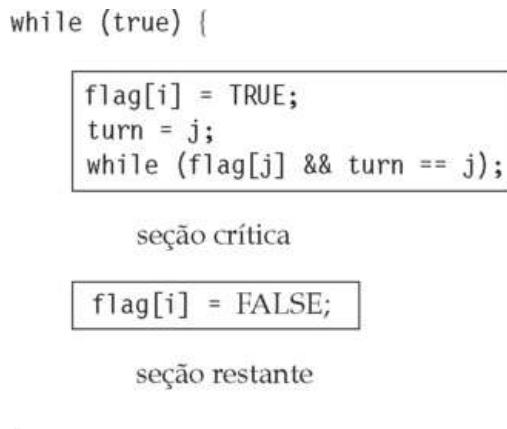


FIGURA 6.2 A estrutura do processo P_i na solução de Peterson.

Para entrar na seção crítica, o processo P_i primeiro define `flag[i]` para ser `true` e depois define `turn` para o valor j , assegurando assim que, se o outro processo quer entrar na seção crítica, ele pode fazer isso. Se os dois processos tentarem entrar ao mesmo tempo, `turn` será definido como i e j aproximadamente ao mesmo tempo. Somente uma dessas atribuições permanecerá; a outra ocorrerá mas será modificada imediatamente. O valor final de `turn` decide qual dos dois processos terá permissão para entrar em sua seção crítica primeiro.

Agora, provamos que essa solução está correta. Precisamos mostrar que:

1. A exclusão mútua é preservada.
2. O requisito de progresso é satisfeito.
3. O requisito de espera limitada é atendido.

Para provar a propriedade 1, observamos que cada P_i entra em sua seção crítica somente se `flag[j] == true` ou `turn == i`. Observe também que se os dois processos puderem executar em suas seções críticas ao mesmo tempo, então `flag[0] == flag[1] == true`. Essas duas observações significam que P_0 e P_1 não poderiam ter executado com sucesso suas instruções `while` ao mesmo tempo, pois o valor de `turn` pode ser 0 ou 1, mas não pode ser ambos. Logo, um dos processos – digamos, P_j – precisa ter executado a instrução `while` com sucesso, enquanto P_i teve que executar pelo menos uma instrução adicional (“`turn == j`”). Contudo, como nesse momento `flag[j] == true` e `turn == j`, e essa condição persistirá enquanto P_j estiver em sua seção crítica, o resultado é o seguinte: a exclusão mútua é preservada.

Para provar as propriedades 2 e 3, observamos que um processo P_i só pode ser impedido de entrar na seção crítica se estiver preso no loop “while” com a condição $\text{flag}[j] == \text{true}$ e $\text{turn} == j$; esse loop é o único possível. Se P_j não estiver pronto para entrar na seção crítica, então $\text{flag}[j] == \text{false}$, e P_i pode entrar em sua seção crítica. Se P_j tiver definido $\text{flag}[j]$ como true e também estiver executando em sua instrução `while`, então $\text{turn} == i$ ou $\text{turn} == j$. Se $\text{turn} == i$, então P_i entrará na seção crítica. Se $\text{turn} == j$, então P_j entrará na seção crítica. Contudo, quando P_j sair de sua seção crítica, ele retornará $\text{flag}[j]$ para false , permitindo que P_i entre em sua seção crítica. Se P_j definir $\text{flag}[j]$ como true , ele também precisará definir turn como i . Assim, como P_i não muda o valor da variável turn enquanto executa a instrução `while`, P_i entra na seção crítica (progresso) depois de, no máximo, uma entrada por P_j (espera limitada).

6.4 Hardware de sincronismo

Acabamos de descrever uma solução baseada em software para o problema da seção crítica. Porém, como dissemos, as soluções baseadas em software, como a de Peterson, não têm garantias de que funcionarão nas modernas arquiteturas de computador. Em vez disso, podemos dizer que qualquer solução para o problema da seção crítica requer uma ferramenta simples - um **lock** (tranca). As condições de corrida são impedidas exigindo-se que as regiões críticas sejam protegidas por locks. Ou seja, um processo deverá adquirir um lock antes de entrar em uma seção crítica; ele libera o lock quando sair da seção crítica. Isso é ilustrado na [Figura 6.3](#).

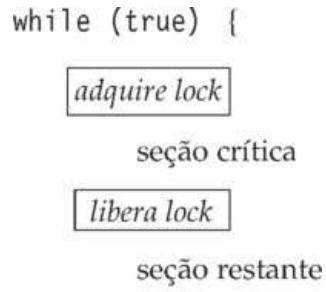


FIGURA 6.3 Solução para o problema da seção crítica usando locks.

Nas discussões a seguir, exploramos várias outras soluções para o problema da seção crítica, usando técnicas que variam desde o hardware até APIs baseadas em software, disponíveis aos programadores de aplicação. Todas essas soluções são baseadas na premissa de lock; porém, conforme veremos, o projeto desses locks pode ser bastante sofisticado.

Vamos começar apresentando algumas instruções de hardware simples, disponíveis em muitos sistemas, e mostrando como podem ser usadas efetivamente na solução do problema de seção crítica. Os recursos do hardware podem tornar a tarefa de programação mais fácil e melhorar a eficiência do sistema.

O problema de seção crítica poderia ser solucionado de forma simples em um ambiente monoprocessado se pudéssemos impedir a ocorrência de interrupções enquanto uma variável compartilhada estivesse sendo modificada. Dessa maneira, poderíamos estar certos de que a sequência atual de instruções teria permissão para ser executada na ordem, sem preempção. Nenhuma outra instrução seria executada, de modo que nenhuma modificação inesperada poderia ser feita à variável compartilhada. Essa técnica é normalmente tomada pelos kernels não preemptivos.

Infelizmente, essa solução não é viável em um ambiente multiprocessado. Desabilitar interrupções em multiprocessadores pode ser algo demorado, enquanto a mensagem é passada a todos os processadores. Essa troca de mensagem atrasa a entrada em cada seção crítica e a eficiência do sistema diminui. Além disso, considere o efeito sobre o clock de um sistema, se o clock for mantido atualizado pelas interrupções.

Muitos sistemas computadorizados modernos, portanto, proveem instruções de hardware especiais, que nos permitem testar e modificar o conteúdo de uma palavra ou trocar o conteúdo de duas palavras, **atomicamente**, ou seja, como uma unidade ininterrupta. Podemos usar essas instruções especiais para solucionar o problema da seção crítica de maneira relativamente simples. Em vez de discutirmos uma instrução específica para uma máquina específica, separamos os conceitos principais por trás desses tipos de instruções. A classe `HardwareData`, mostrada na [Figura 6.4](#), ilustrará as instruções.

```

public class HardwareData
{
    private boolean value = false;

    public HardwareData(boolean value) {
        this.value = value;
    }

    public boolean get( ) {
        return value;
    }

    public void set(boolean newValue) {
        value = newValue;
    }

    public boolean getAndSet(boolean newValue) {
        boolean oldValue = this.get( );
        this.set(newValue);

        return oldValue;
    }

    public void swap(HardwareData other) {
        boolean temp = this.get( );

        this.set(other.get( ));
        other.set(temp);
    }
}

```

FIGURA 6.4 Estrutura de dados para soluções de hardware

O método `getAndSet()` implementando a instrução *get-and-set* aparece na [Figura 6.4](#). A característica importante é que essa instrução é executada atomicamente. Assim, se duas instruções *get-and-set* forem executadas de maneira concorrente (cada uma em uma CPU diferente), elas serão executadas em sequência, em uma ordem qualquer.

Se a máquina admitir a instrução *get-and-set*, poderemos implementar a exclusão mútua declarando `lock` como um objeto da classe `HardwareData` e inicializando-o como `false`. Todas as threads compartilharão o acesso a `lock`. A [Figura 6.5](#) ilustra a estrutura de uma thread qualquer. Observe que essa thread utiliza o método `yield()` apresentado na [Seção 5.7](#). A chamada de `yield()` mantém a thread no estado executável, mas também permite que a JVM selecione outra thread executável para executar.

```

// lock é compartilhado por todas as threads
HardwareData lock = new HardwareData(false);

while (true) {
    while (lock.getAndSet(true))
        Thread.yield( );

    // seção crítica
    lock.set(false);
    // seção restante
}

```

FIGURA 6.5 Thread usando o lock *get-and-set*

A instrução *swap*, definida no método *swap()* na [Figura 6.4](#), opera sobre o conteúdo de duas palavras; assim como a instrução *get-and-set*, ela é executada atomicamente. Se a máquina aceita a instrução *swap*, então a exclusão mútua pode ser fornecida da seguinte maneira. Todas as threads compartilham um objeto *lock*, da classe *HardwareData*, que é inicializado como *false*. Além disso, cada thread também possui um objeto *HardwareData* local, chamado *key*. A estrutura de uma thread arbitrária aparece na [Figura 6.6](#).

```

// lock é compartilhado por todas as threads
HardwareData lock = new HardwareData(false);

// cada thread possui uma cópia local de key
HardwareData key = new HardwareData(true);

while (true) {
    key.set(true);

    do {
        lock.swap(key);
    }
    while (key.get( ) == true);

    // seção crítica
    lock.set(false);
    // seção restante
}

```

FIGURA 6.6 Thread usando a instrução *swap*

Infelizmente, para os designers de hardware implementar instruções *testAndSet()* em multiprocessadores não é uma tarefa fácil. Tais implementações são discutidas em livros sobre arquitetura de computadores.

6.5 Semáforos

As soluções baseadas em hardware para o problema de seção crítica, apresentadas na [Seção 6.4](#), são complicadas para programadores de aplicação. Para contornar essa dificuldade, podemos usar uma ferramenta de sincronismo chamada **semáforo**.

Um semáforo S contém uma variável inteira que, fora a inicialização, é acessada apenas por duas operações-padrão: `acquire()` e `release()`. Essas operações inicialmente se chamavam P (do holandês *proberen*, significando “testar”) e V (de *verhogen*, significando “incrementar”).

Supondo que `value` represente o valor inteiro do semáforo, as definições de `acquire()` e `release()` são mostradas na [Figura 6.7](#). As modificações feitas no valor inteiro do semáforo nas operações `acquire()` e `release()` precisam ser executadas de modo indivisível, ou seja, quando uma thread modificar o valor do semáforo, nenhuma outra thread poderá modificar esse mesmo valor de semáforo simultaneamente. Além disso, no caso de `acquire()`, o teste do valor inteiro de semáforo (`value <= 0`) e sua possível modificação (`value--`) também devem ser executados sem interrupção. Na [Seção 6.5.2](#), veremos como essas operações podem ser implementadas; primeiro, vejamos como os semáforos podem ser usados

```
acquire( ) {
    while (value <= 0)
        ; // nenhuma operação
    value--;
}

release( ) {
    value++;
}
```

FIGURA 6.7 As definições de `acquire()` e `release()`

6.5.1 Utilização

Os sistemas operacionais distinguem entre semáforos contadores e binários. O valor de um **semáforo contador** pode variar por um domínio irrestrito. O valor de um **semáforo binário** só pode variar entre 0 e 1. Em alguns sistemas, os semáforos binários são conhecidos como **mutex locks**, pois são os locks que fornecem exclusão mútua (*mutual exclusion*).

Podemos usar o semáforo binário para controlar o acesso à seção crítica para um processo ou thread. A estratégia geral é a seguinte (supondo que o semáforo seja inicializado em 1):

```
Semaphore sem = new Semaphore(1);

sem.acquire();
// seção crítica

sem.release();
// seção restante
```

Uma solução generalizada para várias threads aparece no programa Java da [Figura 6.8](#). Cinco threads separadas são criadas, mas somente uma pode estar em sua seção crítica em determinado momento. O semáforo `sem`, compartilhado por todas as threads, controla o acesso à seção crítica.

```

public class Worker implements Runnable
{
    private Semaphore sem;

    public Worker(Semaphore sem) {
        this.sem = sem;
    }

    public void run( ) {
        while (true) {
            sem.acquire( );
            criticalSection();
            sem.release( );
            remainderSection();
        }
    }
}

public class SemaphoreFactory
{
    public static void main(String args[ ]) {
        Semaphore sem = new Semaphore(1);
        Thread[ ] bees = new Thread[5];

        for (int i = 0; i < 5; i++)
            bees[i] = new Thread(new Worker(sem));
        for (int i = 0; i < 5; i++)
            bees[i].start( );
    }
}

```

FIGURA 6.8 Sincronismo usando semáforos

Semáforos contadores podem ser usados para vários objetivos, como controlar o acesso a determinado recurso consistindo em um número finito de instâncias. O semáforo é inicializado para o número de recursos disponíveis. Cada thread que deseja usar um recurso realiza uma operação `acquire()` sobre o semáforo (decrementando, assim, o contador). Quando uma thread libera um recurso, ela realiza uma operação `release()` (incrementando o contador). Quando o contador para o semáforo chega a 0, todos os recursos estão sendo usados. Depois disso, as threads que desejam usar um recurso serão bloqueadas até o contador voltar a ser maior que 0.

Também podemos utilizar semáforos para solucionar diversos problemas de sincronismo. Por exemplo, considere dois processos em execução concorrente: P_1 com uma instrução S_1 e P_2 com uma instrução S_2 . Suponha que exijamos que S_2 seja executada apenas depois que S_1 tiver terminado. Podemos implementar esse esquema prontamente permitindo que P_1 e P_2 compartilhem um semáforo comum `synch`, inicializado em 0, e inserindo as instruções

```

 $S_1;$ 
synch.release( );

```

no processo P_1 e as instruções

```

synch.acquire( );
 $S_2;$ 

```

no processo P_2 . Como `synch` é inicializado em 0, P_2 executará S_2 somente depois que P_1 tiver chamado `synch.release()`, depois que a instrução S_1 tiver sido executada.

6.5.2 Implementação

A principal desvantagem do tipo de semáforo que acabamos de descrever é que ele exige a **espera ocupada (busy waiting)**. Enquanto um processo está em sua seção crítica, qualquer outro processo que tenta entrar em sua seção crítica precisa ficar em um loop contínuo no código de entrada. Esse loop contínuo certamente é um problema em um sistema multiprogramado, no qual uma única CPU é compartilhada entre muitos processos. A espera ocupada desperdiça ciclos de CPU que algum outro processo poderia usar de forma produtiva. Um semáforo que produz esse resultado também é chamado de **spinlock**, pois o processo “gira” (spin) enquanto espera pelo lock. (Spinlocks possuem uma vantagem porque nenhuma troca de contexto é necessária quando um processo precisa esperar por um lock, e uma troca de contexto pode levar um tempo considerável. Assim, quando os locks precisam ser mantidos por períodos curtos, os spinlocks são úteis. Eles normalmente são empregados em sistemas multiprocessados, onde uma thread pode “girar” em um processador enquanto outra realiza sua seção crítica em outro processador.)

Para contornar a necessidade de espera ocupada, podemos modificar as definições das operações de semáforo `acquire()` e `release()`. Quando um processo executa a operação `acquire()` e descobre que o valor do semáforo não é positivo, ele precisa esperar. Entretanto, em vez de usar a espera ocupada, o processo pode se *bloquear*. A operação de lock coloca um processo em uma fila de espera associada ao semáforo, e o estado do processo é passado para o estado esperando. Em seguida, o controle é transferido para o escalonador da CPU, que seleciona outro processo para ser executado.

Um processo bloqueado, esperando por um semáforo S , deve ser reiniciado quando algum outro processo executar uma operação `release()`. O processo é reiniciado por uma operação *wakeup* (*acordar*), que muda o processo do estado esperando para o estado pronto. O processo é, então, colocado na fila de prontos (ready queue). (A CPU pode ou não ser passada do processo em execução para o processo que acabou de estar pronto, dependendo do algoritmo de escalonamento da CPU.)

Para implementar semáforos sob essa definição, determinamos um semáforo como (1) um valor inteiro; e (2) uma lista de processos. Quando um processo precisa esperar por um semáforo, ele é acrescentado à lista de processos para esse semáforo. A operação `release()` remove um processo da lista de processos esperando e acorda esse processo.

As operações de semáforo agora podem ser definidas como

```
acquire() {
    value--;
    if (value < 0) {
        acrescenta esse processo à lista
        block();
    }
}

release() {
    value++;
    if (value <= 0) {
        remove um processo P da lista
        wakeup(P);
    }
}
```

A operação `block()` suspende o processo que a chama. A operação `wakeup(P)` retoma a execução de um processo bloqueado P . Essas duas operações são fornecidas pelo sistema operacional como chamadas de sistema básicas.

Observe que nessa implementação o semáforo pode ter valores negativos, embora o valor do semáforo nunca seja negativo sob a definição clássica de semáforos com espera ocupada. Se for negativo, sua magnitude é a quantidade de processos esperando por esse semáforo. Esse fato é o resultado da troca da ordem do decremento e do teste na implementação da operação `acquire()`.

A lista de processos esperando pode ser facilmente implementada por um link em cada bloco de controle de processo (PCB). Cada semáforo contém um valor inteiro e um ponteiro para uma lista de PCBs. Um modo de acrescentar e remover processos da lista, que garante a espera vinculada, é usar uma fila FIFO, na qual o semáforo contém os ponteiros de cabeça e cauda para a fila. Todavia, em geral, a lista pode usar *qualquer* estratégia de enfileiramento. O uso correto dos semáforos não depende de uma estratégia de enfileiramento específica para as listas de semáforo.

Um aspecto crítico dos semáforos é que eles não são executados atomicamente. Precisamos garantir que dois processos não poderão executar as operações `acquire()` e `release()` sobre o mesmo semáforo ao mesmo tempo. Essa situação cria um problema de seção crítica, que pode ser solucionado de duas maneiras.

Em um ambiente de processador único, podemos desabilitar as interrupções durante o tempo em que as operações `acquire()` e `release()` estão sendo executadas. Quando as interrupções estão desabilitadas, as instruções de diferentes processos não podem ser intercaladas. Apenas o processo em execução é executado, até que as interrupções sejam habilitadas e o escalonador possa retomar o controle.

Entretanto, em um ambiente multiprocessado, desabilitar interrupções não funciona. As instruções de diferentes processos (executando em diferentes processadores) podem ser intercaladas de alguma maneira arbitrária. Se o hardware não fornecer quaisquer instruções especiais, poderemos empregar qualquer uma das soluções de software corretas para o problema de seção crítica ([Seção 6.2](#)), em que as seções críticas consistem nas operações `acquire()` e `release()`.

É importante admitir que ainda não eliminamos completamente a espera ocupada com essa definição das operações `acquire()` e `release()`. Em vez disso, passamos a espera ocupada para as seções críticas dos programas de aplicação. Além do mais, limitamos a espera ocupada apenas às seções críticas das operações `acquire()` e `release()`. Essas seções são curtas (se codificadas dessa maneira, não deverão ter mais do que dez instruções). Assim, a seção crítica quase nunca é ocupada; a espera ocupada raramente ocorre, e somente por um curto período. Existe uma situação totalmente diferente com os programas de aplicação, cujas seções críticas podem ser longas (minutos ou até mesmo horas) ou quase sempre podem estar ocupadas. Nesse caso, a espera ocupada é extremamente ineficaz. No decorrer deste capítulo, focalizamos questões de desempenho e mostramos as técnicas para evitar a espera ocupada. Na [Seção 6.8.7.2](#), veremos como os semáforos podem ser implementados na API Java.

6.5.3 Deadlocks e starvation

A implementação de um semáforo com uma fila de espera pode resultar em uma situação em que dois ou mais processos aguardam indefinidamente por um evento que só poderá ser causado por um dos processos aguardando. O evento em questão é a execução de uma operação `release()`. Quando esse estado é alcançado, considera-se que esses processos estão em um **deadlock** (ou impasse).

Como ilustração, consideramos um sistema consistindo em dois processos, P_0 e P_1 , cada um acessando dois semáforos, S e Q , definidos com o valor 1:

P_0	P_1
<code>S.acquire();</code>	<code>Q.acquire();</code>
<code>Q.acquire();</code>	<code>S.acquire();</code>
<code>.</code>	<code>.</code>
<code>.</code>	<code>.</code>
<code>.</code>	<code>.</code>
<code>S.release();</code>	<code>Q.release();</code>
<code>Q.release();</code>	<code>S.release();</code>

Suponha que P_0 execute `S.acquire()` e depois P_1 execute `Q.acquire()`. Quando P_0 executar `Q.acquire()`, ele terá de esperar até P_1 executar `Q.release()`. De modo semelhante, quando P_1 executar `S.acquire()`, ele terá de esperar até P_0 executar `S.release()`. Como essas operações de sinal não podem ser executadas, P_0 e P_1 estão em um deadlock.

Dizemos que um conjunto de processos está em deadlock quando cada processo no conjunto está esperando que um evento possa ser causado somente por outro processo no conjunto. Os principais eventos com os quais estamos preocupados aqui são aquisição e liberação de recursos; porém, outros tipos de eventos podem resultar em deadlocks, como mostraremos no [Capítulo 7](#), onde descrevemos diversos mecanismos para lidar com o problema do deadlock.

Outro problema relacionado com deadlocks é o **bloqueio indefinido** ou **starvation** - uma situação em que os processos esperam indefinidamente dentro do semáforo. O starvation pode ocorrer se acrescentarmos ou removermos processos da lista associada a um semáforo na ordem LIFO (último a entrar, primeiro a sair).

6.5.4 Inversão de prioridade

Existe um desafio no escalonamento quando um processo com prioridade mais alta precisa ler ou modificar dados do kernel que estão atualmente sendo acessados por um processo de prioridade

mais baixa - ou uma cadeia de processos de prioridade mais baixa. Como os dados do kernel normalmente são protegidos com um lock, o processo com prioridade mais alta terá que esperar até que o processo com prioridade mais baixa acabe de usar o recurso. A situação torna-se mais complicada se o processo com prioridade mais baixa for preemptado em favor de outro processo com uma prioridade mais alta. Por exemplo, suponha que tenhamos três processos, B , M e A , cujas prioridades seguem a ordem $B < M < A$. Suponha que o processo A exija o recurso R , que atualmente está sendo acessado pelo recurso B . Normalmente, o processo A esperará até que B acabe de usar o recurso R . Entretanto, agora suponha que o processo M torne-se executável, preemptando, assim, o processo L . Indiretamente, um processo com uma prioridade mais baixa - o processo M - afetou o tempo que o processo A terá que esperar até que B abra mão do recurso R .

Esse problema é conhecido como **inversão de prioridade**. Ele ocorre somente em sistemas com mais de duas prioridades, e portanto uma solução é ter apenas duas prioridades. Porém, isso não é suficiente para a maioria dos sistemas operacionais de uso geral. Normalmente, esses sistemas resolvem o problema implementando um protocolo de herança de prioridade. De acordo com esse protocolo, todos os processos que estão acessando recursos necessários por um processo com prioridade mais alta herdam a prioridade mais alta até que acabem de usar os recursos em questão. Quando terminarem, suas prioridades retornam aos seus valores originais. No exemplo anterior, um protocolo de herança de prioridade permitiria que o processo B temporariamente herdasse a prioridade do processo A , impedindo, assim, que o processo M preemptasse sua execução. Quando o processo B acabasse de usar o recurso R , ele abriria mão de sua prioridade herdada de A e assumiria sua prioridade original. Como o recurso R não estaria disponível, o processo A - e não M - seria executado em seguida.

INVERSÃO DE PRIORIDADE E A MARS PATHFINDER

A inversão de prioridade pode ser mais do que uma inconveniência de escalonamento. Em sistemas com restrições de tempo estritas (como em sistemas de tempo real - ver [Capítulo 19](#)), a inversão de prioridade pode fazer um processo levar mais tempo do que deveria para realizar uma tarefa. Quando isso acontece, outras falhas podem ser propagadas em cascata, resultando na falha geral do sistema.

Considere a sonda Mars Pathfinder, uma sonda espacial da NASA que pousou um robô, o dispositivo Sojourner, em 1997 na superfície de Marte, para realizar experiências. Logo depois que o Sojourner iniciou sua operação, ele começou a experimentar reinicializações frequentes no computador. Cada um deles reinicializava todo o hardware e software, incluindo as comunicações. Se o problema não tivesse sido resolvido, o Sojourner teria fracassado em sua missão.

O problema foi causado pelo fato de que uma tarefa de alta prioridade, "bc_dist", estava levando mais tempo do que o esperado para concluir seu trabalho. Essa tarefa estava sendo forçada a esperar por um recurso compartilhado que estava sendo mantido pela tarefa com prioridade mais baixa "ASI/MET", que, por sua vez, era preemptada por diversas tarefas de prioridade média. A tarefa "bc_dist" atrasaria a espera pelo recurso compartilhado, e, por fim, a tarefa "bc_sched" descobriria o problema e realizaria a reinicialização. O Sojourner estava sofrendo de um caso típico de inversão de prioridade.

O sistema operacional no Sojourner era o VxWorks ([Seção 19.6](#)), que tinha uma variável global para habilitar a herança de prioridade em todos os semáforos. Após o teste, a variável foi definida no Sojourner (em Marte!) e o problema foi resolvido.

Uma descrição completa do problema, sua detecção e sua solução foi escrita pelo líder da equipe de software e está disponível em research.microsoft.com/mbj/Mars_Pathfinder/Authoritative_Account.html.

6.6 Problemas clássicos de sincronismo

Nesta seção, apresentamos uma série de problemas de sincronismo diferentes como exemplos para uma grande classe de problemas de controle de concorrência. Esses problemas são usados para testar quase todo esquema de sincronismo recém-proposto. Em nossas soluções para os problemas, usamos semáforos para o sincronismo.

6.6.1 O problema do bounded-buffer

O problema do bounded-buffer foi introduzido na [Seção 6.1](#); ele normalmente é usado para ilustrar o poder das primitivas de sincronismo. Uma solução é mostrada na [Figura 6.9](#). Um produtor coloca um item no buffer chamando o método `insert()` ([Figura 6.10](#)); os consumidores removem itens invocando `remove()` ([Figura 6.11](#)).

```
public class BoundedBuffer<E> implements Buffer<E>
{
    private static final int BUFFER_SIZE = 5;
    private E[ ] buffer;
    private int in, out;
    private Semaphore mutex;
    private Semaphore empty;
    private Semaphore full;

    public BoundedBuffer( ) {
        // buffer está inicialmente vazio
        in = 0;
        out = 0;
        mutex = new Semaphore(1);
        empty = new Semaphore(BUFFER_SIZE);
        full = new Semaphore(0);

        buffer = (E[]) new Object[BUFFER_SIZE];
    }

    public void insert(E item) {
        // Figura 6.10
    }

    public E remove( ) {
        // Figura 6.11
    }
}
```

FIGURA 6.9 Solução para o problema de bounded-buffer (produtor-consumidor) utilizando semáforos

```

// Produtores chamam este método
public void insert(E item) {
    empty.acquire();
    mutex.acquire();

    // acrescenta um item ao buffer
    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;

    mutex.release();
    full.release();
}

```

FIGURA 6.10 O método `insert()`

```

// Consumidores chamam este método
public E remove() {
    E item;

    full.acquire();
    mutex.acquire();

    // remove um item do buffer
    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;

    mutex.release();
    empty.release();

    return item;
}

```

FIGURA 6.11 O método `remove()`

O semáforo `mutex` provê exclusão mútua para os acessos ao pool de buffers e é inicializado com 1. Os semáforos `empty` e `full` contam o número de buffers vazios e cheios, respectivamente. O semáforo `empty` é inicializado com a capacidade do buffer - `BUFFER_SIZE`; o semáforo `full` é inicializado com 0.

A thread produtor é mostrada na [Figura 6.12](#). O produtor alterna entre dormir por um tempo produzindo uma mensagem e tentar colocar essa mensagem no buffer por meio do método `insert()`.

```

import java.util.Date;

public class Producer implements Runnable
{
    private Buffer<Date> buffer;

    public Producer(Buffer<Date> buffer) {
        this.buffer = buffer;
    }

    public void run( ) {
        Date message;

        while (true) {
            // dorme por um tempo
            SleepUtilities.nap( );

            // produz um item e o insere no buffer
            message = new Date( );
            buffer.insert(message);
        }
    }
}

```

FIGURA 6.12 O produtor

A thread consumidor aparece na [Figura 6.13](#). O consumidor alterna entre dormir e consumir um item usando o método `remove()`.

```

import java.util.Date;

public class Consumer implements Runnable
{
    private Buffer<Date> buffer;

    public Consumer(Buffer<Date> buffer) {
        this.buffer = buffer;
    }

    public void run( ) {
        Date message;

        while (true) {
            // dorme por um tempo
            SleepUtilities.nap( );

            // consome um item do buffer
            message = (Date)buffer.remove( );
        }
    }
}

```

FIGURA 6.13 O consumidor

A classe `Factory` ([Figura 6.14](#)) cria as threads produtor e consumidor, passando a cada uma delas uma referência ao objeto `BoundedBuffer`.

```

import java.util.Date;

public class Factory
{
    public static void main(String args[ ] ) {
        Buffer<Date> buffer = new BoundedBuffer<Date>( );

        // Cria as threads produtor e consumidor
        Thread producer = new Thread(new
Producer(buffer));
        Thread consumer = new Thread(new
Consumer(buffer));

        producer.start( );
        consumer.start( );
    }
}

```

FIGURA 6.14 A classe Factory

6.6.2 O problema dos leitores-escritores

Suponha que um banco de dados deva ser compartilhado entre diversas threads concorrentes. Algumas dessas threads podem querer apenas ler o banco de dados, enquanto outras podem querer atualizar o banco de dados (ou seja, ler e escrever nele). Distinguimos entre esses dois tipos de threads nos referindo ao primeiro como **leitores** e ao segundo como **escritores**. É claro que, se dois leitores acessarem os dados compartilhados concorrentemente, não teremos qualquer efeito adverso. Todavia, se um escritor e alguma outra thread (seja um leitor ou um escritor) acessar o banco de dados concorrentemente, poderá ocorrer um desastre.

Para garantir que essas dificuldades não surjam, exigimos que os escritores tenham acesso exclusivo ao banco de dados compartilhado. Esse requisito leva ao problema de **leitores-escritores**. Desde sua enunciação, esse problema tem sido usado para testar quase toda nova primitiva de sincronismo. O problema possui diversas variações, todas envolvendo prioridades. A mais simples, conhecida como o *primeiro* problema de leitores-escritores, exige que nenhum leitor seja mantido esperando, a menos que um escritor já tenha obtido permissão para usar o banco de dados compartilhado. Em outras palavras, nenhum leitor deverá esperar outros leitores terminarem porque um escritor está esperando. O *segundo* problema de leitores-escritores exige que, quando um escritor estiver pronto, ele realize sua escrita o mais breve possível. Em outras palavras, se um escritor estiver esperando para acessar o objeto, nenhum novo leitor poderá começar a ler.

Uma solução para qualquer um desses problemas pode resultar em starvation. No primeiro caso, os escritores podem estagnar; no segundo caso, os leitores podem estagnar. Por esse motivo, outras variantes do problema têm sido propostas. A seguir, apresentamos os arquivos de classe Java para uma solução para o primeiro problema de leitores-escritores. Ela não focaliza a starvation. (Nos exercícios ao final do capítulo, você modificará a solução para torná-la livre de starvation.) Cada thread leitor alterna entre dormir e ler, como mostra a [Figura 6.15](#). Quando um leitor deseja ler o banco de dados, ele invoca o método `acquireReadLock()`; quando ele tiver terminado a leitura, chamará `releaseReadLock()`. Cada thread escritor ([Figura 6.16](#)) funciona de modo semelhante.

```

public class Reader implements Runnable
{
    private ReadWriteLock db;

    public Reader(ReadWriteLock db) {
        this.db = db;
    }

    public void run() {
        while (true) {
            // dorme por um tempo
            SleepUtilities.nap();

            db.acquireReadLock();

            // você tem acesso para ler do banco de dados
            SleepUtilities.nap();

            db.releaseReadLock();
        }
    }
}

```

FIGURA 6.15 Um leitor

```

public class Writer implements Runnable
{
    private ReadWriteLock db;

    public Writer(ReadWriteLock db) {
        this.db = db;
    }

    public void run() {
        while (true) {
            // dorme por um tempo
            SleepUtilities.nap();

            db.acquireWriteLock();

            // você tem acesso para escrever no banco de dados
            SleepUtilities.nap();

            db.releaseWriteLock();
        }
    }
}

```

FIGURA 6.16 Um escritor

Os métodos chamados em cada thread leitor e escritor são definidos na interface `ReadWriteLock` da [Figura 6.17](#). A classe `Database` da [Figura 6.18](#) implementa essa interface. O `readerCount` regista o número de leitores. O semáforo `mutex` é usado para garantir a exclusão mútua quando `readerCount` for atualizado. O semáforo `db` funciona como um semáforo de exclusão mútua para os escritores. Ele também é usado pelos leitores para impedir que os escritores entrem no banco de dados enquanto o banco de dados está sendo lido. O primeiro leitor realiza uma operação `acquire()` sobre `db`, evitando, assim, que quaisquer escritores entrem no banco de dados. O leitor final realiza uma operação `release()` sobre `db`. Observe que, se um escritor estiver ativo no banco de dados e n leitores estiverem aguardando, então um leitor é enfileirado em `db` e $n - 1$ leitores são enfileirados em `mutex`. Observe também que quando um escritor executa `db.release()`, podemos retomar a execução de

quaisquer leitores aguardando ou um único escritor aguardando. A seleção é feita pelo escalonador.

```
public interface ReadWriteLock
{
    public void acquireReadLock();
    public void acquireWriteLock();
    public void releaseReadLock();
    public void releaseWriteLock();
}
```

FIGURA 6.17 A interface para o problema dos leitores-escritores

```
public class Database implements ReadWriteLock
{
    private int readerCount;
    private Semaphore mutex;
    private Semaphore db;

    public Database( ) {
        readerCount = 0;
        mutex = new Semaphore(1);
        db = new Semaphore(1);
    }

    public void acquireReadLock( ) {
        // Figura 6.19
    }

    public void releaseReadLock( ) {
        // Figura 6.19
    }

    public void acquireWriteLock( ) {
        // Figura 6.20
    }

    public void releaseWriteLock( ) {
        // Figura 6.20
    }
}
```

FIGURA 6.18 O banco de dados para o problema dos leitores-escritores

Os locks de leitura-escrita são mais úteis nas seguintes situações:

- Em aplicações fáceis de identificar quais threads só leem dados compartilhados e quais só escrevem dados compartilhados.
- Em aplicações que possuem mais leitores do que escritores. Isso porque os locks de leitura-escrita em geral exigem mais custo adicional para serem estabelecidos do que os semáforos ou locks de exclusão mútua, e o custo adicional para configurar um lock de leitura-escrita é equilibrado pela maior concorrência da permissão de leitores múltiplos.

```

public void acquireReadLock( ) {
    mutex.acquire( );

    /**
     * O primeiro leitor indica que
     * o banco de dados está sendo lido.
     */
    ++readerCount;
    if (readerCount == 1)
        db.acquire( );

    mutex.release( );
}

public void releaseReadLock( ) {
    mutex.acquire( );

    /**
     * O último leitor indica que
     * o banco de dados não está mais sendo lido.
     */
    --readerCount;
    if (readerCount == 0)
        db.release( );

    mutex.release( );
}

```

FIGURA 6.19 Métodos chamados pelos leitores

6.6.3 O problema dos filósofos na mesa de jantar

Imagine cinco filósofos que gastam a vida pensando e comendo. Os filósofos compartilham uma mesa redonda, cercada por cinco cadeiras, cada uma pertencendo a um filósofo. No centro da mesa existe uma tigela de arroz, e a mesa está disposta com apenas cinco hashis ([Figura 6.21](#)). Quando um filósofo pensa, ele não interage com os colegas. De vez em quando, um filósofo tem fome e tenta pegar os dois hashis próximos a ele (os hashis que estão entre ele e seus vizinhos da esquerda e da direita). Um filósofo só pode pegar um hashi de cada vez. Obviamente, ele não pode pegar um hashi que já esteja na mão de um vizinho. Quando um filósofo com fome tem dois hashis ao mesmo tempo, ele come sem largar os hashis. Quando termina de comer, ele coloca os dois hashis na mesa e recomeça a pensar.



FIGURA 6.21 A situação dos filósofos à mesa de jantar

O **problema dos filósofos na mesa de jantar** é considerado um problema clássico de sincronismo, não por causa de sua importância prática nem porque os cientistas de computador não gostam de filósofos, mas porque é um exemplo de uma grande classe de problemas de controle de concorrência. Essa é uma representação simples da necessidade de alocar vários recursos entre vários processos de uma maneira sem deadlock e sem starvation.

Uma solução simples é representar cada hashi por um semáforo. Um filósofo tenta apanhar o hashi executando uma operação `acquire()` sobre esse semáforo; ele solta um hashi executando a operação `release()` sobre os semáforos apropriados. Assim, os dados compartilhados são

```
Semaphore hashi[ ] = new Semaphore [5];
```

```
for(int i = 0; i < 5; i++)
    hashi[i] = new Semaphore(1);
```

onde todos os elementos de `hashi` são inicializados com 1. A estrutura do filósofo i aparece na Figura 6.22.

```
while (true) {
    // apanha hashi da esquerda
    hashi[i].acquire();
    // apanha hashi da direita
    hashi[(i + 1) % 5].acquire();

    eating();

    // retorna hashi da esquerda
    hashi[i].release();
    // retorna hashi da direita
    hashi[(i + 1) % 5].release();

    thinking();
}
```

FIGURA 6.22 A estrutura do filósofo i

Embora essa solução garanta que dois filósofos vizinhos não estarão comendo simultaneamente, ela precisa ser rejeitada porque tem a possibilidade de criar um deadlock. Suponha que todos os cinco filósofos fiquem com fome ao mesmo tempo e cada um apanhe o hashi à sua esquerda. Agora, todos os elementos de `chopstick` serão iguais a 0. Quando cada filósofo tentar pegar o hashi da direita, ele esperará indefinidamente.

Várias soluções possíveis para o problema de deadlock são listadas a seguir. Essas soluções

impedem o deadlock colocando restrições sobre os filósofos:

- Permitir que, no máximo, quatro filósofos sentem simultaneamente à mesa.
- Só permitir que um filósofo apanhe os hashis se os dois hashis estiverem disponíveis (observe que ele precisa apanhá-los em uma seção crítica).
- Usar uma solução assimétrica; por exemplo, um filósofo ímpar apanha primeiro o hashi da esquerda e depois o da direita, enquanto um filósofo par apanha o hashi da direita e depois o da esquerda.

Na [Seção 6.7](#), apresentamos uma solução para o problema dos filósofos na mesa de jantar, que garante ausência de deadlocks. Observe, porém, que qualquer solução satisfatória para o problema dos filósofos à mesa de jantar precisa de proteção contra a possibilidade de um dos filósofos ainda morrer de fome. Uma solução sem deadlock não elimina necessariamente a possibilidade de starvation.

6.7 Monitores

Embora os semáforos forneçam um mecanismo conveniente e eficaz para o sincronismo de processos, usá-los incorretamente pode resultar em erros de temporização difíceis de detectar, pois esses erros só acontecem se ocorrerem determinadas sequências de execução, e essas sequências nem sempre ocorrem.

Vimos um exemplo desses erros com o uso de contadores em nossa solução para o problema de produtor-consumidor ([Seção 6.1](#)). Naquele exemplo, o problema de temporização só aconteceu raramente, e mesmo assim o valor do contador pareceu ser razoável – deslocado apenas em 1. Apesar disso, a solução não é aceitável. É por esse motivo que os semáforos foram introduzidos em primeiro lugar.

Infelizmente, esses erros de temporização ainda podem ocorrer quando são usados semáforos. Para ilustrar como, revemos a solução de semáforo para o problema de seção crítica. Todos os processos compartilham uma variável semáforo `mutex`, inicializada como 1. Cada processo precisa executar `mutex.acquire()` antes de entrar na seção crítica e `mutex.release()` depois disso. Se essa sequência não for observada, dois processos podem estar em suas seções críticas simultaneamente. Vamos examinar as várias dificuldades resultantes. Observe que essas dificuldades surgirão mesmo que um único processo não se comporte bem. Essa situação pode ser causada por um erro de programação honesto ou por um programador não cooperativo.

- Suponha que um processo troque a ordem em que são executadas as operações `acquire()` e `release()` sobre o semáforo `mutex`, resultando na seguinte execução:

```
mutex.release( );
...
seção crítica
...
mutex.acquire( );
```

Nessa situação vários processos podem estar executando em suas seções críticas concorrentemente, violando o requisito de exclusão mútua. Esse erro só poderá ser descoberto se vários processos estiverem ativos de forma simultânea em suas seções críticas. Observe que essa situação nem sempre poderá ser reproduzível.

- Suponha que um processo substitua `mutex.release()` por `mutex.acquire()`, ou seja, ele executa

```
mutex.acquire( );
...
seção crítica
...
mutex.acquire( );
```

Nesse caso, ocorrerá um deadlock.

- Suponha agora que um processo omite o `mutex.acquire()` ou o `mutex.release()`, ou ambos. Nesse caso, ou a exclusão mútua é violada ou haverá um deadlock.

Esses exemplos ilustram que vários tipos de erros podem ser gerados com facilidade quando os programadores utilizam semáforos incorretamente para solucionar o problema de seção crítica. Problemas semelhantes podem surgir nos outros modelos de sincronismo discutidos na [Seção 6.6](#).

Para lidar com tais erros, os pesquisadores desenvolveram construções de linguagem de alto nível. Nesta seção, vamos descrever uma construção fundamental para o sincronismo de alto nível, o tipo **monitor**.

6.7.1 Uso

Um *tipo*, ou um tipo de dado abstrato, encapsula dados privados com métodos públicos para operar sobre esses dados. Um tipo monitor apresenta um conjunto de operações definidas pelo programador, que recebem exclusão mútua dentro do monitor. O tipo monitor também contém a declaração de variáveis cujos valores definem o estado de uma instância desse tipo, junto com os corpos dos procedimentos ou funções que operam sobre essas variáveis. A síntese de um monitor é

mostrada na [Figura 6.23](#). A representação de um tipo monitor não pode ser usada pelos diversos processos. Portanto, um procedimento definido dentro de um monitor só pode acessar as variáveis declaradas localmente dentro do monitor e seus parâmetros formais. De modo semelhante, as variáveis locais só podem ser acessadas pelos procedimentos locais.

```
monitor nome do monitor
{
    // declarações de variáveis compartilhadas

    procedure P1 (...) {
        ...
    }

    procedure P2 (...) {
        ...
    }

    .
    .

    procedure Pn (...) {
        ...
    }

    initialization code (...) {
        ...
    }
}
```

FIGURA 6.23 Sintaxe de um monitor

A construção do monitor assegura que somente um processo pode estar ativo dentro do monitor por vez. Como consequência, o programador não precisa codificar essa restrição de sincronismo explicitamente ([Figura 6.24](#)). Contudo, a construção do monitor, conforme definida até aqui, não é suficientemente poderosa para modelar alguns esquemas de sincronismo. Para essa finalidade, precisamos definir mecanismos de sincronismo adicionais. Esses mecanismos são fornecidos pela construção da **variável de condição**. Um programador que precisa escrever seu próprio esquema de sincronismo personalizado pode definir uma ou mais variáveis do tipo `Condition`:

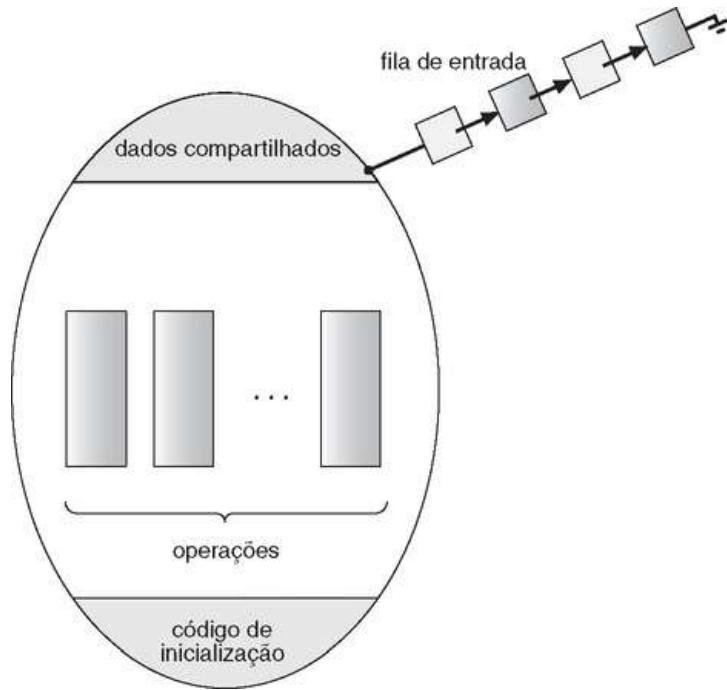


FIGURA 6.24 Visão esquemática de um monitor

`Condition x, y;`

As únicas operações que podem ser chamadas em uma variável de condição são `wait()` e `signal()`. A operação

`x.wait();`

significa que o processo que a chama é suspenso até outro processo chamar

`x.signal();`

A operação `x.signal()` reassume exatamente um processo suspenso. Se nenhum processo estiver suspenso, então a operação `signal()` não terá efeito; ou seja, o estado de `x` é o mesmo como se a operação nunca tivesse sido executada ([Figura 6.25](#)). Compare essa operação com a operação `signal()` associada aos semáforos, que sempre afeta o estado do semáforo.

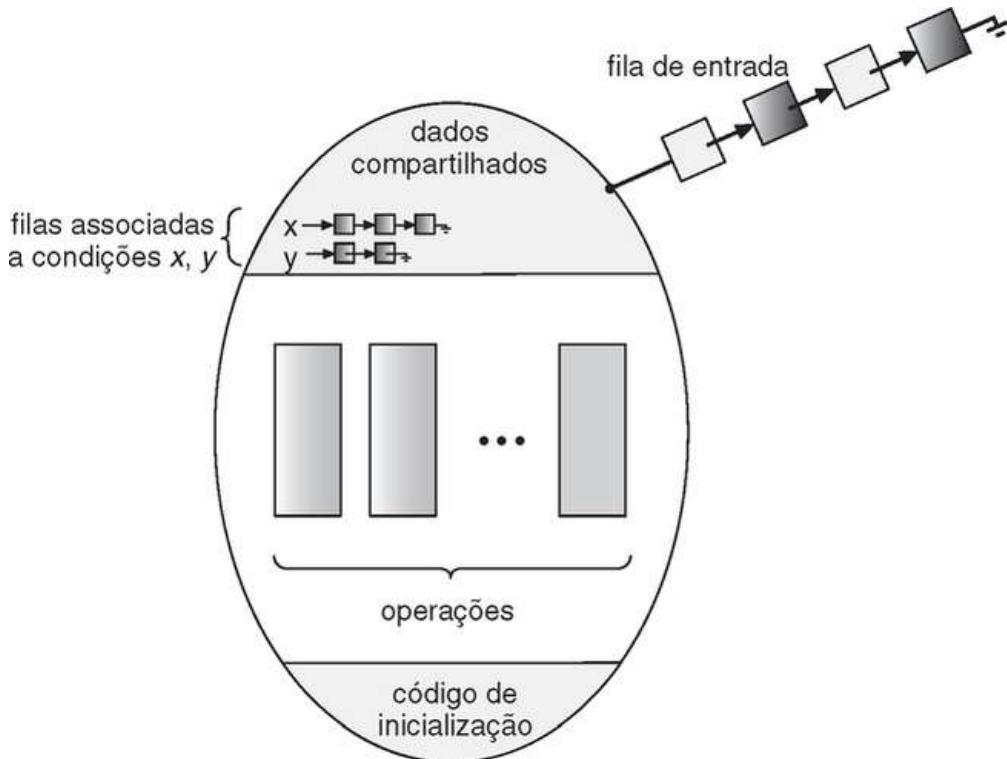


FIGURA 6.25 Monitor com variáveis de condição

Agora suponha que quando a operação `x.signal()` é chamada por um processo P , exista um processo suspenso Q associado à condição x . Logicamente, se o processo suspenso Q tiver permissão para reassumir sua execução, o processo sinalizador P terá de esperar. Caso contrário, P e Q estarão ativos simultaneamente dentro do monitor. No entanto, observe que os dois processos, em conceito, podem continuar com sua execução. Existem duas possibilidades:

1. **Sinalizar (signal) e esperar (wait).** P espera até Q sair do monitor ou espera outra condição.
2. **Sinalizar (signal) e continuar (continue).** Q espera até P sair do monitor ou espera por outra condição.

Existem argumentos razoáveis em favor da adoção de cada uma dessas opções. Por um lado, como P sempre esteve em execução no monitor, o método `signal-econtinue` parece ser mais razoável. Por outro lado, se permitirmos que a thread P continue, então, quando Q for reassumida, a condição lógica pela qual Q estava esperando pode não existir mais. Um meio-termo entre essas duas opções foi adotado na linguagem Concurrent Pascal. Quando a thread P executa a operação `signal()`, ela imediatamente sai do monitor. Logo, Q é imediatamente reassumida.

Muitas linguagens de programação têm incorporado a ideia do monitor descrito nesta seção, incluindo Concurrent Pascal, Mesa, C# (pronuncia-se *C sharp*) e Java. Outras linguagens – como Erlang – fornecem algum tipo de suporte de concorrência usando um mecanismo semelhante. Discutimos os mecanismos de sincronismo da Java por completo na [Seção 6.8](#).

6.7.2 Solução de filósofos na mesa de jantar usando monitores

Ilustramos esses conceitos apresentando uma solução sem deadlock para o problema dos filósofos à mesa de jantar. Essa solução impõe uma restrição de que um filósofo só pode apanhar os hashis se ambos estiverem disponíveis. Para codificar essa solução, precisamos distinguir entre os três estados em que podemos encontrar um filósofo. Para essa finalidade, introduzimos as seguintes estruturas de dados:

```
enum State {THINKING, HUNGRY, EATING};  
State states = new State[5];
```

O filósofo i só pode definir a variável `state[i] = State.EATING` se seus dois vizinhos não estiverem comendo; ou seja, as condições (`state [(i + 4) % 5] != State.EATING`) e (`state[(i + 1) % 5] != State.EATING`) são verdadeiras.

Também precisamos declarar

```
Condition[ ] self = new Condition[5];
```

onde o filósofo i pode se atrasar quando estiver com fome, mas não consegue obter os hashis de que precisa.

Agora, estamos em posição de descrever nossa solução para o problema dos filósofos à mesa de jantar. A distribuição dos hashis é controlada pelo monitor dp , que é uma instância do tipo de monitor `DiningPhilosophers`. Na [Figura 6.26](#), mostramos a definição desse tipo de monitor usando um pseudocódigo tipo Java. Cada filósofo, antes de começar a comer, precisa chamar a operação `takeForks()`. Esse ato pode resultar na suspensão da thread do filósofo. Depois do término bem-sucedido da operação, o filósofo poderá comer. Depois de comer, o filósofo chama a operação `returnForks()` e começa a pensar. Assim, o filósofo i precisa chamar as operações `takeForks()` e `returnForks()` na seguinte sequência:

```
monitor DiningPhilosophers
{
    enum State {THINKING, HUNGRY, EATING};
    State[ ] states = new State[5];
    Condition[ ] self = new Condition[5];

    public DiningPhilosophers {
        for (int i = 0; i < 5; i++)
            state[i] = State.THINKING;
    }

    public void takeForks(int i) {
        state[i] = State.HUNGRY;
        test(i);
        if (state[i] != State.EATING)
            self[i].wait();
    }

    public void returnForks(int i) {
        state[i] = State.THINKING;
        // testa vizinhos da esquerda e da direita
        test((i + 4) % 5);
        test((i + 1) % 5);
    }

    private void test(int i) {
        if ( (state[(i + 4) % 5] != State.EATING) &&
            (state[i] == State.HUNGRY) &&
            (state[(i + 1) % 5] != State.EATING) )
        {
            state[i] = State.EATING;
            self[i].signal();
        }
    }
}
```

FIGURA 6.26 Uma solução de monitor para o problema dos filósofos na mesa de jantar

```
dp.takeForks(i);
eat( );
dp.returnForks(i);
```

É fácil mostrar que essa solução garante que dois filósofos vizinhos não estarão comendo simultaneamente e que não haverá deadlocks. Contudo, observamos que é possível um filósofo

morrer de fome. Não apresentamos uma solução para esse problema, mas deixamos que você a desenvolva nos exercícios.

6.8 Sincronismo em Java

Agora que temos uma base em teoria de sincronismo, podemos descrever como a Java sincroniza a atividade das threads, permitindo que o programador desenvolva soluções generalizadas para impor a exclusão mútua entre as threads. Quando uma aplicação garante que os dados permanecem coerentes mesmo quando estão sendo acessados concorrentemente por várias threads, a aplicação é considerada **segura para thread (thread-safe)**.

6.8.1 Bounded buffer

A solução de memória compartilhada para o problema de bounded buffer foi descrita no [Capítulo 3](#). Essa solução sofre de dois problemas. Primeiro, o produtor e o consumidor utilizam loops de espera ocupada se o buffer estiver cheio ou vazio. Segundo, conforme mostramos novamente na [Seção 6.1](#), a condição de corrida sobre a variável `count` é compartilhada pelo produtor e pelo consumidor. Esta seção resolve esses e outros problemas enquanto desenvolve uma solução por meio dos mecanismos de sincronismo Java.

6.8.1.1 Espera ocupada e livelock

A espera ocupada foi introduzida na [Seção 6.5.2](#), onde examinamos uma implementação das operações de semáforo `acquire()` e `release()`. Naquela seção, descrevemos como um processo poderia se bloquear como uma alternativa à espera ocupada. Um modo de realizar tal bloqueio em Java é fazer uma thread chamar o método `Thread.yield()`. Lembre-se de que, como vimos na [Seção 5.7](#), quando uma thread invoca o método `yield()`, ela permanece no estado executável, mas permite que a JVM selecione outra thread executável para ser executada. O método `yield()` faz um uso mais eficaz da CPU do que a espera ocupada.

No entanto, nesse caso, usar a espera ocupada *ou* a concessão poderá gerar outro problema, conhecido como **livelock**. O livelock é semelhante a um deadlock; ambos evitam o prosseguimento de duas ou mais threads, mas as threads não podem prosseguir por motivos diferentes. O deadlock ocorre quando cada thread em um conjunto está bloqueada esperando por um evento que só pode ser causado por outra thread bloqueada no conjunto. O livelock ocorre quando uma thread tenta continuamente realizar uma ação que falha.

Aqui está um cenário que poderia causar a ocorrência de um livelock. Lembre-se de que a JVM escalona threads usando um algoritmo baseado em prioridade, favorecendo as threads com alta prioridade em relação às threads com prioridade menor. Se o produtor tiver uma prioridade maior do que a do consumidor e o buffer estiver cheio, o produtor entrará no loop `while` e na espera ocupada ou usará `yield()` para renunciar a outra thread executável, enquanto espera que `count` seja decrementado para menos do que `BUFFER_SIZE`. Desde que o consumidor tenha uma prioridade inferior à do produtor, ele nunca poderá ser escalonado pela JVM para execução e, portanto, nunca poderá ser capaz de consumir um item e liberar espaço no buffer para o produtor. Nessa situação, o produtor está impedido, esperando que o consumidor libere espaço no buffer. Logo veremos que existe uma alternativa melhor do que a espera ocupada ou a concessão enquanto se espera a ocorrência de um evento desejado.

6.8.1.2 Condição de corrida

Na [Seção 6.1](#), vimos um exemplo das consequências de uma condição de corrida sobre a variável compartilhada `count`. A [Figura 6.27](#) ilustra como o tratamento do acesso concorrente em Java aos dados compartilhados impede as condições de corrida.

```

// Produtores chamam este método
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE)
        Thread.yield( );

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;
}

// Consumidores chamam este método
public synchronized E remove() {
    E item;

    while (count == 0)
        Thread.yield( );

    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    --count;

    return item;
}

```

FIGURA 6.27 Métodos sincronizados `insert()` e `remove()`

Ao descrever essa situação, introduzimos uma nova palavra-chave: `synchronized`. Cada objeto em Java possui, associado a ele, um único lock. O lock de um objeto só pode pertencer a uma única thread. Normalmente, quando um objeto está sendo referenciado (ou seja, quando seus métodos estão sendo invocados), o lock é ignorado. Entretanto, quando um método é declarado como `synchronized`, para chamar o método é preciso obter o lock do objeto. Se o lock já estiver pertencendo a outra thread, a thread que chama o método `synchronized` é bloqueada e colocada no **conjunto de entrada (entry set)** do lock do objeto. O conjunto de entrada representa o conjunto das threads esperando até que o lock fique disponível. Se o lock estiver disponível quando um método `synchronized` for chamado, a thread que chama se torna a proprietária do lock do objeto e pode entrar no método. O lock é liberado quando a thread sai do método. Se o conjunto de entrada do lock não estiver vazio quando o lock for liberado, a JVM seleciona arbitrariamente uma thread desse conjunto para ser a proprietária do lock. (Quando dizemos “arbitrariamente”, queremos dizer que a especificação não exige que as threads nesse conjunto sejam organizadas com qualquer ordem específica. Todavia, na prática, a maioria das máquinas virtuais ordena as threads no conjunto de espera de acordo com uma política FIFO.) A [Figura 6.28](#) ilustra como o conjunto de entrada opera.

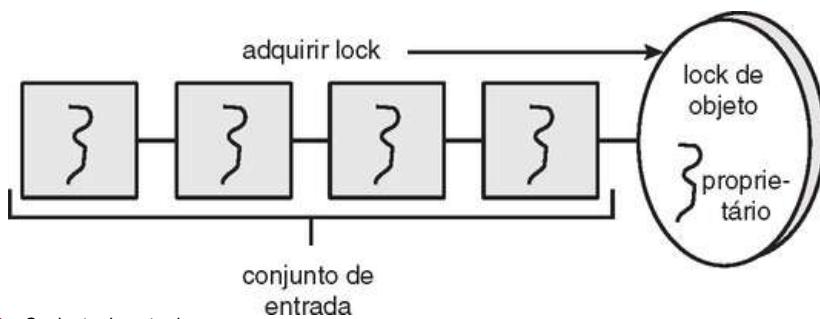


FIGURA 6.28 Conjunto de entrada

Se o produtor chamar o método `insert()`, como mostra a [Figura 6.27](#), e o lock do objeto estiver disponível, o produtor se torna o proprietário do lock; ele pode, então, entrar no método, onde pode alterar o valor de `count` e outros dados compartilhados. Se o consumidor tentar chamar o método `synchronized remove()` enquanto o produtor tem a posse do lock, o consumidor estará bloqueado, porque o lock não está disponível. Quando o produtor sai do método `insert()`, ele libera o lock. O consumidor agora pode obter o lock e entrar no método `remove()`.

6.8.1.3 Deadlock

À primeira vista, essa técnica parece pelo menos solucionar o problema de ter uma condição de corrida na variável `count`. Como o método `insert()` e o método `remove()` são declarados como `synchronized`, garantimos que somente uma thread poderá estar ativa em um desses métodos de cada vez. Porém, a propriedade do lock levou a outro problema.

Suponha que o buffer esteja cheio e o consumidor esteja dormindo. Se o produtor chamar o método `insert()`, ele pode continuar, pois o lock estará disponível. Quando o produtor invoca o método `insert()`, ele vê que o buffer está cheio e realiza o método `yield()`. Em todo o tempo, o produtor ainda tem a posse do lock do objeto. Quando o consumidor desperta e tenta chamar o método `remove()` (que, por fim, liberaria o espaço no buffer para o produtor), ele é bloqueado, pois não tem a posse do lock do objeto. Assim, tanto o produtor quanto o consumidor são incapazes de prosseguir porque (1) o produtor está bloqueado esperando que o consumidor libere o espaço no buffer, e (2) o consumidor está bloqueado esperando que o produtor libere o lock.

Declarando cada método como `synchronized`, evitamos a condição de corrida sobre as variáveis compartilhadas. Entretanto, a presença do loop `yield()` levou a um possível deadlock.

6.8.1.4 Wait e notify

A [Figura 6.29](#) focaliza o loop `yield()`, introduzindo dois novos métodos Java: `wait()` e `notify()`. Além de ter um lock, cada objeto também tem, associado a ele, um **conjunto de espera (wait set)**, que consiste em um conjunto de threads. Esse conjunto de espera está inicialmente vazio. Quando uma thread entra em um método `synchronized`, ela tem a posse do lock do objeto. Todavia, essa thread pode determinar que ela é incapaz de continuar, porque determinada condição não foi atendida. Isso acontecerá, por exemplo, se o produtor chamar o método `insert()` e o buffer estiver cheio. A thread, então, liberará o lock e esperará pela condição que permita continuar, evitando a situação de deadlock que existia anteriormente.

```

// Produtores chamam este método
public synchronized void insert(E item) {
    while (count == BUFFER_SIZE) {
        try {
            wait();
        }
        catch (InterruptedException e) { }
    }

    buffer[in] = item;
    in = (in + 1) % BUFFER_SIZE;
    ++count;

    notify();
}

// Consumidores chamam este método
public synchronized E remove() {
    E item;

    while (count == 0) {
        try {
            wait();
        }
        catch (InterruptedException e) { }
    }

    item = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
    --count;

    notify();
}

return item;
}

```

FIGURA 6.29 Métodos `insert()` e `remove()` usando `wait()` e `notify()`

Quando uma thread chama o método `wait()`, acontece o seguinte:

1. A thread libera o lock do objeto.
2. O estado da thread é definido como bloqueado.
3. A thread é colocada no conjunto de espera do objeto.

Considere o exemplo da [Figura 6.29](#). Se o produtor chamar o método `insert()` e descobrir que o buffer está cheio, ele chamará o método `wait()`. Essa chamada libera o lock, bloqueia o produtor e coloca o produtor no conjunto de espera do objeto. Como o produtor liberou o lock, o consumidor por fim entra no método `remove()`, onde libera o espaço no buffer para o produtor. A [Figura 6.30](#) ilustra os conjuntos de entrada e espera por um lock. (Observe que `wait()` pode resultar em uma `InterruptedException` sendo lançada. Veremos isso na [Seção 6.8.6](#).)

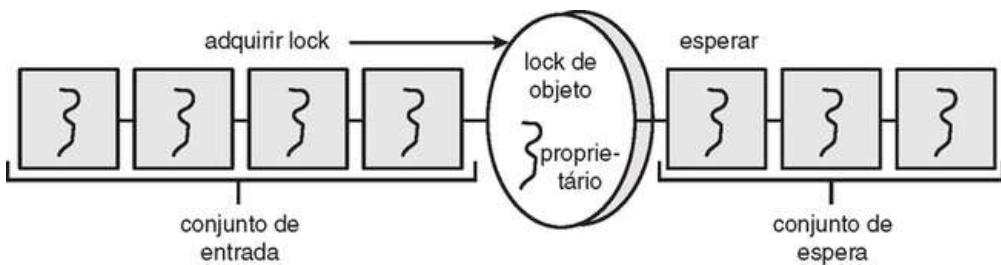


FIGURA 6.30 Conjuntos de entrada e espera

Como a thread do consumidor sinaliza que o produtor agora pode prosseguir? Em geral, quando

uma thread sai de um método `synchronized`, a ação-padrão é que a thread que sai libera apenas o lock associado ao objeto, possivelmente removendo uma thread do conjunto de entrada e dando-lhe a posse do lock. Entretanto, ao final dos métodos `insert()` e `remove()`, temos uma chamada para o método `notify()`. A chamada para `notify()`:

1. Apanha uma thread τ qualquer na lista de threads no conjunto de espera.
2. Move τ do conjunto de espera para o conjunto de entrada.
3. Define o estado de τ de bloqueado para executável.

τ agora pode competir pelo lock com as outras threads. Quando τ tiver retomado o controle do lock, ele retornará da chamada a `wait()`, onde pode verificar o valor de `count` novamente.

Em seguida, descrevemos os métodos `wait()` e `notify()` em termos do programa listado na [Figura 6.29](#). Consideraremos que o buffer está cheio e que o lock do objeto está disponível.

- O produtor chama o método `insert()`, vê que o lock está disponível e entra no método. No método, o produtor descobre que o buffer está cheio e chama `wait()`. A chamada a `wait()` libera o lock do objeto, define o estado do produtor como bloqueado e coloca o produtor na fila de espera pelo objeto.
- O consumidor por fim chama o método `remove()`, pois o lock do objeto agora está disponível. O consumidor remove um item do buffer e chama `notify()`. Observe que o consumidor ainda tem a posse do lock do objeto.
- A chamada a `notify()` remove o produtor do conjunto de espera pelo objeto, move o produtor para o conjunto de entrada e define o estado do produtor como executável.
- O consumidor sai do método `remove()`. A saída desse método libera o lock do objeto.
- O produtor tenta readquirir o lock e tem sucesso. Ele retoma a execução da chamada a `wait()`. O produtor testa o loop `while`, determina que existe espaço disponível no buffer e prossegue com o restante do método `insert()`. Se nenhuma thread estiver no conjunto de espera do objeto, a chamada a `notify()` é ignorada. Quando o produtor sai do método, ele libera o lock do objeto.

A classe `BoundedBuffer` mostrada na [Figura 6.31](#) representa a solução completa para o problema do bounded buffer usando o sincronismo Java. Essa classe pode substituir a classe `BoundedBuffer` usada na solução baseada em semáforo para esse problema, na [Seção 6.6.1](#).

```
public class BoundedBuffer<E> implements Buffer<E>
{
    private static final int BUFFER_SIZE = 5;

    private int count, in, out;
    private E[ ] buffer;

    public BoundedBuffer( ) {
        // buffer inicialmente está vazio
        count = 0;
        in = 0;
        out = 0;
        buffer = (E[ ]) new Object[BUFFER_SIZE];
    }

    public synchronized void insert(E item) {
        // Figura 6.29
    }

    public synchronized E remove( ) {
        // Figura 6.29
    }
}
```

FIGURA 6.31 Bounded buffer

6.8.2 Notificações múltiplas

Conforme descrevemos na [Seção 6.8.1.4](#), a chamada a `notify()` seleciona arbitrariamente uma thread a partir da lista de threads no conjunto de espera para um objeto. Essa técnica funciona bem quando somente uma thread está no conjunto de espera, mas considere o que pode acontecer

quando existem várias threads no conjunto de espera e mais de uma condição para esperar. É possível que uma thread cuja condição ainda não tenha sido atendida seja a que receberá a notificação.

Suponha, por exemplo, que existam cinco threads $\{T_0, T_1, T_2, T_3, T_4\}$ e uma variável compartilhada `turn` indicando de qual thread é a vez. Quando uma thread deseja realizar um trabalho, ela chama o método `doWork()` na [Figura 6.32](#). Somente uma thread cujo número combina com o valor de `turn` pode prosseguir; todas as outras threads precisam esperar sua vez.

```
/*
 * myNumber é o número da thread
 * que deseja realizar algum trabalho
 */
public synchronized void doWork(int myNumber) {
    while (turn != myNumber) {
        try {
            wait();
        }
        catch (InterruptedException e) { }
    }

    // Faz algum trabalho por um tempo

    /*
     * Terminou o trabalho. Agora, indica à
     * próxima thread esperando que seja sua vez
     * de realizar algum trabalho.
     */
    turn = (turn + 1) % 5;

    notify();
}
```

FIGURA 6.32 Método `doWork()`

Considere:

- `turn = 3.`
- T_1, T_2 e T_4 estão no conjunto de espera do objeto.
- T_3 está atualmente no método `doWork()`.

Quando a thread T_3 termina, ela define `turn` como 4 (indicando que é a vez de T_4) e chama `notify()`. A chamada a `notify()` seleciona arbitrariamente uma thread no conjunto de espera. Se T_2 receber a notificação, ela retomará a execução a partir da chamada a `wait()` e testará a condição no loop `while`. T_2 descobre que essa não é sua vez e, por isso, chama `wait()` novamente. Por fim, T_3 e T_0 chamarão `doWork()` e também chamarão o método `wait()`, pois não é a vez nem de T_3 nem de T_0 . Agora, todas as cinco threads estão bloqueadas no conjunto de espera do objeto. Assim, temos outro deadlock para tratar.

Como a chamada a `notify()` apanha uma única thread de forma aleatória no conjunto de espera, o desenvolvedor não tem controle sobre qual thread será escolhida. Felizmente, a Java fornece um mecanismo que permite a notificação a todas as threads no conjunto de espera. O método `notifyAll()` é semelhante a `notify()`, exceto que *cada* thread em espera é removida do conjunto de espera e colocada no conjunto de entrada. Se a chamada a `notify()` em `doWork()` for substituída por uma chamada a `notifyAll()`, quando T_3 terminar e definir `turn` para 4, ela chamará `notifyAll()`. Essa chamada tem o efeito de remover T_1, T_2 e T_4 do conjunto de espera. As três threads, então, competem pelo lock do objeto mais uma vez. Por fim, T_1 e T_2 chamam `wait()` e somente T_4 prossegue com o método `doWork()`.

Em suma, o método `notifyAll()` é um mecanismo que desperta todas as threads aguardando e permite que decidam entre elas quais devem ser executadas em seguida. Em geral, `notifyAll()` é uma operação mais dispendiosa do que `notify()`, pois desperta todas as threads, mas é considerada uma estratégia mais conservadora, apropriada para situações em que várias threads podem estar no conjunto de espera de um objeto.

Na próxima seção, examinamos uma solução baseada em Java para o problema de leitores-escritores, que exige o uso de `notify()` e `notifyAll()`.

6.8.3 O problema de leitores-escritores

Agora, podemos fornecer uma solução para o primeiro problema de leitores-escritores, usando o sincronismo Java. Os métodos chamados pela thread de cada leitor e escritor são definidos na classe Database da Figura 6.33, que implementa a interface ReadWriteLock mostrada na Figura 6.17. O readerCount registra o número de leitores; um valor > 0 indica que o banco de dados está sendo lido. dbWriting é uma variável booleana que indica se o banco de dados está sendo acessado por um escritor. acquireReadLock(), releaseReadLock(), acquireWriteLock() e releaseWriteLock() são declarados como synchronized para garantir a exclusão mútua para as variáveis compartilhadas.

```
public class Database implements ReadWriteLock
{
    private int readerCount;
    private boolean dbWriting;

    public Database( ) {
        readerCount = 0;
        dbWriting = false;
    }

    public synchronized void acquireReadLock( ) {
        // Figura 6.34
    }

    public synchronized void releaseReadLock( ) {
        // Figura 6.34
    }

    public synchronized void acquireWriteLock( ) {
        // Figura 6.35
    }

    public synchronized void releaseWriteLock( ) {
        // Figura 6.35
    }
}
```

FIGURA 6.33 Solução para o problema dos leitores-escritores usando o sincronismo Java

```

public synchronized void acquireReadLock( ) {
    while (dbWriting == true) {
        try {
            wait( );
        }
        catch(InterruptedException e) { }
    }

    ++readerCount;
}

public synchronized void releaseReadLock( ) {
    --readerCount;

    /**
     * O último leitor indica que
     * o banco de dados não está mais sendo lido.
     */
    if (readerCount == 0)
        notify( );
}

```

FIGURA 6.34 Métodos chamados pelos leitores

```

public synchronized void acquireWriteLock( ) {
    while (readerCount > 0 || dbWriting == true) {
        try {
            wait( );
        }
        catch(InterruptedException e) { }
    }

    /**
     * Quando não houver mais leitores ou um escritor,
     * indica que o banco de dados está sendo escrito.
     */
    dbWriting = true;
}

public synchronized void releaseWriteLock( ) {
    dbWriting = false;

    notifyAll( );
}

```

FIGURA 6.35 Métodos chamados pelos escritores

Quando um escritor deseja começar a escrever, primeiro ele verifica se o banco de dados está sendo lido ou escrito. Se o banco de dados estiver sendo lido ou escrito, o escritor entra no conjunto de espera do objeto. Caso contrário, ele define `dbWriting` como `true`. Quando um escritor termina seu trabalho, ele define `dbWriting` como `false`. Quando um leitor chama `acquireReadLock()`, ele primeiro verifica se o banco de dados está sendo escrito. Se não estiver disponível, o leitor entra no conjunto de espera do objeto; caso contrário, ele incrementa `readerCount`. O último leitor chamando `releaseReadLock()` invoca `notify()`, notificando, assim, um escritor que está esperando. Contudo, quando um escritor chama `releaseWriteLock()`, ele chama o método `notifyAll()`, em vez de `notify()`. Considere o efeito sobre os leitores. Se vários leitores desejarem ler o banco de dados enquanto

estiver sendo escrito, e o escritor chamar `notify()` depois que tiver acabado de escrever, somente um leitor receberá a notificação. Outros leitores permanecerão no conjunto de espera, embora o banco de dados esteja disponível para leitura. Chamando `notifyAll()`, um escritor de saída pode notificar a todos os leitores aguardando.

6.8.4 Sincronismo em bloco

A quantidade de tempo entre o momento em que um lock é adquirido e quando ele é liberado é definida como **escopo** do lock. A Java também permite a declaração de blocos de código como `synchronized`, pois um método `synchronized` com apenas uma pequena porcentagem do seu código manipulando dados compartilhados pode gerar um escopo muito grande. Nesse caso, pode ser melhor sincronizar apenas o bloco de código que manipula dados compartilhados do que sincronizar o método inteiro. Esse projeto resulta em um escopo de lock menor. Assim, além da declaração de métodos `synchronized`, a Java também permite o sincronismo em bloco, conforme ilustrado na [Figura 6.36](#). O acesso ao método `criticalSection()` na [Figura 6.36](#) exige a propriedade do lock do objeto `mutexLock`.

```
Object mutexLock = new Object();
...
public void someMethod() {
    nonCriticalSection();

    synchronized(mutexLock) {
        criticalSection();
    }

    remainderSection();
}
```

FIGURA 6.36 Sincronismo em bloco

Também podemos usar os métodos `wait()` e `notify()` em um bloco sincronizado. A única diferença é que precisam ser chamados com o mesmo objeto usado para sincronismo. Essa técnica é apresentada na [Figura 6.37](#).

```
Object mutexLock = new Object();
...
synchronized(mutexLock) {
    try {
        mutexLock.wait();
    }
    catch (InterruptedException ie) { }

    synchronized(mutexLock) {
        mutexLock.notify();
    }
}
```

FIGURA 6.37 Sincronismo em bloco usando `wait()` e `notify()`

6.8.5 Regras de sincronismo

A palavra-chave `synchronized` é uma construção simples, mas é importante conhecer algumas regras sobre seu comportamento.

1. Uma thread que tem a posse do lock do objeto pode entrar em outro método (ou bloco) `synchronized` para o mesmo objeto. Isso é conhecido como lock **recursivo** ou **reentrante**.
2. Uma thread pode aninhar chamadas de método `synchronized` para diferentes objetos. Assim, uma thread pode possuir simultaneamente o lock para vários objetos diferentes.
3. Se um método não for declarado como `synchronized`, ele pode ser invocado independentemente da propriedade do lock, mesmo enquanto outro método `synchronized` do mesmo objeto está sendo

executado.

4. Se o conjunto de espera de um objeto estiver vazio, uma chamada a `notify()` ou `notifyAll()` não terá efeito.

5. `wait()`, `notify()` e `notifyAll()` só podem ser invocados de métodos ou blocos sincronizados; caso contrário, será gerada uma exceção do tipo `IllegalMonitorStateException`.

Também é possível declarar métodos `static` como `synchronized`. Isso porque, junto com os locks que estão associados a instâncias de objeto, existe um único **lock de classe** associado a cada classe. Assim, para determinada classe, pode haver vários locks de objeto, um por instância de objeto. Porém, há apenas um lock de classe.

Além de usar o lock de classe para declarar métodos `static` como `synchronized`, podemos usá-lo em um bloco `synchronized` colocando “*nome da classe. class*” dentro da instrução `synchronized`. Por exemplo, se quiséssemos usar um bloco `synchronized` com o lock de classe para a classe `SomeObject`, usariamos o seguinte:

```
synchronized(SomeObject.class) {  
    /**  
     * bloco de código sincronizado  
     */  
}
```

6.8.6 Tratando de `InterruptedException`

Observe que a chamada do método `wait()` requer sua colocação em um bloco `try-catch`, pois `wait()` pode resultar em uma exceção `InterruptedException`. Lembre-se de que, como vimos no [Capítulo 4](#), o método `interrupt()` é a técnica preferida para interromper uma thread em Java. Quando `interrupt()` é chamado em uma thread, o **status de interrupção** dessa thread é marcado. Uma thread pode verificar seu status de interrupção usando o método `isInterrupted()`, que retorna `true` se seu status de interrupção estiver marcado.

O método `wait()` também verifica o status de interrupção de uma thread. Se estiver definido, `wait()` lançará uma `InterruptedException`. Isso permite a interrupção de uma thread que está bloqueada no conjunto de espera. (Observe também que quando `InterruptedException` é lançada, o status de interrupção da thread é desmarcado.) Para fins de clareza e simplicidade no código, decidimos ignorar essa exceção em nossos exemplos de código. Ou seja, todas as chamadas a `wait()` aparecem como:

```
try {  
    wait();  
}  
catch (InterruptedException ie) { /* ignorar */ }
```

No entanto, se decidíssemos tratar da `InterruptedException`, permitiríamos a interrupção de uma thread bloqueada em um conjunto de espera. Isso leva em conta aplicações mais robustas, com múltiplas threads, pois fornece um mecanismo para interromper uma thread bloqueada ao tentar adquirir um lock de exclusão mútua. Uma estratégia para lidar com isso é permitir que a `InterruptedException` se propague, ou seja, em métodos em que `wait()` é chamado, primeiro removemos os blocos `try-catch` ao chamar `wait()` e declaramos esses métodos como lançando `InterruptedException`. Assim, permitimos que a `InterruptedException` se propague a partir do método onde `wait()` está sendo chamado. No entanto, permitir que essa exceção se propague exige colocar chamadas nesses métodos dentro de blocos `try-catch` (`InterruptedException`).

6.8.7 Recursos de concorrência em Java

Antes da Java 1.5, os únicos recursos de concorrência disponíveis em Java eram os comandos `synchronized`, `wait()` e `notify()`, que são baseados em locks isolados para cada objeto. A Java 1.5 introduziu uma API rica, consistindo em vários recursos de concorrência, incluindo diversos mecanismos para sincronizar threads concorrentes. Nesta seção, abordamos (1) locks reentrantes, (2) semáforos e (3) variáveis de condição disponíveis nos pacotes `java.util.concurrent` e `java.util.concurrent.locks`. Os leitores interessados nos recursos adicionais desses pacotes poderão

consultar a API da Java.

6.8.7.1 Locks reentrantes

Talvez o mecanismo de lock mais simples disponível na API seja ReentrantLock. De muitas maneiras, um ReentrantLock atua como a instrução `synchronized` descrita na [Seção 6.8.1.2](#): um ReentrantLock pertence a uma única thread e é usado para fornecer acesso mutuamente exclusivo a um recurso compartilhado. Contudo, o ReentrantLock fornece vários recursos adicionais, como a definição de um parâmetro de *fairness*, que favorece a concessão do lock à thread esperando por mais tempo. (Lembre-se, da [Seção 6.8.1.2](#): a especificação para a JVM não indica que as threads no conjunto de espera para um lock de objeto devem ser ordenadas de alguma maneira específica.)

Uma thread adquire um lock ReentrantLock chamando seu método `lock()`. Se o lock estiver disponível – ou se a thread chamando `lock()` já o possuir, motivo pelo qual ele se chama *reentrante* –, `lock()` atribuirá à thread que chama a propriedade do lock e retorna o controle. Se o lock estiver indisponível, a thread que chama será bloqueada até que, por fim, receba o lock quando seu proprietário invocar `unlock()`. ReentrantLock implementa a interface Lock; seu uso é:

```
Lock key = new ReentrantLock( );
key.lock( );
try {
    // seção crítica
}
finally {
    key.unlock( );
}
```

O idioma de programação do uso de `try` e `finally` requer alguma explicação. Se o lock for adquirido por meio do método `lock()`, é importante que ele seja liberado de modo semelhante. Delimitando `unlock()` em uma cláusula `finally`, garantimos que o lock será liberado quando a seção crítica terminar ou se ocorrer uma exceção dentro do bloco `try`. Observe que não colocamos a chamada para `lock()` dentro da cláusula `try`, pois `lock()` não lança quaisquer exceções verificadas. Considere o que acontece se colocarmos `lock()` dentro da cláusula `try` e uma exceção não verificada ocorrer quando `lock()` for chamado (como `OutOfMemoryError`). A cláusula `finally` dispara a chamada para `unlock()`, que então lança a `IllegalMonitorStateException` não verificada, pois o lock nunca foi adquirido. Essa `IllegalMonitorStateException` substitui a exceção não verificada que ocorreu quando `lock()` foi chamado, obscurecendo assim o motivo pelo qual o programa falhou inicialmente.

6.8.7.2 Semáforos

A API Java 5 fornece um semáforo de contagem, conforme descrito na [Seção 6.5](#). O construtor para o semáforo aparece como

```
Semaphore(int value);
```

onde `value` especifica o valor inicial do semáforo (um valor negativo é permitido). O método `acquire()` lança uma `InterruptedException` se a thread que adquire for interrompida ([Seção 6.8.6](#)). O exemplo a seguir ilustra o uso de um semáforo para exclusão mútua:

```

Semaphore sem = new Semaphore(1);

try {
    sem.acquire();
    // seção crítica
}
catch (InterruptedException ie) { }
finally {
    sem.release();
}

```

Observe que colocamos a chamada a `release()` na cláusula `finally` para garantir que o semáforo seja liberado.

6.8.7.3 Variáveis de condição

O último utilitário do qual falaremos na API Java é a variável de condição. Assim como o `ReentrantLock` ([Seção 6.8.7.1](#)) é semelhante à instrução `synchronized` da Java, as variáveis de condição fornecem funcionalidade semelhante aos métodos `wait()`, `notify()` e `notifyAll()`. Portanto, para fornecer exclusão mútua a ambos, uma variável de condição precisa estar associada a um lock reentrante.

Produzimos uma variável de condição primeiro criando um `ReentrantLock` e chamando seu método `newCondition()`, que retorna um objeto `Condition` representando a variável de condição para o `ReentrantLock` associado. Isso é ilustrado nas seguintes instruções:

```

Lock key = new ReentrantLock();
Condition condVar = key.newCondition();

```

Quando a variável de condição é obtida, podemos chamar seus métodos `await()` e `signal()`, que funcionam como os comandos `wait()` e `signal()` descritos na [Seção 6.7](#).

Como dissemos, os locks reentrantes e as variáveis de condição na API Java funcionam de modo semelhante às instruções `synchronized`, `wait()` e `notify()`. Contudo, uma vantagem no uso dos recursos disponíveis na API é que eles normalmente fornecem mais flexibilidade e controle do que seus correspondentes `synchronized/wait() /notify()`. Outra distinção refere-se ao mecanismo de lock da Java, que é baseado em cada objeto tendo seu próprio lock. De várias maneiras, esse lock atua como um monitor. Cada objeto Java, assim, possui um monitor associado, e uma thread pode adquirir o monitor de um objeto entrando com um método ou bloco `synchronized`.

Vamos examinar essa distinção mais de perto. Lembre-se de que, com monitores descritos na [Seção 6.7](#), as operações `wait()` e `signal()` podem ser aplicadas a variáveis de condição *nomeadas*, permitindo que uma thread espere por uma condição específica ou que seja notificada quando uma condição específica for atendida. No nível de linguagem, a Java não fornece suporte para variáveis de condição nomeadas. Cada monitor Java está associado a apenas uma variável de condição não nomeada, e as operações `wait()`, `notify()` e `notifyAll()` se aplicam apenas a essa única variável de condição. Quando uma thread Java é acordada por meio de `notify()` ou `notifyAll()`, ela não recebe informação sobre por que foi acordada. Fica a critério da thread reativada verificar por si só se a condição pela qual estava esperando foi atendida. O método `doWork()`, mostrado na [Figura 6.32](#), destaca essa questão; `notifyAll()` precisa ser invocado para acordar todas as threads esperando, e - uma vez acordada - cada thread precisa verificar por si só se a condição que estava esperando foi atendida (ou seja, se é a vez dessa thread).

Ilustramos ainda mais essa distinção reescrevendo o método `doWork()` da [Figura 6.32](#) usando variáveis de condição. Primeiro, criamos um `ReentrantLock` e cinco variáveis de condição (representando as condições que as threads estão esperando) para sinalizar a thread cuja vez é a próxima. Isso pode ser visto a seguir:

```

Lock lock = new ReentrantLock( );
Condition[ ] condVars = new Condition[5];

for (int i = 0; i < 5; i++)
    condVars[i] = lock.newCondition( );

```

O método `doWork()` modificado é mostrado na [Figura 6.38](#). Observe que `doWork()` não é mais declarado como `synchronized`, pois o `ReentrantLock` provê exclusão mútua. Quando uma thread chama `await()` na variável de condição, ela libera o `ReentrantLock` associado, permitindo que outra thread adquira o lock de exclusão mútua. De modo semelhante, quando `signal()` é chamado, somente a variável de condição é sinalizada; o lock é liberado por meio da chamada a `unlock()`.

```

/**
 * myNumber é o número da thread
 * que deseja realizar algum trabalho
 */
public void doWork(int myNumber) {
    lock.lock( );

    try {
        /**
         * Se não for minha vez, então espero
         * até que eu seja sinalizado
         */
        if (myNumber != turn)
            condVars[myNumber].await( );

        // Realiza algum trabalho por um
        tempo...

        /**
         * Acabou de trabalhar. Agora indica à
         * próxima thread esperando que seja
         * a sua
         * vez de fazer algum trabalho.
         */
        turn = (turn + 1) % 5;
        condVars[turn].signal( );
    }
    catch (InterruptedException ie) { }
    finally {
        lock.unlock( );
    }
}

```

FIGURA 6.38 Método `doWork()` com variáveis de condição

6.9 Exemplos de sincronismo

Em seguida, descrevemos os mecanismos de sincronismo fornecidos pelos sistemas operacionais Solaris, Windows XP e Linux, assim como a API Pthreads. Escolhemos esses três sistemas porque eles fornecem bons exemplos de diferentes técnicas para sincronismo do kernel, e incluímos a API Pthreads porque é bastante utilizada para criação e sincronismo de threads por desenvolvedores nos sistemas UNIX e Linux. Como se verá nesta seção, os métodos de sincronismo disponíveis nesses sistemas variam de maneiras sutis e significativas.

6.9.1 Sincronismo no Solaris

Para controlar o acesso às seções críticas, o Solaris fornece mutexes adaptativos, variáveis de condição, semáforos, locks de leitor-escritor e turnstiles. O Solaris implementa semáforos e variáveis de condição conforme são apresentados nas Seções 6.5 e 6.7. Nesta seção, descrevemos os mutexes adaptativos, locks de leitura-escrita e turnstiles.

Um **mutex adaptativo** protege o acesso a cada item de dados crítico. Em um sistema com mutiprocessadores, um mutex adaptativo começa como um semáforo-padrão implementado como um spinlock. Se os dados estiverem bloqueados e, portanto, já estiverem em uso, o mutex adaptativo tem duas opções. Se o lock for mantido por uma thread em execução em outra CPU, a thread “girará” enquanto espera que o lock esteja disponível, pois a thread que mantém o lock provavelmente logo terminará. Se a thread que mantém o lock não estiver no estado de execução, a thread é bloqueada, indo dormir até ser despertada pela liberação do lock. Ela é colocada para dormir de modo a evitar girar enquanto estiver em espera, visto que o lock não será liberado logo. Um lock mantido por uma thread dormindo provavelmente estará nessa categoria. Em um sistema de processador único, a thread que mantém o lock nunca estará em execução se o lock estiver sendo testado por outra thread, pois somente uma thread pode ser executada em determinado momento. Portanto, nesse tipo de sistema, as threads sempre dormem, em vez de girar, se encontrarem um lock.

O Solaris usa o método do mutex adaptativo para proteger apenas os dados acessados por segmentos de código curtos, ou seja, um mutex é usado se um lock for mantido por menos do que algumas centenas de instruções. Se o segmento de código for maior do que isso, o método de espera pelo giro será bastante ineficaz. Para esses segmentos de código maiores, as variáveis de condição e os semáforos são utilizados. Se a thread já mantém o lock desejado, ela emite uma espera e dorme. Quando uma thread libera o lock, ela emite um sinal para a próxima thread dormindo na fila. O custo extra de colocar uma thread para dormir e despertá-la, com as trocas de contexto associadas, é menor do que o custo de desperdiçar várias centenas de instruções esperando em um spinlock.

Os locks de leitores-escritores são usados para proteger dados acessados com frequência, mas normalmente são acessados de uma maneira apenas de leitura. Nessas circunstâncias, os locks leitores-escritores são mais eficientes do que os semáforos, pois várias threads podem ler dados concorrentemente, enquanto os semáforos sempre mantêm o acesso aos dados em série. Os locks de leitores-escritores são relativamente dispendiosos de implementar, de modo que são usados apenas em longas seções de código.

O Solaris utiliza turnstiles para ordenar a lista de threads aguardando para adquirir um mutex adaptativo ou um lock de leitor-escritor. Um **turnstile** é uma estrutura de fila contendo threads bloqueadas em um lock. Por exemplo, se uma thread possui o lock para um objeto sincronizado, todas as outras threads tentando obter a posse do lock serão bloqueadas e entrará no turnstile para esse lock. Quando o lock é liberado, o kernel seleciona uma thread do turnstile como próximo proprietário do lock. Cada objeto sincronizado com pelo menos uma thread bloqueada no lock de um objeto exige um turnstile separado. Entretanto, em vez de associar um turnstile a cada objeto sincronizado, o Solaris dá a cada thread do kernel seu próprio turnstile. Como uma thread só pode ser bloqueada em um objeto de cada vez, isso é mais eficiente do que ter um turnstile por objeto.

O turnstile para a primeira thread a bloquear em um objeto sincronizado torna-se o turnstile para o próprio objeto. As threads subsequentes bloqueando em um lock serão acrescentadas a esse turnstile. Quando a thread inicial por fim liberar o lock, ela receberá um novo turnstile a partir de uma lista de turnstiles livres, mantidos pelo kernel. Para impedir uma **inversão de prioridade**, os turnstiles são organizados segundo um **protocolo de herança de prioridade**. Se uma thread de menor prioridade mantiver um lock no qual uma thread de maior prioridade está bloqueada, a thread com a menor prioridade herdará temporariamente a prioridade da de maior prioridade. Ao liberar o lock, a thread retornará à sua prioridade original.

Observe que os mecanismos de lock utilizados pelo kernel também são implementados para as threads no nível do usuário, de modo que os mesmos tipos de locks estão disponíveis dentro e fora do kernel. Uma diferença crucial na implementação é o protocolo de herança de prioridade. As rotinas de lock do kernel aderem aos métodos de herança de prioridade do kernel usados pelo escalonador, conforme descritos na [Seção 19.4](#); os mecanismos de lock da thread no nível do usuário não fornecem essa funcionalidade.

Para otimizar o desempenho no Solaris, os desenvolvedores refinaram e ajustaram os métodos de lock. Como os locks são usados com frequência e para funções cruciais do kernel, o ajuste de sua implementação e uso pode produzir ganhos tremendos no desempenho.

6.9.2 Sincronismo no Windows XP

O sistema operacional Windows XP é um kernel multithread que também provê suporte para aplicações de tempo real e multiprocessadores. Quando o kernel do Windows XP acessa um recurso global em um sistema de processador único, ele mascara temporariamente as interrupções para todos os tratadores de interrupção que também podem acessar o recurso global. Em um sistema de multiprocessadores, o Windows XP protege o acesso aos recursos globais usando spinlocks. Assim como no Solaris, o kernel utiliza spinlocks somente para proteger segmentos de código pequenos. Além do mais, por motivos de eficiência, o kernel garante que uma thread nunca será preemptada enquanto mantém um spinlock.

Para o sincronismo da thread fora do kernel, o Windows XP provê **objetos despachantes (dispatcher objects)**. Usando um objeto despachante, as threads são sincronizadas de acordo com diversos mecanismos diferentes, incluindo mutexes, semáforos e eventos. O sistema protege os dados compartilhados exigindo que uma thread obtenha a posse de um mutex para acessar os dados e o libere quando terminar. Os semáforos comportam-se como descrito na [Seção 6.5](#). Os **eventos** são semelhantes às variáveis de condição, ou seja, eles podem notificar uma thread aguardando quando ocorrer uma condição desejada. Por fim, os temporizadores são usados para notificar a uma thread (ou mais) de que uma quantidade de tempo determinada expirou.

Os objetos despachantes podem estar em um estado sinalizado ou não sinalizado. O **estado sinalizado** indica que um objeto está disponível e uma thread não será bloqueada quando adquirir o objeto. Um **estado não sinalizado** indica que um objeto não está disponível e uma thread será bloqueada quando tentar adquirir o objeto. Ilustramos as transições de estado de um objeto despachante de lock mutex na [Figura 6.39](#).

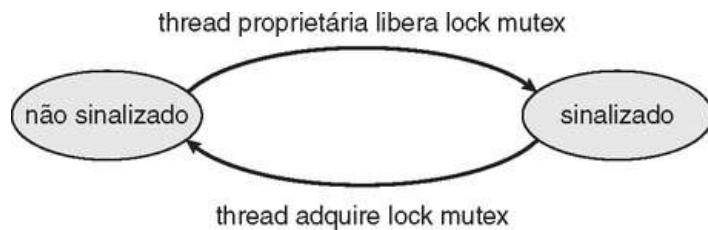


FIGURA 6.39 Objeto despachante mutex

Existe um relacionamento entre o estado de um objeto despachante e o estado de uma thread. Quando uma thread for bloqueada em um objeto despachante não sinalizado, seu estado mudará de pronto (ready) para esperando (waiting), e a thread será colocada na fila de espera para esse objeto. Quando o estado para o objeto despachante passar para sinalizado, o kernel verificará se quaisquer threads estão esperando no objeto. Se houver alguma, o kernel moverá uma thread - ou, possivelmente, mais threads - do estado de espera para o estado pronto, onde poderá retomar a execução. A quantidade de threads que o kernel seleciona na fila de esperando (waiting queue) depende do tipo de objeto despachante pelo qual está esperando. O kernel selecionará apenas uma thread da fila de espera por um mutex, pois um objeto mutex pode ser “possuído” somente por uma única thread. Para um objeto de evento, o kernel selecionará todas as threads que estão esperando pelo evento.

Podemos usar o mutex como exemplo ilustrativo de objetos despachantes e estados da thread. Se uma thread tentar obter um objeto despachante mutex que esteja em um estado não sinalizado, essa thread será suspensa e colocada em uma fila de espera para o objeto mutex. Quando o mutex passa para o estado sinalizado (pois outra thread liberou o mutex), a thread esperando na frente da fila pelo mutex será movida do estado de espera para o estado pronto e obterá o mutex.

Ao final deste capítulo, fornecemos um projeto de programação que utiliza locks mutex e semáforos na API Win32.

6.9.3 Sincronismo no Linux

Antes da versão 2.6, o Linux era um kernel não preemptivo, o que significa que um processo executando nesse modo não pode ser interrompido, mesmo que um processo com prioridade mais alta fique disponível para execução. Agora, porém, o kernel do Linux é totalmente preemptivo, de modo que uma tarefa pode ser interrompida quando estiver executando no kernel.

O kernel do Linux provê spinlocks e semáforos (além de versões leitor-escritor desses dois locks) para lock no kernel. Em máquinas SMP, o mecanismo fundamental de lock é um spinlock, e o kernel

é projetado de modo que o spinlock seja mantido apenas por pequenas durações. Em máquinas com único processador, os spinlocks são impróprios para uso, e são substituídos pela ativação e desativação da preempção do kernel. Ou seja, em máquinas com único processador, em vez de manter um spinlock, o kernel desativa a preempção do kernel; e, em vez de liberar o spinlock, ele permite a preempção do kernel. Isso é resumido na tabela a seguir:

único processador	múltiplos processadores
Desativa preempção do kernel	Adquire spinlock
Ativa preempção do kernel	Libera spinlock

O Linux utiliza uma técnica interessante para desativar e ativar a preempção do kernel. Ele provê duas chamadas de sistema simples - `preempt_disable()` e `preempt_enable()` - para essas tarefas. Além disso, o kernel não pode ser interrompido se uma tarefa no modo kernel estiver mantendo um lock. Para impor essa regra, cada tarefa no sistema possui uma estrutura `thread-info`, contendo um contador, `preempt_count`, para indicar o número de locks mantidos pela tarefa. Quando um lock é adquirido, `preempt_count` é incrementado. Depois ele é decrementado quando um lock é liberado. Se o valor de `preempt_count` para a tarefa atualmente em execução for maior que zero, não é seguro interromper o kernel, pois essa tarefa atualmente mantém um lock. Se o contador for zero, o kernel pode seguramente ser interrompido (supondo que não existam chamadas pendentes para `preempt_disable()`).

Os spinlocks - junto com a ativação e a desativação da preempção do kernel - são usados no kernel apenas quando um lock (ou uma desativação da preempção do kernel) é mantido por curta duração. Quando um lock tiver que ser mantido por um período maior, o uso de semáforos é mais apropriado.

6.9.4 Sincronismo no Pthreads

A API Pthreads provê mutex, variáveis de condição e locks de leitores-escritores para sincronismo de thread. Essa API está disponível para programadores e não faz parte de qualquer kernel em particular. Os mutexes são a técnica de sincronismo fundamental utilizada com Pthreads. Um mutex é usado para proteger seções críticas do código, ou seja, uma thread adquire o lock antes de entrar em uma seção crítica e o libera ao sair da seção crítica. As variáveis de condição no Pthreads se comportam em grande parte conforme descrevemos na [Seção 6.7](#). Os locks de leitores-escritores comportam-se de maneira similar ao mecanismo de lock descrito na [Seção 6.6.2](#). Muitos sistemas que implementam Pthreads também fornecem semáforos, embora não façam parte do padrão Pthreads e, em vez disso, pertençam ao padrão POSIX SEM. Outras extensões à API Pthreads incluem spinlocks, mas nem todas são consideradas portáveis de uma implementação para outra.

6.10 Transações atômicas

A exclusão mútua de seções críticas garante que as seções críticas sejam executadas atomicamente - ou seja, como uma unidade que não pode ser interrompida. Se duas seções críticas forem executadas ao mesmo tempo, o resultado será equivalente à sua execução sequencial em alguma ordem desconhecida. Embora essa propriedade seja útil em vários domínios de aplicação, em muitos casos gostaríamos de ter certeza de que uma seção crítica forma uma única unidade de trabalho lógica, realizada em sua totalidade ou não realizada de forma alguma. Um exemplo é a transferência de fundos, em que uma conta é debitada e outra é creditada. Logicamente, é essencial para a coerência dos dados garantir que as operações de crédito e débito ocorram ou que nenhuma delas ocorra.

A coerência dos dados, juntamente com o armazenamento e a recuperação de dados, é um problema com frequência tratado pelos **sistemas de banco de dados**. Recentemente, tem havido um surto crescente de interesse no uso de técnicas de sistemas de banco de dados nos sistemas operacionais. Os sistemas operacionais podem ser vistos como manipuladores de dados; dessa forma, podem se beneficiar com as técnicas avançadas e os modelos disponíveis pela pesquisa do banco de dados. Por exemplo, muitas das técnicas ocasionais usadas nos sistemas operacionais para gerenciar arquivos poderiam ser mais flexíveis e poderosas se métodos de banco de dados mais formais fossem usados em seu lugar. Nas [Seções 6.10.2 a 6.10.4](#) descrevemos algumas dessas técnicas de banco de dados e explicamos como podem ser usadas pelos sistemas operacionais. Primeiramente, lidaremos com a questão geral da atomicidade da transação. É essa propriedade que as técnicas de bancos de dados devem abordar.

6.10.1 Modelo do sistema

Uma coleção de instruções (ou operações) que realiza uma única função lógica é chamada de **transação**. Um aspecto importante no processamento de transações é a preservação da atomicidade, apesar da possibilidade de falhas dentro do sistema computadorizado.

Podemos considerar a transação uma unidade de programa que acessa e talvez atualize diversos itens de dados que podem residir no disco dentro de alguns arquivos. Pelo nosso ponto de vista, tal transação é uma sequência de operações de leitura e escrita, terminada por uma operação **commit** ou uma operação **abort**. Uma operação **commit** significa que a transação terminou sua execução com sucesso, enquanto uma operação **abort** significa que a transação terminou sua execução normal por causa de um erro lógico ou por causa de uma falha no sistema. Se uma transação terminou e completou sua execução com sucesso, ela está **confirmada**; caso contrário, ela é **cancelada**.

Como uma transação cancelada já pode ter modificado os dados acessados, o estado desses dados pode não ser igual ao estado que haveria se a transação tivesse sido executada atomicamente. Porém, se a atomicidade tiver que ser garantida, uma transação cancelada não poderá ter efeito sobre o estado dos dados. Assim, o estado dos dados acessados por uma transação abortada precisa ser restaurado ao que existia imediatamente antes de a transação iniciar sua execução. Dizemos que tal transação foi **revertida**. Faz parte da responsabilidade do sistema garantir essa propriedade.

Para determinar como o sistema deve garantir a atomicidade, precisamos primeiro identificar as propriedades dos dispositivos usados para armazenar os diversos dados acessados pelas transações. Diversos tipos de meios de armazenamento são distinguidos por sua velocidade relativa, capacidade e recuperação de falhas.

■ **Armazenamento volátil.** As informações que residem no armazenamento volátil normalmente não sobrevivem a falhas no sistema. Alguns exemplos desse tipo de armazenamento são memória principal e cache. O acesso ao armazenamento volátil é bem rápido, tanto por causa da velocidade de acesso à própria memória quanto porque é possível acessar diretamente qualquer item de dados no armazenamento volátil.

■ **Armazenamento não volátil.** As informações que residem no armazenamento não volátil normalmente sobrevivem a falhas no sistema. Alguns exemplos de meios de armazenamento desse tipo são discos e fitas magnéticas. Os discos são mais confiáveis do que a memória principal, mas são menos confiáveis do que as fitas magnéticas. Entretanto, tanto discos quanto fitas estão sujeitos a falhas, que podem resultar em perda de informações. Hoje em dia, o armazenamento não volátil é mais lento do que o armazenamento volátil em várias ordens de grandeza, pois os dispositivos de disco e fita são eletromecânicos e exigem a movimentação física para o acesso aos dados.

MEMÓRIA TRANSACIONAL

Com o surgimento de sistemas multicore houve uma pressão cada vez maior para que se desenvolvam aplicações multithreaded que tirem proveito dos diversos núcleos de processamento. Contudo, as aplicações multithreaded apresentam um risco maior de condições de corrida e deadlocks. Tradicionalmente, técnicas como locks, semáforos e monitores têm sido

usadas para tratar dessas questões. Porém, a **memória transacional** oferece uma estratégia alternativa para o desenvolvimento de aplicações concorrentes seguras para thread.

Uma **transação de memória** é uma sequência de operações de leitura-escrita da memória que são indivisíveis (ou atômicas). Se todas as operações em uma transação forem concluídas, a transação de memória é confirmada; caso contrário, as operações precisam ser abortadas e revertidas. Os benefícios da memória transacional podem ser obtidos por meio de recursos acrescentados a uma linguagem de programação.

Considere um exemplo. Suponha que tenhamos uma função `update()` que modifica dados compartilhados. Tradicionalmente, essa função seria escrita usando locks, da seguinte maneira:

```
update ( ) {
    acquire( );
    /* modifica dados compartilhados */
    release( );
}
```

Entretanto, o uso de mecanismos de sincronismo, como locks e semáforos, envolve muitos problemas em potencial, incluindo deadlocks. Além do mais, à medida que a quantidade de threads aumenta, o bloqueio tradicional não acompanha muito bem esse aumento.

Como uma alternativa aos métodos tradicionais, novos recursos que tiram proveito da memória transacional podem ser acrescentados a uma linguagem de programação. Em nosso exemplo, suponha que acrescentemos a construção `atomic{S}`, que garante que as operações em `S` serão executadas como uma transação. Isso nos permite reescrever o método `update()` da seguinte maneira:

```
update ( ) {
    atomic;
    /* modifica dados compartilhados */
}
```

A vantagem do uso de um mecanismo desse tipo, em vez de locks, é que o sistema de memória transacional – não o desenvolvedor – fica responsável por garantir a atomicidade. Além disso, o sistema pode identificar quais instruções nos blocos indivisíveis podem ser executadas simultaneamente, como o acesso de leitura concorrente a uma variável compartilhada. Naturalmente, é possível que um programador identifique essas situações e use locks de leitura-escrita, mas a tarefa torna-se muito mais difícil quando a quantidade de threads dentro de uma aplicação aumenta.

A memória transacional pode ser implementada tanto no software quanto no hardware. A memória transacional por software (STM), como o nome indica, implementa a memória transacional exclusivamente no software – sem a necessidade de qualquer hardware especial. A STM funciona inserindo código de instrumentação dentro dos blocos de transação. O código é inserido por um compilador e gerencia cada transação examinando onde as instruções podem ser executadas simultaneamente e onde é necessário haver um bloqueio específico de baixo nível. A memória transacional por hardware (HTM pequena) utiliza hierarquias de cache no hardware e protocolos de coerência de cache para gerenciar e resolver conflitos envolvendo dados compartilhados que residem nas memórias cache de processadores separados. A HTM não exige instrumentação de código especial e, portanto, possui menos trabalho adicional do que a STM. No entanto, a HTM exige que as hierarquias de cache e os protocolos de coerência de cache existentes sejam modificados para dar suporte à memória transacional.

A memória transacional já existe há muitos anos, mas sem uma implementação generalizada. Porém, o crescimento de sistemas multicore e a ênfase associada à programação concorrente estimularam uma pesquisa significativa nessa área por parte de acadêmicos e fornecedores de hardware, inclusive Intel e Sun Microsystems.

■ **Armazenamento estável.** As informações residindo no armazenamento estável *nunca* são perdidas (*nunca* deve ser considerado com certo cuidado, pois teoricamente esses fatos absolutos não podem ser garantidos). Para implementar uma aproximação de tal armazenamento, precisamos replicar informações em diversos caches de armazenamento não volátil (normalmente, discos) com modos de falha independentes e atualizar as informações de maneira

controlada ([Seção 12.8](#)).

Aqui, estamos preocupados apenas em garantir a atomicidade de transações em um ambiente em que falhas resultam em perda de informações no armazenamento volátil.

6.10.2 Recuperação baseada em log

Uma maneira de garantir a atomicidade é registrar, no armazenamento estável, informações descrevendo todas as modificações feitas pela transação nos diversos dados acessados. O método mais utilizado para conseguir essa forma de registro é o **log de escrita antecipada**. O sistema mantém, no armazenamento estável, uma estrutura de dados chamada **log**. Cada registrador do log descreve uma única operação de uma transação de escrita e possui os seguintes campos:

- **Nome da transação.** O nome exclusivo da transação que realizou a operação `write`.
- **Nome do item de dados.** O nome exclusivo do item de dados escrito.
- **Valor antigo.** O valor do item de dados antes da operação `write`.
- **Valor novo.** O valor que o item de dados terá após a escrita.

Existem outros registros de log especiais para registrar eventos significativos durante o processamento da transação, como o início de uma transação e a confirmação ou cancelamento de uma transação.

Antes de uma transação T_i iniciar sua execução, o registrador $\langle T_i \text{ starts} \rangle$ é escrito no log. Durante sua execução, qualquer operação `write` por T_i é *precedida* pela escrita do novo registro apropriado no log. Quando T_i é confirmada, o registrador $\langle T_i \text{ commits} \rangle$ é escrito no log.

Como a informação no log é usada na reconstrução do estado dos itens de dados acessados pelas diversas transações, não podemos permitir que a atualização real de um item de dados ocorra antes de o registrador de log correspondente ser gravado no armazenamento estável. Portanto, exigimos que, antes da execução de uma operação `write` (X), os registros de log correspondentes a X sejam gravados no armazenamento estável.

Observe a penalidade no desempenho inerente a esse sistema. Duas escritas físicas são necessárias para cada escrita lógica solicitada. Além disso, mais armazenamento é necessário: para os próprios dados e para o log que está registrando as mudanças. Contudo, nos casos em que os dados são extremamente importantes e a recuperação rápida de falhas é necessária, a funcionalidade compensa o preço a ser pago.

Usando o log, o sistema pode lidar com qualquer falha que não resulte na perda de informações no armazenamento não volátil. O algoritmo de recuperação utiliza dois procedimentos:

- `undo(T_i)`, que restaura o valor de todos os dados atualizados pela transação T_i aos valores antigos.
- `redo(T_i)`, que define o valor de todos os dados atualizados pela transação T_i como os novos valores.

O conjunto de dados atualizados por T_i e os valores antigos e novos adequados podem ser encontrados no log.

Observe que as operações `undo` e `redo` precisam ser *coerentes* (ou seja, várias execuções de uma operação precisam ter o mesmo resultado de uma execução), para garantir o comportamento correto, mesmo que ocorra uma falha durante o processo de recuperação.

Se uma transação T_i for cancelada, então podemos restaurar o estado dos dados atualizados executando `undo(T_i)`. Se ocorrer uma falha no sistema, restauramos o estado de todos os dados atualizados, consultando o log para determinar quais transações precisam ser refeitas e quais precisam ser desfeitas. Essa classificação de transações é realizada da seguinte maneira:

- A transação T_i precisa ser desfeita se o log tiver o registrador $\langle T_i \text{ starts} \rangle$, mas não tiver o registro $\langle T_i \text{ commits} \rangle$.
- A transação T_i precisa ser refeita se o log tiver os registros $\langle T_i \text{ starts} \rangle$ e $\langle T_i \text{ commits} \rangle$.

6.10.3 Pontos de verificação

Quando ocorre uma falha no sistema, precisamos consultar o log para determinar as transações que precisam ser refeitas e as que precisam ser desfeitas. A princípio, precisamos pesquisar o log inteiro para determinar isso. Existem duas grandes desvantagens nessa técnica:

1. O processo de pesquisa é demorado.
2. A maioria das transações que, de acordo com nosso algoritmo, precisam ser refeitas já atualizou realmente os dados que o log diz que precisam de modificação. Embora refazer as modificações aos dados não cause prejuízo (devido à coerência), a recuperação levará mais tempo.

Para reduzir esses tipos de custo adicional, introduzimos o conceito de **pontos de verificação** (checkpoints). Durante a execução, o sistema mantém o log de escrita antecipada. Além disso, o sistema periodicamente cria pontos de verificação que exigem a seguinte sequência de ações:

1. Enviar todos os registros de log atualmente residindo no armazenamento volátil (em geral, a memória principal) para o armazenamento estável.
2. Enviar todos os dados modificados residindo no armazenamento volátil para o armazenamento estável.

3. Enviar um registro de log <checkpoint> para o armazenamento estável.

A presença de um registro <checkpoint> no log permite ao sistema agilizar seu procedimento de recuperação. Considere uma transação T_i que tenha sido confirmada antes do ponto de verificação (checkpoint). O registro < T_i commits> aparece no log antes do registro <checkpoint>. Quaisquer modificações feitas em T_i precisam ter sido gravadas no armazenamento estável antes do checkpoint ou como parte do próprio checkpoint. Assim, no momento da verificação, não é preciso realizar uma operação redo sobre T_i .

Essa observação permite refinar nosso algoritmo de recuperação anterior. Depois de haver uma falha, a rotina de recuperação examinará o log para determinar a transação T_i mais recente, que começou a executar antes de ocorrer o checkpoint mais recente. Ela encontrará essa transação pesquisando o log de trás para a frente, até encontrar o primeiro registro <checkpoint>, e depois procurará o registro < T_i starts> subsequente.

Quando a transação T_i tiver sido identificada, as operações redo e undo só precisarão ser aplicadas à transação T_i e a todas as transações T_j que iniciaram sua execução após a transação T_i . Vamos indicar essas transações pelo conjunto T . O restante do log pode ser ignorado. As operações de recuperação exigidas são as seguintes:

- Para todas as transações T_k em T , tal que o registro < T_k commits> aparece no log, executar $\text{redo}(T_k)$.
- Para todas as transações T_k em T que não possuem um registro < T_k commits> no log, executar $\text{undo}(T_k)$.

6.10.4 Transações atômicas concorrentes

Estivemos considerando um ambiente no qual apenas uma transação pode ser executada por vez. Agora voltamos ao caso no qual transações múltiplas são ativadas simultaneamente. Como cada transação é atômica, a execução concorrente de transações precisa ser a mesma que executá-las em série, em alguma ordem qualquer. Uma maneira de manter essa propriedade, chamada **serialização**, é pela simples execução de cada transação dentro de uma seção crítica, ou seja, todas as transações compartilham um *mutex* de semáforo comum, inicializado em 1. Quando uma transação começa a executar, sua primeira ação é executar wait(mutex) . Depois de a transação ser confirmada ou cancelada, ela executa signal(mutex) .

Embora esse esquema garanta a atomicidade de todas as transações em execução simultânea, ele é muito restritivo. Conforme veremos, em muitos casos podemos permitir que as transações sobreponham sua execução enquanto mantêm a serialização. Diversos algoritmos de **controle de concorrência** diferentes garantem a serialização. Eles são descritos a seguir.

6.10.4.1 Serialização

Considere um sistema com dois itens de dados A e B lidos e escritos por duas transações T_0 e T_1 . Suponha que essas transações sejam executadas atomicamente na ordem T_0 seguida por T_1 . Essa sequência de execução, chamada **roteiro** (schedule), está representada na [Figura 6.40](#). Nesse roteiro, indicado como roteiro 1, a sequência de etapas de instrução está em ordem cronológica de cima para baixo, com as interfaces de T_0 aparecendo na coluna da esquerda e as instruções de T_1 aparecendo na coluna da direita.

T_0	T_1
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

FIGURA 6.40 Roteiro 1: Um roteiro serial em que T_0 é seguida por T_1

Um roteiro em que cada transação é executada atomicamente é denominado **roteiro serial**. Um roteiro serial consiste em uma sequência de instruções de várias transações, na qual as instruções pertencentes a uma transação específica aparecem juntas. Assim, para um conjunto de n transações, existem $n!$ roteiros seriais válidos diferentes. Cada roteiro serial é correto, pois é equivalente à

execução atômica das transações participantes, em alguma ordem arbitrária.

Se permitirmos que as duas transações sobreponham sua execução, então o roteiro resultante não será mais serial. Um **roteiro não serial** não implica necessariamente uma execução incorreta (ou seja, uma execução que não é equivalente a uma representada por um roteiro serial). Para ver que isso acontece, precisamos definir a noção de **operações conflitantes**.

Considere um roteiro R em que existam duas operações consecutivas, O_i e O_j , das transações T_i e T_j , respectivamente. Dizemos que O_i e O_j são *conflitantes* se acessarem o mesmo item de dados e pelo menos uma dessas operações for uma operação `write`. Para ilustrar o conceito de operações conflitantes, consideraremos o roteiro não serial 2 da [Figura 6.41](#). A operação `write(A)` de T_0 entra em conflito com a operação `read(A)` de T_1 . Todavia, a operação `write(A)` de T_1 não entra em conflito com a operação `read(B)` de T_0 , pois as duas operações acessam itens de dados diferentes.

T_0	T_1
<code>read(A)</code>	
<code>write(A)</code>	
	<code>read(A)</code>
	<code>write(A)</code>
<code>read(B)</code>	
<code>write(B)</code>	
	<code>read(B)</code>
	<code>write(B)</code>

FIGURA 6.41 Roteiro 2: Um roteiro serializável concorrente

Podemos tirar proveito dessa situação por meio da troca (*swapping*). Sejam O_i e O_j operações consecutivas de um roteiro S . Se O_i e O_j forem operações de transações diferentes e O_i e O_j não estiverem em conflito, então podemos trocar a ordem de O_i e O_j para produzir um novo roteiro S' . Esperamos que S' seja equivalente a S , pois todas as operações aparecem na mesma ordem nos dois roteiros, exceto para O_i e O_j , cuja ordem não importa.

Podemos ilustrar a ideia de troca considerando novamente o roteiro 2 da [Figura 6.41](#). Como a operação `write(A)` de T_1 não entra em conflito com a operação `read(B)` de T_0 , podemos trocar essas operações para gerar um roteiro equivalente, da seguinte forma:

- Trocar a operação `read(B)` de T_0 pela operação `read(A)` de T_1 .
- Trocar a operação `write(B)` de T_0 pela operação `write(A)` de T_1 .
- Trocar a operação `write(B)` de T_0 pela operação `read(A)` de T_1 .

O resultado final dessas trocas é o roteiro 1 na [Figura 6.40](#), que é um roteiro serial. Assim, mostramos que o roteiro 2 é equivalente a um roteiro serial. Se o roteiro S puder ser transformado em um roteiro serial S' por uma série de trocas de operações não conflitantes, dizemos que um roteiro S é **serializável por conflito**. Assim, o roteiro 2 é serializável por conflito, pois pode ser transformado no roteiro serial 1.

6.10.4.2 Protocolo de Lock

Um modo de garantir a serialização é associar um lock a cada item de dados e exigir que cada transação siga um **protocolo de lock** que controle como os locks são adquiridos e liberados. Nesta seção, restringimos nossa atenção a dois modos:

- **Compartilhado.** Se uma transação T_i tiver obtido um lock no modo compartilhado (indicado por S) sobre o item de dados Q , então T_i pode ler Q , mas não pode escrever nele.
- **Exclusivo.** Se uma transação T_i tiver obtido um lock no modo exclusivo (indicado por X) sobre o item de dados Q , então T_i pode ler e escrever Q .

Exigimos que cada transação requisite um lock em um modo apropriado sobre o item de dados Q , dependendo do tipo das operações que realizará sobre Q .

Para acessar um item de dados Q , a transação T_i primeiro precisa efetuar o lock em Q no modo apropriado. Se Q não estiver correntemente com o lock efetuado, então o lock é concedido e T_i agora pode acessá-lo. No entanto, se o item de dados Q estiver correntemente com o lock efetuado por alguma outra transação, então T_i pode ter de esperar. Mais especificamente, suponha que T_i requisite um lock exclusivo sobre Q . Nesse caso, T_i precisa esperar até o lock sobre Q ser liberado. Se T_i requisitar um lock compartilhado sobre Q , T_i terá de esperar, se Q estiver bloqueado no modo exclusivo. Caso contrário, ela poderá obter o lock e acessar Q . Observe que esse esquema é bastante

familiar ao algoritmo de leitores-escritores, discutido na [Seção 6.6.2](#).

Uma transação pode desbloquear um item de dados que bloqueou anteriormente. Entretanto, ela precisa manter um lock sobre um item de dados enquanto estiver acessando esse item. Além do mais, nem sempre é desejável que uma transação remova o lock de um item de dados imediatamente depois do seu último acesso a esse item de dados, pois a serialização pode não ser garantida.

Um protocolo que garante a serialização é o **protocolo de lock em duas fases (two-phase locking protocol)**. Esse protocolo exige que cada transação emita requisições de lock e unlock em duas fases:

- **Fase de crescimento.** Uma transação pode obter lock, mas não pode liberar nenhum lock.
- **Fase de encolhimento.** Uma transação pode liberar locks, mas não pode obter quaisquer novos locks.

Inicialmente, uma transação está na fase de crescimento. A transação obtém locks conforme a necessidade. Quando a transação liberar um lock, ela entrará na fase de encolhimento, e nenhuma outra requisição de lock poderá ser emitida.

O protocolo de lock em duas fases em geral garante a serialização de conflitos ([Exercício 6.34](#)). Contudo, isso não garante que não haverá deadlock. Além disso, é possível que, para determinado conjunto de transações, existam escalonamentos serializáveis por conflito que não possam ser obtidos pelo uso do protocolo de lock em duas fases. Para melhorar o desempenho em relação ao lock em duas fases, precisamos ter informações adicionais sobre as transações ou impor alguma estrutura ou ordenação sobre o conjunto de dados.

6.10.4.3 Protocolos baseados em estampa de tempo

Nos protocolos de lock descritos anteriormente, a ordem seguida pelos pares de transações em conflito é determinada no momento da execução. Outro método para determinar a ordem de serialização é selecionar uma ordenação com antecedência. O método mais comum para isso é usar um esquema de ordenação por **estampa de tempo**.

Com cada transação T_i no sistema, associamos uma estampa de tempo fixa e exclusiva, indicada por $TS(T_i)$. Essa estampa de tempo é atribuída pelo sistema antes de a transação T_i iniciar sua execução. Se uma transação T_i tiver recebido a estampa de tempo $TS(T_i)$ e, mais tarde, uma nova transação T_j entrar no sistema, então $TS(T_i) < TS(T_j)$. Existem dois métodos simples para implementar esse esquema:

- Usar o valor do relógio do sistema como estampa de tempo, ou seja, a estampa de tempo de uma transação é igual ao valor do relógio quando a transação entrar no sistema. Esse método não funcionará para transações que ocorrem em sistemas separados ou para processadores que não compartilham um único relógio.
- Usar um contador lógico como estampa de tempo, ou seja, a estampa de tempo de uma transação é igual ao valor do contador quando a transação entrar no sistema. O contador é incrementado após a atribuição de uma nova estampa de tempo.

As estampas de tempo das transações determinam a ordem de serialização. Assim, se $TS(T_i) < TS(T_j)$, então o sistema precisa garantir que o roteiro produzido é equivalente a um roteiro serial em que a transação T_i aparece antes da transação T_j .

Para implementar esse esquema, associamos a cada item de dados Q dois valores de estampa de tempo:

- **estampa W(Q)**, que indica a maior estampa de tempo de qualquer transação que executou `write(Q)` com sucesso.
- **estampa R(Q)**, que indica a maior estampa de tempo de qualquer transação que executou `read(Q)` com sucesso.

Essas estampas de tempo são atualizadas sempre que uma nova instrução `read(Q)` ou `write(Q)` for executada.

O protocolo de ordenação de estampa de tempo garante que quaisquer operações `read` e `write` conflitantes sejam executadas na ordem da estampa de tempo. Esse protocolo opera da seguinte maneira:

- Suponha que a transação T_i emita `read(Q)`:
 - Se $TS(T_i) < \text{estampa } W(Q)$, então T_i precisa ler um valor de Q que já foi modificado. Logo, a operação `read` é rejeitada e T_i é revertida.
 - Se $TS(T_i) \geq \text{estampa } W(Q)$, então a operação `read` é executada, e a estampa $R(Q)$ é definida para o maior valor entre estampa $R(Q)$ e $TS(T_i)$.
- Suponha que a transação T_i emita `write(Q)`:
 - Se $TS(T_i) < \text{estampa } R(Q)$, então o valor de Q que T_i está produzindo foi necessário anteriormente e T_i considerou que esse valor nunca seria produzido. Logo, a operação `write` é rejeitada e T_i é revertida.
 - Se $TS(T_i) < \text{estampa } W(Q)$, então T_i está tentando escrever um valor obsoleto de Q . Logo, essa operação `write` é rejeitada e T_i é revertida.
 - Caso contrário, a operação `write` é executada.

Uma transação T_i que seja revertida como resultado da emissão de uma operação read ou write recebe uma nova estampa de tempo e é reiniciada.

Para ilustrar esse protocolo, consideramos o roteiro 3 da [Figura 6.42](#) com as transações T_2 e T_3 . Consideramos que uma transação recebe uma estampa de tempo imediatamente antes de sua primeira instrução. Assim, no roteiro 3, $TS(T_2) < TS(T_3)$, e o roteiro é possível sob o protocolo de estampa de tempo.

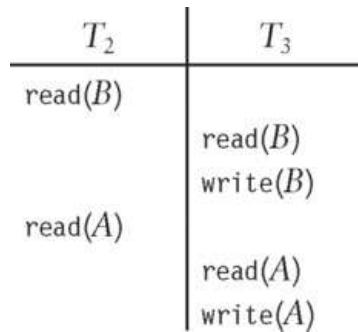


FIGURA 6.42 Roteiro 3: Um roteiro possível sob o protocolo de estampa de tempo

Essa execução também pode ser produzida pelo protocolo de lock em duas fases. Todavia, alguns roteiros são possíveis sob o protocolo de lock de duas fases, mas não sob o protocolo de estampa de tempo, e vice-versa ([Exercício 6.7](#)).

O protocolo por estampa de tempo garante a serialização de conflitos. Essa capacidade vem do fato de as operações com conflito serem processadas na ordem da estampa de tempo. O protocolo também garante ausência de deadlock, pois nenhuma transação fica esperando.

6.11 Resumo

Dada uma coleção de processos ou threads sequenciais cooperativas que compartilham dados, a exclusão mútua precisa ser fornecida para garantir que uma seção crítica do código seja usada somente por um processo ou thread de cada vez. Normalmente, o hardware do computador oferece diversas operações para garantir a exclusão mútua. No entanto, essas soluções baseadas no hardware são muito complicadas para a maioria dos desenvolvedores utilizar. Os semáforos contornam essa dificuldade. Os semáforos podem ser usados para solucionar diversos problemas de sincronismo e podem ser implementados com eficiência especialmente se o suporte do hardware para operações atômicas estiver disponível.

Diversos problemas de sincronismo (como os problemas de bounded buffer, leitores-escritores e filósofos à mesa de jantar) são importantes, em especial porque são exemplos de uma grande classe de problemas de controle de concorrência. Esses problemas são usados para testar quase todo esquema de sincronismo proposto recentemente.

O sistema operacional precisa fornecer meios de proteger contra erros de temporização. Diversas construções da linguagem foram propostas para enfrentar esses problemas. Monitores fornecem o mecanismo de sincronismo para compartilhamento de tipos de dados abstratos. Uma variável de condição provê um método pelo qual um procedimento de monitor pode bloquear sua execução até que seja sinalizado para continuar.

A Java provê várias ferramentas para coordenar as atividades de múltiplas threads acessando dados compartilhados por meio dos mecanismos `synchronized`, `wait()`, `notify()` e `notifyAll()`. Além disso, a API Java fornece suporte para locks de exclusão mútua, semáforos e variáveis de condição.

Os sistemas operacionais também proveem suporte para sincronismo. Por exemplo, Solaris, Windows XP e Linux fornecem mecanismos como semáforos, mutexes, spinlocks e variáveis de condição, para controlar o acesso aos dados compartilhados. A API Pthreads provê suporte para mutexes e variáveis de condição.

Uma transação é uma unidade de programa que precisa ser executada atomicamente, ou seja, ou todas as operações associadas a ela são executadas até o fim ou nenhuma é executada. Para garantir a atomicidade, apesar de falha do sistema, podemos usar um log de escrita antecipada. Todas as atualizações são registradas no log, que é mantido no armazenamento estável. Se houver uma falha no sistema, as informações no log são usadas na restauração do estado dos itens de dados atualizados, o que é feito com o uso de operações `undo` e `redo`. Para reduzir o custo adicional na pesquisa do log após o surgimento de uma falha do sistema, podemos usar um esquema de checkpoint.

Para garantir a serialização quando a execução de várias transações se sobrepõe, temos de usar um esquema de controle de concorrência. Vários esquemas de controle de concorrência garantem a serialização, atrasando uma operação ou abortando a transação que emitiu a operação. Os mais comuns são os protocolos de lock e os esquemas de ordenação com estampa de tempo.

Exercícios práticos

- 6.1. Na [Seção 6.4](#), mencionamos que a desativação de interrupções constantemente pode afetar o relógio do sistema. Explique por que isso pode acontecer e como esses efeitos podem ser minimizados.
- 6.2. **O problema dos fumantes de cigarro.** Considere um sistema com três processos *fumantes* e um processo *agente*. Cada fumante enrola continuamente um cigarro de palha e depois o fuma. Mas para enrolar e fumar um cigarro, ele precisa de três ingredientes: tabaco, palha e fósforos. Um dos processos fumantes tem palha, outro tem o tabaco e o terceiro tem fósforos. O agente tem um estoque infinito de todos os três materiais. O agente coloca dois dos ingredientes sobre a mesa. O fumante que tiver o ingrediente restante, então, faz e fuma um cigarro, sinalizando ao agente quando terminar. O agente, então, coloca dois outros dos três ingredientes, e o ciclo se repete. Escreva um programa para sincronizar o agente e os fumantes usando a sincronismo Java.
- 6.3. Explique por que Solaris, Windows XP e Linux implementam mecanismos de locking múltiplo. Descreva as circunstâncias sob as quais eles utilizam spinlocks, mutexes, semáforos, mutexes adaptativos e variáveis de condição. Em cada caso, explique por que o mecanismo é necessário.
- 6.4. Descreva as diferenças entre armazenamento volátil, não volátil e estável em relação ao custo.
- 6.5. Explique a finalidade do mecanismo de checkpoint. Com que frequência os checkpoints devem ser realizados? Descreva como a frequência dos checkpoints afeta:
- O desempenho do sistema quando não ocorrem falhas.
 - O tempo gasto para a recuperação de uma falha do sistema.
 - O tempo gasto para a recuperação de uma falha do disco.
- 6.6. Explique o conceito de atomicidade da transação.
- 6.7. Mostre que alguns roteiros são possíveis sob o protocolo de lock em duas fases, mas não sob o protocolo de estampa de tempo, e vice-versa.

Exercícios

- 6.8. Condições de corrida são possíveis em diversos sistemas de computação. Considere um sistema bancário com dois métodos: `depositar(valor)` e `sacar(valor)`. Esses dois métodos recebem o `valor` que deve ser depositado ou sacado de uma conta bancária. Suponha que um marido e sua esposa compartilhem uma conta bancária e que, simultaneamente, o marido chame o método `sacar()` e a esposa chame `depositar()`. Descreva como pode haver uma condição de corrida nessa situação e o que poderia ser feito para impedir sua ocorrência.
- 6.9. A primeira solução de software correta conhecida para o problema de seção crítica para dois processos foi desenvolvida por Dekker. Os dois processos, P_0 e P_1 , compartilham as seguintes variáveis:

```
boolean flag[2]; /* inicialmente falso */
int turn;
```

A estrutura do processo P_i ($i == 0$ ou 1) aparece na [Figura 6.43](#); o outro processo é P_j ($j == 1$ ou 0). Prove que o algoritmo satisfaz a todos os três requisitos para o problema da seção crítica.

```
do {
    flag[i] = true;

    while (flag[j]) {
        if (turn == j) {
            flag[i] = false;
            while (turn == j)
                ; // não faz nada
            flag[i] = true;
        }
    }

    // seção crítica

    turn = j;
    flag[i] = false;

    // seção restante
} while (true);
```

FIGURA 6.43 A estrutura do processo P_i no algoritmo de Dekker

- 6.10. A primeira solução de software correta conhecida para o problema da seção crítica para n processos com um limite inferior na espera de $n - 1$ vezes foi apresentada por Eisenberg e McGuire. Os processos compartilham as seguintes variáveis:

```
enum pstate{idle, want_in, in_cs};
pstate flag[n];
int turn
```

Todos os elementos de `flag` são inicialmente `idle`; o valor inicial de `turn` é imaterial (entre 0 e $n-1$). A estrutura do processo P_i aparece na [Figura 6.44](#). Mostre que o algoritmo satisfaz a todos os três requisitos para o problema da seção crítica.

```

do {
    while (true) {
        flag[i] = want_in;
        j = turn;

        while (j != i) {
            if (flag[j] != idle) {
                j = turn;
            } else
                j = (j + 1) % n;
        }

        flag[i] = in_cs;
        j = 0;

        while ( (j < n) && (j == i || flag[j] != in_cs) )
            j++;

        if ( (j >= n) && (turn == i || flag[turn] == idle) )
            break;
    }

    // seção crítica

    j = (turn + 1) % n;

    while (flag[j] == idle)
        j = (j + 1) % n;

    turn = j;
    flag[i] = idle;

    // seção restante
} while (true);

```

FIGURA 6.44 A estrutura do processo P_i no algoritmo de Eisenberg e McGuire.

- 6.11. Qual é o significado do termo *espera ocupada*? Que outros tipos de espera existem em um sistema operacional? A espera ocupada pode ser completamente evitada? Explique sua resposta.
- 6.12. Explique por que os spinlocks não são apropriados para sistemas de único processador, enquanto são muito utilizados em sistemas multiprocessados.
- 6.13. Explique por que a implementação de primitivas de sincronismo desativando as interrupções não é apropriada no sistema de único processador se as primitivas de sincronismo tiverem que ser usadas em programas no nível do usuário.
- 6.14. Explique por que as interrupções não são apropriadas para a implementação de primitivas de sincronismo em sistemas multiprocessados.
- 6.15. Descreva duas estruturas de dados do kernel nas quais as condições de corrida são possíveis. Não se esqueça de incluir uma descrição de como uma condição de corrida pode ocorrer.
- 6.16. Descreva como a instrução `swap()` pode ser usada para melhorar a exclusão mútua que satisfaça o requisito de espera limitada.
- 6.17. Mostre que, se as operações de semáforo `acquire()` e `release()` não forem executadas atomicamente, então a exclusão mútua poderá ser violada.
- 6.18. O Windows Vista oferece uma nova ferramenta de sincronismo leve, chamada **slim reader-writer lock**. Embora a maioria das implementações de slim reader-writer lock favoreça leitores ou escritores, ou talvez ordene as threads que aguardam utilizando uma política FIFO (primeiro a entrar, primeiro a sair), os slim reader-writer lock não favorecem nem leitores nem escritores, e não ordenam as threads aguardando em uma fila FIFO. Explique os benefícios de oferecer essa ferramenta de sincronismo.
- 6.19. Mostre como implementar as operações de semáforo `acquire()` e `release()` em ambientes multiprocessados usando a instrução `getAndSet()`. A solução deverá exibir a espera ocupada mínima.
- 6.20. Demonstre que monitores e semáforos são equivalentes em se tratando de poderem ser usados para implementar os mesmos tipos de problemas de sincronismo.
- 6.21. Escreva um monitor de buffer limitado no qual os buffers (porções) estejam embutidos dentro do próprio monitor.
- 6.22. A exclusão mútua estrita dentro de um monitor torna o monitor de buffer limitado do Exercício 6.21 adequado principalmente para pequenas porções.
 - a. Explique por que isso é verdadeiro.
 - b. Crie um novo esquema que seja adequado para porções maiores.
- 6.23. Discuta as escolhas entre justiça e throughput de operações no problema dos leitores-escritores. Proponha um método para solucionar o problema dos leitores-escritores sem causar starvation.
- 6.24. Como a operação `signal()` associada a monitores difere da operação `release()` correspondente

- definida para semáforos?
- 6.25. Suponha que a instrução `signal()` só possa aparecer como a última instrução em um procedimento de monitor. Sugira como a implementação descrita na [Seção 6.7](#) pode ser simplificada nesta situação.
- 6.26. Considere um sistema consistindo em processos P_1, P_2, \dots, P_n , cada um com um número de prioridade exclusivo. Escreva um monitor que aloca três impressoras idênticas a esses processos, usando os números de prioridade para decidir a ordem de alocação.
- 6.27. Um arquivo deve ser compartilhado entre diferentes processos, cada um deles com um número exclusivo. O arquivo pode ser acessado simultaneamente por diversos processos, sujeito à seguinte restrição: a soma de todos os números exclusivos associados a todos os processos acessando atualmente o arquivo deve ser menor que n . Escreva um monitor para coordenar o acesso ao arquivo.
- 6.28. Quando um sinal é realizado sobre uma condição dentro de um monitor, o processo de sinalização pode continuar sua execução ou transferir o controle ao processo que é sinalizado. Qual é a diferença entre a solução do exercício anterior e essas duas formas diferentes de realizar a sinalização?
- 6.29. Suponha que substituamos as operações `wait()` e `signal()` dos monitores por uma única construção `await(B)`, onde B é uma expressão booleana geral que faz o processo que a está executando esperar até que B se torne `true`.
- Escreva um monitor usando esse esquema para implementar o problema dos leitores-escritores.
 - Explique por que, de um modo geral, essa construção não pode ser implementada de modo eficiente.
 - Que restrições precisam ser colocadas na instrução `await()` para que possa ser implementada de modo eficiente? (Dica: Restrinja a generalidade de B ; ver [Kessels \[1977\]](#).)
- 6.30. Escreva um monitor que implemente um *relógio despertador* permitindo que um programa que o chama seja adiado por um número especificado de unidades de tempo (*tiques*). Você pode considerar a existência de um relógio de hardware real, que chama um procedimento *tique* no seu monitor em intervalos regulares.
- 6.31. O padrão de projeto **Singleton** garante que será criada somente uma instância de um objeto. Por exemplo, considere que temos uma classe chamada `Singleton` e só queremos permitir uma instância dela. Em vez de criar um objeto `Singleton` usando seu construtor, declaramos o construtor como `private` e fornecemos um método `public static` – como `getInstance()` – para a criação de objeto:

```
Singleton sole = Singleton.getInstance( );
```

A [Figura 6.45](#) fornece uma estratégia para implementar o padrão Singleton. A ideia por trás dessa técnica é usar a **inicialização lazy**, pela qual criamos uma instância do objeto somente quando ela é necessária, ou seja, quando `getInstance()` é chamada inicialmente. Porém, a [Figura 6.45](#) sofre de uma condição de corrida. Identifique a condição de corrida.

```
public class Singleton {
    private static Singleton instance = null;

    private Singleton { }

    public static Singleton getInstance( ) {
        if (instance == null)
            instance = new Singleton( );

        return instance;
    }
}
```

FIGURA 6.45 Primeira tentativa no padrão de projeto Singleton

A [Figura 6.46](#) mostra uma estratégia alternativa que resolve a condição de corrida usando o **idioma de lock duplamente verificado**. Usando essa estratégia, primeiro verificamos se `instance` é `null`. Se for, em seguida obtemos o lock para a classe `Singleton` e depois verificamos novamente se `instance` ainda é `null` antes de criar o objeto. Essa estratégia resulta em algumas

condições de corrida? Nesse caso, identifique-as e conserte-as. Caso contrário, ilustre por que esse exemplo de código é seguro para thread.

```
public class Singleton {  
    private static Singleton instance = null;  
  
    private Singleton { }  
  
    public static Singleton getInstance( ) {  
        if (instance == null){  
            synchronized(Singleton.class) {  
                if (instance == null)  
                    instance = new Singleton( );  
            }  
        }  
  
        return instance;  
    }  
}
```

FIGURA 6.46 Padrão de projeto Singleton usando o lock duplamente verificado

- 6.32. Por que Solaris, Linux e Windows XP utilizam spinlocks como mecanismo de sincronismo somente em sistemas multiprocessados e não em sistemas de único processador?
- 6.33. Em sistemas baseados em log, que proveem suporte para transações, as atualizações nos itens de dados não podem ser realizadas antes que as entradas correspondentes sejam registradas em log. Por que essa restrição é necessária?
- 6.34. Mostre que o protocolo de lock em duas fases garante a serialização por conflito.
- 6.35. Quais são as implicações de atribuir uma nova estampa de tempo a uma transação que é revertida? Como o sistema processa transações que foram emitidas depois da transação revertida, mas que possuem estampas de tempo menores que a nova estampa de tempo da transação revertida?

Problemas de programação

- 6.36. O Exercício 4.20 requer que a thread principal espere pelas threads de classificação e de mesclagem usando o método `join()`. Modifique sua solução para este exercício de modo que ele use semáforos em vez do método `join()`. (Dica: Recomendamos a leitura cuidadosa da API Java sobre o construtor para os objetos `Semaphore`.)
- 6.37. A classe `HardwareData` da Figura 6.4 utiliza a ideia de instruções *get-and-set* e *swap*. Entretanto, essa classe não é considerada segura para threads, pois várias threads podem acessar seus métodos ao mesmo tempo e a segurança da thread exige que cada método seja executado atomicamente. Reescreva a classe `HardwareData` usando o sincronismo Java, de modo que seja segura para threads.
- 6.38. Os servidores podem ser projetados para limitar o número de conexões abertas. Por exemplo, um servidor pode querer ter apenas N conexões de socket abertas em determinado momento. Assim que N conexões forem feitas, o servidor não aceitará outra conexão que chegue até uma conexão existente ser liberada.
- 6.39. Considere que um número finito de recursos de um único tipo de recurso tenha que ser gerenciado. Os processos podem solicitar uma série desses recursos e, uma vez terminando, os retornará. Como exemplo, muitos pacotes de software comerciais proveem um determinado número de **licenças**, indicando o número de aplicações que podem ser executadas simultaneamente. Quando a aplicação é iniciada, a contagem de licenças é decrementada. Quando a aplicação termina, a contagem de licenças é incrementada. Se todas as licenças estiverem em uso, as solicitações para iniciar a aplicação serão negadas. Essas solicitações só serão concedidas quando alguém que mantiver uma licença existente terminar de usar a aplicação e uma licença for retornada.
- A classe Java a seguir é usada para gerenciar um número finito de instâncias de um recurso disponível. Observe que quando um processo quiser obter um determinado número de recursos, ele chama o método `decreaseCount()`. De modo semelhante, quando um processo deseja retornar determinado número de recursos, ele chama `increaseCount()`.

```
public class Manager
{
    public static final int MAX_RESOURCES = 5;
    private int availableResources = MAX_RESOURCES;

    /**
     * Diminui availableResources por count recursos.
     * retorna 0 se houver recursos suficientes à
     * disposição, ou então retorna -1
     */
    public int decreaseCount(int count) {
        if (availableResources < count)
            return -1;
        else {
            availableResources -= count;

            return 0;
        }
    }

    /* Aumenta availableResources por count recursos. */
    public void increaseCount(int count) {
        availableResources += count;
    }
}
```

Entretanto, o segmento de programa que apresentamos produz uma condição de corrida. Faça o seguinte:

- Identifique os dados envolvidos na condição de corrida.
- Identifique o local (ou locais) no código onde a condição de corrida ocorre.
- Usando o sincronismo da Java, conserte a condição de corrida. Modifique também `decreaseCount()` de modo que uma thread seja bloqueada se não houver recursos suficientes à disposição.

- 6.40. Implemente a interface `Channel` (Figura 3.20) de modo que os métodos `send()` e `receive()` causem bloqueios. Ou seja, uma thread chamando `send()` será bloqueada se o canal estiver cheio. Se o canal estiver vazio, uma thread chamando `receive()` será bloqueada. Isso exigirá o armazenamento de mensagens em um array de tamanho fixo. Cuide para que sua implementação seja segura para a thread (utilizando o sincronismo Java) e que as mensagens sejam armazenadas na ordem FIFO.

```

        public void acquireWriteLock( ) {
            db.acquire( );
        }

        public void releaseWriteLock( ) {
            db.release( );
        }
    }

```

FIGURA 6.20 Métodos chamados pelos escritores

6.41. Uma **barreira** é um mecanismo de sincronismo de thread para permitir que várias threads sejam executadas por um período, mas depois força todas as threads a esperarem até todas terem atingido um certo ponto. Quando todas as threads tiverem atingido esse ponto (a barreira), elas poderão continuar. Uma interface para uma barreira se parece com o seguinte:

```

public interface Barrier
{
    /**
     * Cada thread chama este método quando
     * atinge a barreira. Todas as threads
     * são liberadas para continuar processando
     * quando a última thread chamar este método.
     */
    public void waitForOthers();

    /**
     * Libera todas as threads da espera pela
     * barreira.
     * Quaisquer chamadas futuras para waitForOthers()
     * não esperarão
     * até que a barreira seja definida novamente com
     * uma chamada ao construtor.
     */
    public void freeAll();
}

```

O segmento de código a seguir estabelece uma barreira e cria 10 threads `Worker` que serão sincronizadas de acordo com a barreira:

```

public static final int THREAD_COUNT = 10;

Barrier jersey = new BarrierImpl(THREAD_
COUNT);
for (int i = 0; i < THREAD_COUNT; i++)
    (new Worker(jersey)).start();

```

Observe que a barreira precisa ser inicializada com o número de threads que estão sendo sincronizadas e que cada thread possui uma referência ao mesmo objeto de barreira – `jersey`. Cada `Worker` seria executado da seguinte forma:

```

// Todas as threads têm acesso a esta
barreira Barrier jersey;
Barrier jersey;

// faz algum trabalho por um tempo...

// agora espera pelos outros
jersey.waitForOthers( );

// agora trabalha mais...

```

Quando uma thread chamar o método `waitForOthers()`, ela será bloqueada até todas as threads terem alcançado esse método (a barreira). Quando todas as threads tiverem alcançado esse método, elas poderão prosseguir com o restante do seu código. O método `freeAll()` evita a necessidade de esperar que as threads alcancem a barreira; assim que `freeAll()` é chamado, todas as threads aguardando pela barreira são liberadas.

Implemente uma interface `Barrier` usando o sincronismo Java.

- 6.42. Implemente a interface `Buffer` ([Figura 3.15](#)) como um buffer limitado, usando as variáveis de condição da Java. Teste a sua solução usando a [Figura 6.14](#).
- 6.43. Implemente a interface `ReadWriteLock` ([Figura 6.17](#)) usando as variáveis de condição da Java. Você poderá achar necessário examinar o método `signalAll()` na API `Condition`.
- 6.44. **O problema do barbeiro dorminhoco.** Uma barbearia consiste em uma sala de espera com n cadeiras e um salão de barbeiro com uma cadeira. Se não houver clientes para serem atendidos, o barbeiro vai dormir. Se um cliente entrar na barbearia e todas as cadeiras estiverem ocupadas, o cliente vai embora. Se o barbeiro estiver ocupado, mas houver cadeiras disponíveis, o cliente senta em uma das cadeiras vagas. Se o barbeiro estiver dormindo, o cliente acorda o barbeiro. Escreva um programa para coordenar o barbeiro e os clientes usando o sincronismo Java.

Projetos de programação

Os projetos a seguir discutirão três temas distintos: projeto de um gerenciador de pid, projeto de um banco de threads e a implementação de uma solução para um problema de filósofos à mesa de um jantar usando variáveis de condição do Java.

Projeto 1: Projeto de um gerenciador de pid

Um gerenciador de pid é responsável por gerenciar identificadores de processo (pids). Quando um processo é inicialmente criado, ele recebe um pid exclusivo pelo gerenciador de pid. O pid é retornado ao seu gerenciador quando o processo termina sua execução. O gerenciador de pid pode, mais tarde, reatribuir esse pid. Os identificadores de processo são discutidos com mais detalhe na [Seção 3.3.1](#). O mais importante aqui é reconhecer que os identificadores de processo precisam ser exclusivos; dois processos ativos não podem ter o mesmo pid.

A interface Java mostrada na [Figura 6.47](#) identifica os métodos básicos para obter e liberar um pid. Os identificadores de processo são atribuídos dentro do intervalo entre `MIN_PID` e `MAX_PID` (inclusive). A diferença fundamental entre `getPID()` e `getPIDWait()` é que, se nenhum pid estiver disponível, `getPID()` retorna `-1`, enquanto `getPIDWait()` bloqueia o processo que chama até que um pid esteja disponível. Assim como a maioria dos dados do kernel, a estrutura de dados para manter um conjunto de pids precisa estar livre de condições de corrida e deadlock. Um resultado possível de uma condição de corrida é que o mesmo pid será atribuído simultaneamente a mais de um processo. (No entanto, um pid pode ser reutilizado depois que tiver sido retornado por meio da chamada a `releasePID()`.) Para conseguir o comportamento de bloqueio em `getPIDWait()`, você poderá usar qualquer um dos mecanismos de sincronismo baseados em Java, discutidos na [Seção 6.8](#).

```

/**
 * Uma interface para um gerenciador de PID.
 *
 * O intervalo de PID's permitidos é
 * MIN_PID .. MAX_PID (inclusive)
 *
 * Uma implementação desta interface
 * deve garantir a segurança para thread.
 */

public interface PIDManager
{
    /** O intervalo de PIDs permitidos (inclusive) */
    public static final int MIN_PID = 4;
    public static final int MAX_PID = 127;

    /**
     * Retorna um PID válido ou -1 se
     * não houver um disponível
     */
    public int getPID();

    /**
     * Retorna um PID válido, possivelmente bloqueando
     * o processo que chama até que haja um disponível.
     */
    public int getPIDWait();

    /**
     * Libera o pid
     * Lança uma IllegalArgumentException se o pid
     * estiver fora do intervalo de valores de PID.
     */
    public void releasePID(int pid);
}

```

FIGURA 6.47 Interface Java para obter e liberar um pid

Projeto 2: Projeto de um banco de threads

Crie um banco de threads (ver [Capítulo 4](#)) usando o sincronismo Java. Seu banco de threads implementará a seguinte API:

ThreadPool ()	Cria um banco de threads de tamanho padrão.
ThreadPool (int size)	Cria um banco de threads com o tamanho size.
void add (Runnable task)	Acrescenta uma tarefa a ser realizada por uma thread no banco.
void stopPool ()	Termina todas as threads no banco.

Seu banco primeiro criará uma série de threads ociosas que aguardam trabalho. O trabalho será submetido ao banco por meio do método `add()`, que acrescenta uma tarefa implementando a interface `Runnable`. O método `add()` colocará a tarefa `Runnable` em uma fila. Quando uma thread no banco se tornar disponível para trabalhar, ela verificará a fila em busca de quaisquer tarefas `Runnable`. Se houver tais tarefas, a thread ociosa removerá a tarefa da fila e chamará seu método `run()`. Se a fila estiver vazia, a thread ociosa esperará para ser notificada quando o trabalho estiver disponível. (O método `add()` realizará um `notify()` quando colocar uma tarefa `Runnable` na fila para possivelmente despertar uma thread ociosa esperando trabalho.) O método `stopPool()` terminará todas as threads no banco chamando seu método `interrupt()` ([Seção 4.5.2](#)). Naturalmente, isso exige que as tarefas `Runnable` executadas pelo banco de threads verifiquem seu status de interrupção.

Existem várias maneiras diferentes de testar a sua solução para este problema. Uma sugestão é modificar sua resposta para o [Exercício 3.17](#), de modo que o servidor possa responder a cada requisição do cliente usando um banco de threads.

Projeto 3: Filósofos à mesa de jantar

Na [Seção 6.7.2](#), fornecemos um esboço de uma solução para o problema dos filósofos à mesa de jantar usando monitores. Este exercício exigirá a implementação dessa solução usando as variáveis de condição da Java.

Comece criando cinco filósofos, cada um identificado por um número de 0 a 4. Cada filósofo é executado como uma thread separada. Os filósofos alternarão entre pensar e comer. Quando um filósofo quer comer, ele chama o método `takeForks(phiNumber)`, onde `phiNumber` identifica o número do filósofo que deseja comer. Quando um filósofo termina de comer, ele chama `returnForks(phiNumber)`.

Sua solução implementará a seguinte interface:

```
public interface DiningServer
{
    /* Chamada por um filósofo quando ele quiser comer */
    public void takeForks(int phiNumber);

    /* Chamada por um filósofo quando ele acabar de comer */
    public void returnForks(int phiNumber);
}
```

A implementação da interface segue o esboço da solução fornecida na [Figura 6.26](#). Use as variáveis de condição da Java para sincronizar a atividade dos filósofos e impedir o deadlock.

Notas bibliográficas

Os problemas de exclusão mútua foram discutidos inicialmente no trabalho clássico de [Dijkstra \[1965a\]](#). O algoritmo de Dekker ([Exercício 6.9](#)) - a primeira solução de software correta para o problema de exclusão mútua de dois processos - foi desenvolvido pelo matemático holandês T. Dekker. Esse algoritmo também foi discutido por [Dijkstra \[1965a\]](#). Uma solução mais simples para o problema de exclusão mútua de dois processos desde então foi apresentado por [Peterson \[1981\]](#) ([Figura 6.2](#)).

[Dijkstra \[1965b\]](#) apresentou a primeira solução para o problema de exclusão mútua para n processos. Porém, essa situação não coloca um limite superior sobre a quantidade de tempo que um processo precisa esperar antes de poder entrar na seção crítica. [Knuth \[1966\]](#) apresentou o primeiro algoritmo com um limite; seu limite foi de 2^n vezes. Um refinamento do algoritmo de Knuth, por [deBruijn \[1967\]](#), reduziu o tempo de espera a n^2 vezes; depois disso, [Eisenberg e McGuire \[1972\]](#) tiveram sucesso na redução do tempo para o limite mais baixo de $n - 1$ vezes. Outro algoritmo que também requer $n - 1$ vezes, porém mais fácil de programar e compreender, é o algoritmo da padaria, que foi desenvolvido por [Lamport \[1974\]](#). [Burns \[1978\]](#) desenvolveu o algoritmo de solução por hardware que satisfaz o requisito de espera limitada.

Discussões gerais referentes ao problema de exclusão mútua foram oferecidas por [Lamport \[1986\]](#) e [Lamport \[1991\]](#). Uma coleção de algoritmos para exclusão mútua foi dada por [Raynal \[1986\]](#).

O conceito de semáforo foi sugerido por [Dijkstra \[1965a\]](#). [Patil \[1971\]](#) examinou se os semáforos podem resolver todos os problemas de sincronismo possíveis. [Parnas \[1975\]](#) discutiu algumas das falhas nos argumentos de Patil. [Kosaraju \[1973\]](#) deu continuidade ao trabalho de Patil para produzir um problema que não pode ser resolvido pelas operações `wait()` e `signal()`. [Lipton \[1974\]](#) discutiu as limitações de diversas primitivas de sincronismo.

Os problemas clássicos de coordenação de processos que descrevemos são paradigmas para uma grande classe de problemas de controle de concorrência. O problema de bounded-buffer, o problema dos filósofos à mesa de jantar e o problema do barbeiro dorminhoco ([Exercício 6.44](#)) foram sugeridos por [Dijkstra \[1965a\]](#) e [Dijkstra \[1971\]](#). O problema dos fumantes ([Exercício 6.2](#)) foi desenvolvido por [Patil \[1971\]](#). O problema dos leitores-escritores foi sugerido por [Courtois e outros \[1971\]](#). A questão da leitura e escrita simultânea foi discutida por [Lamport \[1977\]](#). O problema do sincronismo de processos independentes foi discutido por [Lamport \[1976\]](#).

O conceito de região crítica foi sugerido por [Hoare \[1972\]](#) e por [Brinch-Hansen \[1972\]](#). O conceito de monitor foi desenvolvido por [Brinch-Hansen \[1973\]](#). Uma descrição completa do monitor foi dada por [Hoare \[1974\]](#). [Kessels \[1977\]](#) propôs uma extensão ao monitor para permitir a sinalização automática. A experiência obtida com o uso de monitores em programas concorrentes foi discutida em [Lampson e Redell \[1979\]](#). Eles também examinaram o problema de inversão de prioridade. Discussões gerais referentes à programação concorrente foram oferecidas por [Ben-Ari \[1990\]](#) e [Birrell \[1989\]](#).

A otimização do desempenho de primitivas de lock foi discutida em muitos trabalhos, como

[Lamport \[1987\]](#), [Mellor-Crummey e Scott \[1991\]](#) e [Anderson \[1990\]](#). O uso de objetos compartilhados que não exigem o uso de seções críticas foi discutido em [Herlihy \[1993\]](#), [Bershad \[1993\]](#) e [Kopetz e Reisinger \[1993\]](#). Novas instruções de hardware e sua utilidade na implementação de primitivas de sincronismo foram descritas em trabalhos como [Culler e outros \[1998\]](#), [Goodman e outros \[1989\]](#), [Barnes \[1993\]](#) e [Herlihy e Moss \[1993\]](#).

Alguns detalhes dos mecanismos de lock usados no Solaris foram apresentados em [Mauro e McDougall \[2007\]](#). Observe que os mecanismos de lock usados pelo kernel são implementados também para as threads no nível do usuário, de modo que os mesmos tipos de locks estão disponíveis dentro e fora do kernel. Os detalhes do sincronismo do Windows 2000 podem ser encontrados em [Solomon e Russinovich \[2000\]](#). O sincronismo de thread em Java é explicado em [Oaks e Wong \[2004\]](#); [Goetz e outros \[2006\]](#) apresentam uma discussão detalhada sobre programação simultânea em Java, bem como o pacote `java.util.concurrent`.

O esquema de log de escrita antecipada foi introduzido inicialmente no System R por [Gray e outros \[1981\]](#). O conceito de serialização foi formulado por [Eswaran e outros \[1976\]](#) em conjunto com seu trabalho sobre controle de concorrência para System R. O protocolo de lock em duas fases foi introduzido por [Eswaran e outros \[1976\]](#). O esquema de controle de concorrência baseado em estampa de tempo foi providenciado por [Reed \[1983\]](#). Uma exposição dos diversos algoritmos de controle de concorrência baseada em estampa de tempo foi apresentada por [Bernstein e Goodman \[1980\]](#). [Adl-Tabatabai e outros \[2007\]](#) discutem a memória transacional.

CAPÍTULO 7

Deadlocks

Em um ambiente multiprogramado, vários processos podem competir por um número finito de recursos. Um processo requisita recursos; se os recursos não estiverem disponíveis nesse instante, o processo entra em um estado de espera. Às vezes, os processos em espera podem nunca mais mudar de estado porque os recursos requisitados estão retidos por outros processos no estado de espera. Essa situação é chamada de **deadlock**. Esse assunto foi discutido rapidamente no [Capítulo 6](#), em conjunto com os semáforos.

Talvez a melhor ilustração de um deadlock possa ser tirada de uma lei submetida à legislatura do Kansas no início do século XX. Ela dizia, em parte: “Quando dois trens se aproximarem um do outro, em um cruzamento, ambos deverão parar completamente e nenhum deverá dar partida novamente até que o outro tenha passado.”

Neste capítulo, descrevemos métodos que um sistema operacional pode usar para prevenir ou lidar com deadlocks. Embora algumas aplicações possam identificar programas que possam causar deadlock, a maioria dos sistemas operacionais normalmente não provê facilidades de prevenção de deadlock, e continua sendo responsabilidade dos programadores garantir o projeto de programas livres de deadlock. Os problemas de deadlock só podem se tornar mais comuns, dadas as tendências atuais, incluindo grande quantidade de processos, programas multithreads, muito mais recursos dentro de um sistema e ênfase em servidores de arquivos e banco de dados de longa duração, em vez de sistemas batch.

OBJETIVOS DO CAPÍTULO

- Desenvolver uma descrição de deadlocks, que impedem que grupos de processos concorrentes completem suas tarefas.
- Apresentar diversos métodos para prevenir ou evitar deadlocks em um sistema computadorizado.

7.1 Modelo do sistema

Um sistema consiste em uma quantidade finita de recursos a serem distribuídos entre uma série de processos em competição. Os recursos são particionados em diversos tipos, cada um consistindo em alguma quantidade de instâncias idênticas. Espaço de memória, ciclos de CPU, arquivos e dispositivos de E/S (como impressoras e unidades de DVD) são exemplos de tipos de recursos. Se um sistema tiver duas CPUs, então o recurso *CPU* terá duas instâncias. Da mesma forma, o recurso *impressora* poderá ter cinco instâncias.

Se um processo requisitar uma instância de um recurso, a alocação de *qualquer* instância do tipo satisfará a requisição. Caso contrário, as instâncias não são idênticas, e as classes do recurso não terão sido definidas corretamente. Por exemplo, um sistema pode ter duas impressoras. Essas duas impressoras podem ser definidas como estando na mesma classe de recurso, se ninguém se importar com qual impressora imprimirá qual saída. No entanto, se uma impressora estiver no nono andar e a outra estiver no porão, então as pessoas no nono andar podem não ver as duas impressoras como equivalentes, e uma classe de recurso separada poderá ser definida para cada impressora.

Um processo precisa requisitar um recurso antes de usá-lo e deverá liberar o recurso depois de usá-lo. Um processo pode requisitar tantos recursos quantos necessários para executar sua tarefa designada. É claro que a quantidade de recursos requisitada não poderá ultrapassar a quantidade total de recursos disponíveis no sistema. Em outras palavras, um processo não pode requisitar três impressoras se o sistema tiver apenas duas.

Sob o modo de operação normal, um processo pode utilizar um recurso apenas na seguinte sequência:

1. **Requisitar.** O processo requisita o recurso. Se a requisição não puder ser concedida imediatamente (por exemplo, se o recurso estiver sendo usado por outro processo), o processo requisitante precisará esperar até poder obter o recurso.
2. **Usar.** O processo pode operar no recurso (por exemplo, se o recurso for uma impressora, o processo pode imprimir na impressora).
3. **Liberar.** O processo libera o recurso.

A requisição e a liberação de recursos são chamadas de sistema (system calls), conforme explicamos no [Capítulo 2](#). Alguns exemplos são as chamadas de sistema `request()` e `release()` `device`, `open()` `file` e `allocate()` e `free()` `memory`. A requisição e a liberação de recursos que não são gerenciados pelo sistema operacional podem ser feitas usando as operações `acquire()` e `release()` sobre semáforos ou usando a aquisição e liberação de um lock de objeto, por meio da palavra-chave `synchronized` da Java. Para cada uso de um recurso por um processo ou thread, o sistema operacional verifica para garantir que o processo tenha requisitado e recebido um recurso. Uma tabela do sistema registra se cada recurso está liberado ou alocado; para cada recurso alocado, a tabela também registra o processo ao qual é alocada. Se um processo requisitar um recurso atualmente alocado a outro processo, ele pode ser acrescentado a uma fila de processos esperando por esse recurso.

Um conjunto de processos está em um estado de deadlock quando cada processo no conjunto está esperando por um evento que só pode ser causado por outro processo no conjunto. Os eventos aos quais nos referimos aqui são, principalmente, aquisição e liberação de recursos. Os recursos podem ser recursos físicos (por exemplo, impressoras, unidades de fita, espaço de memória e ciclos de CPU) ou recursos lógicos (por exemplo, arquivos, semáforos e monitores). Entretanto, outros tipos de eventos podem resultar em deadlocks (por exemplo, as facilidades de IPC discutidas no [Capítulo 3](#)).

Para ilustrar um estado de deadlock, considere um sistema com três unidades de CD RW. Suponha que cada um de três processos mantenha uma dessas unidades de CD RW. Se cada processo agora requisitar outra unidade, os três processos estarão em estado de deadlock. Cada um está esperando pelo evento “CD RW está liberado”, que só pode ser causado por um dos outros processos esperando. Esse exemplo ilustra um deadlock envolvendo o mesmo tipo de recurso.

Os deadlocks também podem envolver diferentes tipos de recursos. Por exemplo, considere um sistema com uma impressora e uma unidade de DVD. Suponha que o processo P_i esteja mantendo a unidade de DVD e o processo P_j esteja mantendo a impressora. Se P_i requisitar a impressora e P_j requisitar a unidade de DVD, haverá um deadlock.

Um programador que estiver desenvolvendo aplicações multithreads deverá prestar atenção especial a esse problema: os programas desse tipo são bons candidatos ao deadlock, pois diversas threads podem competir pelos recursos compartilhados.

7.2 Caracterização do deadlock

Em um deadlock, os processos nunca terminam de executar, e os recursos do sistema ficam retidos, impedindo que outras tarefas sejam iniciadas. Antes de discutirmos os diversos métodos para lidar com o problema de deadlock, vamos examinar mais de perto as características dos deadlocks.

7.2.1 Condições necessárias

Uma situação de deadlock pode surgir se as quatro condições a seguir forem atendidas simultaneamente em um sistema:

1. **Exclusão mútua.** Pelo menos um recurso precisa estar retido em modo não compartilhado, ou seja, somente um processo por vez pode usar o recurso. Se outro processo requisitar esse recurso, o processo requisitante deverá ser adiado até o recurso ter sido liberado.
2. **Manter e esperar.** Um processo precisa estar de posse de pelo menos um recurso e esperando para obter a posse de recursos adicionais que estão em posse de outros processos.
3. **Não preempção.** Os recursos não podem ser preemptados, ou seja, um recurso só pode ser liberado voluntariamente pelo processo que o retém depois de esse processo ter concluído sua tarefa.
4. **Espera circular.** Existe um conjunto $\{P_0, P_1, \dots, P_n\}$ de processos esperando, de modo que P_0 aguarda um recurso retido por P_1 , P_1 aguarda um recurso retido por P_2, \dots, P_{n-1} , P_{n-1} aguarda um recurso retido por P_n , e P_n aguarda um recurso retido por P_0 .

Enfatizamos que todas as quatro condições precisam ser atendidas para ocorrer um deadlock. A condição de espera circular contém condição **manter e esperar**, de modo que as quatro condições não são independentes. Contudo, veremos, na [Seção 7.4](#), que é útil considerar cada condição separadamente.

7.2.2 Grafo de alocação de recursos

Os deadlocks podem ser descritos com mais precisão em termos de um grafo direcionado, chamado **grafo de alocação de recursos do sistema**. Esse grafo consiste em um conjunto de vértices V e um conjunto de arestas A . O conjunto de vértices V é partitionado em dois tipos de nós diferentes: $P = \{P_1, P_2, \dots, P_n\}$, o conjunto consistindo em todos os processos ativos no sistema, e $R = \{R_1, R_2, \dots, R_m\}$, o conjunto consistindo em todos os tipos de recursos no sistema.

Uma aresta direcionada do processo P_i para o recurso R_j é indicada por $P_i \rightarrow R_j$; isso significa que o processo P_i requisitou uma instância do recurso R_j e está esperando por esse recurso. Uma aresta direcionada do recurso R_j para o processo P_i é indicada por $R_j \rightarrow P_i$, o que significa que uma instância do recurso R_j foi alocada ao processo P_i . Uma aresta direcionada $P_i \rightarrow R_j$ é chamada de **aresta de requisição**; uma aresta direcionada $R_j \rightarrow P_i$ é chamada de **aresta de atribuição**.

Representamos graficamente cada processo P_i como um círculo e cada recurso R_j como um retângulo. Como o recurso R_j pode ter mais de uma instância, representamos cada instância como um ponto dentro do retângulo. Observe que uma aresta de requisição aponta somente para o retângulo R_j , enquanto uma aresta de atribuição também deve designar um dos pontos no retângulo.

Quando o processo P_i requisita uma instância do recurso R_j , uma aresta de requisição é inserida no grafo de alocação de recursos. Quando essa requisição puder ser atendida, a aresta de requisição será *instantaneamente* transformada em uma aresta de atribuição. Quando o processo não precisar mais acessar o recurso, ele irá liberá-lo; como resultado, a aresta de atribuição será excluída.

O grafo de alocação de recursos mostrado na [Figura 7.1](#) representa a seguinte situação:

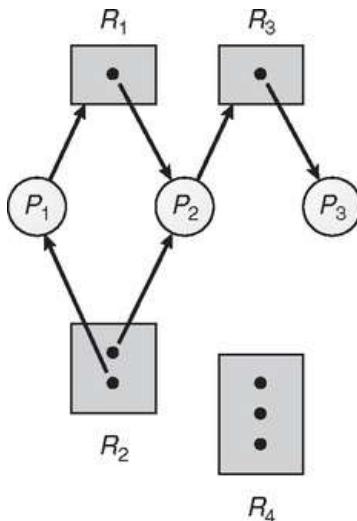


FIGURA 7.1 Grafo de alocação de recursos.

■ Os conjuntos P , R e A :

- $P = \{P_1, P_2, P_3\}$.
- $R = \{R_1, R_2, R_3, R_4\}$.
- $A = \{P_1 \rightarrow R_1, P_2 \rightarrow R_3, R_1 \rightarrow P_2, R_2 \rightarrow P_2, R_2 \rightarrow P_1, R_3 \rightarrow P_3\}$.

■ Instâncias de recurso:

- Uma instância do recurso R_1 .
- Duas instâncias do recurso R_2 .
- Uma instância do recurso R_3 .
- Três instâncias do recurso R_4 .

■ Estados de processo:

- O processo P_1 tem a posse de uma instância do recurso R_2 e está aguardando uma instância do recurso R_1 .
- O processo P_2 tem a posse de uma instância de R_1 e R_2 e está aguardando uma instância do recurso R_3 .
- O processo P_3 tem a posse de uma instância de R_3 .

Dada a definição de um grafo de alocação de recursos, pode ser mostrado que, se o grafo não contém ciclos, então nenhum processo no sistema está em deadlock. Se o grafo tiver um ciclo, pode haver um deadlock.

Se cada recurso tiver uma instância, então um ciclo indica que ocorreu um deadlock. Se o ciclo envolver apenas um conjunto de recurso, cada qual com apenas uma única instância, então ocorreu um deadlock. Cada processo envolvido no ciclo está em deadlock. Nesse caso, um ciclo no grafo é uma condição necessária e suficiente para a existência do deadlock.

Se cada recurso tiver várias instâncias, então um ciclo não significa que ocorreu um deadlock. Nesse caso, um ciclo no grafo é uma condição necessária mas não suficiente para a existência de deadlock.

Para ilustrar esse conceito, vamos retornar ao grafo de alocação de recursos representado na Figura 7.1. Suponha que o processo P_3 requisite uma instância do recurso R_2 . Como nenhuma instância de recurso está disponível, uma aresta de requisição $P_3 \rightarrow R_2$ é acrescentada ao grafo (Figura 7.2). Nesse ponto, existem dois ciclos mínimos no sistema:

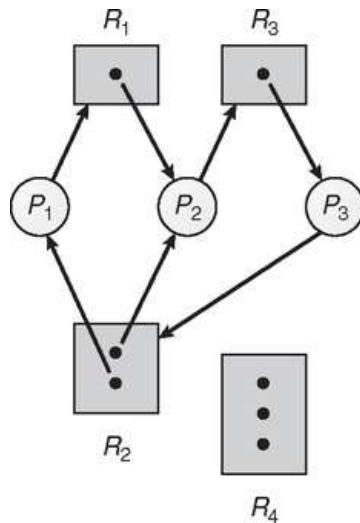


FIGURA 7.2 Grafo de alocação de recursos com um deadlock.

$$\begin{aligned} P_1 &\rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1 \\ P_2 &\rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2 \end{aligned}$$

Os processos P_1 , P_2 e P_3 estão em deadlock. O processo P_2 está aguardando o recurso R_3 , em posse do processo P_3 . O processo P_3 está aguardando que o processo P_1 ou o processo P_2 libere o recurso R_2 . Além disso, o processo P_1 está aguardando o processo P_2 liberar o recurso R_1 .

Agora, considere o grafo de alocação de recursos na [Figura 7.3](#). Nesse exemplo, também temos um ciclo

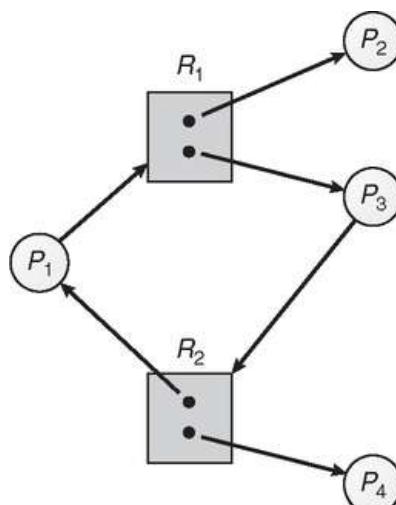


FIGURA 7.3 Grafo de alocação de recursos com um ciclo, mas sem deadlock.

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

Entretanto, não existe deadlock. Observe que o processo P_4 pode liberar sua instância do recurso R_2 . Esse recurso pode, então, ser alocado a P_3 , rompendo o ciclo.

Resumindo, se um grafo de alocação de recursos não tiver um ciclo, então o sistema *não* está em estado de deadlock. Se houver um ciclo, então o sistema pode ou não estar em estado de deadlock.

Essa observação é importante quando lidamos com o problema de deadlock.

7.2.2.1 Deadlock em um programa Java multithreads

```
public static void main(String arg[]) {  
    Lock lockX = new ReentrantLock();  
    Lock lockY = new ReentrantLock();  
  
    Thread threadA = new Thread(new A(lockX,lockY));  
    Thread threadB = new Thread(new B(lockX,lockY));  
  
    threadA.start();  
    threadB.start();  
}
```

FIGURA 7.5 Criando threads (continuação da [Figura 7.4](#)).

Antes de prosseguirmos para uma discussão do tratamento de deadlocks, vejamos como o deadlock pode ocorrer em um programa Java multithreads, como mostra a [Figura 7.4](#). Nesse exemplo, temos duas threads – `threadA` e `threadB` –, além de dois locks reentrantes – `first` e `second`. (Lembre-se, do [Capítulo 6](#), que um lock reentrante atua como um lock simples de exclusão mútua.) Neste exemplo, `threadA` tenta adquirir os locks na ordem (1) `first`, (2) `second`; `threadB` tenta usar a ordem (1) `second`, (2) `first`. O deadlock é possível no seguinte cenário:

```

class A implements Runnable
{
    private Lock first, second;

    public A(Lock first, Lock second) {
        this.first = first;
        this.second = second;
    }

    public void run( ) {
        try {
            first.lock( );
            // faz alguma coisa
            second.lock( );
            // faz algo mais
        }
        finally {
            first.unlock( );
            second.unlock( );
        }
    }
}

class B implements Runnable
{
    private Lock first, second;

    public A(Lock first, Lock second) {
        this.first = first;
        this.second = second;
    }

    public void run( ) {
        try {
            second.lock( );
            // faz alguma coisa
            first.lock( );
            // faz algo mais
        }
        finally {
            second.unlock( );
            first.unlock( );
        }
    }
}

public class DeadlockExample {
    // Figura 7.5
}

```

FIGURA 7.4 Exemplo de deadlock.

threadA → second → threadB → first → threadA

Observe que, embora o deadlock seja possível, ele não ocorrerá se threadA for capaz de obter e liberar os locks do first e second antes de threadB tentar obter os locks. Esse exemplo ilustra um problema com o tratamento de deadlocks: é difícil identificar e testar os deadlocks que podem ocorrer somente sob certas circunstâncias.

7.3 Métodos para tratamento de deadlocks

De modo geral, podemos lidar com o problema de deadlock de três maneiras:

- Podemos usar um protocolo para prevenir ou evitar deadlocks, garantindo que o sistema *nunca* entrará em estado de deadlock.
- Podemos permitir que o sistema entre em estado de deadlock, detectá-lo e recuperar.
- Podemos ignorar o problema e fingir que os deadlocks nunca ocorrem no sistema.

A terceira solução é aquela utilizada pela maioria dos sistemas operacionais, incluindo UNIX e Windows. A JVM também não faz nada para controlar os deadlocks. Fica a critério do desenvolvedor da aplicação escrever programas que tratem dos deadlocks nesses sistemas.

A seguir, vamos fazer uma breve descrição de cada um desses três métodos para tratamento de deadlocks. Depois, nas [Seções 7.4 a 7.7](#), apresentaremos algoritmos detalhados. Contudo, antes de prosseguir, devemos mencionar que alguns pesquisadores argumentam que nenhuma das técnicas básicas isoladas é apropriada para o espectro inteiro de problemas de alocação de recursos nos sistemas operacionais. As técnicas básicas podem ser combinadas, permitindo a seleção de uma técnica ideal para cada classe de recursos em um sistema.

7.3.1 Três métodos principais

Para garantir que os deadlocks nunca ocorrerão, o sistema pode usar um esquema para prevenir ou evitar deadlock. **Prevenção de deadlock** é um conjunto de métodos para garantir que pelo menos uma das condições necessárias ([Seção 7.2.1](#)) não poderá ser satisfeita. Esses métodos previnem deadlocks restringindo o modo como as requisições de recursos podem ser feitas. Discutiremos esses métodos na [Seção 7.4](#).

Evitar deadlock exige que o sistema operacional receba com antecedência informações adicionais com relação a quais recursos um processo requisitará e usará durante seu tempo de vida. Com esse conhecimento adicional, podemos decidir, para cada requisição, se o processo deve ou não esperar. Para decidir se a requisição atual pode ser satisfeita ou precisa ser adiada, o sistema precisa considerar os recursos disponíveis, os recursos alocados a cada processo e as requisições e liberações futuras de cada processo. Discutiremos esses esquemas na [Seção 7.5](#).

Se um sistema não emprega um algoritmo para prevenção ou para evitar deadlock, então uma situação de deadlock poderá ocorrer. Nesse ambiente, o sistema pode prover um algoritmo que examina o estado do sistema para determinar se ocorreu um deadlock e um algoritmo para se recuperar do deadlock (se um deadlock tiver ocorrido). Discutimos essas questões nas [Seções 7.6 e 7.7](#).

Na ausência de algoritmos para detecção e recuperação de deadlock, então podemos chegar a uma situação em que o sistema está em estado de deadlock mas não tem como reconhecer o que aconteceu. Nesse caso, o deadlock não detectado resultará em degradação do desempenho do sistema, pois os recursos estão sendo mantidos por processos que não podem ser executados e porque mais e mais processos, à medida que fizerem requisições dos recursos, entrarão em estado de deadlock. Por fim, o sistema deixará de funcionar e precisará ser reiniciado manualmente.

Embora esse método possa não ser uma técnica viável para o problema do deadlock, ele é usado na maioria dos sistemas operacionais, conforme já mencionamos. Em muitos sistemas, os deadlocks ocorrem com pouca frequência (digamos, uma vez por ano); assim, esse método é menos dispendioso do que os métodos para prevenir, evitar ou detectar e recuperar, que precisam ser usados constantemente. Além disso, em algumas circunstâncias, um sistema está em estado “congelado”, mas não em estado de deadlock. Vemos essa situação, por exemplo, com um processo de tempo real sendo executado na mais alta prioridade (ou qualquer processo sendo executado em um escalonador não preemptivo) e nunca retornando o controle para o sistema operacional. O sistema precisa ter métodos de recuperação manuais para essas condições sem deadlock e pode usar essas técnicas também para a recuperação do deadlock.

7.3.2 Lidando com deadlocks no Java

Como já observamos, a JVM não faz nada para controlar os deadlocks; fica a critério do desenvolvedor da aplicação a escrita de programas sem deadlock. No restante desta seção, vamos ilustrar como o deadlock é possível quando se usam métodos selecionados da API central da Java, e como o programador pode desenvolver programas que tratem do deadlock de forma apropriada.

No [Capítulo 4](#), apresentamos as threads Java e parte da API que permite que os usuários criem e manipulem threads. Dois métodos adicionais da classe Thread, os métodos `suspend()` e `resume()`, foram marcados para serem descontinuados nas próximas versões da Java porque podem levar ao deadlock. (Um método descontinuado indica que o método ainda faz parte da API Java, mas seu uso é desencorajado.)

O método `suspend()` suspende a execução da thread executada. O método `resume()` retoma a execução de uma thread suspensa. Quando uma thread tiver sido suspensa, ela só poderá continuar

se outra thread a retomar. Além do mais, uma thread suspensa continua a manter todos os locks enquanto está bloqueada. O deadlock é possível se uma thread suspensa mantiver um lock de um objeto e a thread que pode retomá-la precisa ter a posse desse lock antes de poder retomar a thread suspensa.

Outro método, `stop()`, também foi marcado para ser descontinuado, mas não porque pode levar ao deadlock. Ao contrário da situação em que uma thread foi suspensa, quando uma thread tiver terminado sua execução, ela libera todos os locks que possui. Contudo, os locks em geral são usados na seguinte sequência: (1) obter o lock; (2) acessar uma estrutura de dados compartilhada; e (3) liberar o lock. Se uma thread estiver no meio da etapa 2 quando for terminada, ela liberará o lock, mas pode deixar a estrutura de dados em um estado incoerente. Na [Seção 4.5.2](#), explicamos como terminar uma thread usando o cancelamento adiado em vez do cancelamento assíncrono de uma thread usando o método `stop()`. Aqui, apresentamos uma estratégia para suspender e retomar uma thread sem usar os métodos desaprovados `suspend()` e `resume()`.

O programa mostrado na [Figura 7.6](#) é um applet multithreads, que exibe a hora do dia. Quando esse applet é iniciado, ele cria uma segunda thread (que chamaremos *thread do relógio*), que mostra a hora do dia. O método `run()` da thread do relógio alterna entre dormir por um segundo e depois chamar o método `repaint()`. O método `repaint()`, por fim, chama o método `paint()`, que desenha a data e hora atuais na janela do navegador.

```

import java.applet.*;
import java.awt.*;

public class ClockApplet extends Applet implements Runnable
{
    private Thread clockThread;
    private boolean ok = false;
    private Object mutex = new Object();

    public void run( ) {
        while (true) {
            try {
                // dorme por 1 segundo
                Thread.sleep(1000);

                // agora reapresenta a data e a hora
                repaint( );

                // vê se precisamos nos suspender
                synchronized (mutex) {
                    while (ok == false)
                        mutex.wait( );
                }
            }
            catch (InterruptedException e) {}
        }
    }

    public void start( ) {
        // Figura 7.7
    }

    public void stop( ) {
        // Figura 7.7
    }

    public void paint(Graphics g) {
        g.drawString(new java.util.Date( ).toString( ), 10, 30);
    }
}

```

FIGURA 7.6 Applet que exibe a data e a hora do dia.

Esse applet foi projetado de modo que a thread do relógio esteja executando enquanto o applet está visível; se o applet não estiver sendo exibido (por exemplo, quando a janela do navegador for minimizada), a thread do relógio é suspensa de sua execução. Isso é feito pela substituição dos métodos `start()` e `stop()` da classe `Applet`. (Cuidado para não confundir esses métodos com `start()`

e `stop()` da classe `Thread`.) O método `start()` de um applet é chamado quando um applet é criado. Se o usuário sair da página Web, se o applet sair da tela ou se a janela do navegador for minimizada, o método `stop()` do applet será chamado. Se o usuário retornar à página Web do applet, o método `start()` do applet será chamado novamente.

O applet usa uma variável booleana `ok` para indicar se a thread do relógio pode ser executada ou não. A variável será definida como verdadeira no método `start()` do applet, indicando que a thread do relógio pode ser executada. O método `stop()` do applet a definirá como falsa. A thread do relógio verificará o valor dessa variável booleana em seu método `run()` e não prosseguirá se for verdadeira. Como a thread para o applet e a thread do relógio estarão compartilhando essa variável, o acesso a ela será controlado por um bloco `synchronized`. Esse programa aparece na [Figura 7.7](#).

```
/*
 * Este método é chamado quando o applet
 * é iniciado ou retornamos a ele.
 */
public void start( ) {
    ok = true;

    if (clockThread == null ) {
        clockThread = new Thread(this);
        clockThread.start( );
    }
    else {
        synchronized(mutex) {
            mutex.notify( );
        }
    }
}

/*
 * Este método é chamado quando saímos da
 * página onde o applet se encontra.
 */
public void stop( ) {
    synchronized(mutex) {
        ok = false;
    }
}
```

FIGURA 7.7 Métodos `start()` e `stop()` para o applet (continuação da [Figura 7.6](#)).

Se a thread do relógio descobrir que o valor booleano é falso, ela se suspenderá chamando o método `wait()` para o objeto `mutex`. Quando o applet quiser retomar a thread do relógio, ele definirá a variável booleana como verdadeira e chamará `notify()` para o objeto `mutex`. Essa chamada a `notify()` desperta a thread do relógio. Ela verifica o valor da variável booleana e, vendo que ela agora é verdadeira, prossegue no seu método `run()`, exibindo a data e a hora.

7.4 Prevenção de deadlock

Conforme observamos na [Seção 7.2.1](#), para ocorrer um deadlock cada uma das quatro condições necessárias precisa ser atendida. Garantindo que pelo menos uma dessas condições não possa ser atendida, podemos *prevenir* a ocorrência de um deadlock. Elaboramos essa técnica examinando cada uma das quatro condições necessárias separadamente.

7.4.1 Exclusão mútua

A condição de exclusão mútua precisa ser mantida para recursos não compartilháveis. Por exemplo, uma impressora não pode ser compartilhada simultaneamente por vários processos. Recursos compartilháveis, ao contrário, não exigem acesso por exclusão mútua e, portanto, não podem estar envolvidos em um deadlock. Arquivos somente de leitura são um bom exemplo de um recurso compartilhado. Se vários processos tentarem abrir um arquivo somente de leitura ao mesmo tempo, eles poderão receber acesso simultâneo ao arquivo. Um processo nunca precisa esperar por um recurso compartilhado. No entanto, em geral, não podemos prevenir deadlocks negando a condição de exclusão mútua, pois alguns recursos são intrinsecamente não compartilháveis.

7.4.2 Manter e esperar

Para garantir que a condição **manter e esperar** nunca ocorra no sistema, precisamos garantir que, sempre que um processo requisitar um recurso, ele não manterá quaisquer outros recursos. Um protocolo que pode ser usado exige que cada processo requisite e receba a alocação de todos os seus recursos antes de iniciar sua execução. Podemos implementar essa provisão exigindo que as chamadas de sistema que requisitam recursos para um processo precedam todas as outras chamadas de sistema.

Um protocolo alternativo permite que um processo requisite recursos apenas quando não tiver nenhum outro. Um processo pode requisitar alguns recursos e utilizá-los. Todavia, antes de ele poder requisitar quaisquer recursos adicionais, ele precisa liberar todos os recursos alocados atualmente.

Para ilustrar a diferença entre esses dois protocolos, consideramos um processo que copia dados de uma unidade de DVD para um arquivo de disco, classifica o arquivo de disco e depois imprime os resultados em uma impressora. Se todos os recursos tiverem de ser requisitados no início do processo, então o processo precisa requisitar a unidade de DVD, o arquivo de disco e a impressora. Ele manterá a impressora por toda a sua execução, embora só precise dela no final.

O segundo método permite ao processo requisitar apenas a unidade de DVD e o arquivo de disco. Ele copia da unidade de DVD para o disco e depois libera a unidade de DVD e o arquivo de disco. O processo precisa, então, requisitar novamente o arquivo de disco e a impressora. Depois de copiar o arquivo de disco para a impressora, ele libera esses dois recursos e termina.

Esses dois protocolos possuem duas desvantagens principais. Primeiro, a utilização de recursos pode ser baixa, pois os recursos podem estar alocados, mas não são utilizados por um período longo. No exemplo dado, podemos liberar a unidade de DVD e o arquivo de disco e depois requisitar novamente o arquivo de disco e a impressora, mas somente se pudermos ter certeza de que nossos dados permanecerão no arquivo de disco. Se não for possível, então precisamos requisitar todos os recursos no início dos dois protocolos.

Segundo, é possível haver starvation. Um processo que precisa de vários recursos populares pode ter de esperar indefinidamente, pois pelo menos um dos recursos de que precisa sempre estará alocado a algum outro processo.

7.4.3 Não preempção

A terceira condição necessária para os deadlocks é não haver preempção dos recursos já alocados. Para garantir que essa condição não seja satisfeita, podemos usar o protocolo a seguir. Se um processo estiver mantendo alguns recursos e requisitar outro recurso que não possa ser alocado imediatamente a ele (ou seja, o processo precisa esperar), então todos os recursos retidos serão preemptados. Em outras palavras, esses recursos serão implicitamente liberados. Os recursos preemptados são acrescentados à lista de recursos pelos quais o processo está esperando. O processo será reiniciado somente quando puder reaver seus recursos antigos, assim como ao receber os novos que está requisitando.

Como alternativa, se um processo requisitar alguns recursos, primeiro verificamos se estão disponíveis. Se estiverem, nós os alocamos. Se não estiverem, verificamos se estão alocados a algum outro processo que esteja esperando por recursos adicionais. Nesse caso, preemptamos os recursos desejados, vindos do processo que está esperando e os alocamos ao processo requisitante. Se os recursos não estiverem disponíveis nem mantidos por um processo esperando, o processo requisitante terá de esperar. Enquanto está esperando, alguns dos recursos podem ser

preemptados, mas somente se outro processo os requisitar. Um processo só pode ser reiniciado quando tiver alocado os recursos que está requisitando e recuperar quaisquer recursos preemptados enquanto estava esperando.

Esse protocolo é aplicado a recursos cujo estado pode ser salvo e restaurado mais tarde, como registros de CPU e espaço de memória. Ele, em geral, não pode ser aplicado a recursos como impressoras e unidades de fita.

7.4.4 Espera circular

A quarta e última condição para deadlocks é a condição de espera circular. Um modo de garantir que essa condição nunca aconteça é impor uma ordenação total de todos os tipos de recursos e exigir que cada processo requisite recursos em uma ordem crescente de enumeração.

Para ilustrar, consideramos que $R = \{R_1, R_2, \dots, R_m\}$ seja o conjunto de tipos de recursos. Atribuímos a cada recurso um número inteiro exclusivo, que permita comparar dois recursos e determinar se um precede o outro em nossa ordenação. Formalmente, definimos uma função F para um $F: R \rightarrow N$, onde N é o conjunto dos números naturais. Por exemplo, se o conjunto de recursos R incluir unidades de fita, unidades de disco e impressoras, então a função F poderá ser definida da seguinte forma:

$$\begin{aligned}F(\text{unidade de fita}) &= 1 \\F(\text{unidade de disco}) &= 5 \\F(\text{impressora}) &= 12\end{aligned}$$

Agora, podemos considerar o seguinte protocolo para prevenir deadlocks: cada processo só pode requisitar recursos em ordem crescente de enumeração, ou seja, um processo pode requisitar qualquer quantidade de instâncias de um recurso - digamos, R_i . Depois disso, o processo pode requisitar instâncias do recurso R_j se e somente se $F(R_j) > F(R_i)$. Por exemplo, usando a função definida anteriormente, um processo que deseja usar a unidade de fita e a impressora ao mesmo tempo terá primeiro de requisitar a unidade de fita e depois requisitar a impressora. Como alternativa, podemos exigir que, sempre que um processo requisitar uma instância do recurso R_j , ele tenha liberado quaisquer recursos R_i tais que $F(R_i) \geq F(R_j)$. Observe também que se várias instâncias do mesmo recurso forem necessárias, uma *única* requisição para todas elas deverá ser emitida.

Se esses dois protocolos forem usados, então a condição de espera circular não poderá ser satisfeita. Podemos demonstrar esse fato supondo existir uma espera circular (prova por contradição). Seja o conjunto de processos envolvidos na espera circular $\{P_0, P_1, \dots, P_n\}$, onde P_i está esperando pelo recurso R_i , mantido pelo processo P_{i+1} . (O módulo aritmético é usado nos índices, de modo que P_n está esperando por um recurso R_0 mantido por P_0 .) Em seguida, como o processo P_{i+1} está mantendo o recurso R_i , enquanto estivermos requisitando o recurso R_{i+1} , precisamos ter $F(R_i) < F(R_{i+1})$, para todo i . Mas essa condição significa que $F(R_0) < F(R_1) < \dots < F(R_n) < F(R_0)$. Pela transitividade, $F(R_0) < F(R_0)$, o que é impossível. Portanto, não pode haver espera circular.

Podemos conseguir esse esquema em uma aplicação Java desenvolvendo uma ordenação entre todos os objetos de sincronização no sistema. Todas as requisições de objetos de sincronização precisam ser feitas em ordem crescente. Por exemplo, se a ordenação de lock no programa Java mostrado na [Figura 7.4](#) fosse

$$\begin{aligned}F(\text{first}) &= 1 \\F(\text{second}) &= 5\end{aligned}$$

então a threadB não poderia requisitar os locks fora de ordem.

Lembre-se de que o desenvolvimento de uma ordenação (ou hierarquia) por si só não previne o deadlock. Fica a critério dos desenvolvedores de aplicação escrever programas que sigam a ordenação. Observe também que a função F deve ser definida de acordo com a ordem normal de uso dos recursos em um sistema. Por exemplo, como a unidade de fita é necessária antes da impressora, seria razoável definir $F(\text{unidade de fita}) < F(\text{impressora})$.

Para desenvolver uma ordenação de lock, programadores Java são encorajados a usar o método `System.identityHashCode()`, que retorna o valor que seria retornado pelo valor default do método `hashCode()` do objeto. Por exemplo, para obter o valor de `identityHashCode()` para os locks `first` e `second` no programa Java mostrado na [Figura 7.4](#), você usaria as seguintes instruções:

```

int firstOrderingValue = System.
identityHashCode(first);
int secondOrderingValue = System.
identityHashCode(second);

```

Embora assegurar que os recursos sejam obtidos na ordem correta seja responsabilidade dos desenvolvedores de aplicação, certo software poderá ser usado para verificar se os locks são obtidos na ordem apropriada e gerar avisos apropriados quando os locks forem obtidos fora da ordem, quando poderá haver deadlock. Um verificador de ordem de lock, que funciona em versões BSD do UNIX, como FreeBSD, é conhecido como **witness**. Witness usam locks de exclusão mútua que protegem seções críticas, conforme descrevemos no [Capítulo 6](#); ele atua mantendo dinamicamente o relacionamento das ordens dos locks em um sistema. Vamos usar o programa mostrado na [Figura 7.4](#) como exemplo. Suponha que `threadA` seja o primeiro a obter os locks e faça isso na ordem (1) `first`, (2) `second`. A witness registra o relacionamento de que `first` precisa ser adquirido antes de `second`. Se `threadB`, mais tarde, obtiver os locks fora de ordem, a testemunha gerará uma mensagem de aviso no console do sistema.

Finalmente, é importante observar que a imposição de uma ordenação de lock não garante a prevenção de deadlock se os locks pudermos ser adquiridos dinamicamente. Por exemplo, suponha que tenhamos um método que transfere fundos entre duas contas. Para evitar uma condição de corrida, usamos o lock de objeto associado a cada objeto `Account` em um bloco `synchronized`. O código se parece com o seguinte:

```

void transaction(Account from, Account to,
double amount) {
    synchronized(from) {
        synchronized(to) {
            from.withdraw(amount);
            to.deposit(amount);
        }
    }
}

```

O deadlock é possível se duas threads chamarem simultaneamente o método `transaction()`, transpondo diferentes contas. Ou seja, uma thread poderia chamar

```
transaction(checkingAccount, savingsAccount, 25);
```

e outra poderia chamar

```
transaction(savingsAccount, checkingAccount, 50);
```

Vamos deixar como exercício para o leitor, descobrir uma solução para corrigir essa situação.

7.5 Evitar deadlock

Os algoritmos de prevenção de deadlock, discutidos na [Seção 7.4](#), previnem os deadlocks limitando o modo como as requisições são feitas. Os limites garantem que pelo menos uma das condições necessárias para o deadlock não possa ocorrer e, portanto, que os deadlocks não possam ser mantidos. Contudo, os possíveis efeitos colaterais de prevenir deadlocks por esse método são a baixa utilização de dispositivos e a redução do throughput do sistema.

Um método alternativo para evitar deadlocks é exigir informações adicionais sobre como os recursos devem ser requisitados. Por exemplo, em um sistema com uma unidade de fita e uma impressora, poderíamos ser informados de que o processo P requisitará primeiro a unidade de fita e depois a impressora, antes de liberar os dois recursos. Entretanto, o processo Q requisitará primeiro a impressora e depois a unidade de fita. Com esse conhecimento da sequência completa de requisições e liberações para cada processo, o sistema pode decidir, para cada requisição, se o processo deverá ou não esperar, para evitar um possível deadlock futuro. Cada requisição exige que, ao tomar essa decisão, o sistema considere os recursos disponíveis, os recursos alocados a cada processo e as requisições e liberações futuras de cada processo.

Os diversos algoritmos que usam essa técnica diferem na quantidade e no tipo de informações exigidas. O modelo mais simples e mais útil exige que cada processo declare o *número máximo* de recursos de cada tipo que possa ser necessário. Dada essa informação *a priori*, é possível construir um algoritmo que garanta que o sistema nunca entrará em estado de deadlock. Um algoritmo para evitar deadlock examina dinamicamente o estado de alocação de recursos para garantir que a condição de espera circular nunca exista. O *estado* de alocação de recursos é definido pela quantidade de recursos disponíveis e alocados e pelas demandas máximas dos processos. Nas próximas seções, exploraremos dois algoritmos para evitar deadlock.

7.5.1 Estado seguro

Um estado é *seguro* se o sistema puder alocar recursos a cada processo (até o seu máximo) em alguma ordem e ainda evitar um deadlock. Mais formalmente, um sistema está em estado seguro somente se houver uma **sequência segura**. Uma sequência de processos $\langle P_1, P_2, \dots, P_n \rangle$ é uma sequência segura para o estado de alocação atual se, para cada P_i , o recurso requisitar que P_i ainda possa ser satisfeito pelos recursos disponíveis mais os recursos mantidos por todo P_j , com $j < i$. Nessa situação, se os recursos de que o processo P_i precisar não estiverem disponíveis, então P_i poderá esperar até que todo P_j tenha terminado. Quando tiverem terminado, P_i poderá obter todos os recursos necessários, completar sua tarefa designada, retornar seus recursos alocados e terminar. Quando P_i terminar, P_{i+1} poderá obter seus recursos necessários, e assim por diante. Se não houver tal sequência, o estado do sistema será considerado *inseguro*.

Um estado seguro não é um estado com deadlock. Ao contrário, um estado com deadlock é um estado inseguro. Contudo, nem todos os estados inseguros são deadlocks ([Figura 7.8](#)). Um estado inseguro *pode* ocasionar um deadlock. Desde que o estado seja seguro, o sistema operacional poderá evitar estados inseguros (e com deadlock). Em um estado inseguro, o sistema operacional não pode evitar que os processos requisitem recursos de modo que ocorra um deadlock. O comportamento dos processos controla os estados inseguros.

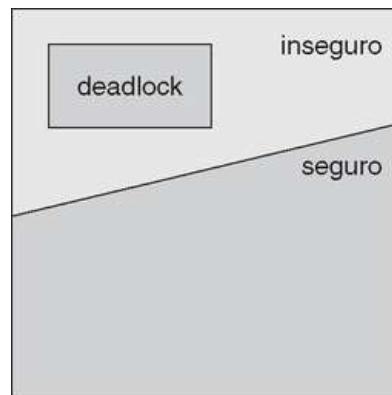


FIGURA 7.8 Espaço dos estados seguro, inseguro e com deadlock.

Para ilustrar, consideramos um sistema com 12 unidades de fita magnética e três processos: P_0 , P_1 e P_2 . O processo P_0 exige 10 unidades de fita, o processo P_1 pode precisar de até 4, e o processo P_2 pode precisar de até 9 unidades de fita. Suponha que, no instante t_0 , o processo P_0 retenha 5

unidades de fita, o processo P_1 retenha 2, e o processo P_2 retenha 2 unidades de fita. (Assim, existem três unidades de fita livres.)

No instante t_0 , o sistema está em estado seguro. A sequência $\langle P_1, P_0, P_2 \rangle$ satisfaz a condição de segurança. O processo P_1 pode alocar todas as unidades de fita e depois retorná-las (o sistema terá 5 unidades de fita disponíveis), depois o processo P_0 pode apanhar todas as unidades de fita e retorná-las (o sistema terá 10 unidades de fita disponíveis) e, finalmente, P_2 poderá apanhar todas as unidades de fita e retorná-las (o sistema terá todas as 12 unidades de fita disponíveis).

	Necessidades máximas	Necessidades atuais
P_0	10	5
P_1	4	2
P_2	9	2

Um sistema pode passar de um estado seguro para um estado inseguro. Suponha que, no instante t_1 , o processo P_2 requisite e receba mais uma unidade de fita. O sistema não está mais em um estado seguro. Nesse ponto, somente o processo P_1 pode receber todas as unidades de fita. Quando ele as retornar, o sistema terá apenas 4 unidades de fita disponíveis. Como o processo P_0 recebeu 5 unidades de fita, mas tem um máximo de 10, ele pode requisitar mais 5 unidades de fita. Se ele fizer isso, como elas não estão disponíveis, o processo P_0 tem de aguardar. De modo semelhante, o processo P_2 pode requisitar mais 6 unidades de fita e ter de aguardar, resultando em um deadlock. Nossa erro foi conceder a requisição do processo P_2 por mais uma unidade de fita. Se tivéssemos feito P_2 aguardar até que um dos outros processos tivesse terminado e liberado seus recursos, poderíamos ter evitado o deadlock.

Dado o conceito de estado seguro, podemos definir os algoritmos para evitar deadlocks que garantem que o sistema nunca estará em condição de deadlock. A ideia é garantir que o sistema sempre permanecerá em estado seguro. A princípio, o sistema está em estado seguro. Sempre que um processo requisitar um recurso disponível, o sistema terá de decidir se o recurso pode ser alocado imediatamente ou se o processo precisa esperar. A requisição só é concedida se a alocação sair do sistema em estado seguro.

Nesse esquema, se um processo requisitar um recurso disponível, ele ainda poderá ter de esperar. Assim, a utilização de recursos pode ser menor do que seria sem um algoritmo para evitar deadlock.

7.5.2 Algoritmo do grafo de alocação de recursos

Se tivermos um sistema de alocação de recursos com somente uma instância de cada recurso, uma variante do grafo de alocação de recursos, definido na [Seção 7.2.2](#), poderá ser usada para evitar deadlock. Além das arestas de requisição e atribuição, apresentamos uma nova aresta, chamada **aresta de pretensão**. Uma aresta de pretensão $P_i \rightarrow R_j$ indica que o processo P_i pode requisitar o recurso R_j em algum momento no futuro. Essa aresta é semelhante a uma aresta de requisição na direção, mas é representada no grafo por uma linha tracejada. Quando o processo P_i requisita o recurso R_j , a aresta de pretensão $P_i \rightarrow R_j$ é convertida para uma aresta de requisição. De modo semelhante, quando um recurso R_j é liberado por P_i , a aresta de atribuição $R_j \rightarrow P_i$ é retornada para uma aresta de pretensão $P_i \rightarrow R_j$.

Observamos que os recursos precisam ser pretendidos *a priori* no sistema, ou seja, antes de o processo P_i iniciar a execução todas as arestas de pretensão já precisam aparecer no grafo de alocação de recursos. Podemos relaxar essa condição permitindo que uma aresta de pretensão $P_i \rightarrow R_j$ seja acrescentada ao grafo somente se todas as arestas associadas ao processo P_i forem arestas de pretensão.

Suponha agora que o processo P_i requisite o recurso R_j . A requisição só pode ser concedida se a conversão da aresta de requisição $P_i \rightarrow R_j$ para uma aresta de atribuição $R_j \rightarrow P_i$ não resultar na formação de um ciclo no grafo de alocação de recursos. Observe que verificamos a segurança usando um algoritmo de detecção de ciclo. Um algoritmo para detectar um ciclo nesse grafo exige uma ordem de n^2 operações, onde n é o número de processos no sistema.

Se não existir um ciclo, então a alocação do recurso deixará o sistema em estado seguro. Se for encontrado um ciclo, então a alocação colocará o sistema em estado inseguro. Portanto, o processo P_i terá que esperar até suas requisições serem satisfeitas.

Para ilustrar esse algoritmo, consideramos o grafo de alocação de recursos da [Figura 7.9](#). Suponha que P_2 requisite R_2 . Embora R_2 esteja livre, não podemos alocá-lo a P_2 , pois essa ação criará um ciclo no grafo ([Figura 7.10](#)). Como já dissemos, um ciclo indica que o sistema está em estado inseguro. Se P_1 requisitar R_2 , e P_2 requisitar R_1 , haverá um deadlock.

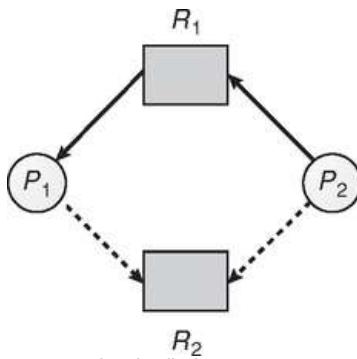


FIGURA 7.9 Grafo de alocação de recursos para evitar deadlock.

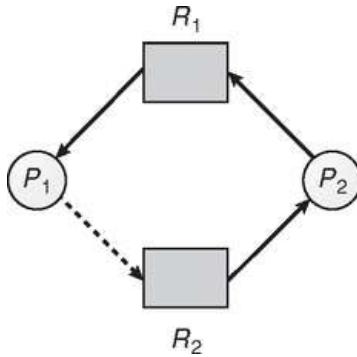


FIGURA 7.10 Um estado inseguro em um grafo de alocação de recursos.

7.5.3 Algoritmo do banqueiro

O algoritmo do grafo de alocação de recursos não se aplica a um sistema de alocação de recursos com múltiplas instâncias de cada recurso. O algoritmo para evitar deadlock que descrevemos a seguir pode ser aplicado a tal sistema, mas é menos eficiente do que o esquema do grafo de alocação de recursos. Esse algoritmo é conhecido como *algoritmo do banqueiro*. O nome foi escolhido porque o algoritmo poderia ser usado em um sistema bancário para garantir que o banco nunca aloque seus caixas disponíveis de modo que não possa mais satisfazer as necessidades de todos os seus clientes.

Quando um novo processo entra no sistema, ele precisa declarar o número máximo de instâncias de cada recurso que pode precisar. Esse número não pode ultrapassar o número total de recursos no sistema. Quando um usuário requisita um conjunto de recursos, o sistema precisa determinar se a alocação desses recursos deixará o sistema em estado seguro. Se deixar, os recursos são alocados; caso contrário, o processo terá de esperar até algum outro processo liberar recursos suficientes.

Diversas estruturas de dados precisam ser mantidas para implementar o algoritmo do banqueiro. Essas estruturas de dados codificam o estado do sistema de alocação de recursos. Precisamos das seguintes estruturas de dados, onde n é o número de processos no sistema e m o número de tipos de recursos:

- **Disponível.** Um vetor de tamanho m indica o número de recursos disponíveis de cada tipo. Se $Disponível[j]$ for igual a k , então haverá k instâncias disponíveis do recurso R_j .
- **Máximo.** Uma matriz $n \times m$ define a demanda máxima de cada processo. Se $Máximo[i][j]$ for igual a k , então o processo P_i pode requisitar no máximo k instâncias do recurso R_j .
- **Alocação.** Uma matriz $n \times m$ define o número de recursos de cada tipo alocado a cada processo. Se $Alocação[i][j]$ for igual a k , então o processo P_i terá alocado k instâncias do recurso R_j .
- **Necessário.** Uma matriz $n \times m$ indica a necessidade de recursos restantes de cada processo. Se $Necessário[i][j]$ for igual a k , então o processo P_i pode precisar de k mais instâncias do recurso R_j para completar sua tarefa. Observe que $Necessário[i][j]$ é igual a $Máximo[i][j] - Alocação[i][j]$.

Essas estruturas de dados variam com o tempo, tanto no tamanho quanto no valor.

Para simplificar a apresentação do algoritmo do banqueiro, estabelecemos agora alguma notação. Sejam X e Y vetores de tamanho n . Dizemos que $X \leq Y$ se e somente se $X[i] \leq Y[i]$ para todo $i = 1, 2, \dots, n$. Por exemplo, se $X = (1, 7, 3, 2)$ e $Y = (0, 3, 2, 1)$, então $Y \leq X$. Além disso, $Y < X$ se $Y \leq X$ e $Y \neq X$.

Podemos tratar cada linha nas matrizes *Alocação* e *Necessário* como vetores e referenciá-los como *Alocação* e *Necessário*. O vetor $Alocação_i$ especifica os recursos alocados ao processo P_i ; o vetor $Necessário_i$ especifica os recursos adicionais que o processo P_i ainda poderá requisitar para

completar sua tarefa.

7.5.3.1 Algoritmo de segurança

Podemos agora apresentar o algoritmo para descobrir se um sistema está ou não em estado seguro. O algoritmo pode ser descrito da seguinte maneira:

1. Sejam $Trabalho$ e Fim vetores de tamanho m e n , respectivamente. Inicialize $Trabalho = Disponível$ e $Fim[i] = \text{false}$ para $i = 0, 1, \dots, n - 1$.

2. Encontre um i tal que
 - a. $Fim[i] == \text{false}$
 - b. $Necessário_i \leq Trabalho$

Se não houver tal i , vá para a etapa 4.

3. $Trabalho = Trabalho + Alocação_i$, $Fim[i] = \text{true}$ Vá para a etapa 2.

4. Se $Fim[i] == \text{true}$ para todo i , então o sistema está em estado seguro.

Esse algoritmo pode exigir uma ordem de $m \times n^2$ operações para determinar se um estado é seguro.

7.5.3.2 Algoritmo de requisição de recursos

Em seguida descrevemos o algoritmo que determina se as requisições podem ser concedidas em segurança.

Seja $Requisição_i$ o vetor de requisição para o processo P_i . Se $Requisição_i[j] == k$, então o processo P_i deseja k instâncias do recurso R_j . Quando uma requisição de recursos for feita para o processo P_i , as seguintes ações são realizadas:

1. Se $Requisição_i \leq Necessário_i$, vá para a etapa 2. Caso contrário, levante uma condição de erro, pois o processo ultrapassou sua pretensão máxima.
2. Se $Requisição_i \leq Disponível$, vá para a etapa 3. Caso contrário, P_i precisa esperar, pois os recursos não estão disponíveis.
3. Faça o sistema fingir ter alocado os recursos requisitados ao processo P_i , modificando o estado da seguinte maneira:

$$\begin{aligned} Disponível &= Disponível - Requisição_i; \\ Alocação_i &= Alocação_i + Requisição_i; \\ Necessário_i &= Necessário_i - Requisição_i; \end{aligned}$$

Se o estado de alocação de recursos resultante for seguro, a transação será completada e o processo P_i receberá a alocação de seus recursos. Entretanto, se o novo estado for inseguro, P_i terá de esperar por $Requisição_i$ e o antigo estado de alocação de recursos será restaurado.

7.5.3.3 Um exemplo ilustrativo

Para ilustrar o uso do conceito do algoritmo do banqueiro, considere um sistema com cinco processos, de P_0 a P_4 , e três tipos de recursos, A , B e C . O tipo de recurso A possui 10 instâncias, o tipo de recurso B possui 5 instâncias e o tipo de recurso C possui 7 instâncias. Suponha que, no instante T_0 , o seguinte instantâneo do sistema tenha sido tirado:

	<i>Alocação</i>	<i>Máximo</i>	<i>Disponível</i>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
P_0	0 1 0	7 5 3	3 3 2
P_1	2 0 0	3 2 2	
P_2	3 0 2	9 0 2	
P_3	2 1 1	2 2 2	
P_4	0 0 2	4 3 3	

O conteúdo da matriz $Necessário$ é definido como sendo $Máximo - Alocação$, e é como a seguir:

<i>Necessário</i>	<i>A B C</i>
P_0	7 4 3
P_1	1 2 2
P_2	6 0 0
P_3	0 1 1
P_4	4 3 1

Pretendemos que o sistema esteja em estado seguro. Na realidade, a sequência $\langle P_1, P_3, P_4, P_2, P_0 \rangle$

satisfaz os critérios de segurança. Suponha agora que o processo P_1 requisite uma instância adicional do recurso A e duas instâncias do recurso C , de modo que $Requisição_1 = (1,0,2)$. Para decidir se essa requisição pode ser concedida, primeiro verificamos se $Requisição_1 \leq Disponível$ – ou seja, $(1,0,2) \leq (3,3,2)$, o que é verdadeiro. Depois, fingimos que a requisição tenha sido atendida e chegamos ao novo estado a seguir:

	<i>Alocação</i>	<i>Necessário</i>	<i>Disponível</i>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
P_0	0 1 0	7 4 3	2 3 0
P_1	3 0 2	0 2 0	
P_2	3 0 2	6 0 0	
P_3	2 1 1	0 1 1	
P_4	0 0 2	4 3 1	

Precisamos determinar se esse novo estado do sistema é seguro. Para isso, executamos nosso algoritmo de segurança e descobrimos que a sequência $\langle P_1, P_3, P_4, P_0, P_2 \rangle$ satisfaz nosso requisito de segurança. Logo, podemos conceder a requisição ao processo P_1 .

No entanto, você poderá ver que quando o sistema está nesse estado, uma requisição para $(3,3,0)$ por P_4 não poderá ser concedida, pois os recursos não estão disponíveis. Além do mais, uma requisição para $(0,2,0)$ por P_0 não pode ser concedida, embora os recursos estejam disponíveis, pois o estado resultante é inseguro.

Deixamos a implementação do algoritmo do banqueiro como um exercício de programação.

7.6 Detecção de deadlock

Conforme observamos, se um sistema não empregar um algoritmo para prevenção de deadlock nem para evitar deadlock, então uma situação de deadlock poderá ocorrer. Nesse ambiente, o sistema precisa prover:

- Um algoritmo que examine o estado do sistema para determinar se ocorreu um deadlock.
- Um algoritmo para se recuperar do deadlock.

Na discussão a seguir, elaboramos esses dois requisitos relacionados com os sistemas com apenas uma instância isolada de cada recurso, bem como aos sistemas com várias instâncias de cada recurso. Nesse ponto, porém, vamos observar que um esquema de detecção e recuperação exige o custo adicional que inclui não apenas os custos em tempo de execução de manter as informações necessárias e executar o algoritmo de detecção, mas também as perdas em potencial inerentes à recuperação de um deadlock.

7.6.1 Única instância de cada recurso

Se todos os recursos tiverem apenas uma única instância, poderemos definir um algoritmo de detecção de deadlock que usa uma variante do grafo de alocação de recursos, chamada grafo *wait-for*. Obtemos esse grafo do grafo de alocação de recursos, removendo os nós de recursos e colapsando as arestas apropriadas.

Mais precisamente, uma borda de P_i a P_j em um grafo *wait-for* implica que o processo P_i está esperando o processo P_j liberar um recurso de que P_i precisa. Existe uma aresta $P_i \rightarrow P_j$ em um grafo *wait-for* se e somente se o grafo de alocação de recursos correspondente tiver duas arestas $P_i \rightarrow R_q$ e $R_q \rightarrow P_j$ para algum recurso R_q . Por exemplo, na [Figura 7.11](#), apresentamos um grafo de alocação de recursos e o grafo *wait-for* correspondente.

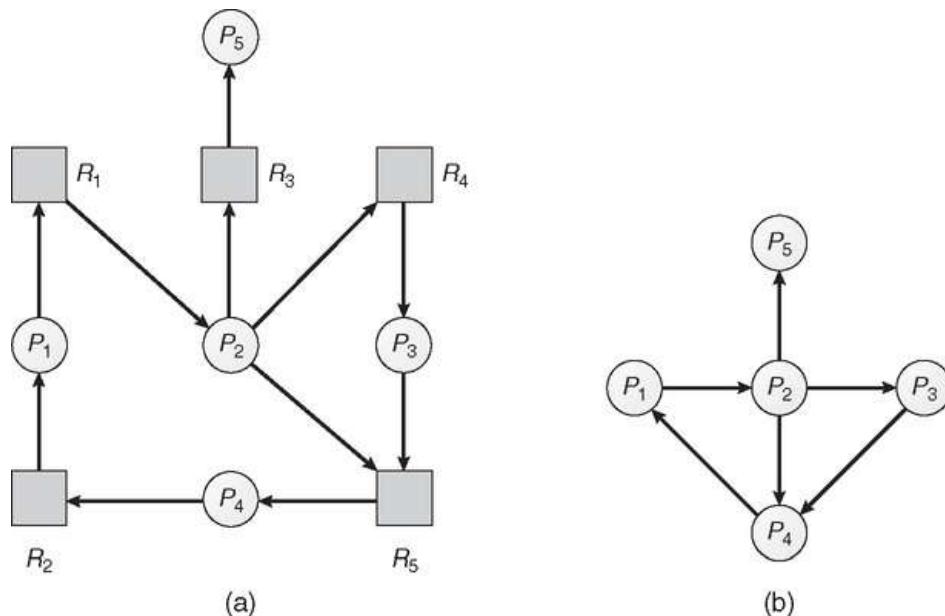


FIGURA 7.11 (a) Grafo de alocação de recursos e (b) grafo *wait-for* correspondente.

Como antes, existe um deadlock no sistema se e somente se o grafo *wait-for* tiver um ciclo. Para detectar deadlocks, o sistema precisa *manter* o grafo *wait-for* e *chamar um algoritmo* periodicamente para procurar um ciclo no grafo. Um algoritmo para detectar um ciclo em um grafo exige uma ordem de n^2 operações, onde n é a quantidade de vértices no grafo.

7.6.2 Várias instâncias de um tipo de recurso

O esquema do grafo *wait-for* não se aplica a um sistema de alocação de recursos com várias instâncias de cada recurso. Agora, passamos a um algoritmo de detecção de recursos que se aplica a tal sistema. O algoritmo emprega diversas estruturas de dados variáveis com o tempo, semelhantes àquelas usadas no algoritmo do banqueiro ([Seção 7.5.3](#)):

- **Disponível.** Um vetor de tamanho m indica o número de recursos disponíveis de cada tipo.
- **Alocação.** Uma matriz $n \times m$ define o número de recursos de cada tipo alocados a cada processo.

■ **Requisição.** Uma matriz $n \times m$ indica a requisição atual de cada processo. Se $Requisição[i][j]$ for igual a k , então o processo P_i está requisitando k mais instâncias do recurso R_j .

A relação \leq entre dois vetores é definida da mesma forma que na [Seção 7.5.3](#). Para simplificar a notação, novamente tratamos as linhas nas matrizes *Alocação* e *Requisição* como vetores; vamos nos referir a elas como *Alocação_i* e *Requisição_i*, respectivamente. O algoritmo de detecção descrito aqui investiga cada sequência de alocação possível em busca dos processos que ainda precisam ser completados. Compare esse algoritmo com o algoritmo do banqueiro, da [Seção 7.5.3](#).

1. Sejam *Trabalho* e *Fim* vetores de tamanho m e n , respectivamente. Inicialize *Trabalho* = *Disponível*. Para $i = 0, 1, \dots, n - 1$, se *Alocação_i* $\neq 0$, então *Fim*[i] = *false*; caso contrário, *Fim*[i] = *true*.
2. Encontre um índice i tal que
 - a. *Fim*[i] == *false*
 - b. *Requisição_i* \leq *Trabalho*

Se não houver tal i , vá para a etapa 4.

3. *Trabalho* = *Trabalho* + *Alocação_i*. *Fim*[i] = *true*. Vá para a etapa 2.

4. Se *Fim*[i] == *false* para algum i , $0 \leq i < n$, então o sistema está em estado de deadlock. Além do mais, se *Fim*[i] == *false*, então o processo P_i está em deadlock.

Esse algoritmo requer uma ordem de operações $m \times n^2$ para detectar se o sistema encontra-se em estado de deadlock.

Você poderá perguntar por que reivindicamos os recursos do processo P_i (na etapa 3) assim que determinamos que $Requisição_i \leq Trabalho$ (na etapa 2b). Sabemos que P_i não está envolvido em um deadlock (pois $Requisição_i \leq Trabalho$). Assim, tomamos uma atitude otimista e supomos que P_i não exigirá mais recursos para completar sua tarefa; portanto, logo retornaremos todos os recursos alocados ao sistema. Se nossa suposição estiver incorreta, um deadlock poderá ocorrer mais tarde. Esse deadlock será detectado da próxima vez em que o algoritmo de detecção de deadlock for invocado.

Para ilustrar esse algoritmo, consideramos um sistema com cinco processos, de P_0 a P_4 , e três tipos de recursos A , B e C . O recurso A possui sete instâncias, o recurso B possui duas instâncias e o recurso C possui seis instâncias. Suponha que, no instante T_0 , tenhamos o seguinte estado de alocação de recursos:

	<i>Alocação</i>	<i>Requisição</i>	<i>Disponível</i>
	<i>A B C</i>	<i>A B C</i>	<i>A B C</i>
P_0	0 1 0	0 0 0	0 0 0
P_1	2 0 0	2 0 2	
P_2	3 0 3	0 0 0	
P_3	2 1 1	1 0 0	
P_4	0 0 2	0 0 2	

Afirmamos que o sistema não se encontra em estado de deadlock. Na realidade, se executarmos nosso algoritmo, veremos que a sequência $\langle P_0, P_2, P_3, P_1, P_4 \rangle$ resulta em *Fim*[i] == *true* para todo i .

Suponha agora que o processo P_2 faça uma requisição adicional para uma instância do tipo C . A matriz de *Requisição* é modificada da seguinte forma:

	<i>Alocação</i>
	<i>A B C</i>
P_0	0 0 0
P_1	2 0 2
P_2	0 0 1
P_3	1 0 0
P_4	0 0 2

Afirmamos que o sistema agora se encontra em deadlock. Embora possamos retomar os recursos mantidos pelo processo P_0 , a quantidade de recursos disponíveis não é suficiente para atender às requisições dos outros processos. Assim, existe um deadlock, consistindo nos processos P_1 , P_2 , P_3 e P_4 .

7.6.3 Uso do algoritmo de detecção

Quando devemos invocar o algoritmo de detecção? A resposta depende de dois fatores:

1. Com que *frequência* provavelmente ocorrerá um deadlock?
2. *Quantos* processos serão afetados pelo deadlock quando ele acontecer?

Se os deadlocks ocorrem com frequência, então o algoritmo de detecção deverá ser invocado também com frequência. Os recursos alocados a processos com deadlock serão ociosos até ele poder ser desfeito. Além disso, a quantidade de processos envolvidos no ciclo de deadlock poderá

crescer.

Os deadlocks ocorrem somente quando algum processo faz uma requisição que não pode ser concedida. Essa requisição pode ser a requisição final, que completa uma cadeia de processos esperando. No caso extremo, portanto, poderíamos invocar o algoritmo de detecção de deadlock toda vez que uma requisição de alocação não puder ser concedida imediatamente. Nesse caso, podemos identificar não apenas o conjunto de processos que estão em deadlock, mas também o processo específico que “causou” o deadlock. (Na realidade, cada um dos processos em deadlock é um elo no ciclo do grafo de recursos, de modo que todos eles, em conjunto, causaram o deadlock.) Se houver muitos tipos de recursos diferentes, uma requisição poderá causar muitos ciclos no grafo de recursos, cada ciclo completado pela requisição mais recente e “causado” por um processo identificável.

Naturalmente, a chamada do algoritmo de detecção de deadlock para cada requisição gerará um custo adicional considerável em tempo de computação. Uma alternativa menos dispendiosa é envolver o algoritmo em intervalos menos frequentes – por exemplo, uma vez por hora ou sempre que a utilização da CPU cair para menos de 40%. (Um deadlock, por fim, degradará o throughput do sistema e fará a utilização da CPU cair.) Se o algoritmo de detecção for chamado em momentos arbitrários, o grafo de recursos poderá conter muitos ciclos. Nesse caso, em geral, não podemos dizer quais dos muitos processos em deadlock “causaram” o deadlock.

7.7 Recuperação do deadlock

O que acontece quando um algoritmo de detecção determina que existe um deadlock? Várias alternativas estão disponíveis. Uma possibilidade é informar ao operador que ocorreu um deadlock e deixar que o operador trate do deadlock manualmente. Outra possibilidade é deixar o sistema se *recuperar* do deadlock automaticamente. Existem duas opções para desfazer um deadlock. Uma é abortar um ou mais processos para interromper a espera circular. A outra é preemptar alguns recursos de um ou mais processos em deadlock.

7.7.1 Término do processo

Para eliminar deadlocks abortando um processo, usamos um dentre dois métodos. Nos dois métodos, o sistema reivindica todos os recursos alocados aos processos terminados.

- **Abortar todos os processos em deadlock.** Esse método desfará o ciclo de deadlock, mas com um custo alto; os processos em deadlock podem ter sido executados por muito tempo, e os resultados da computação parcial precisam ser descartados e provavelmente terão de ser refeitos mais tarde.
- **Abortar um processo de cada vez até que o ciclo de deadlock seja eliminado.** Esse método incorre em um custo adicional considerável, visto que, depois de cada processo ser abortado, um algoritmo de detecção de deadlock precisa ser invocado para determinar se quaisquer processos ainda estão em deadlock.

Abortar um processo pode não ser fácil. Se o processo estiver no meio da atualização de um arquivo, seu cancelamento deixará esse arquivo em estado incorreto. De modo semelhante, se o processo estiver no meio da impressão de dados em uma impressora, o sistema terá de reiniciar a impressora para um estado correto antes de imprimir a próxima tarefa.

Se for utilizado o método do término parcial, então temos de determinar quais processos em deadlock devem ser terminados. Essa determinação é uma decisão política, semelhante às decisões de escalonamento de CPU. A questão é basicamente econômica; temos de abortar aqueles processos cujo término incorrerá no custo mínimo. Infelizmente, o termo *custo mínimo* não é exato. Muitos fatores podem afetar qual processo é escolhido, incluindo:

1. Qual é a prioridade do processo.
2. Por quanto tempo o processo esteve em execução e de quanto tempo mais o processo precisará antes de completar sua tarefa designada.
3. Quantos e que recursos o processo usou (por exemplo, se os recursos são simples de preemptar).
4. De quantos mais recursos o processo precisará para terminar.
5. Quantos processos precisarão ser terminados.
6. Se o processo é interativo ou batch.

7.7.2 Preempção de recursos

Para eliminar deadlocks usando a preempção de recursos, apropriamos alguns recursos dos processos com sucesso e entregamos esses recursos a outros processos, até o ciclo de deadlock ser desfeito.

Se a preempção tiver de lidar com deadlocks, três questões precisarão ser resolvidas:

1. **Seleção de uma vítima.** Quais recursos e quais processos devem ser preemptados? Assim como no término do processo, temos de determinar a ordem da preempção para minimizar o custo. Os fatores de custo podem incluir parâmetros como número de recursos que um processo em deadlock está mantendo e o tempo que o processo consumiu até aqui durante sua execução.
2. **Reversão (Rollback).** Se nos apropriarmos de um recurso de um processo, o que deverá ser feito com esse processo? Logicamente, ele não pode continuar com sua execução normal; ele não possui algum recurso necessário. Temos de efetuar o rollback do processo até algum estado seguro e reiniciá-lo nesse estado.
Como, em geral, é difícil determinar qual é um estado seguro, a solução mais simples é um rollback total. Aborte o processo e depois reinicie-o. Embora seja mais eficiente efetuar o rollback do processo somente até o ponto necessário para desfazer o deadlock, esse método exige que o sistema mantenha mais informações sobre o estado de todos os processos em execução.
3. **Starvation.** Como garantimos que a starvation não ocorrerá? Ou seja, como podemos garantir que nem sempre os recursos do mesmo processo serão preemptados?
Em um sistema em que a seleção da vítima se baseia principalmente em fatores de custo, pode acontecer que o mesmo processo sempre seja escolhido como vítima. Como resultado, esse processo nunca completará sua tarefa designada, uma situação de starvation que precisa ser enfrentada por qualquer sistema prático. Logicamente, temos de garantir que um processo poderá ser escolhido como vítima apenas por um número finito (pequeno) de vezes. A solução

mais comum é incluir o número de rollbacks no fator de custo.

7.8 Resumo

Um estado de deadlock ocorre quando dois ou mais processos estão esperando indefinidamente por um evento que só pode ser causado por um dos processos esperando. Existem três métodos principais para lidar com os deadlocks:

- Usar algum protocolo para prevenir ou evitar deadlocks, garantindo que o sistema nunca entrará em estado de deadlock.
- Permitir que o sistema entre no estado de deadlock, detectá-lo e depois recuperar.
- Ignorar o problema e fingir que os deadlocks nunca ocorrem no sistema.

A terceira solução é utilizada pela maioria dos sistemas operacionais, incluindo UNIX e Windows.

Um deadlock só pode ocorrer se quatro condições necessárias forem satisfeitas simultaneamente no sistema: exclusão mútua, manter e esperar, não preempção e espera circular. Para prevenir deadlocks, podemos garantir que pelo menos uma das condições necessárias nunca seja satisfeita.

Um método para evitar deadlocks, menos rigoroso do que os algoritmos de prevenção, exige que o sistema operacional tenha informações *a priori* sobre como cada processo utilizará os recursos. O algoritmo do banqueiro, por exemplo, exige informações *a priori* sobre o número máximo de cada classe de recurso que pode ser requisitada em cada processo. Usando essas informações, podemos definir um algoritmo para evitar deadlock.

Se um sistema não empregar um protocolo para garantir que os deadlocks nunca ocorrerão, um esquema de detecção e recuperação poderá ser empregado. Um algoritmo de detecção de deadlock precisa ser invocado para determinar se ocorreu um deadlock. Se um deadlock for detectado, o sistema terá de se recuperar terminando alguns de seus processos em deadlock ou apropriando-se dos recursos de alguns dos processos em deadlock.

Onde a preempção for usada para cuidar dos deadlocks, três questões precisam ser focalizadas: seleção de uma vítima, reversão e starvation. Em um sistema que seleciona vítimas para efetuar o rollback principalmente com base nos fatores de custo, a starvation poderá ocorrer, e o processo selecionado nunca completará sua tarefa designada.

Os pesquisadores argumentaram que nenhuma das técnicas básicas isoladamente é apropriada para o espectro inteiro de problemas de alocação de recurso nos sistemas operacionais. As técnicas básicas podem ser combinadas, permitindo a seleção de uma técnica ideal para cada classe de recursos em um sistema.

Exercícios práticos

- 7.1. Liste três exemplos de deadlocks que não estão relacionados com um ambiente de sistema de computação.
- 7.2. Suponha que um sistema esteja em um estado inseguro. Mostre que é possível que os processos concluam sua execução sem entrar em um estado de deadlock.
- 7.3. Uma solução possível para prevenir os deadlocks é ter um único recurso de ordem mais alta que deva ser requisitado antes de qualquer outro recurso. Por exemplo, se várias threads tentarem acessar os objetos de sincronização $A \dots E$, o deadlock é possível. (Esses objetos de sincronização podem incluir mutexes, semáforos, variáveis de condição e coisas desse tipo.) Podemos impedir o deadlock acrescentando um sexto objeto F . Sempre que uma thread quiser obter o lock para qualquer objeto $A \dots E$, ela primeiro terá de obter o lock do objeto F . Essa solução é conhecida como **contenção**: os locks dos objetos $A \dots E$ estão contidos dentro do lock do objeto F . Compare esse esquema com o esquema de espera circular da [Seção 7.4.4](#).
- 7.4. Prove que o algoritmo de segurança apresentado na [Seção 7.5.3](#) requer uma ordem de $m \times n^2$ operações.
- 7.5. Considere um sistema de computação que executa 5.000 jobs por mês e não possui esquema de impedimento de deadlock ou prevenção de deadlock. Os deadlocks ocorrem cerca de duas vezes por mês, e o operador precisa terminar e retornar cerca de 10 jobs por deadlock. Cada job custa cerca de US\$ 2 (em tempo de CPU) e os jobs terminados costumam estar prontos pela metade quando são abortados.
- Um programador de sistemas estimou que um algoritmo para evitar deadlocks (como o algoritmo do banqueiro) poderia ser instalado no sistema, com um aumento no tempo médio de execução por job de cerca de 10%. Como a máquina atualmente possui 30% do tempo ocioso, todos os 5.000 jobs por mês ainda poderiam ser executados, embora o tempo de revezamento aumentasse em cerca de 20%, em média.
- Quais são os argumentos a favor da instalação do algoritmo para evitar deadlocks?
 - Quais são os argumentos contra a instalação do algoritmo para evitar deadlocks?
- 7.6. Um sistema pode detectar que partes de seus processos estão sofrendo de starvation? Se a sua resposta for “sim”, explique como isso é possível. Se a resposta for “não”, explique como o sistema pode lidar com o problema de starvation.
- 7.7. Considere a seguinte política de alocação de recursos. As requisições e liberações de recursos são permitidas a qualquer momento. Se uma requisição de recursos não puder ser satisfeita porque os recursos não estão disponíveis, então verificamos quaisquer processos que estão bloqueados aguardando recursos. Se um processo bloqueado tiver os recursos desejados, então esses recursos são retirados dele e dados ao processo solicitante. O vetor de recursos pelos quais o processo bloqueado está aguardando é aumentado para incluir os recursos que foram tomados.
- Por exemplo, considere um sistema com três tipos de recursos e o vetor *Disponível* inicializado com $(4,2,2)$. Se o processo P_0 solicitar $(2,2,1)$, ele os recebe. Se P_1 solicitar $(1,0,1)$, ele os recebe. Depois, se P_0 solicitar $(0,0,1)$, ele é bloqueado (recurso não disponível). Se P_2 agora solicitar $(2,0,0)$, ele recebe aquele disponível $(1,0,0)$ e aquele que foi alocado a P_0 (pois P_0 está bloqueado). O vetor *Alocação* de P_0 diminui para $(1,2,1)$ e seu vetor *Requisição* sobe para $(1,0,1)$.
- Pode haver um deadlock? Se a sua resposta for “sim”, dê um exemplo. Se a sua resposta for “não”, especifique que condição necessária não pode ocorrer.
 - Pode haver um bloqueio indefinido? Explique sua resposta.
- 7.8. Suponha que você tenha codificado o algoritmo de segurança para evitar deadlock e agora precise implementar um algoritmo de detecção de deadlock. Você poderia fazer isso usando o código do algoritmo de segurança e redefinindo $Máximo[i] = Esperando[i] + Alocação[i]$, onde *Esperando[i]* é um vetor que especifica os recursos pelos quais o processo i está esperando e *Alocação[i]* é definido conforme na [Seção 7.5](#)? Explique sua resposta.
- 7.9. É possível ter um deadlock envolvendo apenas um único processo? Explique sua resposta.

Exercícios

7.10. Considere o deadlock de trânsito representado na [Figura 7.12](#).

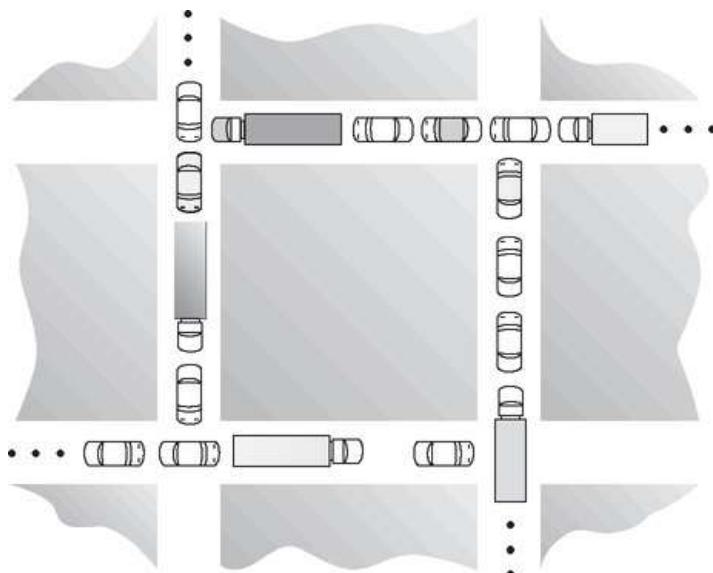


FIGURA 7.12 Deadlock de trânsito para o Exercício 7.10

- a. Mostre que as quatro condições necessárias para o deadlock são atendidas nesse exemplo.
 - b. Indique uma regra simples para evitar deadlocks nesse sistema.
- 7.11. Considere a situação de deadlock que poderia ocorrer no problema dos filósofos à mesa de jantar quando eles apanham os hashis um de cada vez. Discuta como as quatro condições necessárias para o deadlock se mantêm nessas condições. Discuta como os deadlocks poderiam ser evitados eliminando-se qualquer uma das quatro condições necessárias.
- 7.12. Compare o esquema de espera circular com os diversos esquemas para evitar deadlock (como o algoritmo do banqueiro) com relação às seguintes questões:
- a. Overheads de runtime
 - b. Throughput do sistema
- 7.13. Em um sistema computadorizado real, nem os recursos disponíveis nem as demandas dos processos pelos recursos são coerentes por longos períodos (meses). Os recursos falham ou são substituídos, novos processos vêm e vão, novos recursos são comprados e acrescentados ao sistema. Se o deadlock for controlado pelo algoritmo do banqueiro, quais das seguintes mudanças podem ser feitas com segurança (sem introduzir a possibilidade de deadlock) e sob quais circunstâncias?
- a. Aumentar *Disponível* (novos recursos acrescentados).
 - b. Diminuir *Disponível* (recurso removido permanentemente do sistema).
 - c. Aumentar *Máximo* para um processo (o processo precisa de mais recursos do que o permitido).
 - d. Diminuir *Máximo* para um processo (o processo decide que não precisa de tantos recursos).
 - e. Aumentar o número de processos.
 - f. Diminuir o número de processos.
- 7.14. Considere um sistema consistindo em quatro recursos do mesmo tipo, que são compartilhados por três processos, cada um precisando no máximo de dois recursos. Mostre que o sistema está livre de deadlock.
- 7.15. Considere um sistema consistindo em m recursos do mesmo tipo sendo compartilhados por n processos. Os recursos podem ser requisitados e liberados pelos processos somente um de cada vez. Mostre que o sistema está livre de deadlock se as duas condições a seguir forem atendidas:
- a. A necessidade máxima de cada processo está entre 1 e m recursos.
 - b. A soma de todas as necessidades máximas é menor que $m + n$.
- 7.16. A API Java para a classe `Thread` contém um método `destroy()` que foi descontinuado. Consultando a API, explique por que `destroy()` foi descontinuado.
- 7.17. Considere o seguinte instantâneo de um sistema:

	Alocação	Máximo	Disponível
	ABCD	ABCD	ABCD
P_0	0012	0012	1520
P_1	1000	1750	
P_2	1354	2356	
P_3	0632	0652	
P_4	0014	0656	

Responda às seguintes perguntas usando o algoritmo do banqueiro:

- a. Qual é o conteúdo da matriz *Necessário*?
 - b. O sistema está em estado seguro?
 - c. Se uma requisição do processo P_1 chegar para $(0,4,2,0)$, a requisição poderá ser concedida imediatamente?
- 7.18. Podemos obter o algoritmo do banqueiro para um único tipo de recurso do algoritmo do banqueiro geral reduzindo o dimensionamento dos diversos arrays por 1. Mostre, por meio de um exemplo, que não podemos implementar o esquema do banqueiro para múltiplos tipos de recursos pela aplicação do esquema de único recurso a cada recurso individualmente.
- 7.19. Considere o problema do jantar dos filósofos em que os hashis são colocados no centro da mesa e dois deles quaisquer poderiam ser usados por um filósofo. Considere que as requisições de hashis sejam feitas uma de cada vez. Descreva uma regra simples para determinar se determinada requisição poderia ser satisfeita sem causar deadlock, dada a alocação atual de hashis aos filósofos.
- 7.20. Considere novamente o esquema do exercício anterior. Agora, considere que cada filósofo exige três hashis para comer e que as requisições de recursos ainda sejam emitidas separadamente. Descreva algumas regras simples para determinar se determinada requisição poderia ser satisfeita sem causar deadlock, dada a alocação atual de hashis aos filósofos.
- 7.21. Qual é a suposição otimista feita no algoritmo de detecção de deadlock? Como essa suposição poderia ser violada?

Problemas de programação

- 7.22. Na [Seção 7.4.4](#), descrevemos uma situação em que impedimos o deadlock garantindo que todos os locks sejam adquiridos em determinada ordem. Porém, também explicamos que o deadlock é possível nessa situação se os locks forem adquiridos dinamicamente e duas threads chamarem simultaneamente o método `transaction()`. Conserte o método `transaction()` para impedir a ocorrência de deadlock. (Dica: considere o uso de `System.identityHashCode()`.)
- 7.23. Uma ponte de pista única conecta as duas vilas de Vermont, de North Tunbridge e South Tunbridge. Os fazendeiros nas duas vilas utilizam essa ponte para entregar sua produção na cidade vizinha. A ponte pode entrar em um impasse se um fazendeiro do norte e outro do sul entrarem nela ao mesmo tempo (os fazendeiros de Vermont são teimosos e incapazes de recuar). Usando semáforos Java ou a sincronização Java, crie um algoritmo que impeça o deadlock.
 Para testar sua implementação, projete duas threads, uma representando um fazendeiro indo para o norte e a outra representando um fazendeiro indo para o sul. Quando ambos estiverem sobre a ponte, cada um dormirá por um período de tempo aleatório para simular a travessia da ponte. Inicialmente, não se preocupe com starvation (a situação em que os fazendeiros do norte impedem que os fazendeiros do sul usem a ponte ou vice-versa).
- 7.24. Modifique sua solução para o [Exercício 7.23](#) de modo que fique livre de starvation.

Projetos de programação

Neste projeto, você escreverá um programa Java que implemente o algoritmo do banqueiro, discutido na [Seção 7.5.3](#). Vários clientes requisitam e liberam recursos do banco. O banqueiro concederá uma requisição apenas se ela deixar o sistema em um estado seguro. Uma requisição é negada se deixar o sistema em estado inseguro.

O banco

O banco empregará a estratégia esboçada na [Seção 7.5.3](#), na qual considerará as requisições de n clientes por m recursos. O banco acompanhará os recursos usando as seguintes estruturas de dados:

```

int number0fCustomers; // o número de clientes
int number0fResources; // o número de recursos

int[ ] available; // a quantidade disponível
// de cada recurso
int[ ][ ] maximum; // a demanda máxima de cada
// cliente
int[ ][ ] allocation; // a quantidade atualmente
// alocada a cada cliente
int[ ][ ] need; // as necessidades restantes
// de cada cliente

```

A funcionalidade do banco aparece na interface mostrada na [Figura 7.13](#). A implementação dessa interface exigirá a inclusão de um construtor que recebe o número de recursos disponíveis inicialmente. Por exemplo, suponha que tenhamos três tipos de recursos com 10, 5 e 7 recursos disponíveis inicialmente. Nesse caso, podemos criar uma implementação da interface usando a seguinte técnica:

```

public interface Bank
{
    /**
     * Inclui um cliente
     * customerNumber - o número do cliente
     * maximumDemand - a demanda máxima para esse cliente
     */
    public void addCustomer(interface customerNumber,
                           int[ ] maximumDemand);

    /**
     * Mostra o valor de available, maximum,
     * allocation e need
     */
    public void getState( );

    /*Requisita recursos
     * customerNumber - o cliente requisitando recursos
     * request - os recursos sendo requisitados
     */
    public boolean requestResources(int customerNumber,
                                   int[ ] request);

    /**
     * Libera recursos
     * customerNumber - o cliente liberando recursos
     * release - os recursos sendo liberados
     */
    public void releaseResources(int customerNumber,
                               int[ ] release);
}

```

FIGURA 7.13 Interface mostrando a funcionalidade do banco.

```
Bank theBank = new BankImpl(10,5,7);
```

O banco concederá uma requisição se ela satisfizer o algoritmo de segurança esboçado na [Seção 7.5.3.1](#). Se a concessão da requisição não deixar o sistema em estado seguro, ela será negada.

Testando a sua implementação

Esse programa espera que a implementação da interface Bank se chame BankImpl e exige um arquivo de entrada contendo a demanda máxima de cada tipo de recurso para cada cliente. Por exemplo, se houver cinco clientes e três tipos de recursos, o arquivo de entrada poderá conter o seguinte:

```
7,5,3  
3,2,2  
9,0,2  
2,2,2  
4,3,3
```

Isso indica que a demanda máxima para o cliente₀ é 7,5,3; para o cliente₁ é 3,2,2; e assim por diante. Como cada linha do arquivo de entrada representa um cliente separado, o método addCustomer() deve ser chamado à medida que cada linha é lida, inicializando o valor de maximum para cada cliente. (No exemplo anterior, o valor de maximum[0][] é inicializado em 7,5,3 para o cliente₀, maximum[1][] é inicializado em 3,2,2, e assim por diante.)

Além do mais, TestHarness.java também exige o número inicial de recursos disponíveis no banco. Por exemplo, se houver inicialmente 10, 5 e 7 recursos disponíveis, chamamos TestHarness.java da seguinte forma:

```
java TestHarness infile.txt 10 5 7
```

onde infile.txt refere-se a um arquivo contendo a demanda máxima para cada cliente, seguida pelo número de recursos inicialmente disponíveis. O array available será inicializado com os valores passados na linha de comandos.

Notas bibliográficas

Dijkstra [1965a] foi um dos primeiros e mais influentes colaboradores para o tema do deadlock. Holt [1972] foi a primeira pessoa a formalizar a noção de deadlocks em termos de um modelo teórico de grafo, semelhante ao que apresentamos neste capítulo. O starvation foi abordado por Holt [1972]. Hyman [1985] providenciou o exemplo de deadlock da legislatura do Kansas. Um estudo mais recente de tratamento de deadlock é fornecido em Levine [2003].

Os diversos algoritmos de prevenção foram sugeridos por Havender [1968], que idealizou o esquema de ordenação de recursos para o sistema IBM OS/360.

O algoritmo do banqueiro para evitar deadlocks foi desenvolvido para um único tipo de recurso por Dijkstra [1965a] e foi estendido para múltiplos tipos de recursos por Habermann [1969]. Os Exercícios 7.14 e 7.15 são de Holt [1971].

O algoritmo de detecção de deadlock para múltiplas instâncias de um tipo de recurso, descrito na Seção 7.6.2, foi apresentado por Coffman e outros [1971].

Bach [1987] descreve quantos dos algoritmos no kernel do UNIX tradicional tratam do deadlock. As soluções para os problemas de deadlock em redes são discutidas em trabalhos como os de Culler e outros [1998] e os de Rodeheffer e Schroeder [1991].

O verificador de ordem de lock conhecido como witness é apresentado em Baldwin [2002].

PARTE III

GERÊNCIA DE MEMÓRIA

ESBOÇO

[Capítulo 10: Introdução a Gerência de memória](#)

[Capítulo 11: Memória principal](#)

[Capítulo 12: Memória virtual](#)

Introdução a Gerência de memória

A finalidade principal de um sistema computadorizado é executar programas. Esses programas, juntamente com os dados que acessam, precisam estar na memória principal (ao menos parcialmente) durante a execução.

Para melhorar a utilização da CPU e a velocidade de sua resposta aos usuários, o computador precisa manter vários processos na memória. Existem muitos esquemas de gerência de memória, refletindo várias técnicas, e a eficácia de cada algoritmo depende da situação. A seleção de um esquema de gerência de memória para um sistema depende de muitos fatores, em especial do projeto de *hardware* do sistema. A maioria dos algoritmos requer seu próprio suporte do hardware.

CAPÍTULO 8

Memória principal

No [Capítulo 5](#), mostramos como a CPU pode ser compartilhada por um conjunto de processos. Como resultado do escalonamento da CPU, podemos melhorar a utilização da CPU e o tempo de resposta do computador aos seus usuários. No entanto, para observar esse aumento no desempenho, temos de manter vários processos na memória, ou seja, temos de *compartilhar* a memória.

Neste capítulo, discutimos as diversas maneiras de gerenciar a memória. Os algoritmos de gerência de memória variam de uma técnica puramente de máquina até estratégias de paginação e segmentação. Cada técnica possui suas próprias vantagens e desvantagens. A seleção de um método de gerência de memória para um sistema específico depende de muitos fatores, em especial do projeto do *hardware* do sistema. Como veremos, muitos algoritmos exigem suporte do hardware, embora os projetos recentes tenham integrado de perto o hardware e o sistema operacional.

OBJETIVOS DO CAPÍTULO

- Fornecer uma descrição detalhada de várias maneiras de organizar o hardware de memória.
- Discutir diversas técnicas de gerência de memória, incluindo paginação e segmentação.
- Fornecer uma descrição detalhada do Intel Pentium, que admite segmentação pura e segmentação com paginação.

8.1 Conceitos básicos

Como vimos no Capítulo 1, a memória é fundamental para a operação de um sistema computadorizado moderno. A memória consiste em uma grande sequência de words ou bytes, cada uma com seu próprio endereço. A CPU apanha instruções da memória de acordo com o valor do contador de programa. Essas instruções podem ocasionar o carregamento adicional e o armazenamento em endereços específicos da memória.

Um ciclo típico de execução de instrução, por exemplo, primeiro apanha uma instrução da memória. A instrução é decodificada e pode fazer os operandos serem buscados da memória. Depois de a instrução ter sido executada sobre os operandos, os resultados podem ser armazenados de volta para a memória. A unidade de memória vê apenas um fluxo de endereços de memória; ela não sabe como são gerados (pelo contador de programa, indexação, indireção, endereços literais e assim por diante) ou para que servem (instruções ou dados). De acordo com isso, podemos ignorar como um programa gera um endereço de memória. Estamos interessados apenas na sequência de endereços de memória gerada pelo programa em execução.

Começamos nossa discussão abordando várias questões pertinentes às diversas técnicas para gerenciamento de memória. Isso inclui uma visão geral dos aspectos básicos do hardware, o vínculo entre os endereços de memória simbólicos e os endereços físicos reais e a distinção entre endereços lógicos e físicos. Concluímos com uma discussão sobre a carga dinâmica e o vínculo entre código e bibliotecas compartilhadas.

8.1.1 Hardware básico

A memória principal e os registradores embutidos no próprio processador são os únicos tipos de armazenamento acessíveis diretamente pela CPU. Existem instruções de máquina que pegam endereços de memória como argumentos, mas nenhuma pega endereços de disco. Portanto, quaisquer instruções em execução (e quaisquer dados usados pelas instruções) precisam estar em um desses dispositivos de armazenamento com acesso direto. Se os dados não estiverem na memória, eles terão de ser movidos para lá antes que a CPU possa fazer qualquer operação sobre eles.

Os registradores internos da CPU são acessíveis, em geral, em um ciclo do relógio (clock) da CPU. A maioria das CPUs pode decodificar as instruções e realizar operações simples do conteúdo no registrador, na velocidade de uma ou mais operações por clock tick. O mesmo não pode ser dito da memória principal, cujo acesso é feito por meio de uma operação no barramento de memória. O acesso à memória pode exigir muitos ciclos do relógio da CPU para completar, quando o processador normalmente precisa **protelar (stall)** suas operações, pois não tem os dados necessários para completar a instrução que está executando. Essa situação se torna intolerável, pois o acesso à memória é feito com muita frequência. A solução é acrescentar uma memória de acesso rápido entre a CPU e a memória principal. Esse buffer de memória usado para alojar um diferencial de velocidade, chamado **cache**, é descrito na Seção 1.8.3.

Estamos interessados não apenas na velocidade relativa do acesso à memória principal; também precisamos garantir a operação correta para proteger o sistema operacional contra acesso pelos programas do usuário e, além disso, proteger os programas do usuário uns dos outros. Essa proteção precisa ser fornecida pelo hardware. Ela pode ser implementada de várias maneiras, conforme veremos neste capítulo. Nesta seção, esboçamos uma implementação possível.

Primeiro precisamos garantir que cada processo tenha um espaço de memória separado, e para isso precisamos da capacidade de determinar o intervalo de endereços válidos que o programa pode acessar e garantir que o processo pode acessar apenas esses endereços válidos. Podemos fornecer esse tipo de proteção usando dois registradores, normalmente uma base e um limite, conforme ilustramos na Figura 8.1. O **registrador de base** contém o menor endereço válido de memória física; o **registrador de limite** contém o tamanho do intervalo. Por exemplo, se o registrador de base contém 300040 e o registrador de limite for 120900, então o programa poderá acessar legalmente todos os endereços desde 300040 até 420939, inclusive.

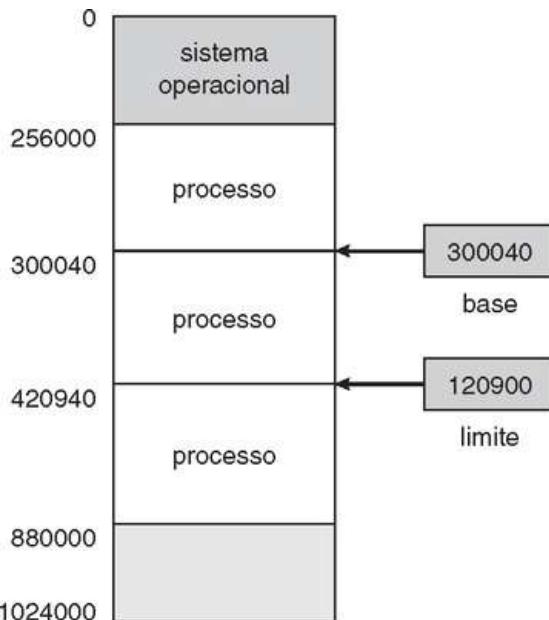


FIGURA 8.1 Um registrador de base e um de limite definem um espaço de endereços lógicos.

A proteção do espaço da memória é obtida quando o hardware da CPU compara *cada* endereço gerado no modo usuário com os registradores. Qualquer tentativa de um programa executando no modo usuário de acessar a memória do monitor ou a memória de outros usuários resulta em um trap para o monitor, que trata a tentativa como um erro fatal ([Figura 8.2](#)). Esse esquema impede que o programa do usuário (accidental ou deliberadamente) modifique o código ou as estruturas de dados do sistema operacional ou de outros usuários.

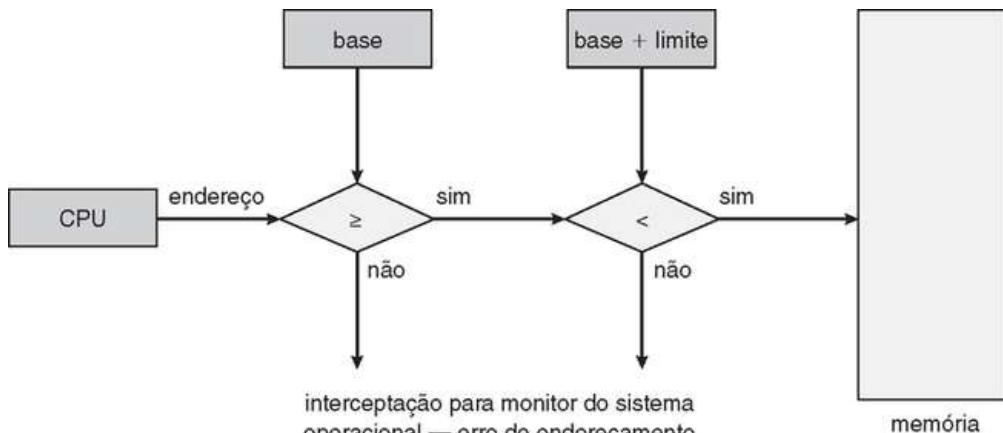


FIGURA 8.2 Proteção de endereço de hardware com registradores de base e limite.

Os registradores de base e limite só podem ser carregados pelo sistema operacional por meio de uma instrução privilegiada especial. Isso porque instruções privilegiadas só podem ser executadas no modo kernel e somente o sistema operacional pode ser executado nesse modo. Esse esquema permite que o monitor mude o valor dos registradores, mas impede que programas do usuário alterem o conteúdo dos mesmos.

Ao sistema operacional, executando no modo kernel, é dado acesso irrestrito à memória do sistema operacional e do usuário. Essa provisão permite que o sistema operacional carregue os programas dos usuários na memória dos usuários, faça o dump desses programas no caso de erros, acesse e modifique parâmetros das chamadas de sistema, e assim por diante.

8.1.2 Associação de endereços

Em geral, um programa reside em um disco como um arquivo executável binário. Para ser executado, ele precisa ser trazido para a memória e colocado dentro de um processo. Dependendo da gerência de memória em uso, o processo pode ser movido entre o disco e a memória durante sua execução. Os processos no disco que estão esperando para serem trazidos para execução na memória formam a **fila de entrada**.

O procedimento normal é selecionar um dos processos na fila de entrada e carregá-lo para a memória. À medida que o processo é executado, ele acessa instruções e dados da memória. Por fim, o processo termina e seu espaço na memória é declarado disponível.

A maioria dos sistemas permite a um processo do usuário residir em qualquer parte da memória física. Assim, embora o espaço de endereços do computador comece em 00000, o primeiro endereço do processo do usuário não precisa ser 00000. Essa técnica afeta os endereços que o programa do usuário pode utilizar. Na maioria dos casos, um programa do usuário passará por várias etapas - algumas podendo ser opcionais - antes de ser executado (Figura 8.3). Os endereços podem ser representados de diferentes maneiras durante essas etapas. Os endereços no programa-fonte costumam ser simbólicos (como *contador*). Um compilador **associará** esses endereços simbólicos a endereços relocáveis (por exemplo, "14 bytes a partir do início deste módulo"). O linkeditor ou loader (carregador) de código, por sua vez, associará os endereços relocáveis a endereços absolutos (como 74014). Cada associação é um mapeamento de um espaço de endereços em outro.

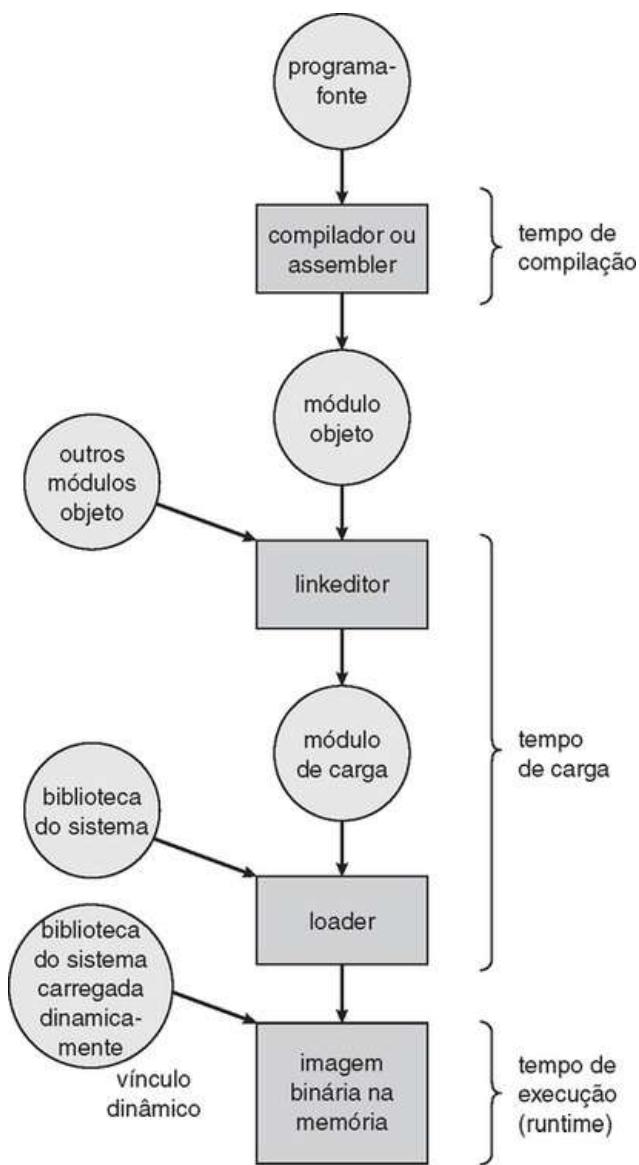


FIGURA 8.3 Processamento de um programa do usuário em múltiplas etapas.

Classicamente, a associação de instruções e dados aos endereços de memória pode ser feita em qualquer etapa ao longo do caminho:

■ **Tempo de compilação.** Se você souber, no momento da compilação, onde o processo residirá na memória, então o **código absoluto** pode ser gerado. Por exemplo, se você sabe que um processo do usuário residirá a partir do local *R*, então o código gerado pelo compilador começará nesse local e se estenderá a partir daí. Se, em algum momento depois disso, o local inicial mudar, será necessário compilar esse código novamente. Os programas no formato .COM do MS-DOS são associados no momento da compilação.

■ **Tempo de carga.** Se, no momento da compilação, não se souber onde o processo residirá na

memória, então o compilador terá de gerar um **código relocável**. Nesse caso, a associação final é deixada para o momento da carga. Se o endereço de partida mudar, só precisamos recarregar o código do usuário para incorporar esse valor alterado.

■ **Tempo de execução (runtime).** Se o processo puder ser movido durante sua execução de um segmento da memória para outro, então a associação terá de ser adiada para o momento da execução. Um hardware especial precisa estar disponível para esse esquema funcionar, conforme discutiremos na [Seção 8.1.3](#). A maioria dos sistemas operacionais de uso geral utiliza esse método.

Uma grande parte deste capítulo é dedicada a mostrar como essas diversas associações podem ser implementadas de forma eficaz em um sistema computadorizado e a discutir o suporte de hardware apropriado.

8.1.3 Espaço de endereços lógicos e físicos

Um endereço gerado pela CPU é denominado **endereço lógico**, enquanto um endereço visto pela unidade de memória - ou seja, aquele carregado no **registrador de endereços de memória** - é considerado um **endereço físico**.

Os métodos de associação de endereço em tempo de compilação e em tempo de carga geram endereços lógicos e físicos idênticos. Contudo, o esquema de associação de endereço no tempo de execução resulta em diferentes endereços lógicos e físicos. Nesse caso, normalmente nos referimos ao endereço lógico como **endereço virtual**. Usamos o *endereço lógico* e o *endereço virtual* para indicar a mesma coisa neste texto. O conjunto de todos os endereços lógicos gerados por um programa é um **espaço de endereços lógicos**; o conjunto de todos os endereços físicos correspondentes a esses endereços lógicos é um **espaço de endereços físicos**. Assim, no esquema de associação de endereço em tempo de execução, os espaços de endereços lógicos e físicos são diferentes.

O mapeamento em tempo de execução dos endereços virtuais para físicos é feito por um dispositivo de hardware chamado **unidade de gerência de memória (Memory-Management Unit - MMU)**. Podemos escolher dentre muitos métodos diferentes para realizar esse mapeamento, conforme discutimos nas [Seções 8.3 a 8.7](#). No momento, ilustramos esse mapeamento com um esquema MMU simples, uma generalização do esquema de registrador de base descrito na [Seção 8.1.1](#). O registrador de base agora é chamado de **registrador de relocação**. O valor no registrador de relocação é *somado* a cada endereço gerado por um processo do usuário no momento em que é enviado para a memória ([Figura 8.4](#)). Por exemplo, se a base estiver em 14000, então uma tentativa do usuário de endereçar o local 0 é relocada dinamicamente para o local 14000; um acesso ao local 346 é mapeado para o local 14346. O sistema operacional MS-DOS executando na família de processadores Intel 80x86 utiliza quatro registradores de relocação quando carrega e executa processos.

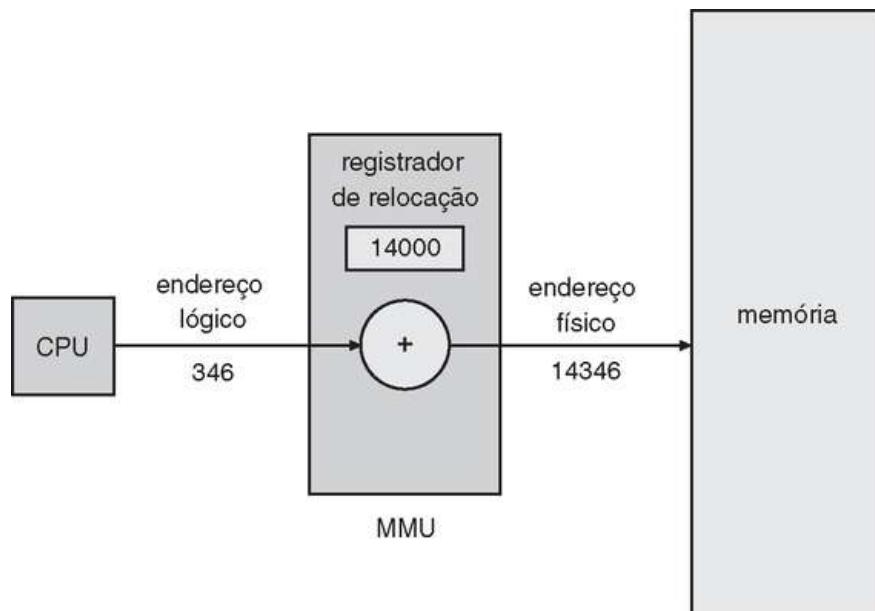


FIGURA 8.4 Relocação dinâmica usando um registrador de relocação.

O programa do usuário nunca vê os endereços físicos *reais*. O programa pode criar um ponteiro para o local 346, armazená-lo na memória, manipulá-lo e compará-lo com outros endereços - tudo como o número 346. Somente quando for usado como um endereço de memória (em um load ou

store indireto, talvez), ele será relocado em relação ao registrador de base. O programa do usuário trata de endereços *lógicos*. O hardware de mapeamento de memória converte os endereços lógicos em endereços físicos. Essa forma de associação em tempo de execução foi discutida na [Seção 8.1.2](#). O local final de um endereço de memória referenciado não é determinado antes de ser feita a referência.

Agora, temos dois tipos diferentes de endereços: endereços lógicos (no intervalo de 0 ao *máximo*) e endereços físicos (no intervalo de $R + 0$ até $R + \text{máximo}$ para um valor de base R). O usuário gera apenas endereços locais e pensa que o processo é executado nos locais de 0 ao *máximo*. Porém, esses endereços lógicos precisam ser mapeados para endereços físicos antes de serem usados.

O conceito de *espaço de endereços lógicos* associado a um *espaço de endereços físicos* separado é fundamental para a gerência de memória apropriada.

8.1.4 Carregamento dinâmico

Em nossa discussão até aqui, o programa inteiro e todos os dados de um processo precisam estar na memória física para o processo ser executado. O tamanho de um processo é, portanto, limitado ao tamanho da memória física. Para obtermos melhor utilização do espaço na memória, podemos usar o **carregamento dinâmico**. Com o carregamento dinâmico, uma rotina não é carregada até ser chamada. Todas as rotinas são mantidas no disco em um formato de carga relocável. O programa principal é carregado para a memória e executado. Quando uma rotina precisar chamar outra rotina, a rotina que chama primeiro verifica se a outra foi carregada. Se não, o loader de código relocável é chamado para carregar a rotina desejada para a memória e atualizar as tabelas de endereço do programa, para refletir essa mudança. Depois, o controle é passado para a rotina recém-carregada.

A vantagem do carregamento dinâmico é que uma rotina não utilizada nunca é carregada. Esse método é útil quando grande quantidade de código é necessária para tratar de casos que ocorrem com pouca frequência, como rotinas de erro. Nesse caso, embora o tamanho total do programa possa ser grande, a parte usada (e, portanto, carregada) pode ser muito menor.

O carregamento dinâmico não exige suporte especial do sistema operacional. É responsabilidade dos usuários projetar seus programas para tirar proveito desse método. Contudo, os sistemas operacionais podem ajudar o programador fornecendo rotinas de biblioteca para implementar o carregamento dinâmico.

8.1.5 Bibliotecas de vínculo dinâmico e compartilhadas

A [Figura 8.3](#) mostra **bibliotecas vinculadas dinamicamente**. Alguns sistemas operacionais admitem apenas o **vínculo estático**, em que as bibliotecas de linguagem do sistema são tratadas como qualquer outro módulo objeto e são combinadas pelo loader na imagem do programa binário. O conceito de vínculo dinâmico é semelhante ao do carregamento dinâmico. Todavia, aqui é o vínculo e não o carregamento que é adiado para o momento da execução. Esse recurso é usado com bibliotecas do sistema, como bibliotecas de sub-rotina da linguagem. Sem esse recurso cada programa em um sistema precisa ter uma cópia de sua biblioteca da linguagem (ou, pelo menos, das rotinas referenciadas pelo programa) incluída na imagem executável. Esse requisito desperdiça espaço de disco e memória principal.

Com o vínculo dinâmico, um *stub* é incluído na imagem para cada referência à rotina da biblioteca. O stub é um pequeno código que indica como localizar a rotina correta na biblioteca residente na memória ou como carregar a biblioteca, se a rotina ainda não estiver presente. Quando o stub é executado, ele verifica se a rotina necessária já está na memória. Se não, o programa carrega a rotina para a memória. De qualquer forma, o próprio stub é substituído pelo endereço da rotina e executa essa rotina. Assim, da próxima vez que determinado segmento de código for alcançado, a rotina da biblioteca será executada diretamente, sem haver custo algum para o vínculo dinâmico. Sob esse esquema, todos os processos que utilizam uma biblioteca da linguagem executam apenas uma cópia do código da biblioteca.

Esse recurso pode ser estendido para atualizações da biblioteca (como reparos de bugs). Uma biblioteca pode ser substituída por uma nova versão, e todos os programas que referenciam a biblioteca usarão automaticamente a nova versão. Sem o vínculo dinâmico, todos esses programas precisariam ser vinculados para obter acesso à nova biblioteca. Para os programas não executarem acidentalmente novas versões incompatíveis das bibliotecas, a informação sobre a versão é incluída no programa e na biblioteca. Mais de uma versão de uma biblioteca poderá ser carregada na memória, e cada programa utiliza sua informação de versão para decidir qual cópia da biblioteca usará. Pequenas mudanças retêm o mesmo número de versão, enquanto grandes mudanças incrementam o número da versão. Assim, somente os programas compilados com a nova versão da biblioteca são afetados pelas mudanças incompatíveis incorporadas a ela. Outros programas vinculados antes da instalação da nova biblioteca continuarão usando a biblioteca mais antiga. Esse sistema também é conhecido como **bibliotecas compartilhadas**.

Ao contrário do carregamento dinâmico, o vínculo dinâmico em geral exige ajuda do sistema

operacional. Se os processos na memória estiverem protegidos uns dos outros, o sistema operacional será a única entidade que poderá verificar se a rotina necessária está no espaço de memória de outro processo ou se pode permitir que vários processos acessem os mesmos endereços de memória. Explicaremos esse conceito com mais detalhes quando discutirmos sobre paginação, na [Seção 8.4.4](#).

8.2 Swapping

Um processo precisa estar na memória para ser executado. Contudo, um processo pode ter de ser **swapped (trocado)** temporariamente entre a memória e um **backing store (armazenamento de apoio)**, para depois ser trazido de volta para a memória, para continuar a execução. Por exemplo, considere um ambiente de multiprogramação com um algoritmo de escalonamento de CPU round robin. Quando o quantum expira, o gerenciador de memória começa a efetuar o swap out (retirada) do processo que terminou e coloca outro processo no espaço de memória liberado (Figura 8.5). Nesse meio-tempo, o escalonador de CPU aloca a fatia de tempo (time slice) para algum outro processo na memória. Quando cada processo terminar seu quantum, ele será trocado por outro processo. O ideal é que o gerenciador de memória possa efetuar o swap dos processos com rapidez suficiente para que alguns processos estejam na memória, prontos para executar, quando o escalonador de CPU quiser realizar o escalonamento. Além disso, o quantum precisa ser suficientemente grande para permitir que quantidades razoáveis de computação sejam feitas entre cada operação de swap.

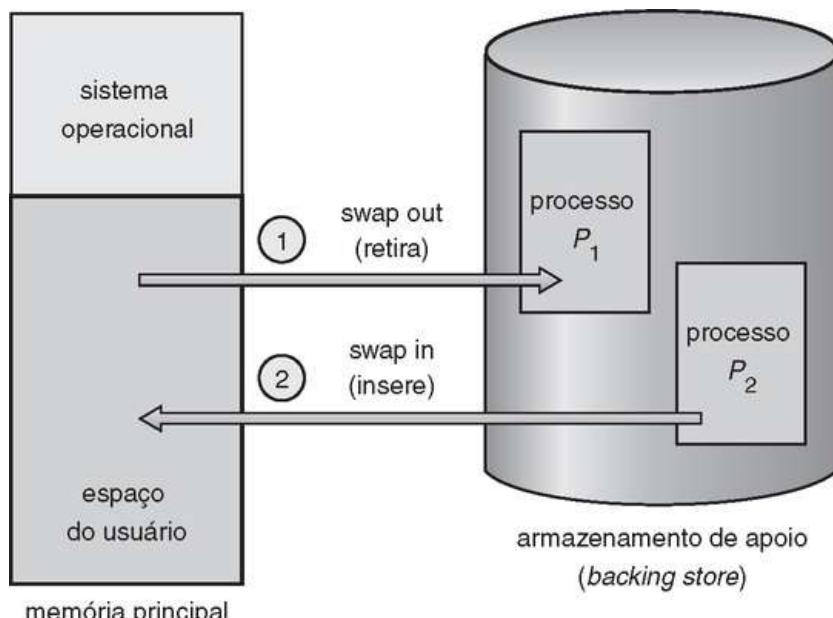


FIGURA 8.5 Swap de dois processos usando um disco como armazenamento de apoio.

Uma variante dessa política swap é usada para algoritmos de escalonamento baseados em prioridade. Se um processo com prioridade mais alta chegar e quiser ser atendido, o gerenciador de memória poderá efetuar o swap do processo de prioridade mais baixa e, em seguida, carregar e executar o processo com maior prioridade. Quando o processo com maior prioridade terminar, o processo com menor prioridade poderá ser levado de volta e continuar sua execução. Essa variante de swap às vezes é chamada **roll out, roll in**.

Em geral, um processo removido será retornado para o mesmo espaço de memória que ocupava anteriormente. Essa restrição é ditada pelo método de associação de endereço. Se a associação for feita em tempo de montagem ou carga, o processo não pode ser movido para um local diferente. Entretanto, se a associação em tempo de execução estiver sendo usada, então um processo pode ser retornado para um espaço de memória diferente, pois os endereços físicos são calculados durante o tempo de execução.

O swap exige um armazenamento de apoio. O armazenamento de apoio é um disco rápido. Ele precisa ser grande o suficiente para acomodar cópias de todas as imagens da memória para todos os usuários e precisa prover acesso direto a essas imagens da memória. O sistema mantém uma **fila de prontos (ready queue)** consistindo em todos os processos cujas imagens da memória estão no armazenamento de apoio ou na memória e prontas para serem executadas. Sempre que o escalonador de CPU decide executar um processo, ele chama o despachante. O despachante verifica se o próximo processo na fila está na memória. Se não estiver, e se não houver uma região livre na memória, o despachante retira (swap out) um processo atualmente na memória e insere (swap in) o processo desejado na memória. Em seguida, recarrega os registradores e transfere o controle para o processo selecionado.

O tempo de troca de contexto nesse sistema swap é bastante alto. Para ter uma ideia do tempo de troca de contexto, vamos considerar que o processo do usuário tenha 100 MB e o armazenamento

de apoio seja um disco rígido comum, com uma taxa de transferência de 50 MB por segundo. A transferência real do processo de 100 MB para dentro e para fora da memória principal leva $100 \text{ MB} / 50 \text{ MB} \text{ por segundo} = 2 \text{ segundos}$

Supondo uma latência média de 8 milissegundos, o tempo de troca é de 2008 milissegundos. Como precisamos fazer o swap out e o swap in, o tempo de troca total é, então, 4016 milissegundos.

Observe que a maior parte do tempo de troca é o tempo de transferência. O tempo de transferência total é diretamente proporcional à *quantidade* de memória trocada. Se tivermos um sistema computadorizado com 4 GB de memória principal e um sistema operacional residente ocupando 1 GB, o tamanho máximo do processo do usuário é de 3 GB. Contudo, muitos processos do usuário podem ser muito menores do que esse tamanho – digamos, 100 MB. Um processo de 100 MB poderia ser retirado da memória em 2 segundos, comparados aos 60 segundos necessários para retirar 3 GB. Logicamente, seria útil saber quanta memória um processo do usuário está usando, e não o quanto ele *poderia* estar usando. Depois, precisaríamos trocar apenas o que está sendo usado, reduzindo o tempo de swap. Para esse método ser eficaz, o usuário precisa manter o sistema informado de quaisquer mudanças nos requisitos de memória. Assim, um processo com requisitos de memória dinâmicos precisará emitir chamadas de sistema (*requisitar memória e liberar memória*) para informar ao sistema operacional quanto às necessidades de memória variáveis.

O swap também é restrito por outros fatores. Se quisermos trocar um processo, precisamos ter certeza de que ele está completamente ocioso (idle). Uma preocupação importante é qualquer E/S pendente. Um processo pode estar aguardando uma operação de E/S quando quisermos retirá-lo para liberar a memória. No entanto, se a E/S estiver acessando, de forma assíncrona, os buffers de E/S na memória do usuário, então o processo não poderá ser retirado. Suponha que a operação de E/S esteja enfileirada porque o dispositivo está ocupado. Se retirássemos o processo P_1 e incluíssemos o processo P_2 , uma operação de E/S poderia, em seguida, tentar usar a memória que agora pertence ao processo P_2 . Existem duas soluções para esse problema: nunca trocar um processo com E/S pendente ou executar operações de E/S somente em buffers do sistema operacional. As transferências entre os buffers do sistema operacional e a memória do processo ocorrerão apenas quando o processo estiver na memória.

Atualmente, o swap-padrão é usado em poucos sistemas. Ele exige muito tempo e fornece pouco tempo de execução para ser uma solução razoável de gerência de memória. Entretanto, versões modificadas do swap são encontradas em muitos sistemas.

Uma modificação do swap é usada em muitas versões do UNIX. O swap é desativado, mas será iniciado se muitos processos estiverem sendo executados e estiverem usando uma quantidade limite de memória. O swap é novamente interrompido quando a carga no sistema for reduzida. O gerenciamento da memória no UNIX na [Seção 21.7](#).

Os primeiros PCs – que não tinham a sofisticação para implementar métodos de gerência de memória mais avançados – executavam vários processos grandes usando uma versão modificada de swap. Um exemplo importante é o sistema operacional Microsoft Windows 3.1, que admite a execução concorrente de processos na memória. Se um novo processo for carregado e não houver memória principal suficiente, um processo antigo será levado para o disco. Todavia, esse sistema operacional não fornece o swap integral, pois o usuário, e não o escalonador, decide quando é hora de passar de um processo para outro. Qualquer processo retirado permanece fora (e sem executar) até o usuário selecionar esse processo para execução. As versões subsequentes dos sistemas operacionais da Microsoft tiram proveito dos recursos avançados da MMU, agora encontrados nos PCs. Vamos explorar esses recursos na [Seção 8.4](#) e no [Capítulo 9](#), onde explicaremos a memória virtual.

8.3 Alocação de memória contígua

A memória principal precisa acomodar o sistema operacional e os diversos processos do usuário. Portanto, precisamos alocar a memória principal da maneira mais eficiente possível. Esta seção explica um método comum, a alocação de memória contígua.

A memória é dividida em duas partições: uma para o sistema operacional residente e uma para os processos do usuário. Podemos colocar o sistema operacional na memória baixa ou na memória alta. O principal fator que afeta essa decisão é o local do vetor de interrupção. Como o vetor de interrupção costuma estar na memória baixa, os programadores colocam o sistema operacional também na memória baixa. Assim, neste texto, discutimos apenas a situação em que o sistema operacional reside na memória baixa. O desenvolvimento da outra situação é semelhante.

Em geral, queremos que vários processos do usuário residam na memória ao mesmo tempo. Portanto, precisamos considerar como alocar a memória disponível aos processos que estão na fila de entrada, esperando para serem trazidos para a memória. Na **alocação de memória contígua**, cada processo está contido em uma única seção de memória contígua.

8.3.1 Proteção e mapeamento da memória

Antes de discutirmos mais sobre a alocação de memória, temos de discutir a questão da proteção e mapeamento da memória. Podemos prover esses recursos usando um registrador de relocação, conforme discutimos na [Seção 8.1.3](#), com um registrador de limite, conforme discutimos na [Seção 8.1.1](#). O registrador de relocação contém o valor do menor endereço físico; o registrador de limite contém o intervalo de endereços lógicos (por exemplo, relocação = 100040 e limite = 74600). Com os registradores de relocação e limite, cada endereço lógico precisa ser menor do que o registrador de limite; a MMU mapeia o endereço lógico *dinamicamente*, somando o valor no registrador de relocação. Esse endereço mapeado é enviado à memória ([Figura 8.6](#)).

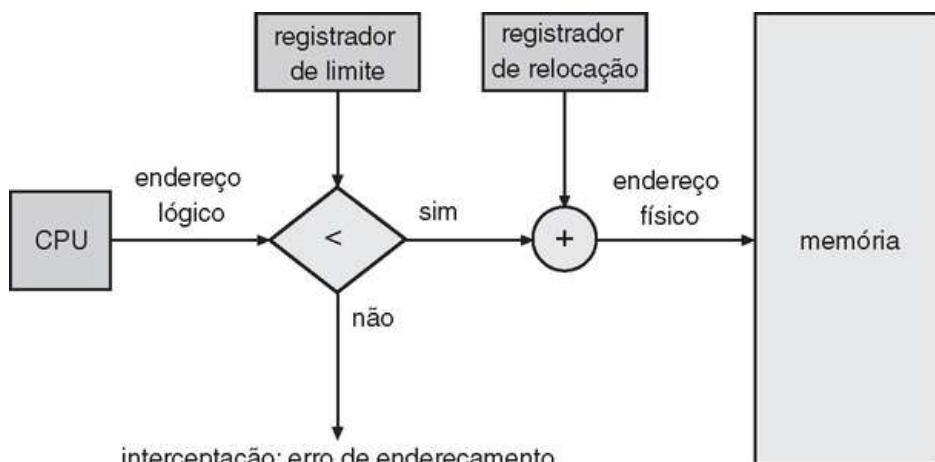


FIGURA 8.6 Suporte do hardware para registradores de relocação e limite.

Quando o escalonador de CPU seleciona um processo para execução, o despachante carrega os registradores de relocação e limite com os valores corretos, como parte da troca de contexto. Como cada endereço gerado por uma CPU é comparado a esses registradores, podemos proteger o sistema operacional e os programas dos outros usuários e dados contra modificação por esse processo em execução.

O esquema de registrador de relocação fornece um modo alternativo de permitir que o tamanho do sistema operacional mude dinamicamente. Essa flexibilidade é desejável em muitas situações. Por exemplo, o sistema operacional contém código e espaço em buffer para drivers de dispositivos. Se um driver de dispositivo (ou outro serviço do sistema operacional) não for usado, não queremos manter o código e seus dados na memória, pois poderíamos usar esse espaço para outras finalidades. Tal código às vezes é chamado de código **transiente** do sistema operacional; ele vem e vai conforme a necessidade. Assim, o uso desse código muda o tamanho do sistema operacional durante a execução do programa.

8.3.2 Alocação de memória

Agora, estamos prontos para passar para a alocação de memória. Um dos métodos mais simples para alocação de memória é dividi-la em várias **partições** de tamanho fixo. Cada partição pode

conter um processo. Assim, o grau de multiprogramação está limitado pelo número de partições. Nesse **método de múltiplas partições**, quando uma partição está livre um processo é selecionado da fila de entrada e é carregado na partição livre. Quando o processo termina, a partição volta a ficar disponível para outro processo. Esse método foi usado originalmente pelo sistema operacional IBM OS/360 (chamado MFT); ele não é mais utilizado. O método descrito a seguir é uma generalização do esquema de partição fixa (chamado MVT); ele é usado em especial em um ambiente batch. Muitas das ideias apresentadas aqui também se aplicam a um ambiente de tempo compartilhado, em que a segmentação pura é utilizada para a gerência de memória ([Seção 8.6](#)).

No esquema de **partição variável**, o sistema operacional mantém uma tabela indicando quais partes da memória estão disponíveis e quais estão ocupadas. A princípio, toda a memória está disponível para os processos do usuário e é considerada um grande bloco de memória disponível, um **buraco**. Por fim, como veremos, a memória contém um conjunto de buracos de vários tamanhos.

À medida que os processos entram no sistema, são colocados em uma fila de entrada. O sistema operacional leva em consideração os requisitos de memória de cada processo e a quantidade de memória disponível para determinar quais processos recebem memória. Quando um processo recebe espaço, ele é carregado na memória e pode então competir pelo tempo da CPU. Quando um processo termina, ele libera sua memória, que o sistema operacional pode preencher com outro processo da fila de entrada.

A qualquer momento, então, temos uma lista de tamanhos de bloco disponíveis e a fila de entrada. O sistema operacional pode ordenar a fila de entrada de acordo com um algoritmo de escalonamento. A memória é alocada aos processos até os requisitos de memória do próximo processo não poderem ser satisfeitos, ou seja, se nenhum bloco de memória disponível (ou buraco) for grande o suficiente para conter esse processo. O sistema operacional pode esperar até existir um bloco grande o suficiente à disposição ou então pode percorrer a fila de entrada para ver se os requisitos de memória menores de algum outro processo podem ser atendidos.

Em geral, conforme mencionamos, os blocos de memória disponíveis compreendem um *conjunto* de buracos de vários tamanhos, espalhados pela memória. Quando um processo chega e precisa de memória, o sistema procura no conjunto um buraco grande o suficiente para esse processo. Se o buraco for muito grande, ele é dividido em duas partes. Uma é alocada ao processo que chega; a outra é retornada ao conjunto de buracos. Quando um processo termina, ele libera seu bloco de memória, que é então colocado de volta no conjunto de buracos. Se o novo buraco for adjacente a outros buracos, eles são mesclados para formar um buraco maior. Nesse ponto, o sistema pode ter de verificar se existem processos esperando pela memória e se essa memória recém-liberada e recombinaada pode satisfazer as demandas de qualquer um desses processos esperando.

Esse procedimento é uma instância específica do **problema de alocação de armazenamento dinâmico** em geral, que se preocupa em como satisfazer uma requisição de tamanho n de uma lista de buracos livres. Existem muitas soluções para esse problema. As estratégias **first-fit**, **best-fit** e **worst-fit** são as mais utilizadas para selecionar um buraco livre do conjunto de buracos disponíveis.

- **First-fit.** Aloca o *primeiro* buraco cujo tamanho possa conter o processo. A busca pode começar no início do conjunto de buracos ou onde terminou a busca anterior. A busca é interrompida assim que encontramos um buraco livre grande o bastante.
- **Best-fit.** Aloca o *menor* buraco cujo tamanho seja suficiente para conter o processo. Temos de pesquisar a lista inteira, a menos que esteja classificada por tamanho. Essa estratégia tende a produzir pequenos buracos não contíguos.
- **Worst-fit.** Aloca o *maior* buraco que possa conter o processo. Novamente, temos de pesquisar a lista inteira, a menos que esteja classificada por tamanho. Essa estratégia tende a produzir grandes buracos, que podem ser mais úteis do que os pequenos buracos, obtidos com a técnica do best-fit.

As simulações têm mostrado que o first-fit e o best-fit são melhores do que o worst-fit em termos de velocidade e utilização de armazenamento. Nem o first-fit nem o best-fit é melhor do que o outro em termos de utilização de armazenamento, mas o first-fit é em geral mais rápido.

8.3.3 Fragmentação

As estratégias first-fit e best-fit para a alocação de memória sofrem de **fragmentação externa**. À medida que os processos são carregados e removidos da memória, o espaço de memória livre é repartido em pequenas partes. A fragmentação externa existe quando há espaço de memória total suficiente para satisfazer uma requisição, mas os espaços disponíveis não são contíguos; o armazenamento é fragmentado em uma grande quantidade de pequenos buracos. Esse problema de fragmentação pode ser muito grave. No pior caso, poderíamos ter um bloco de memória livre (ou desperdiçada) entre cada dois processos. Se, em vez disso, todas essas pequenas partes de memória estivessem em um grande bloco livre, poderíamos executar vários outros processos.

Não importa se estamos usando a estratégia de first-fit ou best-fit, ambas podem afetar a quantidade de fragmentação. Outro fator é qual extremidade de um bloco livre é alocada. (Qual é a parte restante: a que está no início ou a que está no final do bloco?) Não importa o algoritmo utilizado, a fragmentação externa será um problema.

Dependendo da quantidade total de armazenamento de memória e do tamanho médio do processo, a fragmentação externa pode ser um problema pequeno ou grande. A análise estatística do first-fit, por exemplo, revela que, até mesmo com alguma otimização, dados N blocos alocados, outros $0,5 N$ blocos serão perdidos para a fragmentação, ou seja, um terço da memória pode ficar inutilizável! Essa propriedade é conhecida como **regra dos 50%**.

A fragmentação da memória pode ser interna ou externa. Considere um esquema de alocação de múltiplas partições, com um buraco de 18.464 bytes. Suponha que o próximo processo requisite 18.462 bytes. Se alocarmos exatamente o bloco requisitado, ficaremos com um buraco de 2 bytes. O custo adicional de manter esse buraco será muito maior do que o próprio buraco. A técnica geral para evitar esse problema é dividir a memória em blocos de tamanho fixo e alocar memória em unidades baseadas no tamanho do bloco. Com essa técnica, a memória alocada a um processo pode ser ligeiramente maior do que a memória requisitada. A diferença entre esses dois números é a **fragmentação interna** - a memória interna a uma partição, mas que não está sendo usada.

Uma solução para o problema da fragmentação externa é a **compactação**. O objetivo é redistribuir o conteúdo da memória de modo a colocar toda a memória livre junta, em um grande bloco. Contudo, a compactação nem sempre é possível. Se a relocação for estática e for feita em tempo de montagem ou carga, a compactação não poderá ser feita; a compactação só é possível se a relocação for dinâmica e feita em tempo de execução. Se os endereços são relocados dinamicamente, a relocação exige apenas mover o programa e os dados e depois alterar o registrador de base para refletir o novo endereço de base. Quando a compactação é possível, temos de determinar seu custo. O algoritmo de compactação mais simples é mover todos os processos para um extremo da memória; todos os buracos se movem para a outra direção, produzindo um grande buraco de memória disponível. Esse esquema pode ser dispendioso.

Outra solução possível para o problema da fragmentação externa é permitir que o espaço de endereços lógicos de um processo seja não contíguo, permitindo assim que um processo receba memória física onde estiver disponível. Duas técnicas complementares realizam essa solução: paginação ([Seção 8.4](#)) e segmentação ([Seção 8.6](#)). Essas técnicas também podem ser combinadas ([Seção 8.7](#)).

8.4 Paginação

Paginação é um esquema de gerência de memória que permite que o espaço de endereços físicos de um processo seja não contíguo. A paginação evita a fragmentação externa e a necessidade de compactação. Ela também resolve o problema considerável de ajustar trechos de memória de tamanhos variados ao armazenamento de apoio; a maioria dos esquemas de gerência de memória utilizados antes da introdução da paginação sofria com esse problema. O problema surge porque, quando alguns fragmentos de código ou dados residentes na memória principal precisam ser retirados da memória, o espaço precisa ser encontrado no armazenamento de apoio. O armazenamento de apoio também tem os problemas de fragmentação discutidos em relação à memória principal, exceto que o acesso é muito mais lento, de modo que a compactação é impossível. Devido às suas vantagens em relação a outros métodos, a paginação em suas diversas formas normalmente é usada na maioria dos sistemas operacionais.

Como tradição, o suporte para paginação tem sido tratado pelo hardware. Entretanto, projetos recentes implementaram a paginação integrando de perto o hardware e o sistema operacional, em especial em microprocessadores de 64 bits.

8.4.1 Método básico

O método básico para implementar a paginação envolve dividir a memória física em blocos de tamanho fixo, chamados **quadros**, e dividir a memória lógica em blocos do mesmo tamanho, chamados **páginas**. Quando um processo tiver de ser executado, suas páginas são carregadas em quaisquer quadros de memória disponíveis de sua origem (de um sistema de arquivos ou do armazenamento de apoio). O armazenamento de apoio é dividido em blocos de tamanho fixo, que possuem o mesmo tamanho dos quadros de memória.

O suporte do hardware para a paginação é ilustrado na [Figura 8.7](#). Cada endereço gerado pela CPU é dividido em duas partes: um **número de página (p)** e um **deslocamento de página (d)**. O número de página é usado como um índice para a **tabela de página**. A tabela de página contém o endereço de base de cada página na memória física. Esse endereço de base é combinado com o deslocamento de página para definir o endereço da memória física que é enviado para a unidade de memória. O modelo de paginação de memória é mostrado na [Figura 8.8](#).

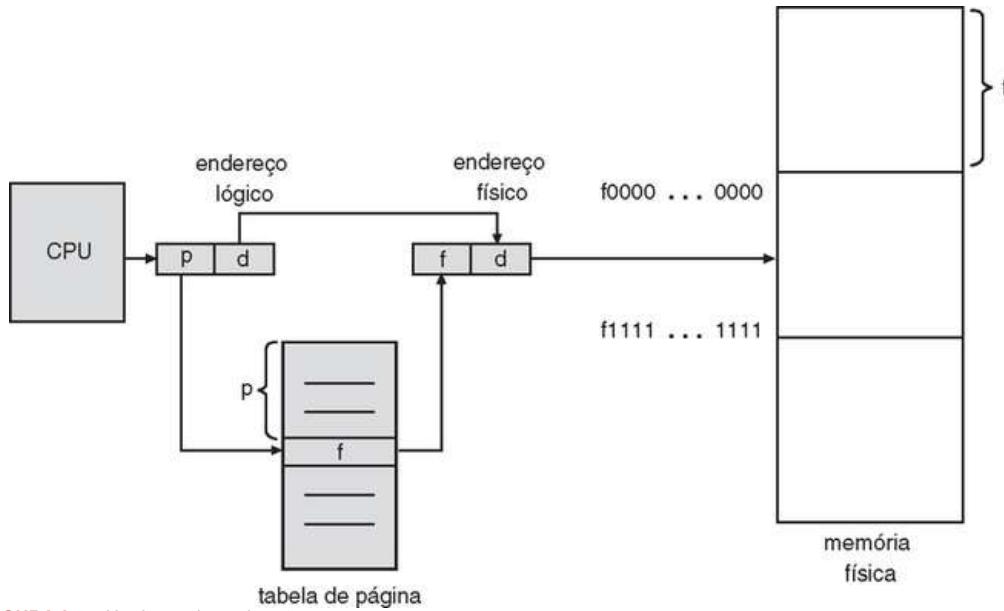


FIGURA 8.7 Hardware de paginação.

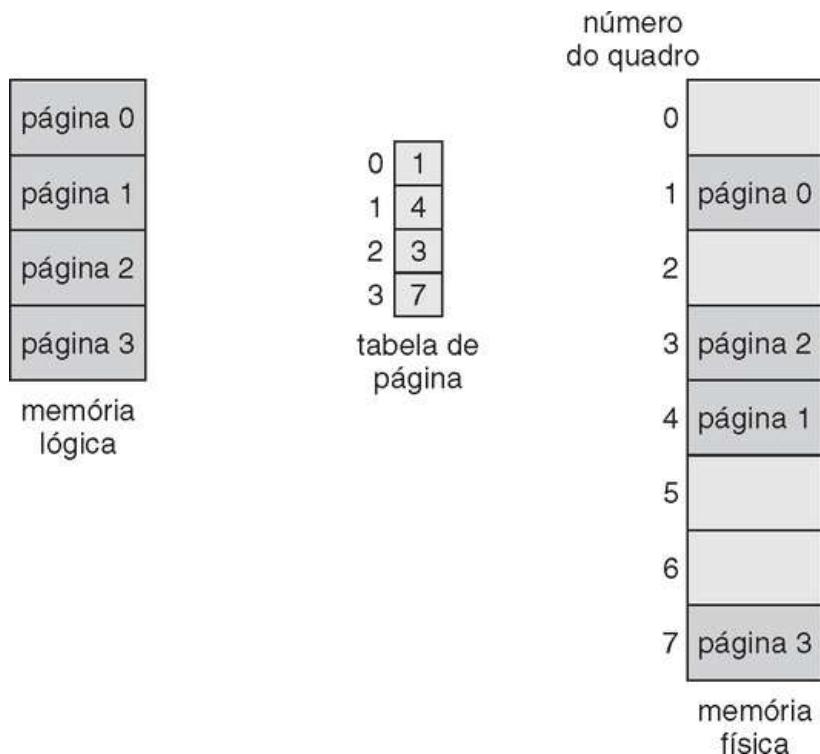


FIGURA 8.8 Modelo de paginação da memória lógica e física.

O tamanho da página (como o tamanho do quadro) é definido pelo hardware. O tamanho de uma página é uma potência de 2, variando entre 512 bytes e 16 MB por página, dependendo da arquitetura do computador. A seleção de uma potência de 2 como tamanho de página facilita bastante a tradução de um endereço lógico para um número de página e deslocamento de página. Se o tamanho do espaço de endereços lógicos for 2^m e o tamanho da página for 2^n unidades de endereçamento (bytes ou words), então os $m - n$ bits de alta ordem de um endereço lógico designam o número de página, e os n bits de baixa ordem designam o deslocamento de página. Assim, o endereço lógico é o seguinte:

número de página	deslocamento de página
p	d
$m - n$	n

onde p é um índice para a tabela de página e d é o deslocamento dentro da página.

Como exemplo concreto (embora minúsculo), considere a memória na [Figura 8.9](#). Aqui, no endereço lógico, $n = 2$ e $m = 4$. Usando um tamanho de página de 4 bytes e uma memória física de 32 bytes (8 páginas), mostramos como a visão da memória pelo usuário pode ser mapeada na memória física. O endereço lógico 0 está na página 0, deslocamento 0. Indexando para a tabela de página, descobrimos que a página 0 está no quadro 5. Assim, o endereço lógico 0 é mapeado para o endereço físico 20 [= $(5 \times 4) + 0$]. O endereço lógico 3 (página 0, deslocamento 3) é mapeado para o endereço lógico 23 [= $(5 \times 4) + 3$]. O endereço lógico 4 é a [página 1](#), deslocamento 0; de acordo com a tabela de página, a [página 1](#) é mapeada para o quadro 6. Assim, o endereço lógico 4 é mapeado para o endereço físico 24 [= $(6 \times 4) + 0$]. O endereço lógico 13 é mapeado para o endereço físico 9.

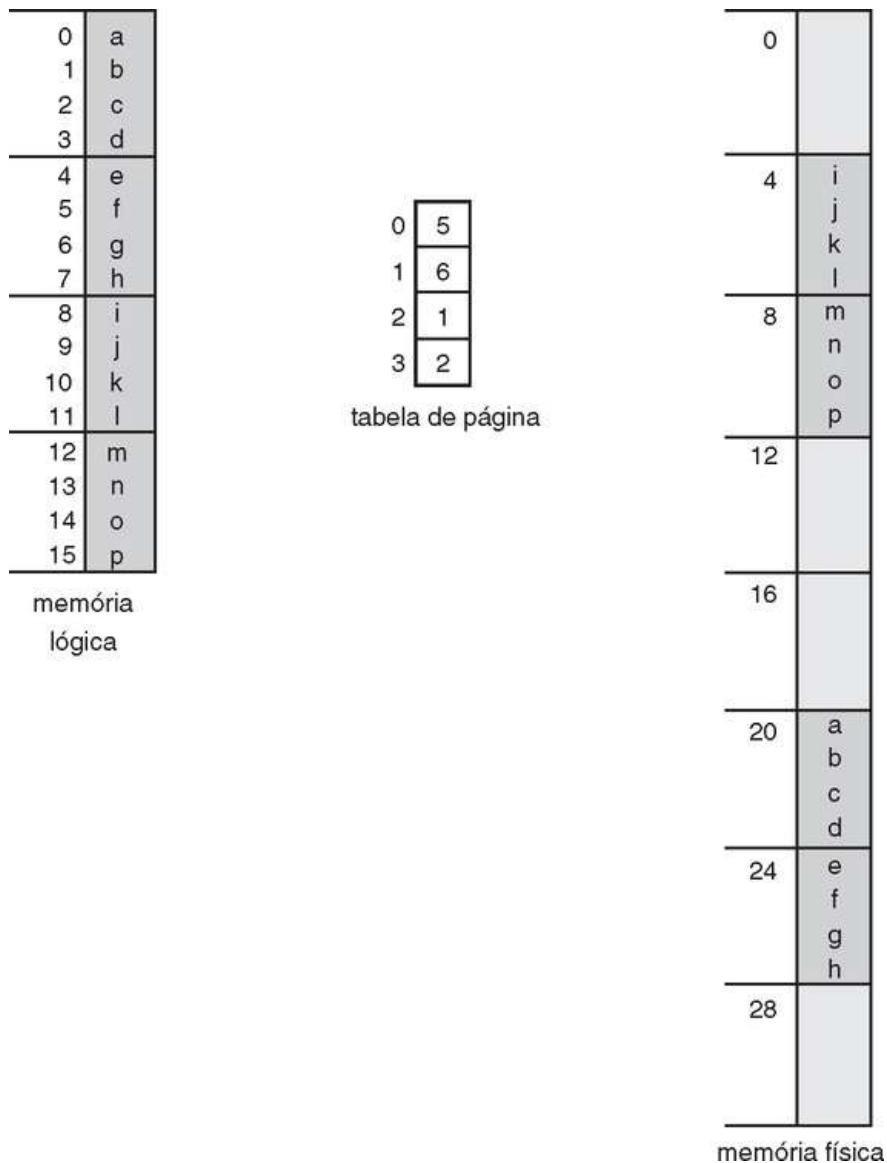


FIGURA 8.9 Exemplo de paginação para uma memória de 32 bytes com páginas de 4 bytes.

Você pode ter notado que a paginação em si é uma forma de relocação dinâmica. Cada endereço lógico é associado pelo hardware de paginação a algum endereço físico. O uso da página é semelhante ao uso de uma tabela de registradores de base (ou relocação), um para cada quadro de memória.

Quando usamos um esquema de paginação, não temos fragmentação externa. *Qualquer* quadro livre pode ser alocado a um processo que precisar dele. Todavia, podemos ter alguma fragmentação interna. Observe que os quadros são alocados como unidades. Se os requisitos de memória de um processo não coincidirem com os limites de página, o *último* quadro alocado pode não estar completamente cheio. Por exemplo, se o tamanho de página for 2.048 bytes, um processo de 72.766 bytes precisaria de 35 páginas mais 1.086 bytes. Ele receberia 36 quadros, resultando em uma fragmentação interna de $2.048 - 1.086 = 962$ bytes. No pior caso, um processo precisaria de n páginas mais um byte. Ele receberia $n + 1$ quadros, resultando em uma fragmentação interna de quase um quadro inteiro.

Se o tamanho do processo for independente do tamanho de página, esperamos que a fragmentação interna seja, na média, meia página por processo. Essa consideração sugere que tamanhos de página menores são desejáveis. No entanto, um custo adicional é envolvido em cada entrada de tabela de página, e esse custo é reduzido quando o tamanho das páginas aumenta. Além disso, a E/S de disco é mais eficiente quando a quantidade de dados sendo transferida é maior ([Capítulo 12](#)). Em geral, tamanhos de página têm crescido com o tempo, à medida que os processos, conjuntos de dados e memória principal têm se tornado maiores. Hoje, as páginas têm um tamanho entre 4 KB e 8 KB, e alguns sistemas admitem tamanhos de página ainda maiores. Algumas CPUs e kernels admitem até mesmo tamanhos de página múltiplos. Por exemplo, o Solaris utiliza tamanhos de página de 8 KB e 4 MB, dependendo dos dados armazenados pelas páginas. Os pesquisadores agora estão desenvolvendo suporte para tamanho de página variável instantânea.

Em geral, cada entrada da tabela de página possui 4 bytes de extensão, mas esse tamanho também pode variar. Uma entrada de 32 bits pode apontar para um dentre 2^{32} quadros da página física. Se o tamanho do quadro for 4 KB, então um sistema com entradas de 4 bytes pode endereçar 2^{44} (ou 16 TB) de memória física.

Quando um processo chega ao sistema para ser executado, seu tamanho, expresso em páginas, é examinado. Cada página do processo precisa de um quadro. Assim, se o processo exige n páginas, pelo menos n quadros precisam estar disponíveis na memória. Se n quadros estiverem disponíveis, são alocados a esse processo que chega. A primeira página do processo é carregada em um dos quadros alocados, e o número do quadro é colocado na tabela de página para esse processo. A próxima página é carregada em outro quadro, e seu número de quadro é colocado na tabela de página, e assim por diante (Figura 8.10).

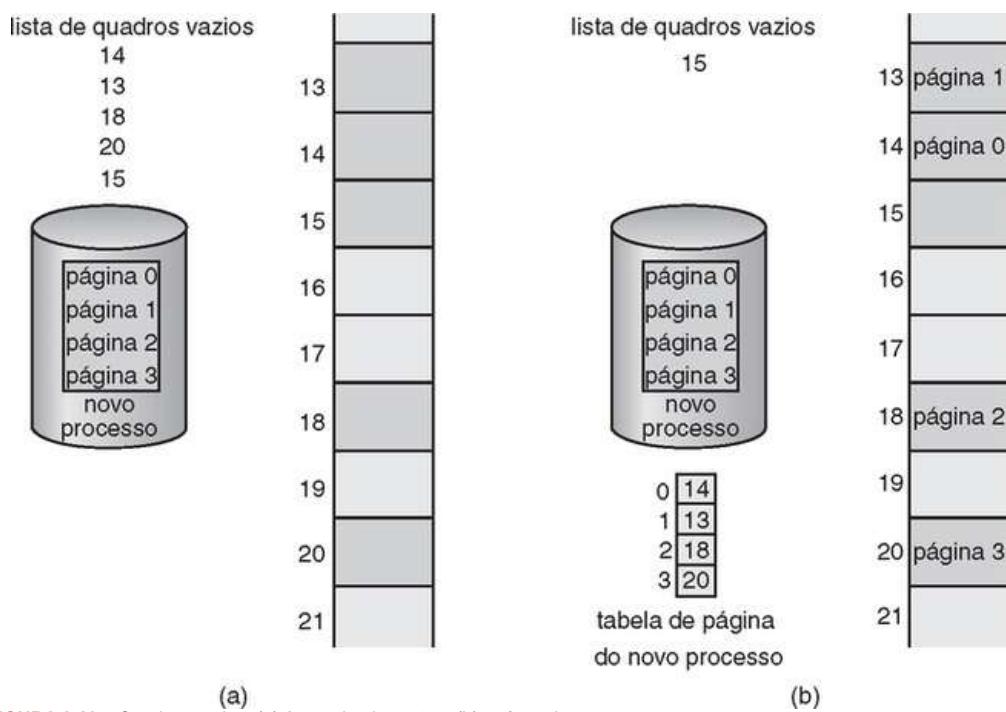


FIGURA 8.10 Quadros vazios. (a) Antes da alocação e (b) após a alocação.

Um aspecto importante da paginação é a separação clara entre a visão da memória pelo usuário e a memória física real. O programa do usuário vê a memória como um único espaço, contendo apenas um único programa. Na verdade, o programa do usuário está espalhado por toda a memória física, que também mantém outros programas. A diferença entre a visão da memória pelo usuário e a memória física real é reconciliada pelo hardware de tradução de endereços. Os endereços lógicos são traduzidos para endereços físicos. Esse mapeamento fica escondido do usuário e é controlado pelo sistema operacional. Observe que o processo do usuário, por definição, é incapaz de acessar a memória que ele não possui. Ele não tem como endereçar a memória fora da sua tabela de página, e a tabela só inclui as páginas que o processo possui.

Como o sistema operacional está gerenciando a memória física, ele precisa estar ciente dos detalhes de alocação da memória física – quais quadros estão alocados, quais estão disponíveis, quantos quadros existem no total, e assim por diante. Essa informação é mantida em uma estrutura de dados chamada **tabela de quadros**. A tabela de quadros possui uma entrada para cada quadro de página física, indicando se este último está livre ou alocado, e, se estiver alocado, para qual página de qual processo ou processos.

Além disso, o sistema operacional precisa estar ciente de que os processos do usuário operam no espaço do usuário, e todos os endereços lógicos precisam ser mapeados para produzir endereços físicos. Se um usuário fizer uma chamada de sistema (para realizar E/S, por exemplo) e prover um endereço como um parâmetro (um buffer, por exemplo), esse endereço terá de ser mapeado para produzir o endereço físico correto. O sistema operacional mantém uma cópia da tabela de página para cada processo, além de uma cópia do contador de instrução e do conteúdo dos registradores. Essa cópia é usada para traduzir endereços lógicos em endereços físicos sempre que o sistema operacional precisar associar um endereço lógico a um endereço físico manualmente. Ela também é usada pelo despachante de CPU para definir a tabela de página de hardware quando um processo tiver de receber tempo da CPU. A paginação, portanto, aumenta o tempo da troca de contexto.

8.4.2 Suporte do hardware

Cada sistema operacional possui seus próprios métodos para armazenar tabelas de página. A maioria aloca uma tabela de página para cada processo. Um ponteiro para a tabela de página é armazenado com os outros valores de registrador (como o contador de instrução) no bloco de controle do processo. Quando o despachante é requisitado a iniciar um processo, ele precisa recarregar os registradores do usuário e definir os valores corretos da tabela de página do hardware a partir da tabela de página do usuário armazenada.

A implementação de hardware da tabela de página pode ser feita de várias maneiras. No caso mais simples, a tabela de página é implementada como um conjunto de **registradores** dedicados. Esses registradores devem ser montados com lógica de altíssima velocidade, para tornar a tradução de endereço de paginação eficiente. Cada acesso à memória precisa passar pelo mapa de paginação, de modo que a eficiência é uma consideração importante. O despachante da CPU recarrega esses registradores, assim como recarrega os outros registradores. Naturalmente, as instruções para carregar ou modificar os registradores da tabela de página são privilegiadas, de modo que somente o sistema operacional pode alterar o mapa da memória. O PDP-11 da DEC é um exemplo dessa arquitetura. O endereço consiste em 16 bits, e o tamanho de página é de 8 KB. A tabela de página, portanto, consiste em oito entradas mantidas em registradores velozes.

O uso dos registradores para a tabela de página é satisfatório se a tabela de página for razoavelmente pequena (por exemplo, 256 entradas). Entretanto, a maioria dos computadores modernos permite que a tabela de página seja muito grande (por exemplo, um milhão de entradas). Para essas máquinas, o uso de registradores velozes para implementar a tabela de página não é viável. Em vez disso, a tabela de página é mantida na memória principal, e um **registrador de base da tabela de página (Page Table Base Register - PTBR)** aponta para a tabela de página. A mudança nas tabelas de página exige somente a mudança nesse registrador, reduzindo bastante o tempo da troca de contexto.

O problema com essa técnica é o tempo exigido para acessar um local da memória do usuário. Se quisermos acessar o local i , teremos de indexar primeiro na tabela de página, usando o valor no PTBR deslocado pelo número de página para i . Essa tarefa exige um acesso à memória. Ela nos fornece um número de quadro, que é combinado com o deslocamento da página para produzir o endereço real. Podemos, então, acessar o local desejado na memória. Com esse esquema, *dois* acessos à memória são necessários para acessar um byte (um para a entrada da tabela de página, um para o byte). Assim, o acesso à memória é atrasado por um fator de 2. Esse atraso seria intolerável sob a maioria das circunstâncias. Também poderíamos lançar mão da técnica de swap!

A solução-padrão para esse problema é usar um cache especial, menor, de pesquisa rápida, chamado **Translation Look-aside Buffer (TLB)**. A TLB é uma memória associativa, de alta velocidade. Cada entrada na TLB consiste em duas partes: uma chave (ou tag) e um valor. Quando a memória associativa recebe um item, o item é comparado com todas as chaves simultaneamente. Se o item for localizado, o campo de valor correspondente é retornado. A pesquisa é rápida; o hardware, porém, é caro. Em geral, o número de entradas em uma TLB é pequeno, entre 64 e 1.024.

A TLB é usada com tabelas de página da seguinte maneira. A TLB contém apenas algumas das entradas da tabela de página. Quando um endereço lógico é gerado pela CPU, seu número de página é apresentado à TLB. Se o número de página for encontrado, seu número de quadro está disponível imediatamente e é usado para acessar a memória. A tarefa inteira pode levar menos de 10% a mais do que levaria se fosse usada uma referência de memória não mapeada.

Se o número de página não estiver na TLB (conhecido como **falha na TLB - TLB miss**), uma referência de memória à tabela de página precisa ser criada. Quando o número do quadro é obtido, podemos usá-lo para acessar a memória ([Figura 8.11](#)). Além disso, acrescentamos o número de página e o número de quadro à TLB, de modo que serão encontrados rapidamente na próxima referência. Se a TLB já estiver cheia de entradas, o sistema operacional terá de selecionar um para substituição. As políticas de substituição variam desde LRU (usado menos recentemente) até o método aleatório. Além do mais, algumas TLBs permitem que certas entradas sejam **fixadas**, significando que não podem ser removidas da TLB. Em geral, as entradas da TLB para o código do kernel são fixadas.

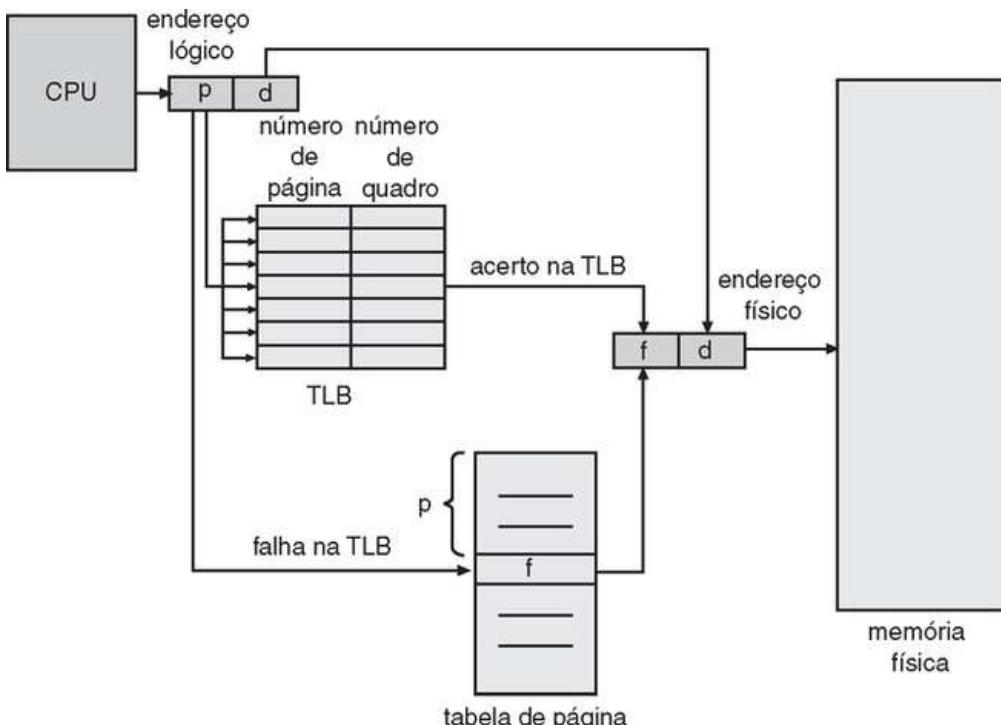


FIGURA 8.11 Hardware de paginação com TLB.

Algumas TLBs armazenam **identificadores do espaço de endereços (Address Space Identifiers - ASIDs)** em cada entrada da TLB. Um ASID identifica cada processo de forma exclusiva e é usado para prover proteção do espaço de endereços para esse processo. Quando a TLB tenta resolver números de página virtuais, ela garante que o ASID para o processo em execução corresponde ao ASID associado à página virtual. Se os ASIDs não combinarem, a tentativa é tratada como uma falha na TLB. Além de prover proteção do espaço de endereços, um ASID permite que a TLB contenha entradas para vários processos diferentes ao mesmo tempo. Se a TLB não aceitar ASIDs separados, então toda vez que uma nova tabela de página for selecionada (por exemplo, a cada troca de contexto), a TLB precisa ser **esvaziada** (ou apagada) para garantir que o próximo processo em execução não usará as informações de tradução erradas. Caso contrário, a TLB poderia incluir entradas antigas, que contêm endereços virtuais válidos, mas endereços físicos incorretos ou inválidos, vindos do processo anterior.

A porcentagem de vezes que determinado número de página é encontrado na TLB é chamada de **taxa de acertos**. Uma taxa de acertos de 80%, por exemplo, significa que encontramos o número de página desejado na TLB em 80% do tempo. Se são necessários 20 nanosegundos para pesquisar a TLB e 100 nanosegundos para acessar a memória, então um acesso à memória mapeada leva 120 nanosegundos quando o número da página está na TLB. Se não encontrarmos o número da página na TLB (20 nanosegundos), então primeiro temos de acessar a memória para obter a tabela de página e o número do quadro (100 nanosegundos) e depois acessar o byte desejado na memória (100 nanosegundos), gerando um total de 220 nanosegundos. Para encontrar o **tempo efetivo de acesso à memória**, precisamos pesar o caso por sua probabilidade:

$$\text{tempo de acesso efetivo} = 0,80 \times 120 + 0,20 \times 220 = 140 \text{ nanosegundos}$$

Nesse exemplo, sofremos um atraso de 40% no tempo de acesso à memória (de 100 para 140 nanosegundos).

Para uma taxa de acertos de 98%, temos

$$\text{tempo de acesso efetivo} = 0,98 \times 120 + 0,02 \times 220 = 122 \text{ nanosegundos}$$

Essa taxa de acertos maior produz apenas um atraso de 22% no tempo de acesso. Exploraremos melhor o impacto da taxa de acertos sobre a TLB no [Capítulo 9](#).

8.4.3 Proteção

A proteção da memória em um ambiente paginado é realizada pela proteção dos bits associados a cada quadro. Esses bits costumam ser mantidos na tabela de página. Cada referência à memória passa por uma tabela de página para encontrar o número de quadro correto. Ao mesmo tempo em que o endereço físico está sendo calculado, os bits de proteção podem ser verificados.

Um bit de proteção pode definir uma página para ser somente de leitura ou de leitura e escrita. Qualquer tentativa de escrever em uma página somente de leitura causa uma interceptação (trap) do hardware para o sistema operacional (ou uma violação de proteção de memória). Podemos facilmente expandir essa técnica para prover um nível de proteção mais apurado. Podemos criar

hardware para prover proteção somente de leitura, leitura/escrita ou somente de execução; ou, então, provendo bits de proteção separados para cada tipo de acesso, podemos permitir qualquer combinação desses acessos. Tentativas ilegais serão interceptadas pelo sistema operacional.

É anexado mais um bit a cada entrada na tabela de página: um bit de **válido-inválido**. Quando esse bit é definido como “válido”, a página associada está no espaço de endereços lógicos do processo e, portanto, é uma página legal (ou válida). Quando o bit é definido como “inválido”, a página não está no espaço de endereços lógicos do processo. Os endereços ilegais são interceptados pelo uso do bit válido-inválido. O sistema operacional define esse bit para cada página, concedendo ou rejeitando os acessos à página.

Suponha, por exemplo, que em um sistema com um espaço de endereços de 14 bits (0 a 16383), tenhamos um programa que deverá usar apenas os endereços de 0 a 10468. Dado um tamanho de página de 2 KB, obtemos a situação mostrada na [Figura 8.12](#). Os endereços nas páginas 0, [1](#), [2](#), [3](#), [4](#) e [5](#) são mapeados pela tabela de página. Contudo, qualquer tentativa de gerar um endereço nas [páginas 6](#) ou [7](#) descobrirá que o bit de válido-inválido está definido como inválido, e o computador interceptará para o sistema operacional (referência de página inválida).

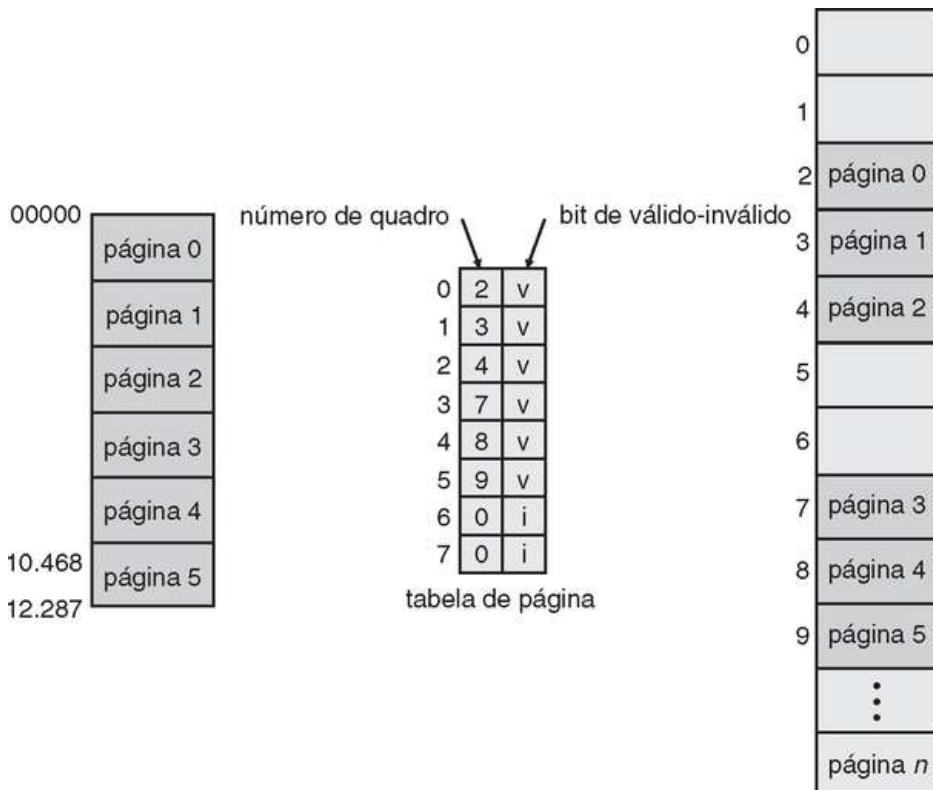


FIGURA 8.12 Bit válido (v) ou inválido (i) em uma tabela de página.

Observe que esse esquema criou um problema. Como o programa se estende somente até o endereço 10.468, qualquer referência além desse endereço é ilegal. No entanto, as referências à [página 5](#) são classificadas como válidas, de modo que os endereços até 12.287 são válidos. Somente os endereços de 12.288 a 16.383 são inválidos. Esse problema é um resultado do tamanho de página de 2 KB e reflete a fragmentação interna da paginação.

Raramente um processo utilizará todo o seu intervalo de endereços. Na verdade, muitos processos usam apenas uma pequena fração do espaço de endereços disponível para eles. Seria um desperdício, nesses casos, criar uma tabela de página com entradas para cada página no intervalo de endereços. A maior parte dessa tabela não seria usada, mas ocuparia um espaço valioso na memória. Alguns sistemas proveem hardware na forma de um **registro de extensão da tabela de página (Page Table Length Register - PTLR)** para indicar o tamanho da tabela de página. Esse valor é comparado com cada endereço lógico, para verificar se o endereço está no intervalo válido para o processo. A falha desse teste causa uma interceptação de erro para o sistema operacional.

8.4.4 Páginas compartilhadas

Uma vantagem da paginação é a possibilidade de *compartilhar* o código comum. Essa consideração é importante particularmente no ambiente de tempo compartilhado. Considere um sistema que admite 40 usuários, cada um executando um editor de textos. Se o editor de textos consiste em 150

KB de código e 50 KB de espaço de dados, precisamos de 8.000 KB para aceitar os 40 usuários. Entretanto, se o código for **código reentrante** (ou **código puro**), ele pode ser compartilhado, como mostra a [Figura 8.13](#). Aqui, vemos um editor de três páginas - cada página com 50 KB (o tamanho de página grande é usado para simplificar a figura) - sendo compartilhado entre três processos. Cada processo tem sua própria página de dados.

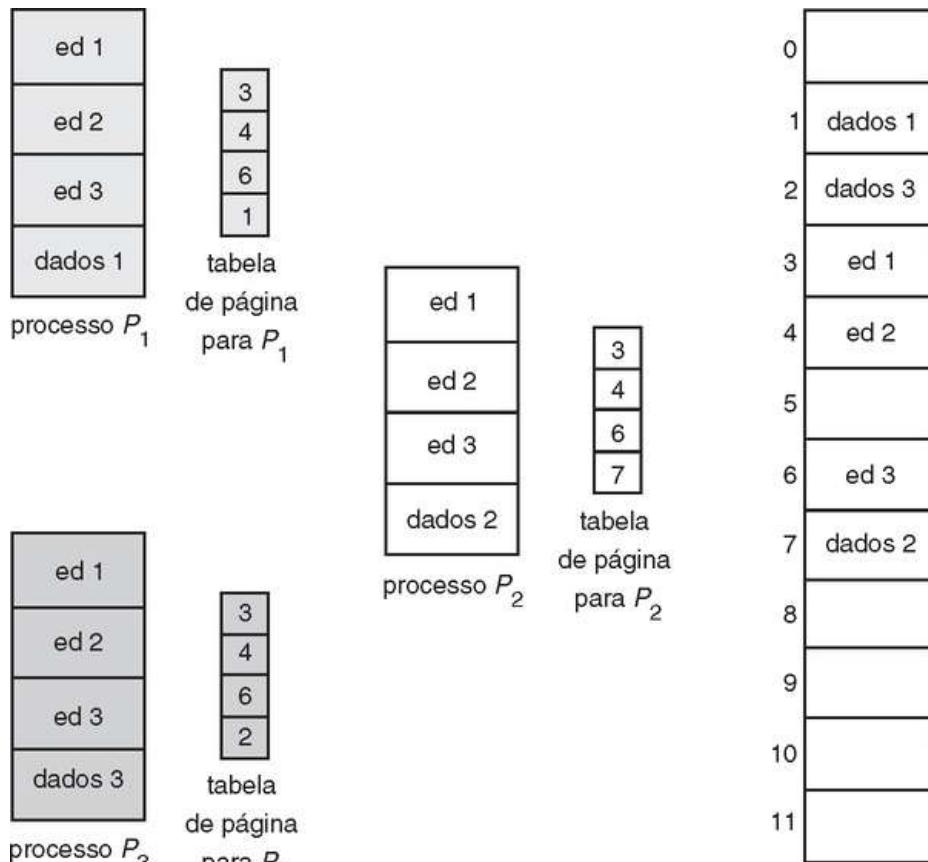


FIGURA 8.13 Compartilhamento de código em um ambiente de paginação.

O código reentrante é o código não automodificável; ele nunca muda durante a execução. Assim, dois ou mais processos podem executar o mesmo código ao mesmo tempo. Cada processo tem sua própria cópia dos registradores e armazenamento de dados para manter os dados para a execução do processo. Os dados para dois processos diferentes variarão para cada processo.

Somente uma cópia do editor precisa ser mantida na memória física. A tabela de página de cada usuário é mapeada na mesma cópia física do editor, mas as páginas de dados são mapeadas para quadros diferentes. Assim, para dar suporte a 40 usuários, precisamos apenas de uma cópia do editor (150 KB), mais 40 cópias dos 50 KB de espaço de dados, uma por usuário. O espaço total exigido agora é 2.150 KB, em vez de 8.000 KB - uma economia significativa.

Somente os programas muito utilizados também podem ser compartilhados - compiladores, sistemas de janela, bibliotecas de tempo de execução, sistemas de banco de dados e assim por diante. Para ser compartilhável, o código precisa ser reentrante. A natureza somente de leitura do código compartilhado não deve ser deixada para a exatidão do código; o sistema operacional deverá impor essa propriedade.

O compartilhamento de memória entre os processos em um sistema é semelhante ao compartilhamento do espaço de endereços de uma tarefa pelas threads, descritas no [Capítulo 4](#). Além do mais, lembre-se de que, no [Capítulo 3](#), descrevemos a memória compartilhada como um método de comunicação entre processos. Alguns sistemas operacionais implementam a memória compartilhada usando páginas compartilhadas.

A organização da memória de acordo com páginas fornece diversos benefícios, além de permitir que vários processos compartilhem as mesmas páginas físicas. Vamos abordar vários outros benefícios no [Capítulo 9](#).

8.5 Estrutura da tabela de página

Nesta seção, vamos explorar algumas das técnicas mais comuns para estruturar a tabela de página: paginação hierárquica, tabelas de página com hash e tabelas de página invertidas.

8.5.1 Paginação hierárquica

A maioria dos sistemas computadorizados modernos admite um grande espaço de endereços lógicos (2^{32} a 2^{64}). Nesse tipo de ambiente, a própria tabela de página se torna excessivamente grande. Por exemplo, considere um sistema com um espaço de endereços lógicos de 32 bits. Se o tamanho de página nesse sistema for 4 KB (2^{12}), então uma tabela de página pode consistir em até um milhão de entradas ($2^{32}/2^{12}$). Supondo que cada entrada consista em 4 bytes, cada processo pode precisar de até 4 MB de espaço de endereço físico apenas para a tabela de página. Logicamente, não gostaríamos de alocar a tabela de página de forma contígua na memória principal. Uma solução simples para esse problema é dividir a tabela de página em pedaços menores. Podemos realizar essa divisão de várias maneiras.

Uma maneira é usar um algoritmo de paginação de dois níveis, em que a própria tabela de página também é paginada (Figura 8.14). Por exemplo, considere novamente o sistema com um espaço de endereços lógicos de 32 bits com um tamanho de página de 4 KB. Um endereço lógico é dividido em um número de página contendo 20 bits e um deslocamento de página contendo 12 bits. Como paginamos a tabela de página, o número da página é dividido ainda em um número de página de 10 bits e um deslocamento de página de 10 bits. Assim, um endereço lógico é representado da seguinte forma:

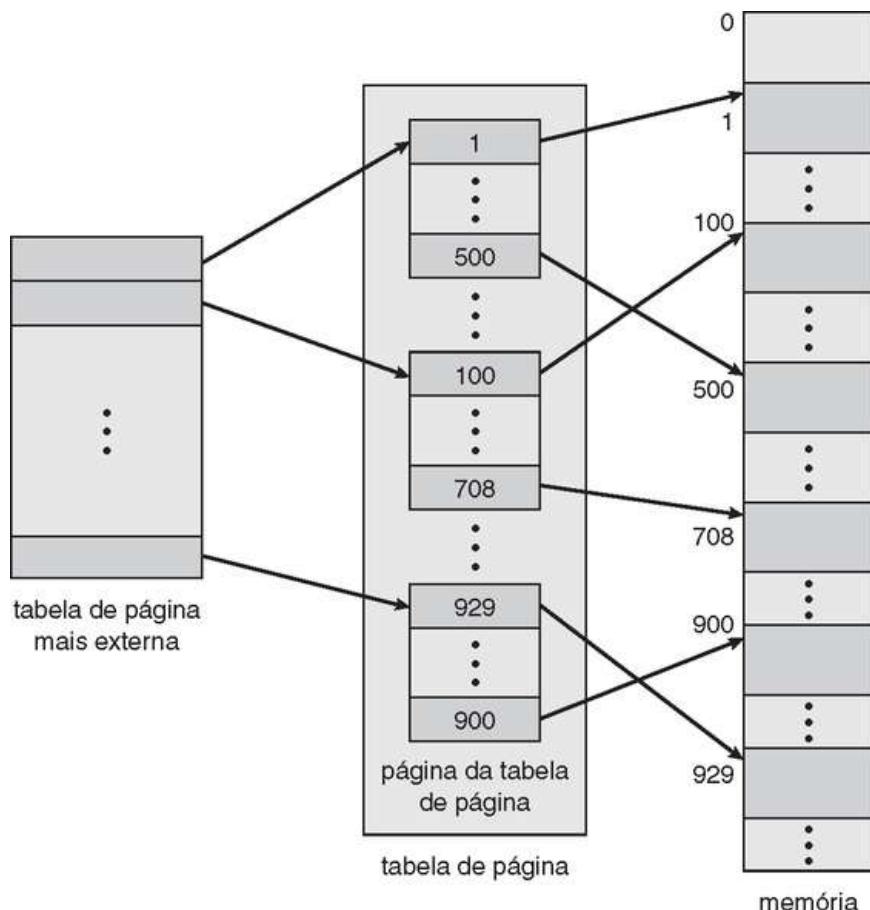


FIGURA 8.14 Um esquema de tabela de página de dois níveis.

número de página	deslocamento de página	
p_1	p_2	d
10	10	12

onde p_1 é um índice para a tabela de página mais externa e p_2 é o deslocamento dentro da página da tabela de página mais externa. O método de tradução de endereço para essa arquitetura aparece na [Figura 8.15](#). Como a tradução de endereço funciona da tabela de página mais externa para dentro, esse esquema também é conhecido como **forward-mapped page table**.

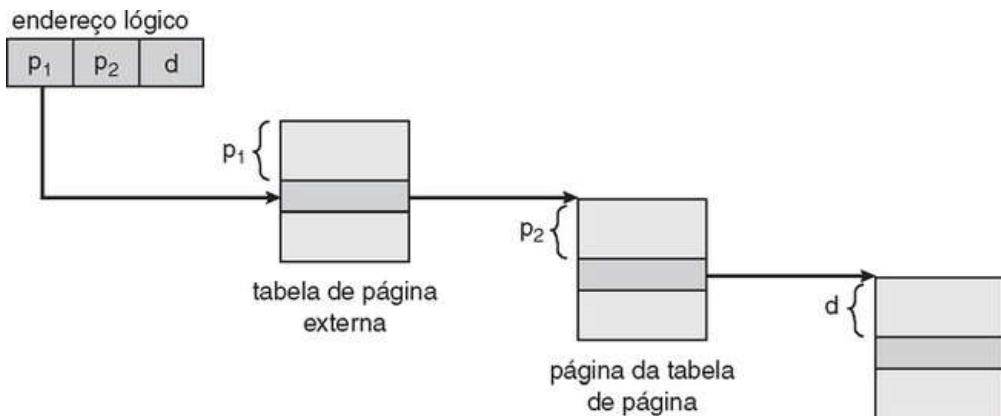


FIGURA 8.15 Tradução de endereço para uma arquitetura de página de 32 bits em dois níveis.

A arquitetura do VAX admite uma variação da página em dois níveis. O VAX é uma máquina de 32 bits com tamanho de página de 512 bytes. O espaço de endereços lógicos de um processo é dividido em quatro seções iguais, cada uma consistindo em 2^{30} bytes. Cada seção representa uma parte diferente do espaço de endereços lógicos de um processo. Os dois primeiros bits de alta ordem do endereço lógico designam a seção apropriada. Os 21 bits seguintes representam o número de página lógico dessa seção, e os 9 bits finais representam um deslocamento na página desejada. Particionando a tabela de página dessa maneira, o sistema operacional pode deixar as partições sem utilização até um processo precisar delas. Um endereço na arquitetura VAX é composto da seguinte maneira:

seção	página	deslocamento
s	p	d
2	21	9

onde s designa o número de seção, p é um índice para a tabela de página e d é o deslocamento dentro da página. Mesmo quando esse esquema é utilizado, o tamanho de uma tabela de página de um nível para um processo VAX usando uma seção ainda é de 2^{21} bits * 4 bytes por entrada = 8 MB. Para o uso da memória principal ser reduzido ainda mais, o VAX pagina as tabelas de página do processo do usuário.

Para um sistema com um espaço de endereços lógicos de 64 bits, um esquema de página de dois níveis não é mais apropriado. Para ilustrar esse ponto, vamos supor que o tamanho de página em tal sistema é de 4 KB (2^{12}). Nesse caso, a tabela de página consistirá em até 2^{52} entradas. Se usarmos um esquema de página de dois níveis, então as tabelas de página mais internas poderão convenientemente possuir uma página de extensão ou 2^{10} entradas de 4 bytes. Os endereços se parecerão com:

página externa	página interna	deslocamento
p_1	p_2	d
42	10	12

A tabela de página externa consistirá em 2^{42} entradas ou 2^{44} bytes. O método óbvio para evitar uma tabela tão grande é dividir a tabela de página externa em partes menores. Essa técnica é utilizada em alguns processadores de 32 bits, para gerar mais flexibilidade e eficiência.

Podemos dividir a tabela de página externa de várias maneiras. Podemos paginar a tabela de

página externa, ficando com um esquema de página de três níveis. Suponha que a tabela de página externa seja composta de páginas de tamanho padrão (2^{10} entradas ou 2^{12} bytes); um espaço de endereços de 64 bits ainda é assustador:

2ª página externa	página externa	página interna	deslocamento
p_1	p_2	p_3	d
32	10	10	12

A tabela de página externa ainda contém um tamanho de 2^{34} bytes.

A próxima etapa seria um esquema de página de quatro níveis, no qual a própria tabela de página externa de segundo nível também é paginada, e assim por diante. A arquitetura UltraSPARC (com o endereçamento de 64 bits) exigiria sete níveis de paginação – um número de acessos à memória proibitivo – para traduzir cada endereço lógico. Você pode observar por este exemplo por que, para arquiteturas de 64 bits, as tabelas de página hierárquicas costumam ser consideradas impróprias.

8.5.2 Tabelas de página com hash

Uma técnica comum para tratar de espaços de endereço maiores do que 32 bits é usar uma **tabela de página com hash (hashed page table)**, com o valor de hash sendo o número da página virtual. Cada entrada na tabela hash contém uma lista encadeada (linked list) de elementos que se dirigem para o mesmo local (para tratar colisões). Cada elemento consiste em três campos: (1) o número de página virtual; (2) o valor do quadro de página mapeado; e (3) um ponteiro para o próximo elemento na lista interligada.

O algoritmo funciona da seguinte maneira: o número de página virtual no endereço virtual é picotado na tabela hash. O número de página virtual é comparado com o campo 1 no primeiro elemento da lista interligada. Se houver uma combinação, o quadro de página correspondente (campo 2) é usado para formar o endereço físico desejado. Se não houver uma correspondência, as entradas subsequentes na lista encadeada são pesquisadas em busca de um número de página virtual que combine. Esse esquema é mostrado na [Figura 8.16](#).

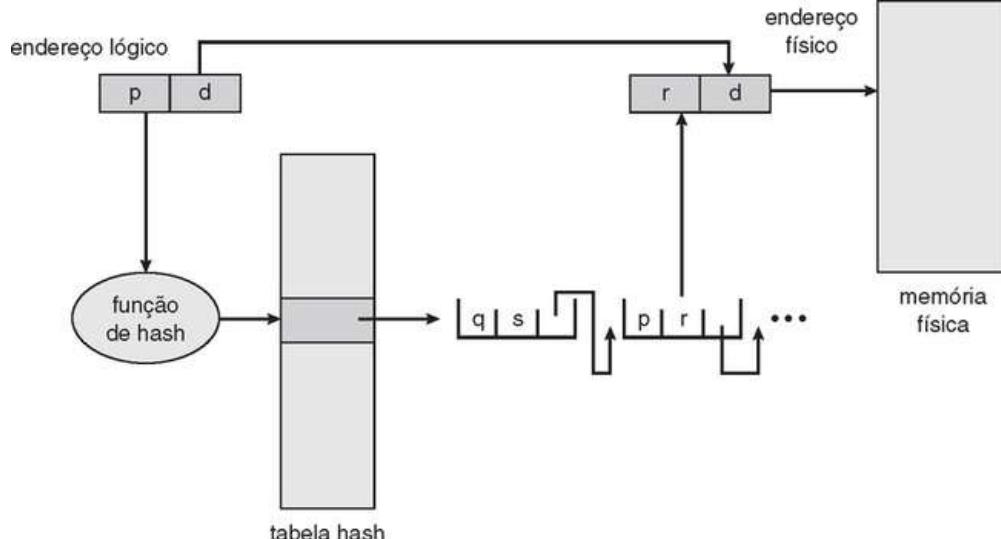


FIGURA 8.16 Tabela de página com hash.

Uma variação desse esquema, que é favorável para espaços de endereço de 64 bits, tem sido proposta. Essa variação utiliza **tabelas de página agrupadas**, que são semelhantes às tabelas de página com hash, exceto que cada entrada na tabela hash refere-se a várias páginas (por exemplo, 16) em vez de uma única página. Portanto, uma única entrada da tabela de página pode armazenar os mapeamentos para vários quadros de página física. As tabelas de página agrupadas são úteis principalmente para espaços de endereços **dispersos**, nos quais as referências de memória não são contíguas, mas espalhadas por todo o espaço de endereços.

8.5.3 Tabelas de página invertidas

Em geral, cada processo possui uma tabela de página associada. A tabela de página possui uma entrada para cada página que o processo esteja usando (ou então uma entrada para cada endereço

virtual, independentemente da validade deste). Essa representação de tabela é natural, pois os processos referenciam páginas por meio dos endereços virtuais das páginas. O sistema operacional precisa, então, traduzir essa referência para um endereço da memória física. Como a tabela é classificada por endereço virtual, o sistema operacional consegue calcular onde a entrada do endereço físico associado está localizada na tabela e usar esse valor diretamente. Uma das desvantagens desse método é que cada tabela de página pode consistir em milhões de entradas. Essas tabelas podem consumir grande quantidade de memória física só para registrar como a outra memória física está sendo usada.

Para resolver esse problema, podemos usar uma **tabela de página invertida**. Essa tabela possui uma entrada para cada página real (ou quadro) de memória. Cada entrada consiste no endereço virtual da página armazenada nesse local da memória real, com informações sobre o processo que possui a página. Assim, somente uma tabela de página está no sistema, e ela tem apenas uma entrada para cada página da memória física. A [Figura 8.17](#) mostra a operação de uma tabela de página invertida. Compare-a com a [Figura 8.7](#), que representa uma tabela de página padrão em operação. As tabelas de página invertidas normalmente exigem que um identificador de espaço de endereço ([Seção 8.4.2](#)) seja armazenado em cada entrada da tabela de página, pois a tabela contém vários espaços de endereço diferentes mapeando a memória física. Armazenar o identificador do espaço de endereços garante que uma página lógica para determinado processo seja mapeada para o quadro da página física correspondente. Alguns exemplos de sistemas que utilizam tabelas de página invertidas são o UltraSPARC de 64 bits e o PowerPC.

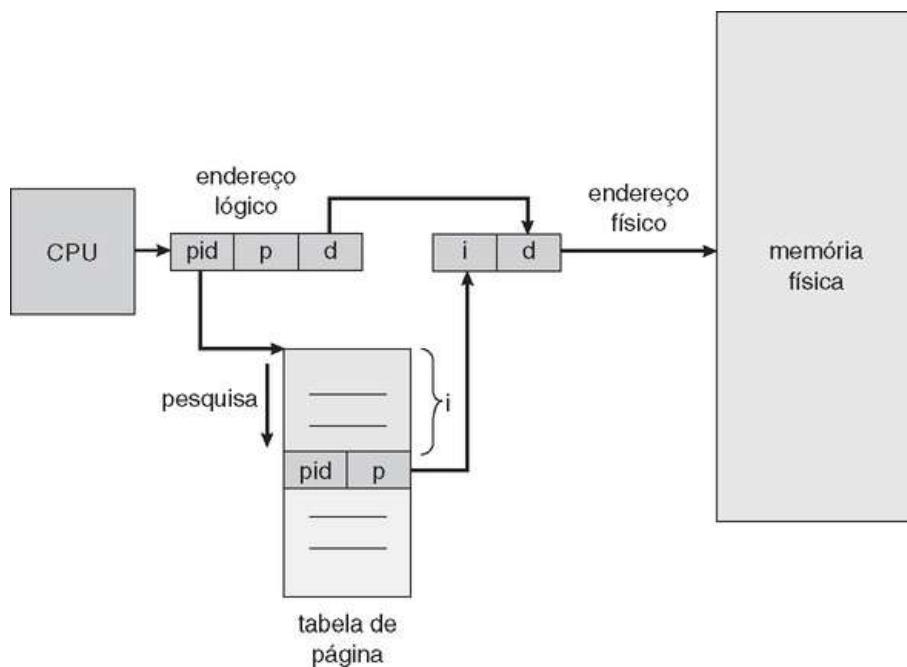


FIGURA 8.17 Tabela de página invertida.

Para ilustrar esse método, descrevemos uma versão simplificada da tabela de página invertida, usada no IBM RT. Cada endereço virtual no sistema consiste em uma tripla

<id-processo, número-página, deslocamento>

Cada entrada da tabela de página invertida é um par <id-processo, número-página>, onde o id-processo assume o papel do identificador do espaço de endereços. Quando ocorre uma referência à memória, parte do endereço virtual, consistindo em <id-processo, número-página>, é apresentado ao subsistema de memória. A tabela de página invertida, em seguida, é pesquisada em busca de uma combinação. Se for encontrada uma combinação - digamos, na entrada i -, o endereço físico < i , deslocamento> será gerado. Se nenhuma combinação for encontrada, então foi realizado um acesso a endereço ilegal.

Embora esse esquema diminua a quantidade de memória necessária para armazenar cada tabela de página, ele aumenta a quantidade de tempo necessário para pesquisar a tabela quando ocorrer uma referência de página. Como a tabela de página invertida é classificada por endereço físico, se as pesquisas ocorrerem nos endereços virtuais a tabela inteira pode precisar ser pesquisada em busca de uma combinação. Essa busca levaria muito tempo. Para aliviar esse problema, usamos uma tabela hash, conforme descrita na [Seção 8.5.2](#), para limitar a pesquisa a uma ou, no máximo, algumas entradas da tabela de página. Naturalmente, cada acesso à tabela hash acrescenta uma referência de memória ao procedimento, de modo que uma referência de memória virtual exige no mínimo duas leituras da memória real: uma para a entrada da tabela hash e uma para a tabela de

página. (Lembre-se de que a TLB é pesquisada primeiro, antes de a tabela hash ser consultada, oferecendo alguma melhoria de desempenho.)

Os sistemas que utilizam tabelas de página invertidas têm dificuldades para implementar a memória compartilhada, que costuma ser implementada como múltiplos endereços virtuais (um para cada processo compartilhando a memória) que são mapeados para um endereço físico. Esse método-padrão não pode ser usado com tabelas de página invertidas; como há somente uma entrada de página virtual para cada página física, uma página física não pode ter dois (ou mais) endereços virtuais compartilhados. Uma técnica simples para resolver essa questão é permitir que a tabela de página só contenha um mapeamento de um endereço virtual para o endereço físico compartilhado. Isso significa que as referências aos endereços virtuais que não são mapeados resultam em faltas de página.

8.6 Segmentação

Um aspecto importante da gerência de memória que se tornou inevitável com a página é a separação entre a visão da memória pelo usuário e a memória física real. Como já vimos, a visão da memória pelo usuário não é igual à memória física real. A visão do usuário é mapeada na memória física. Esse mapeamento permite a diferenciação entre a memória lógica e a memória física.

8.6.1 Método básico

Os usuários pensam na memória como um array linear de bytes, alguns contendo instruções e outros contendo dados? A maioria das pessoas diria que não. Em vez disso, os usuários preferem ver a memória como uma coleção de segmentos de tamanho variável, sem uma ordenação necessária entre os segmentos (Figura 8.18).

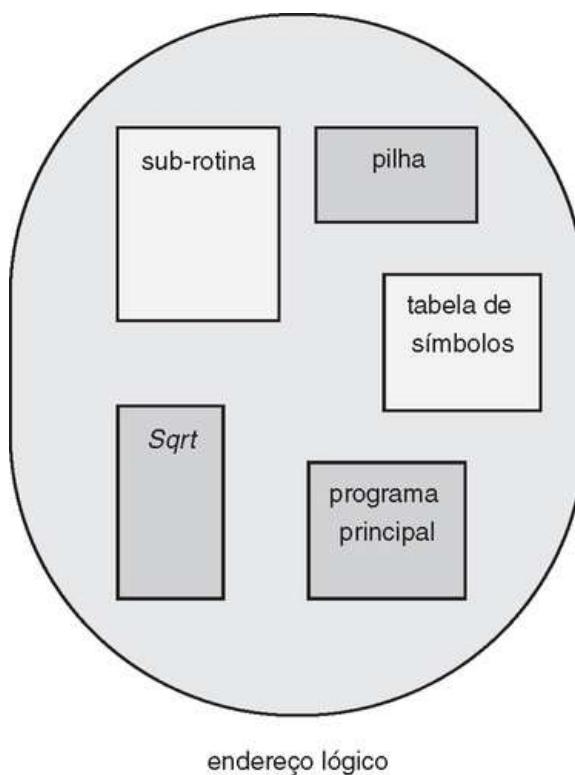


FIGURA 8.18 Visão de um programa pelo usuário.

Considere como você pensa em um programa enquanto escreve. Você pensa nele como um programa principal, com um conjunto de métodos, procedimentos ou funções. Ele também pode incluir diversas estruturas de dados: objetos, arrays, pilhas, variáveis e assim por diante. Cada um desses módulos ou elementos de dados é referenciado por nome. Você fala a respeito da “pilha”, “biblioteca matemática”, “o programa principal”, sem se importar com os endereços que esses elementos ocupam na memória. Você não está preocupado em saber se a pilha está armazenada antes ou depois da função `Sqrt()`. Cada um desses segmentos tem tamanho variável; o tamanho é definido intrinsecamente pela finalidade do segmento no programa. Os elementos dentro de um segmento são identificados por seu deslocamento desde o início do segmento: a primeira instrução do programa, a sétima entrada no quadro pilha, a quinta instrução do `Sqrt()`, e assim por diante.

Segmentação é um esquema de gerência de memória que admite essa visão da memória pelo usuário. Um espaço de endereços lógicos é uma coleção de segmentos. Cada segmento possui um nome e um tamanho. Os endereços especificam o nome do segmento e o deslocamento dentro do segmento. O usuário, portanto, especifica cada endereço por duas quantidades: um nome de segmento e um deslocamento. (Compare esse esquema com o esquema de paginação, em que o usuário especifica apenas um único endereço, que é partitionado pelo hardware em um número de página e um deslocamento, todos invisíveis ao programador.)

Para simplificar, os segmentos são numerados e referenciados com um número de segmento, em vez de um nome de segmento. Assim, um endereço lógico consiste em um *par ordenado*:

<número-segmento, deslocamento>

Normalmente, o programa do usuário é compilado e o compilador constrói os segmentos de

maneira automática, refletindo o programa de entrada.

Um compilador C poderia criar um segmento separado para o seguinte:

1. O código.
2. As variáveis globais.
3. A pilha heap, da qual a memória é alocada.
4. As pilhas usadas por cada thread.
5. A biblioteca C padrão.

As bibliotecas vinculadas durante a compilação poderiam receber segmentos separados. O loader apanharia todos esses segmentos, que receberiam números de segmento.

8.6.2 Hardware

Embora o usuário possa agora se referir aos objetos em um programa por um endereço bidimensional, a memória física real ainda é uma sequência de bytes unidimensional. Assim, precisamos definir uma implementação para associar os endereços bidimensionais definidos pelo usuário aos endereços físicos unidimensionais. Esse mapeamento é efetuado por uma **tabela de segmentos**. Cada entrada na tabela de segmentos possui uma *base de segmento* e um *limite de segmento*. A base de segmento contém o endereço físico inicial no qual o segmento reside na memória, e o limite de segmento especifica a extensão do segmento.

O uso de uma tabela de segmentos é ilustrado na [Figura 8.19](#). Um endereço lógico consiste em duas partes: um número de segmento, s , e um deslocamento dentro desse segmento, d . O número do segmento é usado como índice para a tabela de segmentos. O deslocamento d do endereço lógico precisa estar entre 0 e o limite do segmento. Se não estiver, interceptamos para o sistema operacional (tentativa de endereçamento lógico além do final do segmento). Quando um deslocamento é válido, ele é somado à base do segmento para produzir o endereço na memória física do byte desejado. A tabela de segmentos, portanto, é basicamente um array de pares de registradores de base e limite.

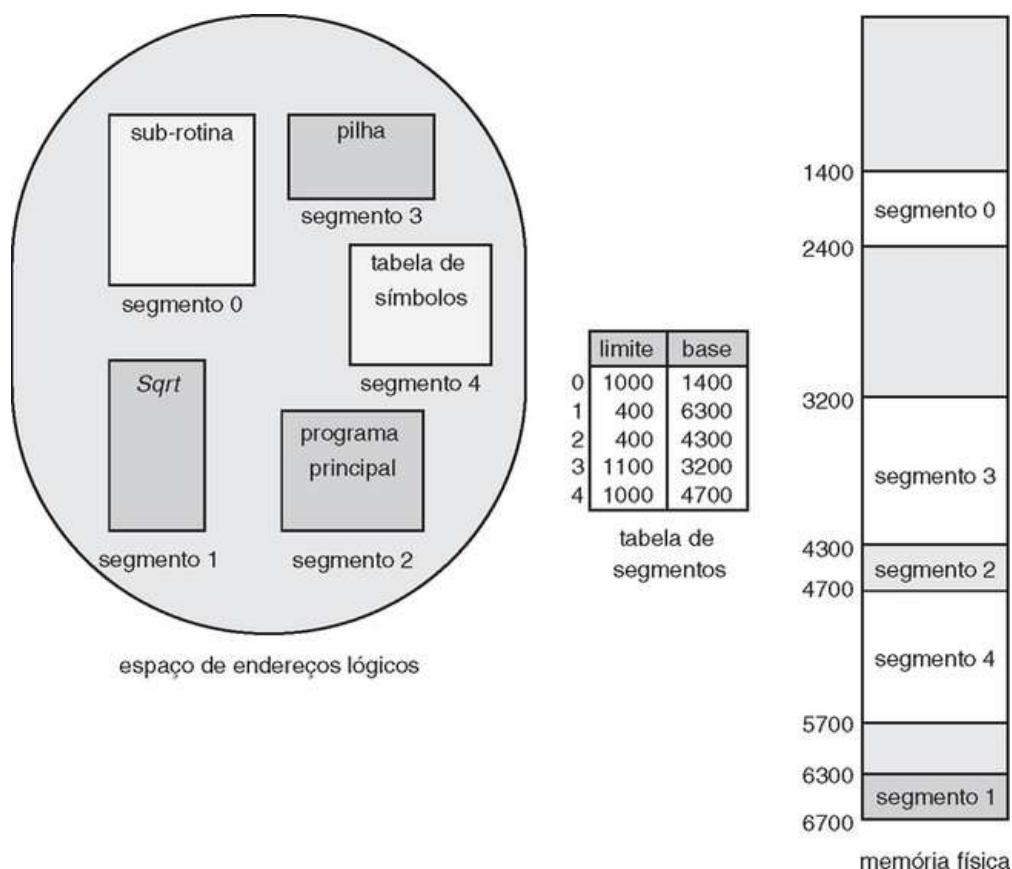


FIGURA 8.20 Exemplo de segmentação.

Como exemplo, considere a situação mostrada na [Figura 8.20](#). Temos cinco segmentos, numerados de 0 a 4. Os segmentos são armazenados na memória física conforme mostramos. A tabela de segmentos possui uma entrada separada para cada segmento, indicando o endereço inicial do segmento na memória física (ou base) e a extensão desse segmento (ou limite). Por exemplo, o segmento 2 possui 400 bytes e começa no local 4300. Assim, uma referência ao byte 53 do segmento

2 é mapeada para o local $4300 + 53 = 4353$. Uma referência ao segmento 3, byte 852, é mapeada para 3200 (a base do segmento 3) + 852 = 4052. Uma referência ao byte 1222 do segmento 0 resultaria em uma interceptação para o sistema operacional, pois esse segmento possui uma extensão de apenas 1.000 bytes.

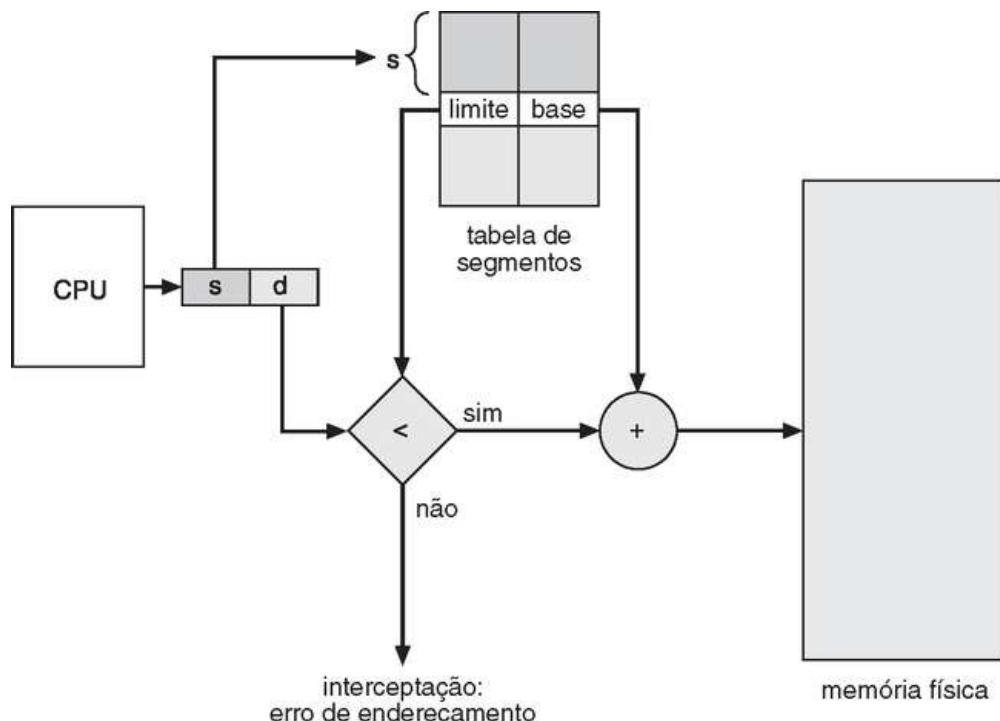


FIGURA 8.19 Hardware de segmentação.

8.7 Exemplo: Intel Pentium

A paginação e a segmentação possuem vantagens e desvantagens. Na verdade, algumas arquiteturas fornecem ambos. Nesta seção, discutimos a arquitetura Intel Pentium, que admite segmentação pura e segmentação com paginação. Não fornecemos uma descrição completa da estrutura de gerência de memória do Pentium neste texto. Em vez disso, apresentamos as principais ideias nas quais ela é baseada. Concluímos nossa discussão com uma visão geral da tradução de endereços do Linux nos sistemas Pentium.

Nos sistemas Pentium, a CPU gera endereços lógicos, que são dados à unidade de segmentação. A unidade de segmentação produz um endereço linear para cada endereço lógico. O endereço linear é dado então à unidade de paginação, que por sua vez gera o endereço físico na memória principal. Assim, as unidades de segmentação e paginação formam o equivalente da unidade de gerência de memória (MMU). Esse esquema é representado na [Figura 8.21](#).

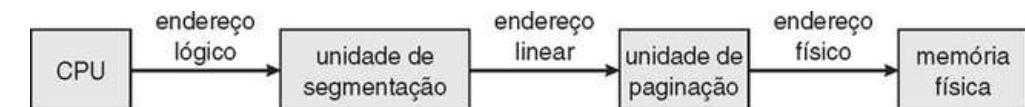


FIGURA 8.21 Tradução de endereço lógico para físico no Pentium.

8.7.1 Segmentação do Pentium

A arquitetura do Pentium permite que um segmento possa ter até 4 GB, e o número máximo de segmentos por processo é 16 KB. O espaço de endereços lógicos de um processo é dividido em duas partições. A primeira partição consiste em segmentos de até 8 KB que são privados a esse processo. A segunda partição consiste em segmentos de até 8 KB compartilhados entre todos os processos. As informações sobre a primeira partição são mantidas na **tabela de descriptor local (Local Descriptor Table - LDT)**; as informações sobre a segunda partição são mantidas na **tabela de descriptor global (Global Descriptor Table - GDT)**. Cada entrada na LDT e na GDT consiste em um descriptor de segmento com 8 bytes, com informações detalhadas sobre um segmento específico, incluindo o local de base e o limite desse segmento.

O endereço lógico é um par (seletor, deslocamento). Nesse par, o seletor é um número de 16 bits:

<i>s</i>	<i>g</i>	<i>p</i>
13	1	2

em que *s* designa o número do segmento, *g* indica se o segmento está na GDT ou na LDT e *p* lida com a proteção. O deslocamento é um número de 32 bits que especifica o local do byte (ou word) dentro do segmento em questão.

A máquina possui seis registradores de segmento, permitindo que seis segmentos sejam endereçados a qualquer momento por um processo. Ela tem seis registradores de microprograma com 8 bytes, para manter os descritores correspondentes da LDT ou da GDT. Esse cache permite que o Pentium evite ter de ler o descriptor da memória para cada referência da memória.

O endereço linear no Pentium possui 32 bits de extensão e é formado da seguinte maneira. O registrador de segmento aponta para a entrada apropriada na LDT ou GDT. As informações de base e limite sobre o segmento em questão são usadas para gerar um **endereço linear**. Primeiro, o limite é usado para verificar a validade do endereço. Se o endereço não for válido, uma falha de memória é gerada, resultando em uma interceptação para o sistema operacional. Se for válido, o valor do deslocamento será somado ao valor da base, resultando em um endereço linear de 32 bits. Isso é representado na [Figura 8.22](#). Na próxima seção, discutimos como a unidade de paginação transforma esse endereço linear em um endereço físico.

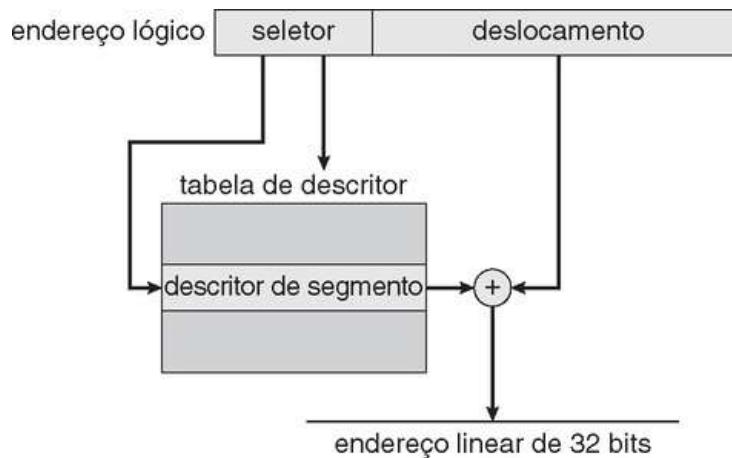


FIGURA 8.22 Segmentação do Intel Pentium.

8.7.2 Paginação do Pentium

A arquitetura do Pentium permite um tamanho de página de 4 KB ou 4 MB. Para páginas de 4 KB, o Pentium usa um esquema de paginação de dois níveis, em que a divisão do endereço linear de 32 bits é a seguinte:

número de página		deslocamento de página
p_1	p_2	d
10	10	12

O esquema de tradução de endereço para essa arquitetura é semelhante ao esquema mostrado na [Figura 8.15](#). A tradução de endereços para processadores Intel Pentium aparece com mais detalhes na [Figura 8.23](#). Os dez bits de alta ordem referenciam uma entrada na tabela de página mais externa, que o Pentium chama de **diretório de página**. (O registrador CR3 aponta para o diretório de página para o processo ativo.) A entrada do diretório de página aponta para uma tabela de página mais interna, que é indexada pelo conteúdo dos 10 bits mais internos no endereço linear. Finalmente, os bits de baixa ordem de 0 a 11 referem-se ao deslocamento na página de 4 KB apontada na tabela de página.

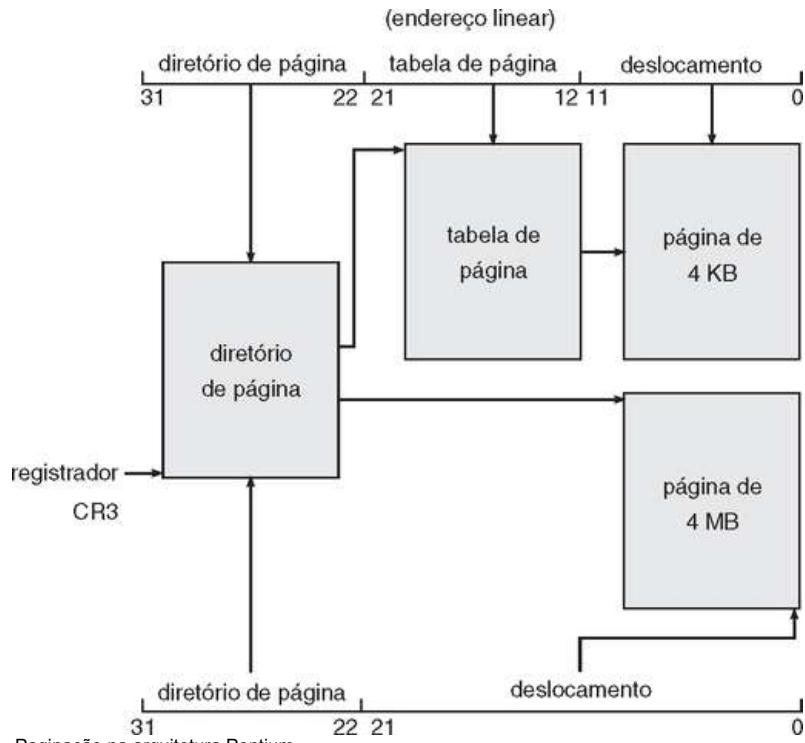


FIGURA 8.23 Paginação na arquitetura Pentium.

Uma entrada no diretório de página é a flag Page Size, que – se marcada – indica que o tamanho do frame de página é de 4 MB e não o padrão de 4 KB. Se essa flag estiver marcada, o diretório de página aponta diretamente para o frame de página de 4 MB, passando a tabela de página mais interna; e os 22 bits de baixa ordem no endereço linear referem-se ao deslocamento no frame de página de 4 MB.

Para melhorar a eficiência do uso de memória física, as tabelas de página do Intel Pentium podem ser passadas para o disco. Nesse caso, um bit de inválido é usado na entrada do diretório de página para indicar se a tabela para a qual a entrada está apontando se encontra na memória ou no disco. Se a tabela estiver no disco, o sistema operacional poderá usar os outros 31 bits para especificar o local da tabela no disco; a tabela pode, então, ser trazida para a memória quando necessário.

8.7.3 Linux no sistema Pentium

Como ilustração, considere o sistema operacional Linux rodando na arquitetura Intel Pentium. Como o Linux foi projetado para rodar em uma série de processadores – muitos deles fornecendo suporte apenas limitado para a segmentação –, esse sistema não conta com a segmentação, e a utiliza o mínimo possível. No Pentium, o Linux usa apenas seis segmentos:

1. Um segmento para o código do kernel do sistema.
2. Um segmento para os dados do kernel.
3. Um segmento para o código do usuário.
4. Um segmento para os dados do usuário.
5. Um segmento de estado de tarefa (Task State Segment – TSS).
6. Um segmento para a LDT-padrão.

Os segmentos para o código do usuário e para os dados do usuário são compartilhados por todos os processos em execução no modo do usuário. Isso é possível porque todos os processos utilizam o mesmo espaço de endereço lógico, e todos os descritores subsequentes são armazenados na tabela de descritor global (GDT). Além do mais, cada processo possui seu próprio segmento de estado de tarefa (TSS), e o descritor para esse segmento está armazenado na GDT. Esse TSS é usado para armazenar o contexto de hardware de cada processo durante as trocas de contexto. O segmento LDT-padrão é armazenado por todos os processos e não é utilizado. Todavia, se um processo exige sua própria LDT, ele pode criar uma e usá-la no lugar da LDT-padrão.

Como foi mencionado, cada seletor de segmento inclui um campo de 2 bits para proteção. Assim, o Pentium permite quatro níveis de proteção. Desses quatro níveis, o Linux só reconhece dois: o modo do usuário e o modo do kernel.

Embora o Pentium utilize um modelo de paginação de dois níveis, o Linux foi projetado para executar em diversas plataformas de hardware, muitas delas sendo plataformas de 64 bits, onde a paginação em dois níveis não é plausível. Portanto, o Linux adotou uma estratégia de paginação em três níveis, que funciona bem para as arquiteturas de 32 e 64 bits.

O endereço linear no Linux é desmembrado nas quatro partes a seguir:

diretório global	diretório do meio	tabela de página	deslocamento
------------------	-------------------	------------------	--------------

A [Figura 8.24](#) destaca o modelo de paginação em três níveis do Linux.

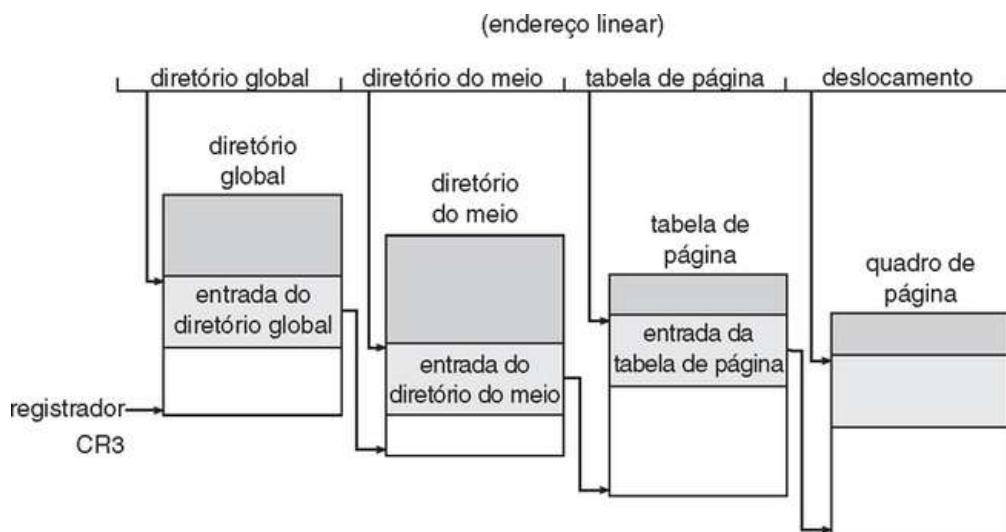


FIGURA 8.24 Paginação em três níveis no Linux.

O número de bits em cada parte do endereço linear varia de acordo com a arquitetura. Porém, conforme descrevemos anteriormente nesta seção, a arquitetura do Pentium utiliza apenas um modelo de página em dois níveis. Como, então, o Linux aplica seu modelo de três níveis no Pentium? Nessa situação, o tamanho do diretório do meio é de zero bit, efetivamente contornando o diretório do meio.

Cada tarefa no Linux tem seu próprio conjunto de tabelas de página e - assim como na [Figura 8.23](#) - o registrador CR3 aponta para o diretório global para a tarefa atualmente sendo executada. Durante uma troca de contexto, o valor do registrador CR3 é salvo e restaurado nos segmentos TSS das tarefas envolvidas na troca de contexto.

8.8 Resumo

Os algoritmos de gerência de memória para sistemas operacionais multiprogramados variam desde a técnica simples do sistema monousuário até a segmentação paginada. O fator mais importante para determinar o método utilizado em um sistema específico é o hardware fornecido. Cada endereço de memória gerado pela CPU precisa ser verificado quanto à sua validade e possivelmente mapeado para um endereço físico. A verificação não pode ser implementada (de forma eficiente) no software. Logo, estamos restritos pelo hardware disponível.

Os vários algoritmos de gerência de memória discutidos (alocação contígua, paginação, segmentação e combinações de página e segmentação) diferem em muitos aspectos. Ao comparar diferentes estratégias de gerência de memória, usamos as seguintes considerações:

- **Suporte do hardware.** Um registrador de base simples ou um par de registradores de base-límite é o suficiente para esquemas com única e múltipla partição, enquanto a paginação e a segmentação precisam de tabelas de mapeamento para definir o mapa de endereços.
- **Desempenho.** À medida que o algoritmo de gerência de memória se torna mais complexo, o tempo necessário para mapear um endereço lógico em um endereço físico aumenta. Para os sistemas simples, só precisamos comparar ou somar ao endereço lógico - são operações rápidas. A paginação e a segmentação podem ser igualmente rápidas se a tabela for implementada em registradores rápidos. Entretanto, se a tabela estiver na memória, os acessos à memória do usuário podem ser degradados substancialmente. Uma TLB pode diminuir a degradação do desempenho para um nível aceitável.
- **Fragmentação.** Um sistema multiprogramado em geral funcionará de modo mais eficiente se tiver um nível mais alto de multiprogramação. Para determinado conjunto de processos, só podemos aumentar o nível de multiprogramação colocando mais processos na memória. Para realizar essa tarefa, temos de reduzir o desperdício de memória ou fragmentação. Os sistemas com unidades de alocação de tamanho fixo, como o esquema de única partição e a paginação, sofrem de fragmentação interna. Os sistemas com unidades de alocação de tamanho variável, como o esquema de múltipla partição e segmentação, sofrem de fragmentação externa.
- **Relocação.** Uma solução para o problema de fragmentação externa é a compactação. A compactação envolve deslocar um programa na memória de modo que o programa não perceba a mudança. Essa consideração exige que os endereços lógicos sejam relocados dinamicamente, durante a execução. Se os endereços forem relocados apenas no momento da carga, não poderemos compactar o armazenamento.
- **Troca.** A troca (swap) pode ser acrescentada a qualquer algoritmo. Em intervalos determinados pelo sistema operacional, normalmente ditados pelas políticas de escalonamento de CPU, os processos são copiados da memória principal para um armazenamento de apoio e, mais tarde, são copiados de volta para a memória principal. Esse esquema permite que mais processos sejam executados do que podem caber na memória ao mesmo tempo.
- **Compartilhamento.** Outro meio de aumentar o nível de multiprogramação é compartilhar o código e os dados entre diferentes usuários. O compartilhamento exige que a paginação ou a segmentação sejam usadas, para prover pequenos pacotes de informação (páginas ou segmentos) que podem ser compartilhados. O compartilhamento é um meio de executar muitos processos com uma quantidade de memória limitada, mas os programas compartilhados e os dados precisam ser projetados com cuidado.
- **Proteção.** Se houver paginação ou segmentação, diferentes seções de um programa do usuário podem ser declaradas como somente de execução, somente de leitura ou de leitura/escrita. Essa restrição é necessária com código ou dados compartilhados e normalmente é útil em qualquer caso para prover verificações simples em tempo de execução para os erros de programação comuns.

Exercícios práticos

- 8.1. Cite duas diferenças entre endereços lógicos e físicos.
- 8.2. Considere um sistema no qual um programa pode estar separado em duas partes: código e dados. A CPU sabe se deseja uma instrução (busca de instrução) ou dados (busca de dados ou armazenamento). Portanto, dois pares de registradores de base-limite são fornecidos: um para instruções e um para dados. O par de registradores de base-limite é automaticamente apenas de leitura, de modo que os programas possam ser compartilhados entre diferentes usuários. Discuta as vantagens e as desvantagens desse esquema.
- 8.3. Por que os tamanhos de página são sempre potências de 2?
- 8.4. Considere um espaço de endereços lógicos de 64 páginas de 1.024 words cada, mapeado em uma memória física de 32 quadros.
 - a. Quantos bits existem no endereço lógico?
 - b. Quantos bits existem no endereço físico?
- 8.5. Qual é o efeito de permitir que duas entradas em uma tabela de página apontem para o mesmo quadro de página na memória? Explique como esse efeito poderia ser usado para diminuir a quantidade de tempo necessária para copiar uma grande quantidade de memória de um local para outro. Que efeito a atualização de algum byte em uma página teria sobre a outra página?
- 8.6. Descreva um mecanismo pelo qual um segmento poderia pertencer ao espaço de endereços de dois processos diferentes.
- 8.7. É possível compartilhar segmentos entre processos sem exigir que eles tenham o mesmo número de segmentos em um sistema de segmentação vinculado dinamicamente.
 - a. Defina um sistema que permita o vínculo estático e o compartilhamento de segmentos sem exigir que os números de segmento sejam os mesmos.
 - b. Descreva um esquema de paginação que permita que as páginas sejam compartilhadas sem exigir que os números de página sejam operações iguais.
- 8.8. No IBM/370, a proteção de memória é fornecida por meio do uso de *chaves*. Uma chave é uma quantidade de 4 bits. Cada bloco de 2 K de memória tem uma chave (a chave de armazenamento) associada a ele. A CPU também tem uma chave (a chave de proteção) associada a ela. Uma operação de armazenamento só tem permissão se as duas chaves forem iguais ou se uma delas for zero. Quais dos seguintes esquemas de gerência de memória poderiam ser usados com sucesso com esse hardware?
 - a. Máquina pura
 - b. Sistema monousuário
 - c. Multiprogramação com um número fixo de processos
 - d. Multiprogramação com um número variável de processos
 - e. Paginação
 - f. Segmentação

Exercícios

- 8.9. Explique a diferença entre fragmentação interna e externa.
- 8.10. Considere o processo a seguir para gerar binários. Um compilador é usado para gerar o código objeto para os módulos individuais, e um editor de vínculo (linkeditor) é usado para combinar múltiplos módulos objeto em um único binário de programa. Como o linkeditor muda o vínculo das instruções e dados para endereços de memória? Que informação precisa ser passada do compilador para o linkeditor para facilitar as tarefas de ligação de memória do linkeditor?
- 8.11. Dadas cinco partições de memória de 100 KB, 500 KB, 200 KB, 300 KB e 600 KB (nesta ordem), como cada um dos algoritmos de first-fit, best-fit e worst-fit incluiria processos de 212 KB, 417 KB, 112 KB e 426 KB (nesta ordem)? Que algoritmo faz o uso mais eficiente da memória?
- 8.12. A maioria dos sistemas permite que os programas aloquem mais memória ao seu espaço de endereços durante a execução. Os dados alocados nos segmentos de heap dos programas são um exemplo dessa memória alocada. O que é necessário para o suporte à alocação dinâmica de memória nos seguintes esquemas?
- Alocação de memória contígua
 - Segmentação pura
 - Paginação pura
- 8.13. Compare os esquemas da organização da memória principal de alocação de memória contígua, segmentação pura e paginação pura com relação aos seguintes aspectos:
- Fragmentação externa
 - Fragmentação interna
 - Capacidade de compartilhar código entre processos
- 8.14. Em um sistema com paginação, um processo não pode acessar a memória que ele não possui. Por quê? Como o sistema operacional poderia permitir o acesso à outra memória? Por que deveria ou não?
- 8.15. Compare a paginação com a segmentação com relação à quantidade de memória exigida pelas estruturas de tradução de endereço a fim de converter endereços virtuais em endereços físicos.
- 8.16. Os binários de programa em muitos sistemas normalmente são estruturados da seguinte forma. O código é armazenado a partir de um pequeno endereço virtual fixo, como 0. O segmento de código é acompanhado pelo segmento de dados, que é usado para armazenar as variáveis do programa. Quando o programa começa a ser executado, a pilha é alocada na outra ponta do espaço de endereços virtuais e pode crescer em direção aos endereços virtuais mais baixos. Qual é o significado dessa estrutura nos esquemas a seguir?
- Alocação de memória contígua
 - Segmentação pura
 - Paginação pura
- 8.17. Considerando um tamanho de página de 1 KB, quais são os números de página e deslocamentos para as seguintes referências de endereço (fornecidas como números decimais)?
- 2375
 - 19366
 - 30000
 - 256
 - 16385
- 8.18. Considere um espaço de endereços lógicos de 32 páginas com 1.024 words cada, mapeado em uma memória física de 16 frames.
- Quantos bits são necessários no endereço lógico?
 - Quantos bits são necessários no endereço físico?
- 8.19. Considere um sistema de computação com um endereço lógico de 32 bits e tamanho de página de 4 KB. O sistema admite até 512 MB de memória física. Quantas entradas existem em cada um dos seguintes?
- Uma tabela de página convencional com um único nível
 - Uma tabela de página invertida
- 8.20. Considere um sistema de paginação com a tabela de página armazenada na memória.
- Se uma referência de memória leva 200 nanosegundos, quanto tempo leva uma referência à memória paginada?
 - Se acrescentarmos TLBs, e 75% de todas as referências à tabela de página forem encontradas nas TLBs, qual é o tempo efetivo de referência à memória? (Considere que a localização de uma entrada de página nas TLBs leve tempo zero, se a entrada existir.)
- 8.21. Por que a segmentação e a paginação às vezes são combinadas em um único esquema?
- 8.22. Explique por que o compartilhamento de um módulo reentrante é mais fácil quando a segmentação é usada do que quando a paginação pura é usada.
- 8.23. Considere a seguinte tabela de segmentos:

Segmento	Base	Tamanho
0	219	600
1	2300	14
2	90	100
3	1327	580
4	1952	96

Quais são os endereços físicos para os endereços lógicos a seguir?

- a. 0,430
- b. 1,10
- c. 2,500
- d. 3,400
- e. 4,112

- 8.24. Qual é a finalidade da paginação das tabelas de página?
- 8.25. Considere o esquema de paginação hierárquico usado pela arquitetura VAX. Quantas operações da memória são realizadas quando um programa do usuário executa uma operação de load da memória?
- 8.26. Compare o esquema de paginação segmentada com o esquema de tabelas de página em hash para o tratamento de grandes espaços de endereços. Sob quais circunstâncias um esquema é preferível em relação ao outro?
- 8.27. Considere o esquema de tradução de endereço da Intel mostrado na [Figura 8.22](#).
- a. Descreva todas as etapas realizadas pelo Intel Pentium na tradução de um endereço lógico para um endereço físico.
 - b. Quais são as vantagens para o sistema operacional do hardware que provê uma tradução de memória tão complicada?
 - c. Existem desvantagens nesse sistema de tradução de endereço? Se houver, quais são elas? Se não, por que isso não é usado por todos os fabricantes?

Problemas de programação

8.28. Supondo que um sistema tenha um endereço virtual de 32 bits, escreva um programa em Java que receba (1) o tamanho de uma página e (2) o endereço virtual. Seu programa informará o número de página e deslocamento do endereço virtual indicado com o tamanho de página especificado. Os tamanhos de página deverão ser especificados como uma potência de 2 e dentro do intervalo de 1024 a 16384 (inclusive). Supondo que tal programa se chame `Address`, ele seria executado desta forma:

```
java Address 4096 19986
```

e a saída correta apareceria como:

```
O endereço 19986 contém:  
número de página = 4  
deslocamento = 3602
```

Notas bibliográficas

A alocação dinâmica de armazenamento foi discutida por [Knuth \[1973\]](#) ([Seção 2.5](#)), que descobriu, por resultados de simulação, que o first-fit geralmente é superior ao best-fit. [Knuth \[1973\]](#) também discutiu a regra dos 50%.

O conceito de paginação pode ser creditado aos projetistas do sistema Atlas, descrito por [Kilburn e outros \[1961\]](#) e por [Howarth e outros \[1961\]](#). O conceito de segmentação foi discutido primeiro por [Dennis \[1965\]](#). A segmentação paginada foi utilizada inicialmente no GE 645, no qual o MULTICS foi implementado a princípio ([Organick \[1972\]](#) e [Daley e Dennis \[1967\]](#)).

As tabelas de página invertidas foram discutidas em um artigo sobre o gerenciador de armazenamento IBM RT por [Chang e Mergen \[1988\]](#).

A tradução de endereços no software é abordada em [Jacob e Mudge \[1997\]](#).

[Hennessy e Patterson \[2002\]](#) discutem os aspectos de hardware das TLBs, caches e MMUs. [Talluri e outros \[1995\]](#) discutem as tabelas de página para os espaços de endereço de 64 bits. Técnicas alternativas para impor a proteção de memória são propostas e estudadas em [Wahbe e outros \[1993a\]](#), [Chase e outros \[1994\]](#), [Bershad e outros \[1995\]](#) e [Thorn \[1997\]](#). As técnicas de gerência da TLB são discutidas por [Dougan e outros \[1999\]](#) e por [Jacob e Mudge \[2001\]](#). [Fang e outros \[2001\]](#) avaliam o suporte para páginas grandes.

[Tanenbaum \[2001\]](#) discute a paginação no Intel 80386. A gerência de memória em diversas arquiteturas - como Pentium II, PowerPC e UltraSPARC - foi descrita por [Jacob e Mudge \[1998a\]](#). A segmentação nos sistemas Linux é apresentada em [Bovet e Cesati \[2002\]](#).

CAPÍTULO 9

Memória virtual

No [Capítulo 8](#), discutimos as diversas estratégias de gerência de memória usadas nos sistemas computadorizados. Todas essas estratégias têm o mesmo objetivo: manter muitos processos na memória simultaneamente, para permitir a multiprogramação. No entanto, elas costumam exigir que um processo inteiro esteja na memória antes de poder ser executado.

A memória virtual é uma técnica que permite a execução de processos que não estão completamente na memória. Uma grande vantagem desse esquema é que os programas podem ser maiores do que a memória física. Além do mais, a memória virtual extrai a memória principal para um conjunto de armazenamento extremamente grande e uniforme, separando a memória lógica, vista pelo usuário, da memória física. Essa técnica livra os programadores dos problemas de limitação do armazenamento de memória. A memória virtual também permite que processos compartilhem arquivos com facilidade e implementem a memória compartilhada. Além disso, ela provê um mecanismo eficiente para a criação do processo. Contudo, a memória virtual não é fácil de implementar e pode diminuir bastante o desempenho, se usada de forma descuidada. Neste capítulo, vamos discutir sobre a memória virtual na forma de paginação por demanda, examinando sua complexidade e custo.

OBJETIVOS DO CAPÍTULO

- Descrever os benefícios de um sistema de memória virtual.
- Explicar os conceitos da paginação por demanda, algoritmos de substituição de página e alocação de quadros de página.
- Discutir os princípios do modelo de conjunto de trabalho.

9.1 Aspectos básicos

Os algoritmos de gerência de memória esboçados no [Capítulo 8](#) são necessários por causa de um requisito básico: as instruções executadas precisam estar na memória física. A primeira técnica para atender a esse requisito é colocar o espaço de endereços lógicos inteiro na memória física. O carregamento dinâmico pode ajudar a aliviar essa restrição, mas exige precauções especiais e trabalho extra pelo programador.

O requisito de que as instruções precisam estar na memória física para serem executadas parece ser necessário e razoável, mas também é muito lamentável, pois limita o tamanho de um programa ao tamanho da memória física. Na verdade, um exame dos programas reais nos mostra que, em muitos casos, o programa inteiro não é necessário. Por exemplo, considere:

- Os programas possuem código para lidar com condições de erro incomuns. Como esses erros raramente ou nunca ocorrem na prática, esse código quase nunca é executado.
- Arrays, listas e tabelas recebem mais memória do que realmente precisam. Um array pode ser declarado com 100 por 100 elementos, embora raramente seja maior do que 10 por 10 elementos. Uma tabela de símbolos do assembler pode ter espaço para 3.000 símbolos, embora o programa em média tenha menos de 200 símbolos.
- Certas opções e recursos de um programa podem ser usados raramente. Por exemplo, as rotinas nos computadores do governo dos Estados Unidos que equilibram o orçamento não são usadas há muitos anos.

Até mesmo nos casos em que o programa inteiro é necessário, ele pode não ser todo necessário ao mesmo tempo.

A capacidade de executar um programa que esteja apenas parcialmente na memória teria muitos benefícios:

- Um programa não seria mais restrito pela quantidade de memória física disponível. Os usuários poderiam escrever programas para um espaço de endereços *virtuais* extremamente grande, simplificando a tarefa de programação.
- Como cada programa do usuário poderia exigir menos memória física, mais programas poderiam ser executados ao mesmo tempo, com um aumento correspondente na utilização de CPU e throughput, mas sem aumento no tempo de resposta ou de retorno.
- Menos E/S seria necessária para carregar ou trocar cada programa do usuário para a memória, de modo que cada programa do usuário seria executado mais rapidamente.

Assim, a execução de um programa que não esteja inteiramente na memória beneficiaria tanto o sistema quanto o usuário.

A **memória virtual** envolve a separação entre a memória lógica, conforme percebida pelos usuários, e a memória física. Essa separação permite que uma memória virtual extremamente grande seja oferecida para os programadores quando somente uma memória física menor estiver disponível ([Figura 9.1](#)). A memória virtual torna a tarefa de programação muito mais fácil, pois o programador não precisa mais se preocupar com a quantidade de memória física disponível; em vez disso, ele pode se concentrar no problema a ser programado.

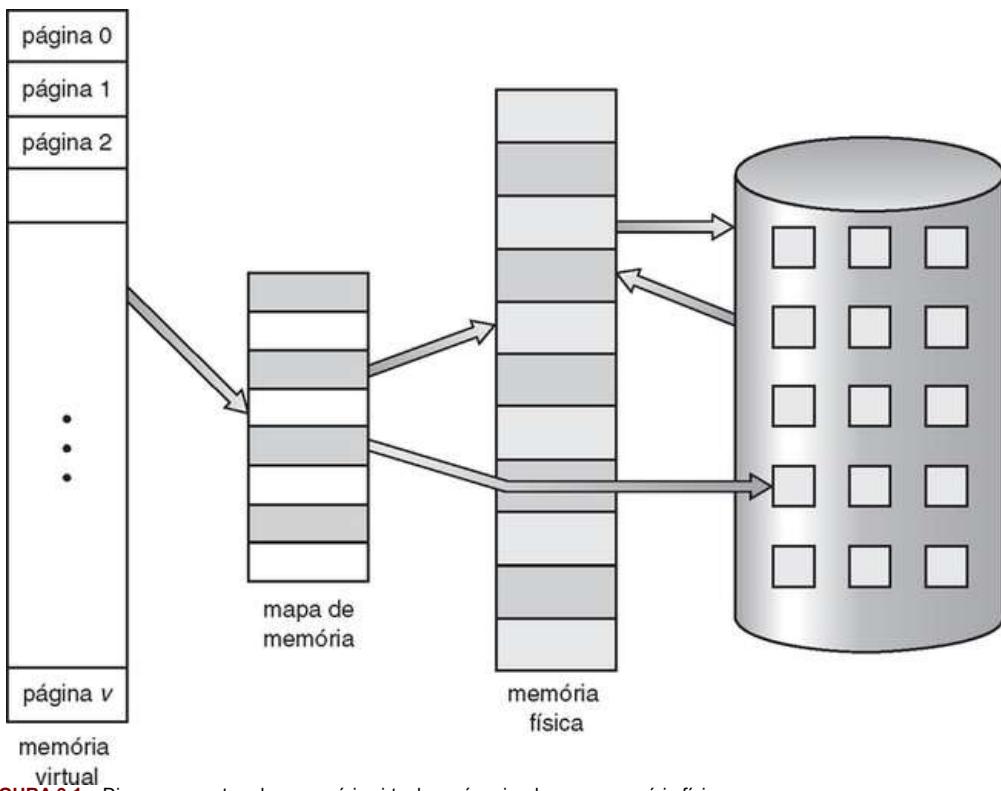


FIGURA 9.1 Diagrama mostrando a memória virtual que é maior do que a memória física.

O **espaço de endereços virtuais** de um processo refere-se à visão lógica (ou virtual) de como um processo é armazenado na memória. Normalmente, essa visão é que um processo começa em determinado endereço lógico – digamos, endereço 0 – e existe em trechos contíguos na memória, como mostra a [Figura 9.2](#). Contudo, como vimos no [Capítulo 8](#), na verdade a memória física pode ser organizada em quadros de página, e os quadros de página físicos atribuídos a um processo podem não ser contíguos. Fica a critério da unidade de gerência de memória (MMU) associar as páginas lógicas aos quadros de página físicos na memória.

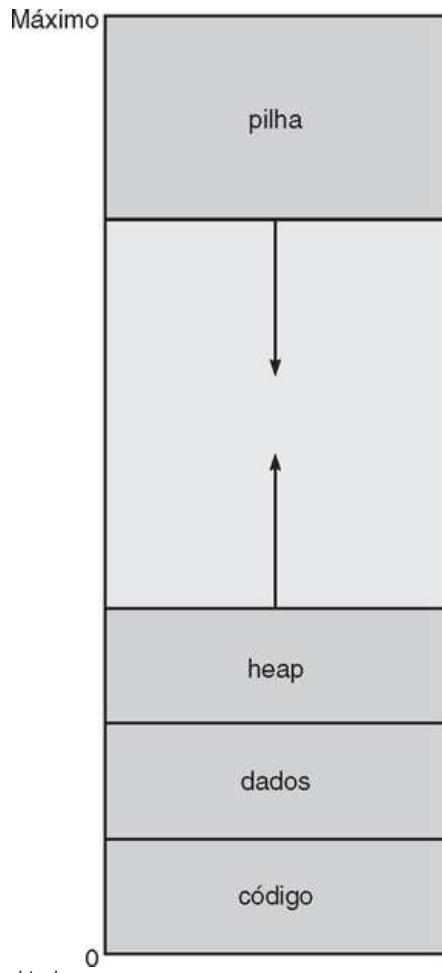


FIGURA 9.2 Espaço de endereços virtuais.

Observe, na [Figura 9.2](#), que permitimos que a heap cresça para cima na memória como é usada para a alocação dinâmica de memória. Da mesma maneira, permitimos que a pilha cresça para baixo na memória por meio de chamadas de função sucessivas. O grande espaço vazio (ou buraco) entre a heap e a pilha faz parte do espaço de endereços virtuais, mas só exigirá páginas físicas reais se a heap ou a pilha crescer. Os espaços de endereço virtual que incluem os buracos são conhecidos como espaços de endereços **esparsos**. O uso de um espaço de endereço esparsos é benéfico porque os buracos podem ser preenchidos enquanto os segmentos de pilha ou heap crescem, ou se quisermos vincular bibliotecas (ou possivelmente outros objetos compartilhados) de forma dinâmica durante a execução do programa.

Além de separar a memória lógica da memória física, a memória virtual também permite que arquivos e memória sejam compartilhados por dois ou mais processos diferentes, por meio do compartilhamento de página ([Seção 8.4.4](#)). Isso ocasiona os seguintes benefícios:

- As bibliotecas do sistema podem ser compartilhadas por vários processos diferentes, por meio do mapeamento do objeto compartilhado para um espaço de endereços virtuais. Embora cada processo considere as bibliotecas compartilhadas parte do seu espaço de endereços virtuais, as páginas reais em que as bibliotecas residem na memória física são compartilhadas por todos os processos ([Figura 9.3](#)). Em geral, uma biblioteca é mapeada apenas em relação ao espaço de cada processo vinculado a ela.

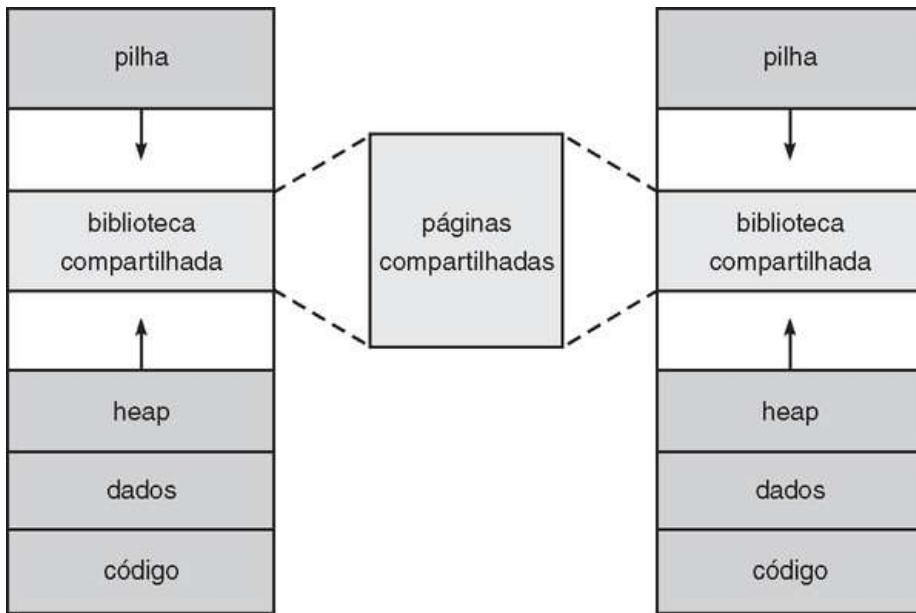


FIGURA 9.3 Biblioteca compartilhada usando memória virtual.

- De modo semelhante, a memória virtual permite aos processos compartilharem memória. Lembre-se de que, como vimos no [Capítulo 3](#), dois ou mais processos podem se comunicar usando memória compartilhada. A memória virtual permite a um processo criar uma região da memória que pode compartilhar com outro processo. Os processos que compartilham essa região a consideram parte do seu espaço de endereços virtuais, embora as páginas físicas da memória sejam compartilhadas, conforme ilustramos na [Figura 9.3](#).
- A memória virtual pode permitir que as páginas sejam compartilhadas durante a criação do processo, por meio da chamada de sistema `fork()`, agilizando, assim, a criação do processo. Exploramos esses e outros benefícios da memória virtual mais adiante neste capítulo. Primeiro, porém, discutimos a implementação da memória virtual por meio da paginação por demanda.

9.2 Paginação por demanda

Considere como um programa executável poderia ser carregado do disco para a memória. Uma opção é carregar o programa inteiro na memória física no momento da execução do programa. Contudo, um problema com essa técnica é que podemos não *precisar* inicialmente do programa inteiro na memória. Considere um programa que começa com uma lista de opções disponíveis, das quais o usuário deve selecionar. Carregar o programa inteiro para a memória resulta na carga do código executável para *todas* as opções, independente de uma opção, ao final, ser selecionada pelo usuário ou não. Uma estratégia alternativa é carregar as páginas apenas à medida que forem necessárias. Essa técnica é conhecida como **paginação por demanda** e normalmente é usada nos sistemas de memória virtual. Com a memória virtual paginada por demanda, as páginas só são carregadas quando forem exigidas durante a execução do programa; as páginas que nunca são acessadas, assim, nunca são carregadas para a memória física.

Um sistema de paginação por demanda é semelhante a um sistema de paginação com swap ([Figura 9.4](#)) no qual os processos residem na memória secundária (que normalmente é um disco). Quando queremos executar um processo, ele é passado para a memória. Todavia, em vez de trocar o processo inteiro para a memória, usamos um **lazzy swapper**. Um lazzy swapper nunca troca uma página para a memória a menos que essa página seja necessária. Como agora estamos exibindo um processo como uma sequência de páginas, em vez de um grande espaço de endereços contíguo, o uso do termo *swapper* é tecnicamente incorreto. Um swapper manipula processos inteiros, enquanto um **paginador** se preocupa com as páginas individuais de um processo. Assim, usamos *paginador*, em vez de *swapper*, em conjunto com a paginação por demanda.

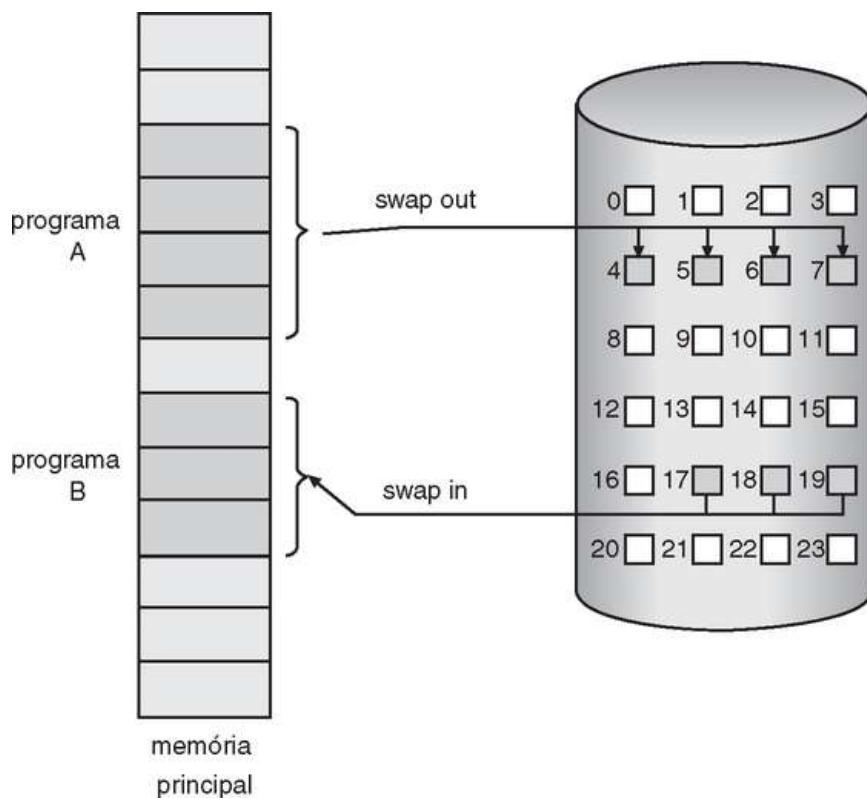


FIGURA 9.4 Transferência de uma memória paginada para o espaço contíguo no disco.

9.2.1 Conceitos básicos

Quando um processo tem de ser trocado para dentro da memória, o paginador imagina quais páginas serão usadas antes de o processo ser novamente removido da memória. Em vez de apanhar um processo inteiro, o paginador traz somente essas páginas para dentro da memória. Assim, ele evita ler páginas da memória que, de qualquer forma, não serão usadas, diminuindo o tempo de swap e a quantidade de memória física necessária.

Com esse esquema, precisamos de alguma forma de suporte do hardware para distinguir entre as páginas que estão na memória e as páginas que estão no disco. O esquema de bit válido-inválido descrito na [Seção 8.4.3](#) pode ser usado para essa finalidade. Entretanto, desta vez, quando esse bit

estiver definido como “válido”, a página associada é válida e está na memória. Se o bit estiver definido como “inválido”, a página não é válida (ou seja, não está no espaço de endereços lógicos do processo) ou é válida, mas atualmente está no disco. A entrada da tabela de página para uma página trazida para a memória é definida normalmente, mas a entrada da tabela de página para uma página que não está na memória é marcada como inválida ou contém o endereço da página no disco. Essa situação é representada na [Figura 9.5](#).

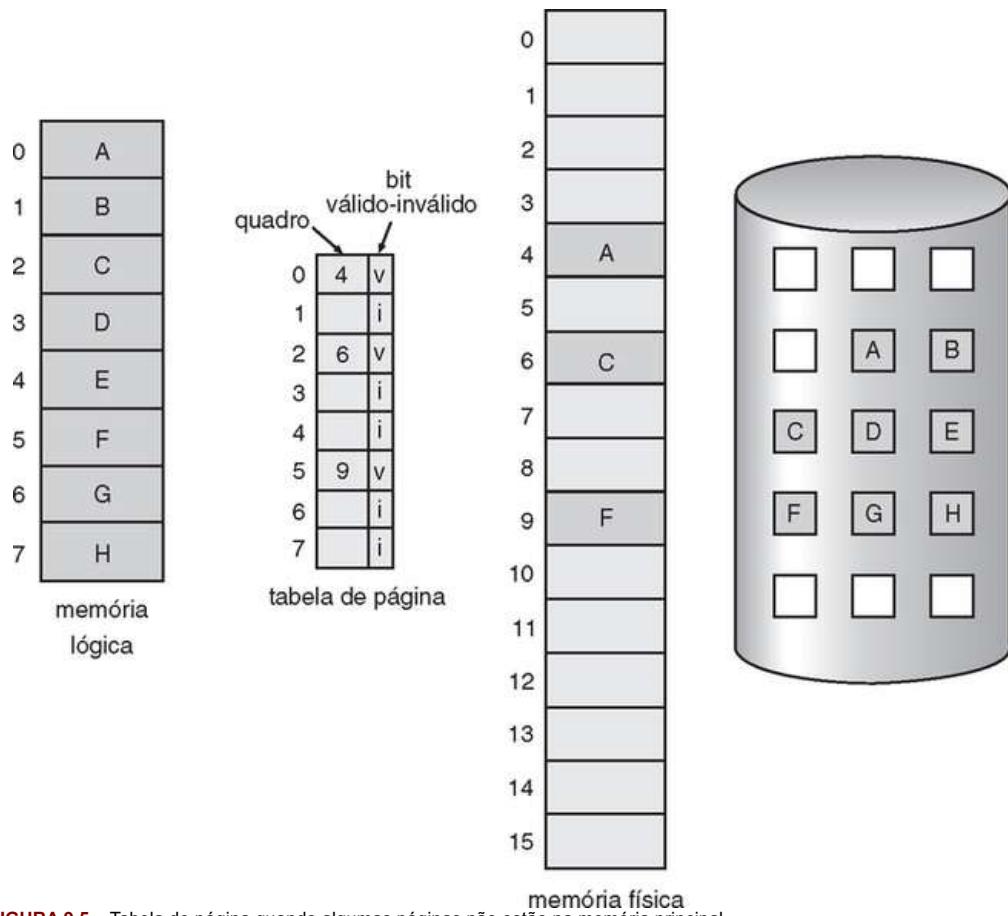


FIGURA 9.5 Tabela de página quando algumas páginas não estão na memória principal.

Observe que a marcação de uma página inválida não terá efeito se o processo nunca tentar acessar essa página. Logo, se fizermos a escolha certa e paginarmos todas e somente as páginas realmente necessárias, o processo será executado como se tivéssemos trazido todas as páginas. Enquanto o processo executa e acessa as páginas **residentes na memória**, a execução prossegue normalmente.

Contudo, o que acontece se o processo tentar acessar uma página que não foi trazida para a memória? O acesso a uma página marcada como inválida causa uma **falha de página (page fault)**. O hardware de paginação, ao traduzir o endereço na tabela de página, notará que o bit de inválido está marcado, causando uma interceptação para o sistema operacional. Essa interceptação é o resultado da falha do sistema operacional ao tentar trazer a página desejada para a memória. O procedimento para tratar dessa falha de página (page fault) é simples ([Figura 9.6](#)):

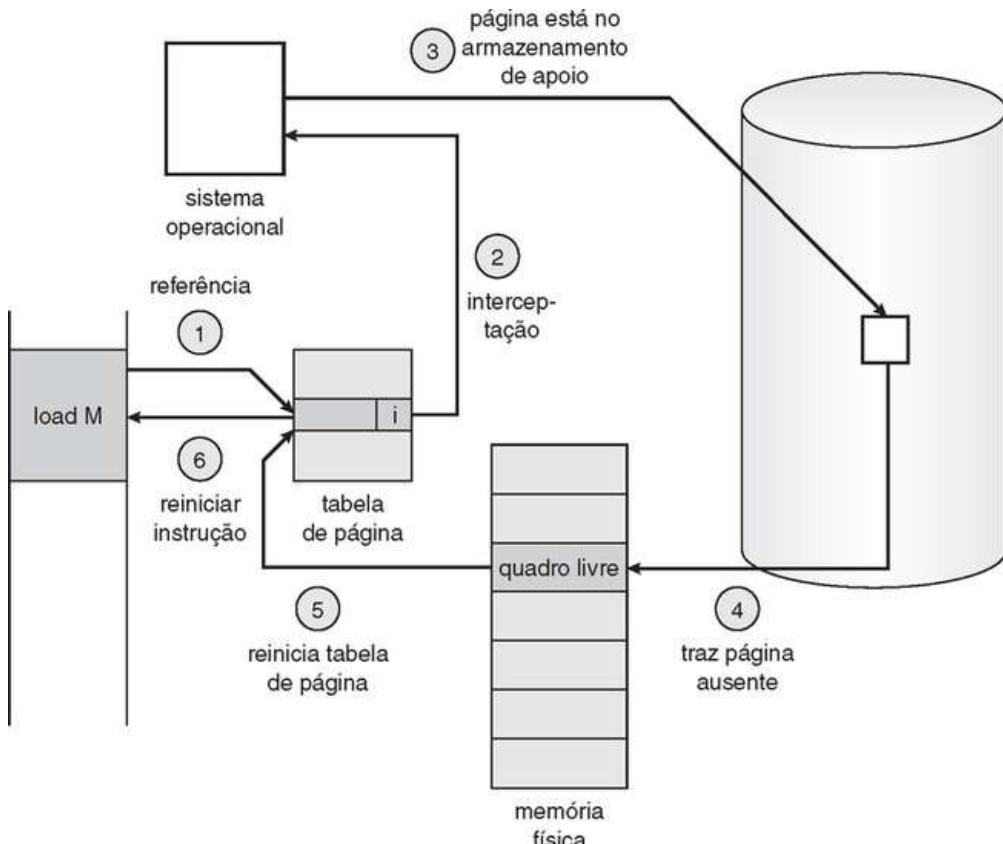


FIGURA 9.6 Etapas no tratamento de uma falha de página (page fault).

1. Verificamos uma tabela interna (mantida com o bloco de controle de processo) para esse processo, a fim de determinar se a referência foi um acesso de memória válido ou inválido.
2. Se a referência foi inválida, terminamos o processo. Se foi válida, mas ainda não trouxemos essa página, então a página é trazida para a memória.
3. Encontramos um quadro livre (apanhando um da lista de quadros livres, por exemplo).
4. Preparamos uma operação de disco para ler a página desejada para o quadro recém-alocado.
5. Quando a leitura do disco tiver sido concluída, modificamos a tabela interna mantida com o processo e a tabela de página, para indicar que a página agora está na memória.
6. Reiniciamos a instrução interrompida pela interceptação. O processo agora pode acessar a página como se sempre tivesse ficado na memória.

No caso extremo, podemos começar a executar um processo *sem* páginas na memória. Quando o sistema operacional definir o ponteiro de instrução para a primeira instrução do processo, que está em uma página não residente na memória, o processo imediatamente detectará a falha da página. Depois que essa página for trazida para a memória, o processo continuará sua execução, recuperando as páginas necessárias, até cada uma estar na memória. Nesse ponto, ele poderá executar sem falhas de página. Esse esquema é pura **paginação por demanda**: nunca trazer uma página para a memória até ser necessário.

Teoricamente, alguns programas podem acessar várias páginas de memória novas na execução de cada instrução (uma página para a instrução e muitas para dados), possivelmente causando várias falhas de página por instrução. Essa situação resultaria em um desempenho inaceitável para o sistema. Felizmente, a análise dos processos em execução mostra que esse comportamento é improvável. Os programas costumam ter **localidade de referência**, descrita na [Seção 9.6.1](#), que resulta em um desempenho razoável em relação à paginação por demanda.

O hardware para dar suporte à paginação por demanda é o mesmo que o hardware para a paginação e swap:

- **Tabela de página.** Essa tabela tem a capacidade de marcar uma entrada como inválida por meio de um bit de válido-inválido ou de um valor especial dos bits de proteção.
- **Memória secundária.** Essa memória mantém as páginas que não estão presentes na memória principal. A memória secundária é um disco de alta velocidade. Ele é conhecido como dispositivo de swap, e a seção do disco usada para essa finalidade é conhecida como **swap space**. A alocação do swap space é discutida no [Capítulo 12](#).

Um requisito crucial para a paginação por demanda é a habilidade de reiniciar qualquer instrução após uma falha de página. Como salvamos o estado (registradores, código de condição, contador de instrução) do processo interrompido quando ocorre uma falha de página, precisamos da capacidade de reiniciar o processo *exatamente* no mesmo local e estado, exceto que a página desejada agora

está na memória e é acessível. Na maioria dos casos, o requisito dessa facilidade de reinício é fácil de ser atendido. Uma falha de página poderia ocorrer em qualquer referência à memória. Se ela ocorrer enquanto estivermos buscando um operando, temos de buscar e decodificar a instrução novamente e depois buscar o operando.

Como exemplo do pior caso, considere uma instrução de três endereços, como somar (ADD) o conteúdo de A a B, colocando o resultado em C. Estas são as etapas para executar essa instrução:

1. Buscar e decodificar a instrução (ADD).
2. Buscar A.
3. Buscar B.
4. Somar A e B.
5. Armazenar a soma em C.

Se houver a falha de página quando tentarmos armazenar em C (porque C está em uma página ausente da memória), então teremos de apanhar a página desejada, colocá-la na memória, corrigir a tabela de página e reiniciar a instrução. O reinício exigirá buscar a instrução novamente, decodificá-la de novo, apanhar os dois operandos e depois somar mais uma vez. No entanto, não há muito trabalho repetido (menos de uma instrução completa), e a repetição é necessária apenas quando ocorre uma falha de página.

A maior dificuldade surge quando uma instrução pode modificar vários locais diferentes. Por exemplo, considere a instrução MVC (move character) do IBM System 360/370, que pode mover até 256 bytes de um local para outro local (possivelmente com overlay). Se um bloco (origem ou destino) ultrapassar um limite de página, uma falha de página poderá ocorrer após a movimentação estar parcialmente realizada. Além disso, se os blocos de origem e destino se sobreponem, o bloco de origem pode ter sido modificado, e nesse caso não poderemos reiniciar a instrução.

Esse problema pode ser resolvido de duas maneiras diferentes. Em uma solução, o microcódigo calcula e tenta acessar as duas extremidades de ambos os blocos. Se ocorrer uma falha de página, ela acontecerá nessa etapa, antes de algo ser modificado. A movimentação poderá ocorrer; sabemos que nenhuma falha de página poderá ocorrer, pois todas as páginas relevantes estão na memória. A outra solução utiliza registradores temporários para manter os valores dos locais modificados. Se houver uma falha de página, todos os valores antigos são escritos de volta à memória antes de ocorrer a interceptação. Essa ação restaura a memória ao seu estado anterior ao início da instrução, de modo que a instrução possa ser repetida.

De forma alguma esse é o único problema arquitetônico resultante do acréscimo da paginação a uma arquitetura existente para permitir a paginação por demanda, mas ilustra algumas das dificuldades. A paginação é acrescentada entre a CPU e a memória em um sistema computadorizado. Ela deverá ser transparente ao processo do usuário. Assim, as pessoas consideram que a paginação pode ser acrescentada a qualquer sistema. Embora essa suposição seja verdadeira para um ambiente de não paginação por demanda, em que uma falha de página representa um erro fatal, ela não é verdadeira onde uma falha de página significa apenas que uma página adicional precisa ser trazida para a memória e o processo precisa ser reiniciado.

9.2.2 Desempenho da paginação por demanda

A paginação por demanda pode afetar significativamente o desempenho de um sistema computadorizado. Para saber por que, vamos calcular o **tempo de acesso efetivo** para uma memória paginada por demanda. Para a maioria dos sistemas computadorizados, o tempo de acesso à memória, indicado como ma , varia de 10 a 200 nanosegundos. Se não tivermos falhas de página, o tempo de acesso efetivo será igual ao tempo de acesso à memória. Entretanto, se houver uma falha de página, primeiro temos de ler a página relevante do disco e depois acessar a palavra desejada.

Considere p a probabilidade de uma falha de página ($0 \leq p \leq 1$). Esperaríamos que p fosse próximo de zero, ou seja, esperaríamos ter apenas algumas poucas falhas de página. O **tempo de acesso efetivo**, portanto, é

$$\begin{aligned} \text{tempo de acesso efetivo} &= (1 - p) \times ma \\ &\quad + p \times \text{tempo da falha de pá}\end{aligned}$$

Para calcular o tempo de acesso efetivo, precisamos saber quanto tempo é necessário para tratar de uma falha de página. Uma falha de página ocasiona a seguinte sequência:

1. Interceptar (trap) para o sistema operacional.
2. Salvar os registradores do usuário e estado do processo.
3. Determinar se a interrupção foi uma falha de página.
4. Verificar se a referência à página foi válida e determinar o local da página no disco.
5. Emitir uma leitura do disco para um quadro livre:

- a. Esperar em uma fila para esse dispositivo até a requisição de leitura ser atendida.
 - b. Esperar pela busca de dispositivo e/ou tempo de latência.
 - c. Iniciar a transferência da página para o quadro livre.
6. Enquanto espera, alocar a CPU a algum outro usuário (escalonamento de CPU, opcional).
 7. Receber uma interrupção do subsistema de E/S de disco (E/S completada).
 8. Salvar os registradores e o estado do processo para o outro usuário (se a etapa 6 for executada).
 9. Determinar se a interrupção foi do disco.
 10. Corrigir a tabela de página e outras tabelas para mostrar que a página desejada agora está na memória.
 11. Esperar a CPU ser alocada a esse processo novamente.
 12. Restaurar os registradores do usuário, estado do processo e nova tabela de página, para depois retomar a instrução interrompida.

Nem todas essas etapas são necessárias em todos os casos. Por exemplo, supomos que, na etapa 6, a CPU seja alocada a outro processo enquanto ocorre a operação de E/S. Esse arranjo permite que a multiprogramação mantenha a utilização de CPU, mas exige um tempo adicional para retomar a rotina de atendimento à falha de página quando a transferência de E/S estiver concluída.

De qualquer forma, encaramos os três componentes principais do tempo de atendimento da falha de página:

1. Atender a interrupção de falha de página.
2. Ler a página para a memória.
3. Reiniciar o processo.

A primeira e a terceira tarefas podem ser reduzidas, com uma codificação cuidadosa, para várias centenas de instruções. Cada uma dessas tarefas pode levar de 1 a 100 microssegundos. No entanto, o tempo de swap de página será próximo de 8 milissegundos. (Um disco rígido típico possui uma latência média de 3 milissegundos, uma busca de 5 milissegundos e um tempo de transferência de 0,05 milissegundo. Assim, o tempo de paginação total é de aproximadamente 8 milissegundos, incluindo o tempo do hardware e do software.) Lembre-se também de que estamos examinando apenas o tempo de atendimento do dispositivo. Se uma fila de processos estiver aguardando pelo dispositivo, temos de somar o tempo de enfileiramento do dispositivo enquanto esperamos que ele esteja livre para atender à nossa requisição, aumentando ainda mais o tempo de swap.

Com um tempo médio para atendimento de falha de página de 8 milissegundos e um tempo de acesso à memória de 200 nanossegundos, então o tempo de acesso efetivo, em nanossegundos, é

$$\begin{aligned}
 \text{temp de acesso efetivo} &= \\
 (1 - p) \times (200) + p(8 \text{ milissegundos}) &= \\
 = (1 - p) \times 200 + p \times 8.000.000 &= \\
 = 200 + 7.999.800 \times p
 \end{aligned}$$

Portanto, podemos ver que o tempo de acesso efetivo é diretamente proporcional à **taxa de falha de página**. Se um acesso dentre 1.000 causar uma falha de página, o tempo de acesso efetivo é de 8,2 microssegundos. O computador seria atrasado por um fator de 40 por causa da paginação por demanda! Se quisermos uma degradação inferior a 10%, então precisamos de

$$\begin{aligned}
 220 > 200 + 7.999.800 \times p, \\
 20 > 7.999.800 \times p \\
 p < 0,0000025
 \end{aligned}$$

Em outras palavras, para manter o retardado devido à paginação em um nível razoável, podemos permitir que menos de um acesso à memória dentre 399.990 tenha falha de página. Em suma, é importante manter baixa a taxa de falha de página em um sistema de paginação por demanda. Caso contrário, o tempo de acesso efetivo aumenta, atrasando bastante a execução do processo.

Um aspecto adicional da paginação por demanda é o tratamento e o uso geral do swap space. A E/S de disco para o swap space em geral é mais rápida do que no sistema de arquivos. Isso porque o swap space é alocado em blocos muito maiores, e os métodos de pesquisa de arquivo e alocação indireta não são usados ([Capítulo 12](#)). O sistema pode, portanto, obter maior throughput de paginação copiando a imagem do arquivo inteiro para o swap space na partida do processo, e depois realizando a paginação por demanda do swap space. Outra opção é requisitar páginas do sistema de

arquivos inicialmente, mas escrever as páginas no swap space à medida que forem substituídas. Essa técnica garantirá que somente as páginas necessárias serão lidas do sistema de arquivos, mas que toda a paginação subsequente será feita a partir do swap space.

Alguns sistemas tentam limitar a quantidade de swap space usada por meio da paginação por demanda de arquivos binários. As páginas de demanda para tais arquivos são trazidas diretamente do sistema de arquivos. Contudo, quando a substituição de página é necessária, esses quadros podem ser substituídos (pois nunca são modificados), e as páginas podem ser lidas do sistema de arquivos para a memória novamente, se necessário. Usando essa técnica, o próprio sistema de arquivos serve como um armazenamento de apoio. Entretanto, o swap space ainda terá de ser usado para as páginas não associadas a um arquivo; essas páginas incluem a pilha e a heap para um processo. Esse método parece ser um meio-termo conveniente e é usado em vários sistemas, inclusive Solaris e BSD UNIX.

9.3 Cópia na escrita

Na [Seção 9.2](#), ilustramos como um processo pode ser iniciado pela simples paginação por demanda na página contendo a primeira instrução. No entanto, a criação de processos usando a chamada de sistema `fork()` pode evitar a necessidade de paginação por demanda usando uma técnica semelhante ao compartilhamento de página (explicado na [Seção 8.4.4](#)). Essa técnica providencia uma rápida criação de processo e minimiza o número de novas páginas que precisam ser alocadas ao processo recém-criado.

Lembre-se de que a chamada de sistema `fork()` cria um processo filho como uma duplicata de seu pai. Tradicionalmente, `fork()` funcionava criando uma cópia do espaço de endereços do pai para o filho, duplicando as páginas pertencentes ao pai. Todavia, considerando que muitos processos filhos invocam a chamada de sistema `exec()` imediatamente após a criação, a cópia do espaço de endereços do pai pode ser desnecessária. Como alternativa, podemos usar uma técnica conhecida como **cópia na escrita** (**copy on write**), que funciona permitindo que os processos pai e filho compartilhem as mesmas páginas. Essas páginas compartilhadas são marcadas como páginas de cópia na escrita, significando que, se um dos processos escrever em uma página compartilhada, uma cópia da página compartilhada será criada. A cópia na escrita é ilustrada nas [Figuras 9.7 e 9.8](#), que mostram o conteúdo da memória física antes e depois que o processo 1 modifica a página C.

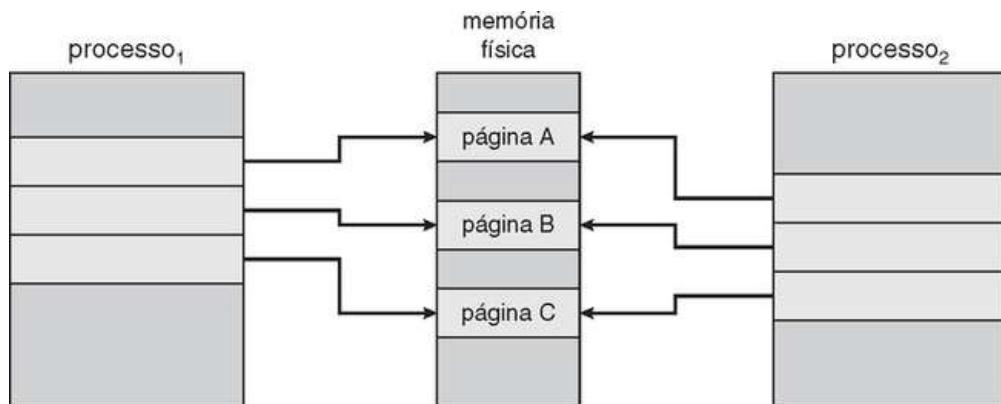


FIGURA 9.7 Antes de o processo 1 modificar a página C.

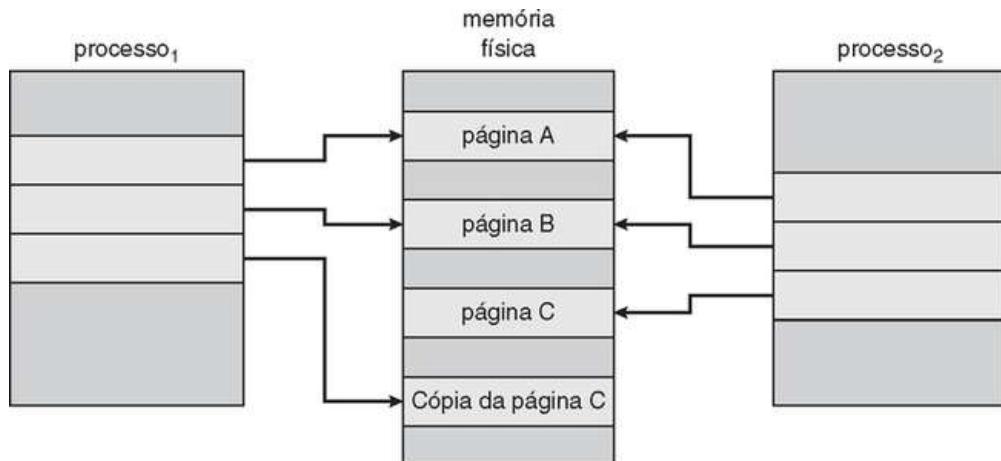


FIGURA 9.8 Depois de o processo 1 modificar a página C.

Por exemplo, suponha que o processo filho tente modificar uma página contendo partes da pilha, com as páginas definidas para a cópia na escrita. O sistema operacional criará, então, uma cópia dessa página, mapeando-a no espaço de endereços do processo filho. O processo filho modificará sua página copiada e não a página pertencente ao processo pai. É claro que quando a técnica de cópia na escrita for utilizada, somente as páginas modificadas por um dos processos serão copiadas; todas as páginas não modificadas podem ser compartilhadas pelos processos pai e filho. Observe, também, que somente as páginas que podem ser modificadas precisam ser marcadas como cópia na escrita. As páginas que não podem ser modificadas (páginas contendo código executável) podem ser

compartilhadas pelo pai e pelo filho. A cópia na escrita é uma técnica comum utilizada por vários sistemas operacionais, inclusive Windows XP, Linux e Solaris.

Quando for determinado que uma página será duplicada usando a cópia na escrita, é importante observar o local no qual a página livre será alocada. Muitos sistemas operacionais proveem um **banco** de páginas livres para tais requisições. Essas páginas livres são alocadas quando a pilha ou a heap para um processo precisa ser expandida ou quando há páginas de cópia na escrita para serem gerenciadas. Os sistemas operacionais alocam essas páginas usando uma técnica conhecida como **zerar por demanda (zero fill on demand)**. As páginas zeradas por demanda são zeradas antes de serem alocadas, apagando todo o conteúdo anterior.

Diversas versões do UNIX (incluindo Solaris e Linux) também proveem uma variação da chamada de sistema `fork()` - `vfork()` (de **virtual memory fork**). O `vfork()` opera de modo diferente de `fork()` com a cópia na escrita. Com `vfork()`, o processo pai é suspenso e o processo filho utiliza o espaço de endereços do pai. Como `vfork()` não usa a cópia na escrita, se o processo filho mudar quaisquer páginas do espaço de endereços do pai as páginas alteradas serão visíveis ao pai depois de ele retomar. Portanto, `vfork()` precisa ser usado com cautela, para garantir que o processo filho não modifique o espaço de endereços do pai. O `vfork()` deverá ser usado quando o processo filho chamar `exec()` após a criação. Como não ocorre cópia de páginas, `vfork()` é um método de criação de processo eficaz e, às vezes, é usado para implementar interfaces shell da linha de comandos no UNIX.

9.4 Substituição de página

Em nossa discussão inicial sobre taxa de falha de página, consideramos até certo ponto que cada página falha no máximo uma vez, quando é referenciada pela primeira vez. No entanto, essa representação não é estritamente exata. Se um processo de dez páginas usa apenas metade delas, então a paginação por demanda economiza a E/S necessária para carregar as cinco páginas que nunca são usadas. Também poderíamos aumentar nosso grau de multiprogramação executando o dobro de processos. Assim, se tivéssemos 40 quadros, poderíamos executar 8 processos, em vez dos 4 que poderiam ser executados se cada um exigisse 10 quadros (5 dos quais nunca foram usados).

Se aumentarmos nosso grau de multiprogramação, estaremos **superalocando** memória. Se executarmos 6 processos, cada um dos quais com 10 páginas de extensão, mas que, na realidade, utilizam apenas 5 páginas cada, teremos maior utilização de CPU e throughput, com 10 quadros poupanços. Entretanto, é possível que cada um desses processos, para determinado conjunto de dados, possa de repente tentar usar todas as suas 10 páginas, resultando em uma necessidade de 60 quadros, quando somente 40 estão disponíveis.

Além do mais, considere que a memória do sistema não é usada apenas para manter páginas de programa. Buffers para E/S também consomem uma quantidade significativa da memória. Esse uso pode aumentar o trabalho dos algoritmos de posicionamento de memória. A decisão de quanta memória deve ser alocada para E/S e quanta deve servir para as páginas de programa é um desafio significativo. Alguns sistemas alocam uma porcentagem fixa de memória para buffers de E/S, enquanto outros permitem que os processos do usuário e o subsistema de E/S disputem toda a memória do sistema.

A superalocação se manifesta da seguinte maneira. Enquanto um processo do usuário está sendo executado, ocorre uma falha de página. O sistema operacional determina onde a página desejada está residindo no disco, mas descobre que *não* existem quadros livres na lista de quadros livres: toda a memória está sendo utilizada ([Figura 9.9](#)).

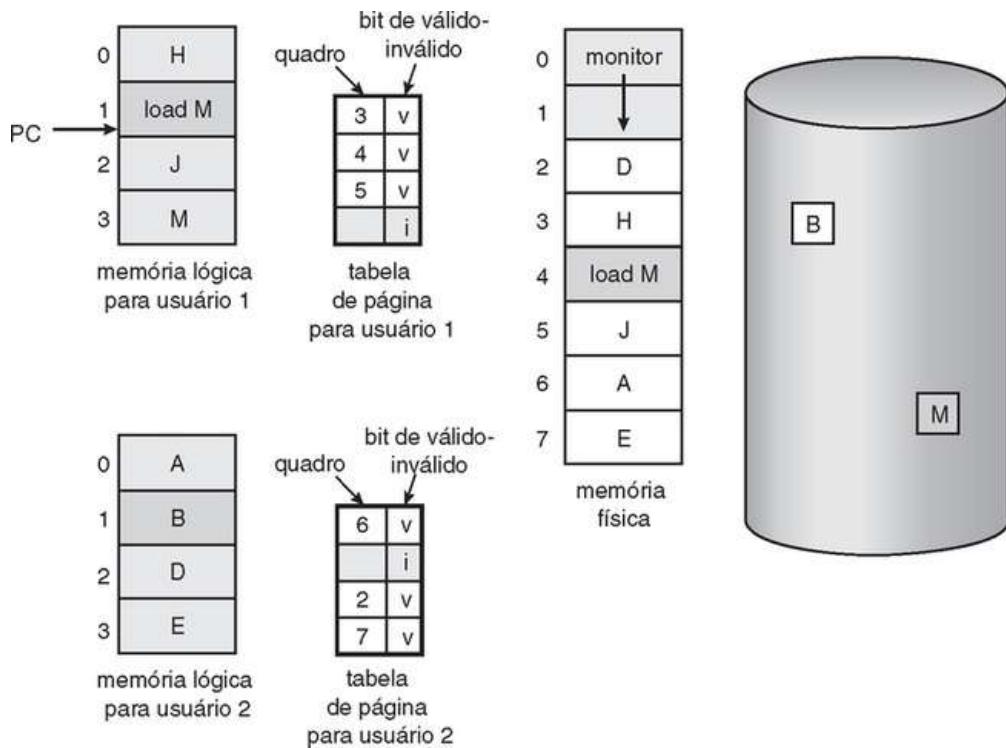


FIGURA 9.9 Necessidade de substituição de página.

O sistema operacional tem várias opções nesse ponto. Ele poderia terminar o processo do usuário. Entretanto, a paginação por demanda é a tentativa do sistema operacional de melhorar a utilização e o throughput do sistema computadorizado. Os usuários não deverão estar cientes de que seus processos estão rodando em um sistema paginado - a paginação deverá ser transparente ao usuário. Assim, essa opção não é a melhor escolha.

O sistema operacional poderia retirar um processo da memória, liberando todos os seus quadros e reduzindo o nível de multiprogramação. Essa opção é boa em certas circunstâncias; vamos considerá-la melhor na [Seção 9.6](#). Aqui, discutiremos a solução mais comum: **substituição de página**.

9.4.1 Substituição de página básica

A substituição de página utiliza a seguinte técnica. Se nenhum quadro estiver livre, encontraremos um que não esteja sendo usado e o liberaremos. Podemos liberar um quadro escrevendo seu conteúdo no swap space e alterando a tabela de página (e todas as outras tabelas) para indicar que a página não está mais na memória (Figura 9.10). Agora, podemos usar o espaço liberado para manter a página da qual o processo notou a falha. Modificamos a rotina de serviço de falha de página para incluir a substituição de página:

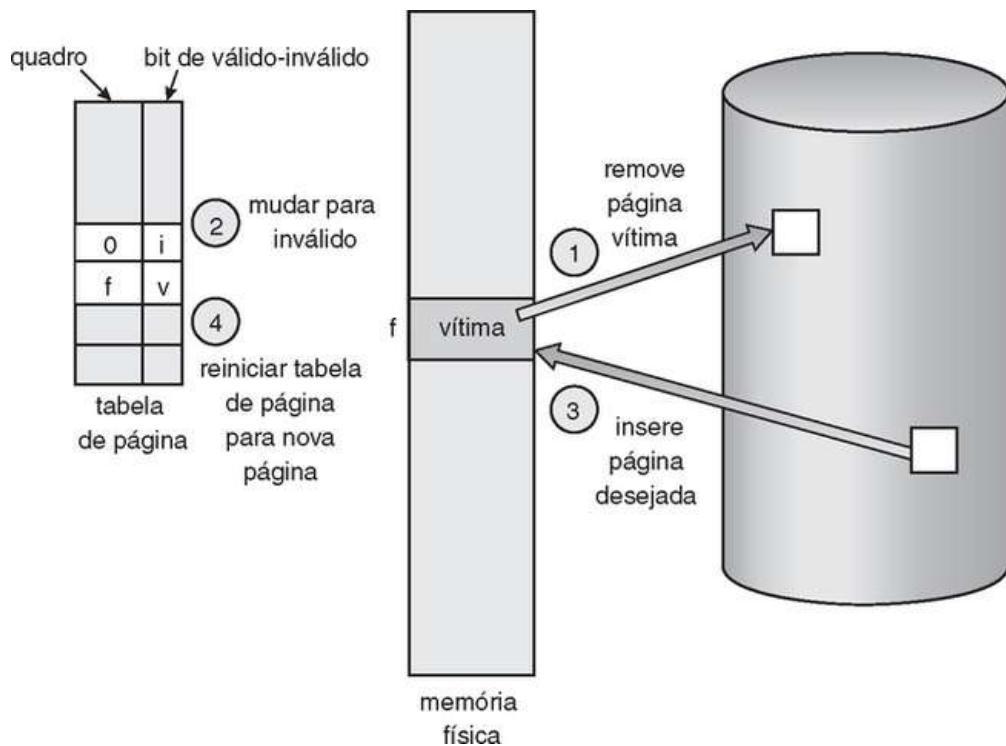


FIGURA 9.10 Substituição de página.

1. Encontrar o local da página desejada no disco.
2. Encontrar um quadro livre:
 - a. Se houver um quadro livre, use-o.
 - b. Se não houver um quadro livre, use um algoritmo de substituição de página para selecionar um **quadro vítima**.
 - c. Envie o quadro vítima para o disco; mude as tabelas de página e quadro de acordo.
3. Ler a página desejada para o quadro recém-liberado; mudar as tabelas de página e quadro.
4. Reiniciar o processo do usuário.

Observe que, se nenhum quadro estiver livre, *duas* transferências de página (uma para fora e outra para dentro) são exigidas. Essa situação efetivamente dobra o tempo de atendimento de falha de página e, consequentemente, aumenta o tempo de acesso efetivo.

Podemos reduzir esse custo adicional usando um **bit de modificação** (ou **bit de sujo**). Quando esse esquema é usado, cada página ou quadro pode ter um bit de modificação associado a ela no hardware. O bit de modificação para uma página é definido pelo hardware sempre que qualquer palavra ou byte forem escritos na página, indicando que a página foi modificada. Quando selecionamos uma página para substituição, examinamos seu bit de modificação. Se o bit estiver marcado, sabemos que a página foi modificada desde que foi lida do disco. Nesse caso, temos de escrever essa página no disco. No entanto, se o bit de modificação não estiver marcado, a página *não* foi modificada desde que foi lida para a memória. Nesse caso, não precisamos movimentar essa página da memória para o disco: ela já está lá. Essa técnica também se aplica a páginas somente de leitura (por exemplo, páginas de código binário). Essas páginas não podem ser modificadas; assim, elas podem ser descartadas quando for desejado. Esse esquema pode reduzir bastante o tempo exigido para atender a uma falha de página, pois reduz o tempo de E/S para a metade se a página não tiver sido modificada.

A substituição de página é algo básico da página por demanda. Ela completa a separação entre memória lógica e memória física. Com esse mecanismo, uma memória virtual enorme pode ser fornecida para os programadores em cima de uma memória física menor. Sem a paginação por demanda, os endereços do usuário são mapeados para endereços físicos, de modo que os dois conjuntos de endereços podem ser diferentes. Contudo, todas as páginas de um processo ainda precisam estar na memória física. Com a paginação por demanda, o tamanho do espaço de endereços lógico não é mais restrito pela memória física. Se tivermos um processo do usuário com

20 páginas, poderemos executá-lo em dez quadros usando a paginação por demanda e um algoritmo de substituição para encontrar um quadro livre sempre que necessário. Se uma página modificada tiver de ser substituída, seu conteúdo será copiado para o disco. Uma referência futura a essa página causará uma falha de página. Nesse momento, a página será trazida de volta para a memória, talvez substituindo alguma outra página no processo.

Temos de resolver dois problemas importantes para implementar a paginação por demanda: desenvolver um **algoritmo de alocação de quadro** e um **algoritmo de substituição de página**. Ou seja, se tivermos vários processos na memória, teremos de decidir quantos quadros serão alocados a cada processo. Além do mais, quando a substituição de página é exigida, temos de selecionar os quadros que devem ser substituídos. O projeto de algoritmos apropriados para solucionar esses problemas é uma tarefa importante, pois a E/S em disco é bastante dispendiosa. Até mesmo pequenas melhorias nos métodos de paginação por demanda geram enormes ganhos no desempenho do sistema.

Existem muitos algoritmos de substituição de páginas diferentes. Cada sistema operacional possui seu próprio esquema de substituição. Como selecionamos um algoritmo de substituição em particular? Em geral, queremos um com a menor taxa de falha de página.

Avaliamos um algoritmo executando-o em um determinado conjunto de referências à memória e calculando a quantidade de falhas de página. Esse conjunto de referências à memória é denominado **string de referência**. Podemos gerar strings de referências artificialmente (usando um gerador de números aleatórios, por exemplo) ou podemos rastrear determinado sistema e registrar o endereço de cada referência à memória. A segunda opção produz uma quantidade maior de dados (na ordem de 1 milhão de endereços por segundo). Para reduzir a quantidade de dados, usamos dois fatos.

Primeiro, para determinado tamanho de página (e o tamanho da página é fixado pelo hardware ou pelo sistema), precisamos considerar apenas o número da página e não o endereço inteiro. Segundo, se tivermos uma referência a uma página p , então quaisquer referências *imediatamente* após a página p nunca causarão uma falha de página (page fault). A página p estará na memória após a primeira referência; as referências seguintes não causarão falha de página.

Por exemplo, se rastreamos determinado processo, poderemos registrar a seguinte sequência de endereços:

```
0100, 0432, 0101, 0612, 0102, 0103, 0104,  
0101, 0611, 0102, 0103, 0104, 0101, 0610,  
0102, 0103, 0104, 0101, 0609, 0102, 0105
```

A 100 bytes por página, essa sequência é reduzida para a seguinte string de referência

```
1, 4, 1, 6, 1, 6, 1, 6, 1, 6, 1
```

Para determinar a quantidade de falhas de página para determinada string de referência e algoritmo de substituição de página, também precisamos saber a quantidade de quadros de página disponíveis. Obviamente, à medida que a quantidade de quadros disponíveis aumenta, o número de falhas de página diminui. Para a string de referência considerada anteriormente, por exemplo, se tivéssemos três ou mais quadros, teríamos apenas três falhas de página, uma falha para a primeira referência a cada página. Ao contrário, com somente um quadro disponível, teríamos uma substituição a cada referência, resultando em 11 falhas. Em geral, esperamos uma curva como a da [Figura 9.11](#). À medida que o número de quadros aumenta, o número de falhas de página cai para algum nível mínimo. Naturalmente, a inclusão de memória física aumenta a quantidade de quadros.

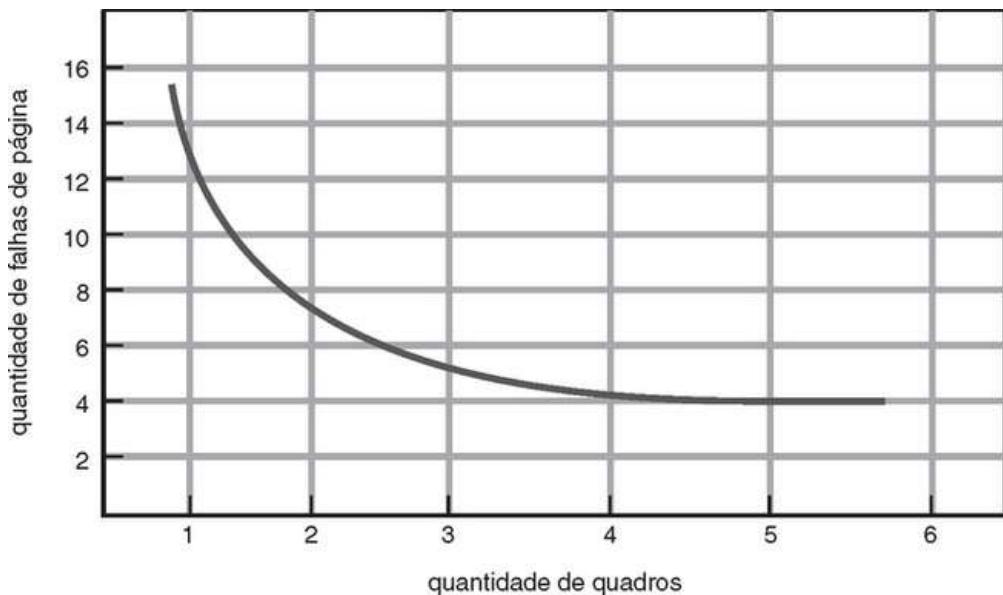


FIGURA 9.11 Gráfico para falhas de página versus quantidade de quadros.

A seguir, ilustramos diversos algoritmos de substituição de página. Ao fazer isso, usamos a string de referência

7, 0, 1, 2, 0, 3, 0, 4, 2, 3, 0, 3, 2, 1, 2, 0, 1, 7, 0, 1

para uma memória com três quadros.

9.4.2 Substituição de página FIFO

O algoritmo de substituição de página mais simples é um algoritmo first-in, first-out (FIFO). Um algoritmo de substituição FIFO associa a cada página a hora em que essa página foi trazida para a memória. Quando uma página tiver de ser substituída, a página mais antiga será escolhida. Observe que não é necessário registrar a hora em que uma página foi trazida. Podemos criar uma fila FIFO para manter todas as páginas na memória. Substituimos a página na cabeça da fila. Quando uma página é trazida para a memória, ela é inserida ao final da fila.

Para a nossa string de referência de exemplo, os três quadros estão inicialmente vazios. As três primeiras referências (7, 0, 1) causam falhas de página e são trazidas para esses quadros vazios. A próxima referência (2) substitui a **página 7**, porque a **página 7** foi trazida em primeiro lugar. Como 0 é a próxima referência e 0 já está na memória, não temos uma falha para essa referência. A primeira referência a 3 resulta na substituição da página 0, pois agora é a primeira na fila. Devido a essa troca, a próxima referência a 0 gerará uma falha. A **página 1** será substituída pela página 0. Esse processo continua como mostra a **Figura 9.12**. Toda vez que ocorre uma falha, mostramos quais páginas estão em nossos três quadros. Existem 15 falhas ao todo.

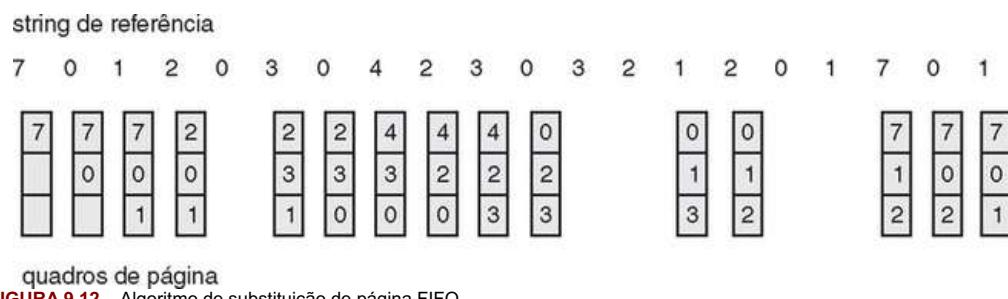


FIGURA 9.12 Algoritmo de substituição de página FIFO.

O algoritmo de substituição de página FIFO é fácil de entender e programar. Contudo, seu desempenho nem sempre é bom. Por um lado, a página substituída pode ser um módulo de inicialização, usado há muito tempo e não mais necessário. Por outro lado, ela poderia conter uma

variável bastante utilizada, inicializada desde cedo e com uso constante.

Observe que, mesmo se selecionarmos para substituição uma página em uso ativo, tudo ainda funciona corretamente. Depois de substituir uma página ativa por uma nova, haverá uma falha quase imediatamente, para apanhar a página ativa. Alguma outra página terá de ser substituída para trazer a página ativa de volta para a memória. Assim, uma escolha de substituição ruim aumenta a taxa de falha de página e atrasa a execução do processo. No entanto, não causa execução incorreta.

Para ilustrar os problemas que são possíveis com um algoritmo de substituição de página FIFO, considere a seguinte string de referência:

1,2,3,4,1,2,5,1,2,3,4,5

A Figura 9.13 mostra a curva de falhas de página para essa string de referência *versus* a quantidade de quadros disponíveis. Observamos que a quantidade de falhas para quatro quadros (dez) é *maior* do que a quantidade de falhas para três quadros (nove)! Esse resultado inesperado é conhecido como **anomalia de Belady**: para alguns algoritmos de substituição de página, a taxa de falha de página pode *aumentar* enquanto a quantidade de quadros alocados aumenta. Poderíamos esperar que, dando mais memória a um processo, seu desempenho melhorasse. Em uma pesquisa antiga, os investigadores notaram que essa suposição nem sempre era verdadeira. A anomalia de Belady foi descoberta como resultado.

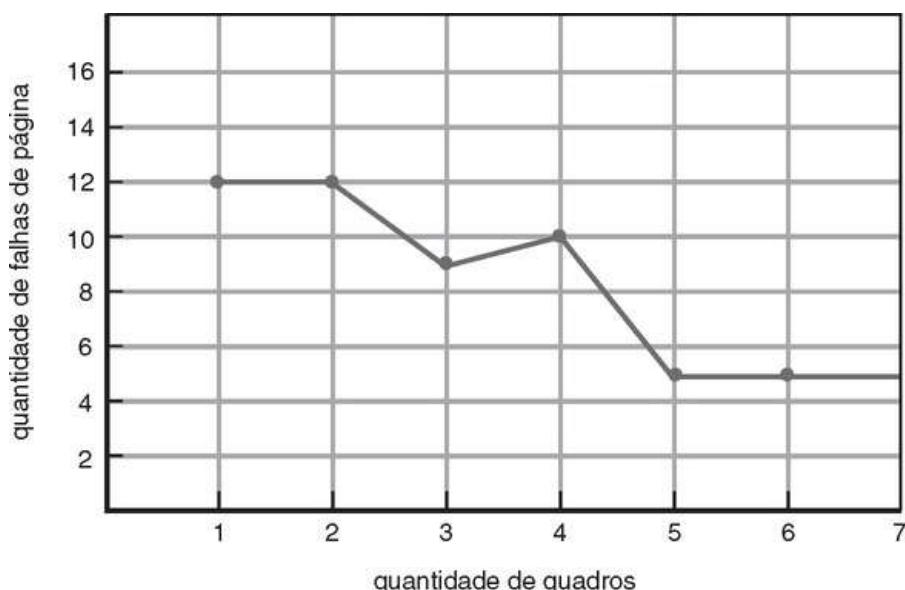


FIGURA 9.13 Curva de falha de página para a substituição FIFO sobre uma string de referência.

9.4.3 Substituição de página ótima

Um resultado da descoberta da anomalia de Belady foi a busca de um **algoritmo de substituição de página ótima**, que possui a menor taxa de falha de página de todos os algoritmos e nunca sofrerá com a anomalia de Belady. Esse algoritmo existe e foi chamado de OPT ou MIN. Ele é simples:

Substitua a página que não será usada
pelo maior período.

O uso desse algoritmo de substituição de página garante a menor taxa de falha de página possível para uma quantidade fixa de quadros.

Por exemplo, em nossa string de referências de exemplo, o algoritmo de substituição de página ótima geraria nove falhas de página, como mostra a Figura 9.14. As três primeiras referências causam falhas de página que preencherão os três quadros vazios. A referência à página 2 substitui a página 7 porque a 7 não será usada até a referência 18, enquanto a página 0 será usada na referência 5, e a página 1 na 14. A referência à página 3 substitui a página 1, pois a página 1 será a última das três páginas na memória a ser referenciada novamente. Com apenas 9 falhas de página, a substituição ideal é muito melhor do que o algoritmo FIFO, que tinha 15 falhas. (Se ignorarmos as

três primeiras falhas, pelas quais todos os algoritmos terão de passar, então a substituição ótima é duas vezes melhor do que a substituição FIFO.) De fato, nenhum algoritmo de substituição pode processar essa string de referência em três quadros com menos de nove falhas.

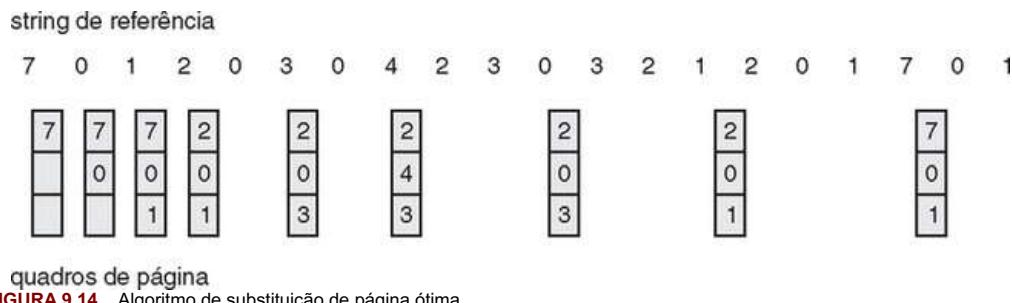


FIGURA 9.14 Algoritmo de substituição de página ótima.

Infelizmente, o algoritmo de substituição de página ótima é difícil de implementar, pois exige conhecimento futuro da string de referência. (Encontramos uma situação semelhante com o algoritmo de escalonamento de CPU SJF, discutido na [Seção 5.3.2](#).) Como resultado, o algoritmo ótimo é usado para estudos de comparação. Por exemplo, pode ser útil saber que, embora um novo algoritmo não seja ideal, está dentro dos 12,3% do ideal, no máximo, de dentro dos 4,7% da média.

9.4.4 Substituição de página LRU

Se o algoritmo ótimo não é viável, talvez uma aproximação do algoritmo ótimo seja possível. A distinção principal entre os algoritmos FIFO e OPT (além de examinar para trás *versus* para a frente no tempo) é que o algoritmo FIFO usa a hora em que uma página foi trazida para a memória, enquanto o algoritmo OPT usa a hora em que uma página deverá ser *usada*. Se usarmos o passado recente como uma aproximação para o futuro próximo, então poderemos substituir a página que *não foi usada* pelo maior período. Essa técnica é o **algoritmo Least Recently Used (menos recentemente usado - LRU)**.

A substituição LRU associa a cada página a hora do último uso dessa página. Quando uma página tiver de ser substituída, a LRU escolhe a página que não foi usada pelo maior período. Podemos considerar essa estratégia o algoritmo de substituição de página ótima examinando para trás no tempo, e não para frente. (Por mais estranho que pareça, se considerarmos S^R o reverso de uma string de referência S , então a taxa de falha de página para o algoritmo OPT sobre S é o mesmo que a taxa de falha de página para o algoritmo OPT sobre S^R . De modo semelhante, a taxa de falha de página para o algoritmo LRU sobre S é o mesmo que a taxa de falha de página para o algoritmo LRU sobre S^R .)

O resultado da aplicação da substituição LRU à nossa string de referência de exemplo aparece na [Figura 9.15](#). O algoritmo LRU produz 12 falhas. Observe que as 5 primeiras falhas são as mesmas da substituição ótima. Todavia, quando ocorre a referência à [página 4](#), a substituição LRU vê que, dos 3 quadros na memória, a página 2 foi usada menos recentemente. Assim, o algoritmo LRU substitui a página 2, sem saber que a página 2 está para ser usada. Quando, mais tarde, faltar a página 2, o algoritmo LRU substituirá a [página 3](#), pois agora é a página usada menos recentemente dentre as três páginas na memória. Apesar desses problemas, a substituição LRU com 12 falhas ainda é melhor do que a substituição FIFO com 15.

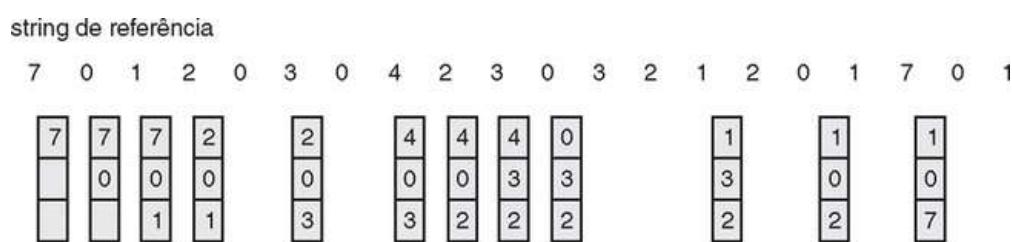


FIGURA 9.15 Algoritmo de substituição de página LRU.

A política LRU é usada como um algoritmo de substituição de página e é considerada boa. O maior problema é *como* implementar a substituição LRU. Um algoritmo de substituição de página LRU pode exigir assistência substancial do hardware. O problema é determinar a ordem para os quadros, definida pelo horário da última utilização. Duas implementações são viáveis:

■ **Contadores.** No caso mais simples, associamos a cada entrada da tabela de página um campo de tempo de uso e acrescentamos à CPU um relógio ou contador lógico. O relógio é incrementado para cada referência à memória. Sempre que é feita uma referência à página, o conteúdo do registrador de relógio é copiado para o campo de horário de uso na entrada de tabela de página para essa página. Desse modo, sempre temos o “horário” da última referência a cada página. Substituímos a página com o menor valor de horário. Esse esquema exige uma pesquisa da tabela de página para encontrar a página LRU e uma escrita na memória (para o campo de horário de uso na tabela de página) para cada acesso à memória. Os horários também precisam ser mantidos quando as tabelas de página são alteradas (devido ao escalonamento de CPU). O estouro do relógio também precisa ser considerado.

■ **Pilha.** Outra técnica para implementar a substituição LRU é manter uma pilha de números de página. Sempre que uma página é referenciada, ela é removida da pilha e colocada no topo. Desse modo, o topo da pilha sempre é a página usada mais recentemente, e a parte inferior é a página LRU (Figura 9.16). Como as entradas precisam ser removidas do meio da pilha, é melhor implementar essa técnica com o uso de uma lista duplamente encadeada, com um ponteiro para cabeça e outro para o final. A remoção de uma página e sua colocação no topo da pilha, então, exige a mudança de seis ponteiros no máximo. Cada atualização é um pouco mais dispendiosa, mas não existe pesquisa para uma substituição; o ponteiro do final aponta para o final da pilha, que é a página LRU. Essa técnica é particularmente apropriada para implementações de software ou microcódigo da substituição LRU.

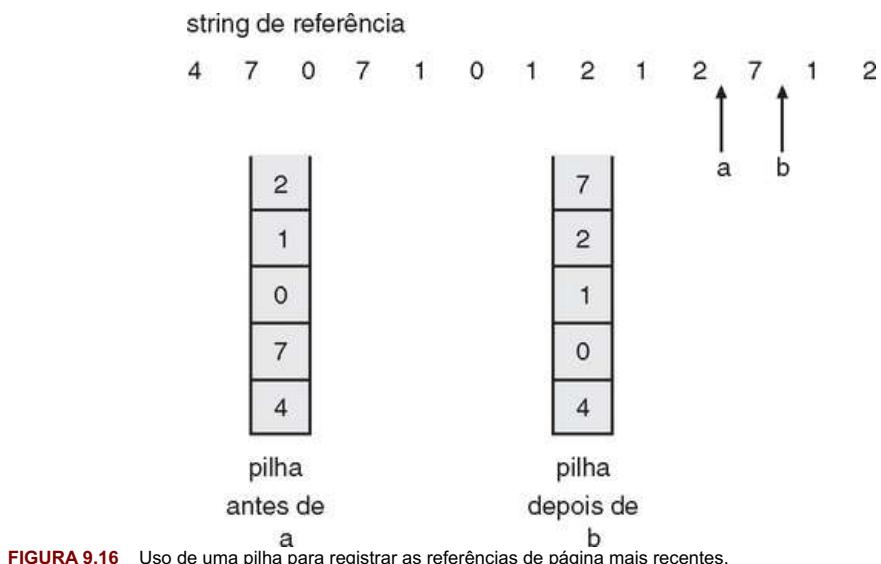


FIGURA 9.16 Uso de uma pilha para registrar as referências de página mais recentes.

Assim como a substituição ótima, a substituição LRU não sofre com a anomalia de Belady. Ambas pertencem a uma classe de algoritmos de substituição de página chamada **algoritmos de pilha**, que nunca poderão apresentar a anomalia de Belady. Um algoritmo de pilha é um algoritmo para o qual podemos mostrar que o conjunto de páginas na memória para n quadros sempre é um *subconjunto* do conjunto de páginas que estaria na memória com $n + 1$ quadros. Para a substituição LRU, o conjunto de páginas na memória seria as n páginas referenciadas mais recentemente. Se a quantidade de quadros for aumentada, essas n páginas ainda serão aquelas referenciadas mais recentemente e, por isso, ainda estarão na memória.

Observe que nenhuma implementação de LRU seria possível sem a assistência do hardware além dos registradores de TLB-padrão. A atualização dos campos de relógio ou da pilha precisa ser feita para *cada* referência à memória. Se tivéssemos de usar uma interrupção para cada referência para permitir ao software atualizar tais estruturas de dados, isso atrasaria cada referência à memória por um fator de pelo menos 10, atrasando cada processo do usuário por um fator de 10. Poucos sistemas poderiam tolerar esse custo adicional à gerência de memória.

9.4.5 Substituição de página por aproximação LRU

Poucos sistemas computadorizados proveem suporte de hardware suficiente para a verdadeira substituição de página LRU. Alguns sistemas não proveem suporte de hardware, e outros algoritmos de substituição de página (como o algoritmo FIFO) precisam ser utilizados. Entretanto, muitos sistemas proveem alguma ajuda na forma de um **bit de referência**. O bit de referência para uma página é definido, pelo hardware, sempre que a página é referenciada (seja com uma leitura ou uma escrita em qualquer byte na página). Os bits de referência estão associados a cada entrada na tabela

de página.

Inicialmente, todos os bits são apagados (com 0) pelo sistema operacional. À medida que um processo do usuário é executado, o bit associado a cada página referenciada é marcado (com 1) pelo hardware. Depois de algum tempo, podemos determinar quais páginas foram usadas e quais não foram usadas, examinando os bits de referência, embora não saibamos a *ordem* do uso. Essa informação é a base para muitos algoritmos de substituição de página, que se aproximam da substituição LRU. A seguir analisaremos mais de perto vários desses algoritmos.

9.4.5.1 Algoritmo de bits de referência adicionais

Podemos obter informações adicionais de ordenação registrando os bits de referência em intervalos regulares. Podemos manter um byte (8 bits) para cada página em uma tabela na memória. Em intervalos regulares (digamos, a cada 100 milissegundos), uma interrupção de temporizador transfere o controle para o sistema operacional. O sistema operacional desloca o bit de referência de cada página para o bit de alta ordem do seu byte de 8 bits, deslocando os outros bits para a direita em 1 bit e descartando o bit de baixa ordem. Esses registradores de deslocamento de 8 bits contêm o histórico do uso da página para os oito últimos períodos. Se o registrador de deslocamento tiver 00000000, por exemplo, então a página não foi usada por oito períodos; uma página usada pelo menos uma vez a cada período teria um valor de registrador de deslocamento igual a 11111111. Uma página com um valor de registrador de histórico igual a 11000100 foi usada mais recentemente do que uma com um valor 01110111. Se interpretarmos esses bytes de 8 bits como inteiros sem sinal, a página com o menor número é a página LRU e poderá ser substituída. Observe que os números precisam ser exclusivos. Podemos substituir (swap) todas as páginas com o menor valor ou usar o método FIFO para escolher entre elas.

A quantidade de bits de histórico incluída no registro de deslocamento pode ser variada, naturalmente, e seria selecionada (dependendo do hardware disponível) para tornar a atualização a mais rápida possível. No caso extremo, o número pode ser reduzido a zero, deixando apenas o próprio bit de referência. Esse algoritmo é chamado **algoritmo de substituição de página da segunda chance**.

9.4.5.2 Algoritmo da segunda chance

O algoritmo básico da substituição da segunda chance é um algoritmo de substituição FIFO. Entretanto, quando uma página tiver sido selecionada, inspecionamos seu bit de referência. Se o valor for 0, prosseguimos substituindo essa página, mas, se o bit de referência for definido como 1, damos uma segunda chance à página e selecionamos a próxima página FIFO. Quando uma página recebe uma segunda chance, seu bit de referência é apagado e sua hora de chegada é reiniciada para a hora atual. Assim, uma página que recebe uma segunda chance não será substituída até todas as outras páginas terem sido substituídas (ou recebido uma segunda chance). Além disso, se uma página for usada com frequência suficiente para manter seu bit de referência marcado, ela nunca será substituída.

Um modo de implementar o algoritmo da segunda chance (às vezes chamado algoritmo de *relógio*) é como uma fila circular. Um ponteiro (imagine um ponteiro do relógio) indica qual página deve ser substituída em seguida. Quando um quadro é necessário, o ponteiro avança até encontrar uma página com um bit de referência 0. Ao avançar, ele apaga os bits de referência ([Figura 9.17](#)). Quando uma página vítima é encontrada, ela é substituída, e a nova página é inserida na fila circular, nessa posição. Observe que, no pior caso, quando todos os bits estão marcados, o ponteiro percorre a fila inteira, dando a cada página uma segunda chance. Ele apaga todos os bits de referência antes de selecionar a próxima página para substituição. A substituição da segunda chance degenera a substituição FIFO se todos os bits estiverem marcados.

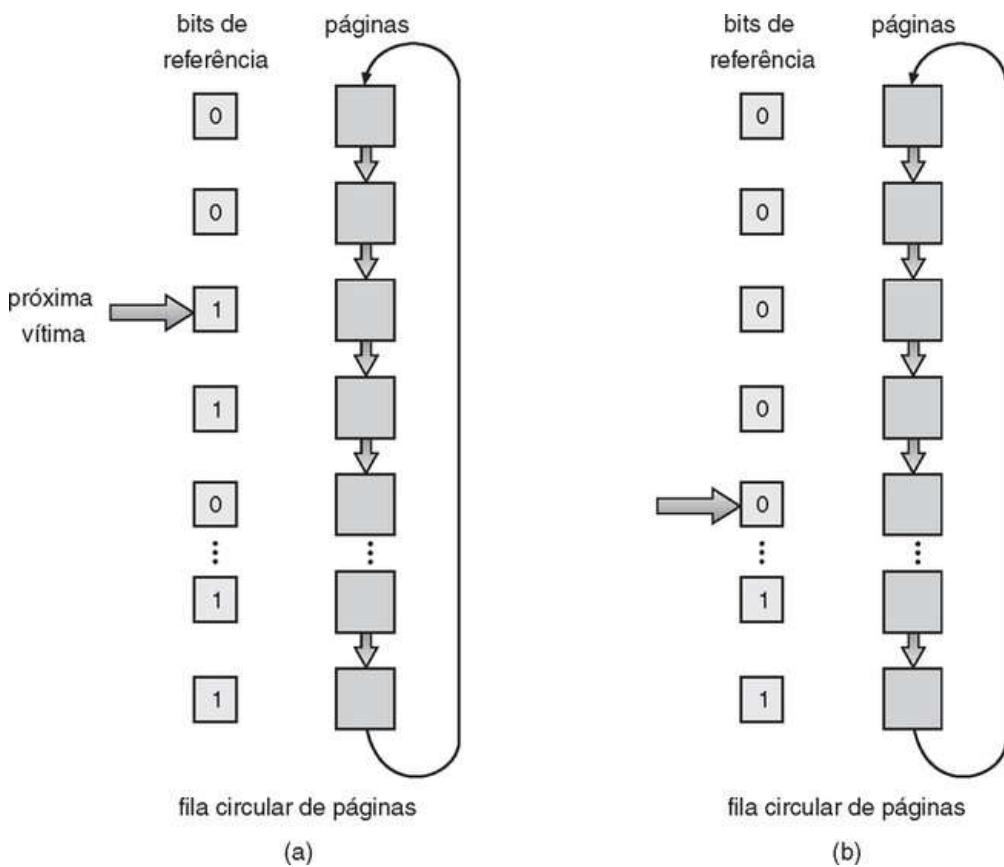


FIGURA 9.17 Algoritmo de substituição de página pela segunda chance (relógio).

9.4.5.3 Algoritmo da segunda chance melhorado

Podemos melhorar o algoritmo da segunda chance considerando o bit de referência e o bit de modificação (Seção 9.4.1) um par ordenado. Com esses dois bits, temos as quatro classes possíveis a seguir:

1. (0, 0) nem usada recentemente nem modificada - melhor página para substituir.
2. (0, 1) não usada recentemente, mas modificada - não tão bom, pois a página precisará ser escrita antes de substituir.
3. (1, 0) usada recentemente mas limpa - é provável que logo seja usada novamente.
4. (1, 1) usada recentemente e modificada - é provável que logo seja usada novamente, e a página terá de ser escrita no disco antes de poder ser substituída.

Cada página está em uma dessas quatro classes. Quando a substituição de página é necessária, usamos o mesmo esquema do algoritmo do relógio, mas, em vez de examinar se a página para a qual estamos apontando possui o bit de referência marcado como 1, examinamos a classe à qual essa página pertence. Substituímos a primeira página encontrada na classe não vazia mais baixa. Observe que podemos ter de varrer uma fila circular várias vezes antes de encontrar uma página a ser substituída.

A diferença principal entre esse algoritmo e o algoritmo de relógio mais simples é que, aqui, damos preferência às páginas modificadas, para reduzir a quantidade de E/S necessária.

9.4.6 Substituição de página baseada em contagem

Existem muitos outros algoritmos que podem ser usados para a substituição de página. Por exemplo, poderíamos manter um contador do número de referências feitas a cada página e desenvolver os dois esquemas a seguir:

- O algoritmo **Least-Frequently-Used (menos frequentemente usada - LFU)** exige que a página com a menor contagem seja substituída. O motivo para essa seleção é que uma página usada de maneira ativa deve ter um contador de referência grande. Contudo, um problema surgirá quando uma página for bastante usada durante a fase inicial de um processo, mas nunca mais for usada novamente. Como ela foi muito usada, terá uma contagem grande e permanecerá na memória mesmo que não seja mais necessária. Uma solução é deslocar as contagens para a direita em 1 bit em intervalos regulares, formando um contador de uso médio diminuindo exponencialmente.
- O **Most-Frequently-Used (mais frequentemente usada - MFU)** é baseado no argumento de que a página com a menor contagem provavelmente acabou de ser trazida para a memória e

ainda está para ser usada.

Como você poderia esperar, as substituições MFU e LFU não são comuns. A implementação desses algoritmos é dispendiosa, e eles não se aproximam muito da substituição OPT.

9.4.7 Algoritmos com buffer de página

Outros procedimentos são usados em conjunto com um algoritmo de substituição de página. Por exemplo, os sistemas mantêm um banco de quadros livres. Quando ocorre uma falha de página, um quadro vítima é escolhido como antes. Entretanto, a página desejada é lida para um quadro livre do banco antes de a vítima ser retirada. Esse procedimento permite ao processo reiniciar assim que possível, sem esperar que a página vítima seja retirada. Quando a vítima mais tarde for retirada, seu quadro será acrescentado ao banco de quadros livres.

Uma expansão dessa ideia é manter uma lista de páginas modificadas. Sempre que o dispositivo de paginação estiver ocioso, uma página modificada será selecionada e escrita no disco. Seu bit de modificação, então, é zerado. Esse esquema aumenta a probabilidade de uma página estar limpa quando for selecionada para substituição, e não precisará ser escrita no disco.

Outra modificação é manter um banco de quadros livres, mas lembrar qual página estava em cada quadro. Como o conteúdo do quadro não é modificado quando um quadro é escrito no disco, a página antiga pode ser reutilizada diretamente do banco de quadros livres se ela for necessária antes de esse quadro ser reutilizado. Nenhuma E/S é necessária nesse caso. Quando ocorrer uma falha de página, primeiro verificamos se a página desejada está no banco de quadros livres. Se não estiver, temos de selecionar um quadro livre e ler uma página para inserir nele.

Essa técnica é usada no sistema VAX/VMS em conjunto com o algoritmo de substituição FIFO. Quando o algoritmo de substituição FIFO por engano substituir uma página que ainda está em uso ativo, essa página será apanhada no banco de quadros livres e nenhuma E/S será necessária. O buffer de quadros livres provê proteção contra o algoritmo de substituição FIFO relativamente fraco, porém simples. Esse método é necessário porque as primeiras versões do VAX não implementavam o bit de referência de maneira correta.

Algumas versões do UNIX utilizam esse método em conjunto com o algoritmo da segunda chance. Também pode ser um aumento útil a qualquer algoritmo de substituição de página reduzir a penalidade incorrida na seleção incorreta da página vítima.

9.4.8 Aplicações e substituição de página

Em certos casos, as aplicações que acessam os dados por meio da memória virtual do sistema operacional funcionam pior do que se o sistema operacional não provesse buffer algum. Um exemplo típico é um banco de dados, que provê sua própria gerência de memória e buffers de E/S. As aplicações desse tipo entendem seu uso de memória e uso de disco melhor do que um sistema operacional que está implementando algoritmos para uso geral. Se o sistema operacional estiver mantendo a E/S em buffers, e a aplicação também estiver fazendo o mesmo, então o dobro da memória estará sendo usado para um conjunto de E/S.

Em outro exemplo, os data warehouses normalmente realizam grandes leituras sequenciais do disco, seguidas por cálculos e escritas. O algoritmo LRU estaria removendo páginas antigas e preservando as novas, enquanto a aplicação estaria lendo mais as páginas antigas do que as novas (enquanto inicia suas leituras sequenciais mais uma vez). Aqui, MFU seria mais eficiente do que LRU.

Por causa desse problema, alguns sistemas operacionais proveem a programas especiais a capacidade de usar uma partição de disco como um grande array sequencial de blocos lógicos, sem quaisquer estruturas de dados do sistema de arquivos. Esse array às vezes é chamado de **raw disk (disco bruto)**, e a E/S nesse array é considerada E/S bruta. A E/S bruta contorna todos os serviços do sistema de arquivos, como paginação por demanda da E/S de arquivos, lock de arquivo, busca antecipada, alocação de espaço, nomes de arquivo e diretórios. Observe que, embora certas aplicações sejam mais eficientes ao implementar seus próprios serviços de armazenamento de uso específico em uma partição bruta, a maioria das aplicações funciona melhor quando utilizam os serviços normais do sistema de arquivos.

9.5 Alocação de quadros

Agora abordaremos a questão da alocação. Como alocamos a quantidade fixa de memória livre entre os diversos processos? Se tivermos 93 quadros livres e dois processos, quantos quadros cada processo obterá?

O caso mais simples é o sistema monousuário. Considere um sistema monousuário com 128 KB de memória, compostos de páginas com tamanho de 1 KB. Esse sistema possui 128 quadros. O sistema operacional pode ocupar 35 KB, deixando 93 quadros para o processo do usuário. Sob a paginação por demanda pura, todos os 93 quadros inicialmente seriam colocados na lista de quadros livres. Quando um processo do usuário iniciasse a execução, ele geraria uma sequência de falhas de página (page fault). Todas as 93 primeiras falhas de página receberiam quadros livres da lista de quadros livres. Quando a lista de quadros livres estivesse esgotada, um algoritmo de substituição de página seria utilizado para selecionar uma das 93 páginas na memória para ser substituída pela 94^a, e assim por diante. Quando o processo terminasse, os 93 quadros seriam mais uma vez colocados na lista de quadros livres.

Existem muitas variações para essa estratégia simples. Podemos exigir que o sistema operacional aloque todo o seu espaço de buffer e tabela da lista de quadros livres. Quando esse espaço não estiver em uso pelo sistema operacional, ele poderá ser usado para dar suporte à paginação do usuário. Podemos tentar manter três quadros livres reservados na lista de quadros livres o tempo todo. Assim, quando houver uma falha de página, haverá um quadro livre disponível para paginar. Quando a troca de página estiver ocorrendo, uma substituição poderá ser selecionada e ser escrita no disco enquanto o processo continua a ser executado. Outras variantes também são possíveis, mas a estratégia básica é clara: o processo do usuário recebe qualquer quadro livre.

9.5.1 Quantidade mínima de quadros

Nossas estratégias para a alocação de quadros são limitadas de várias maneiras. Por exemplo, não podemos alocar mais do que a quantidade total de quadros disponíveis (a menos que haja compartilhamento de página). Também precisamos alocar pelo menos uma quantidade mínima de quadros. Aqui, veremos mais de perto esse último requisito.

Um motivo para alocar pelo menos uma quantidade mínima de quadros envolve o desempenho. Obviamente, à medida que a quantidade de quadros alocados a cada processo diminui, a taxa de falha de página aumenta, atrasando a execução do processo. Além disso, lembre-se de que, quando ocorre uma falha de página antes de uma instrução em execução terminar, a instrução precisa ser reiniciada. Consequentemente, precisamos ter quadros suficientes para manter todas as diferentes páginas que qualquer instrução isolada possa referenciar.

Por exemplo, considere uma máquina em que todas as instruções de referência à memória podem referenciar apenas um endereço de memória. Nesse caso, precisamos pelo menos de um quadro para a instrução e um quadro para a referência à memória. Além disso, se o endereçamento indireto em um nível for permitido (por exemplo, uma instrução load na [página 16](#) pode se referir a um endereço na página 0, uma referência indireta à [página 23](#)), então a paginação exige pelo menos três quadros por processo. Pense no que poderia acontecer se um processo tivesse apenas dois quadros.

A quantidade mínima de quadros é definida pela arquitetura do computador. Por exemplo, a instrução move para o PDP-11 inclui mais de uma palavra para alguns modos de endereçamento, e assim a instrução, em si, pode se estender por duas páginas. Além disso, cada um dos seus dois operandos pode conter referências indiretas, gerando um total de seis quadros. Outro exemplo é a instrução MVC do IBM 370. Como a instrução é de armazenamento para armazenamento, ela exige 6 bytes e pode se espalhar por duas páginas. O bloco de caracteres a mover e a área para a qual ele será movido também podem ser estender por duas páginas. Essa situação exigiria seis quadros. O pior dos casos ocorre quando a instrução MVC é o operando de uma instrução EXECUTE que ultrapassa um limite de página; nesse caso, precisamos de oito quadros.

O cenário do pior caso ocorre nas arquiteturas de computador que permitem múltiplos níveis de indireção (por exemplo, cada palavra de 16 bits poderia conter um endereço de 15 bits mais um indicador de indireção de 1 bit). Teoricamente, uma simples instrução load poderia referenciar um endereço indireto que poderia referenciar um endereço indireto (em outra página) que também poderia referenciar um endereço indireto (em outra página), e assim por diante, até cada página na memória virtual ter sido alcançada. Assim, no pior caso, a memória virtual inteira precisaria estar na memória física. Para contornar essa dificuldade, temos de colocar um limite sobre os níveis de indireção (por exemplo, limitar uma instrução a, no máximo, 16 níveis de indireção). Quando ocorrer a primeira indireção, um contador será definido como 16; o contador, em seguida, é decrementado para cada indireção sucessiva para essa instrução. Se o contador for decrementado até 0, haverá uma interceptação (indireção excessiva). Essa limitação reduz o número máximo de referências à memória por instrução para 17, exigindo a mesma quantidade de quadros.

Embora a quantidade mínima de quadros por processo seja definida pela arquitetura, a

quantidade máxima é definida pela quantidade de memória física disponível. Entre esses, ainda ficamos com uma escolha significativa para a alocação de quadros.

9.5.2 Algoritmos de alocação

O modo mais fácil de dividir m quadros entre n processos é dar a cada um uma fatia igual, m/n quadros. Por exemplo, se houver 93 quadros e 5 processos, cada processo receberá 18 quadros. Os 3 quadros restantes podem ser usados como um banco de buffer para quadros livres. Esse esquema é chamado de **alocação igual (equal allocation)**.

Uma alternativa é reconhecer que diversos processos precisarão de diferentes quantidades de memória. Considere um sistema com um tamanho de quadro de 1 KB. Se um pequeno processo de 10 KB e um banco de dados interativo de 127 KB forem os únicos dois processos executando em um sistema com 62 quadros livres, não faz muito sentido dar 31 quadros a cada processo. O processo menor não precisa de mais do que 10 quadros, de modo que os outros 21 são, rigorosamente falando, desperdiçados.

Para resolver esse problema, podemos utilizar a **alocação proporcional (proportional allocation)**, em que alocamos a memória disponível a cada processo de acordo com seu tamanho. Considere que o tamanho da memória virtual para o processo p_i seja s_i e defina:

$$S = \sum s_i$$

Então, se a quantidade total de quadros disponíveis é m , alocamos a_i quadros ao processo p_i , onde a_i é aproximadamente

$$a_i = s_i / S \times m$$

Naturalmente, precisamos ajustar cada a_i para ser um inteiro maior do que o número mínimo de quadros exigido pelo conjunto de instruções, com a soma não ultrapassando m .

Para a alocação proporcional, dividiríamos 62 quadros entre dois processos, um de 10 páginas e outro de 127 páginas, alocando 4 quadros e 57 quadros, respectivamente, pois

$$\begin{aligned} 10 / 137 \times 62 &\approx 4, \text{ e} \\ 127 / 137 \times 62 &\approx 57 \end{aligned}$$

Dessa maneira, os dois processos compartilham os quadros disponíveis de acordo com suas “necessidades” e não igualmente.

Na alocação igual e proporcional, a alocação pode variar de acordo com o nível de multiprogramação. Se o nível de multiprogramação for aumentado, cada processo perderá alguns quadros para prover a memória necessária para o novo processo. Reciprocamente, se o nível de multiprogramação diminuir, os quadros que foram alocados ao processo que saiu podem ser distribuídos para os processos restantes.

Observe que, com a alocação igual ou proporcional, um processo de alta prioridade é tratado da mesma forma que um processo de baixa prioridade. Por sua definição, porém, podemos querer dar ao processo de alta prioridade mais memória para agilizar sua execução, com o detimento dos processos de baixa prioridade. Uma solução é usar um esquema de alocação proporcional em que a taxa dos quadros depende não dos tamanhos relativos dos processos, mas das prioridades dos processos ou de uma combinação de tamanho e prioridade.

9.5.3 Alocação global versus local

Outro fator importante na forma como os quadros são alocados aos diversos processos é a substituição de página. Com vários processos competindo pelos quadros, podemos classificar os algoritmos de substituição de página em duas grandes categorias: **substituição global** e **substituição local**. A substituição global permite que um processo selecione um quadro de substituição do conjunto de todos os quadros, mesmo que esse quadro esteja atualmente alocado a

algum outro processo, ou seja, um processo pode apanhar um quadro de outro. A substituição local exige a cada processo selecionar somente a partir do seu próprio conjunto de quadros alocados.

Por exemplo, considere um esquema de alocação em que permitimos que processos de alta prioridade selecionem quadros de processos de baixa prioridade para substituição. Um processo pode selecionar uma substituição dentre seus próprios quadros ou dos quadros de qualquer processo de baixa prioridade. Essa técnica permite a um processo de alta prioridade aumentar sua alocação de quadro à custa de um processo de prioridade baixa.

Com uma estratégia de substituição local, a quantidade de quadros alocados a um processo não muda. Com a substituição global, um processo pode selecionar apenas quadros alocados a outros processos, aumentando, assim, a quantidade de quadros alocados a ele (supondo que outros processos não escolham *seus* quadros para substituição).

Um problema com um algoritmo de substituição global é que um processo não pode controlar sua própria taxa de falha de página. O conjunto de páginas na memória para um processo depende não apenas do comportamento de paginação desse processo, mas também do comportamento de paginação de outros processos. Portanto, o mesmo processo pode funcionar de modo muito diferente (por exemplo, levando 0,5 segundo para uma execução e 10,3 segundos para a execução seguinte), devido a circunstâncias externas. Isso não acontece com um algoritmo de substituição local. Sob a substituição local, o conjunto de páginas na memória para um processo é afetado pelo comportamento de paginação somente desse processo. Entretanto, a substituição local poderia atrapalhar um processo, não tornando disponível para ele outras páginas na memória. Assim, a substituição global em geral resulta em maior throughput e é o método mais usado.

9.5.4 Acesso não uniforme à memória

Até aqui em nossa abordagem da memória virtual, consideramos que toda a memória principal é criada da mesma forma – ou, pelo menos, que é acessada de modo igual. Em muitos sistemas de computação, isso não acontece. Geralmente, em sistemas com diversas CPUs ([Seção 1.3.2](#)), determinada CPU pode acessar algumas seções da memória principal mais rapidamente do que outras seções. Essas diferenças de desempenho são causadas pela forma como as CPUs e a memória são interconectadas no sistema. Constantemente, esse tipo de sistema é composto de várias placas de sistema, cada uma contendo diversas CPUs e alguma memória. Essas placas de sistema são interligadas de diversas maneiras, desde barramentos do sistema até conexões de rede de alta velocidade, como a InfiniBand. Como você poderia esperar, as CPUs em determinada placa podem acessar a memória nessa placa com menos atraso do que podem acessar a memória em outras placas no sistema. Sistemas nos quais os tempos de acesso à memória variam significativamente são coletivamente conhecidos como sistemas de **acesso não uniforme à memória** Non-Uniform Memory Access – NUMA e, sem exceção, eles são mais lentos do que os sistemas nos quais memória e CPUs estão localizadas na mesma placa-mãe.

Gerenciar quais quadros de página estão armazenados em quais locais pode afetar significativamente o desempenho em sistemas NUMA. Se tratarmos a memória como uniforme em um sistema desse tipo, as CPUs podem ter que esperar muito mais tempo pelo acesso à memória do que se modificarmos os algoritmos de alocação de memória para levar em consideração o NUMA. Mudanças semelhantes deverão ser feitas no sistema de escalonamento. O objetivo dessas mudanças é ter quadros de memória alocados “o mais próximo possível” da CPU em que o processo está sendo executado. A definição de “próximo” é “dentro da menor latência”, o que geralmente significa na mesma placa de sistema que a CPU.

As mudanças algorítmicas consistem em ter um escalonador rastreando a última CPU em que cada processo foi executado. Se o escalonador tentar escalonar cada processo para a sua CPU anterior, e o sistema de gerência de memória tentar alocar quadros para o processo próximo da CPU em que está sendo escalonado, então o resultado poderá ser visto em melhores taxas de acerto de cache e menores tempos de acesso à memória.

Essa situação se torna mais complicada quando as threads são acrescentadas. Por exemplo, um processo com muitas threads em execução pode acabar com essas threads escalonadas em muitas placas de sistema diferentes. Como a memória deverá ser alocada nesse caso? O Solaris resolve o problema criando uma entidade **lgroup** no kernel. Cada lgroup reúne CPUs e memória próximas. Na verdade, existe uma hierarquia de lgroups com base na quantidade de latência entre os grupos. O Solaris tenta escalonar todas as threads de um processo e alocar toda a memória de um processo dentro de um lgroup. Se isso não for possível, ele apanha lgroups próximos para o restante dos recursos necessários. Dessa maneira, a latência geral da memória é reduzida e as taxas de acerto de cache são aumentadas.

9.6 Thrashing

Se a quantidade de quadros alocados a um processo de baixa prioridade ficar abaixo da quantidade mínima exigida pela arquitetura do computador, precisamos suspender a execução desse processo. Depois, devemos retirar suas páginas restantes, liberando todos os seus quadros alocados. Essa provisão introduz um nível de entrada e saída da memória do escalonamento intermediário da CPU.

De fato, examine qualquer processo que não tenha quadros “suficientes”. Se o processo não tiver a quantidade de quadros de que precisa para dar suporte às páginas em uso ativo, ele terá falha de página. Nesse ponto, ele terá de substituir alguma página. No entanto, como todas as suas páginas estão em uso ativo, ele precisa substituir uma página que será necessária novamente em seguida. Como consequência, ele logo causará outra falha (fault) e, de novo e de novo, substituindo páginas que terá de trazer de volta imediatamente.

Essa alta atividade de página é denominada **thrashing**. Um processo está realizando thrashing se estiver gastando mais tempo paginando do que executando.

9.6.1 Causa do thrashing

O thrashing resulta em problemas de desempenho severos. Considere o cenário a seguir, baseado no comportamento real dos primeiros sistemas de paginação.

O sistema operacional monitora a utilização de CPU. Se a utilização de CPU estiver baixa, aumentamos o grau de multiprogramação introduzindo um novo processo no sistema. Um algoritmo de substituição de página global é utilizado; ele substitui as páginas sem considerar o processo ao qual pertencem. Agora, suponha que um processo entre em uma nova fase em sua execução e precise de mais quadros. Ele começa a causar falhas e a remover quadros de outros processos. Contudo, esses processos precisam dessas páginas e, por isso, também causarão falhas, tomando quadros de outros processos. Esses processos com falhas precisam usar o dispositivo de paginação para enviar e receber páginas. À medida que são enfileirados para o dispositivo de paginação, a fila de prontos se esvazia. Enquanto os processos esperam pelo dispositivo de paginação, a utilização da CPU diminui.

O escalonador de CPU vê a utilização de CPU diminuindo e *aumenta* o grau de multiprogramação como resultado. O novo processo tenta iniciar apanhando quadros dos processos em execução, causando mais falhas de página e uma fila maior para o dispositivo de paginação. Como resultado, a utilização de CPU cai ainda mais e o escalonador de CPU tenta aumentar o grau de multiprogramação. Acontecendo o thrashing, o throughput do sistema decai vertiginosamente. A taxa de falha de página aumenta muito. Como resultado, o tempo efetivo de acesso à memória aumenta. Nenhum trabalho é realizado, pois os processos estão gastando todo o seu tempo paginando.

Esse fenômeno é ilustrado na [Figura 9.18](#), em que a utilização de CPU é comparada com o grau de multiprogramação. À medida que o grau de multiprogramação aumenta, a utilização de CPU também aumenta, embora mais lentamente, até ser alcançado um máximo. Se o grau de multiprogramação for aumentado ainda mais, o thrashing entra em cena, e a utilização de CPU cai bruscamente. Nesse ponto, para aumentar a utilização de CPU e acabar com o thrashing, temos de *diminuir* o grau de multiprogramação.

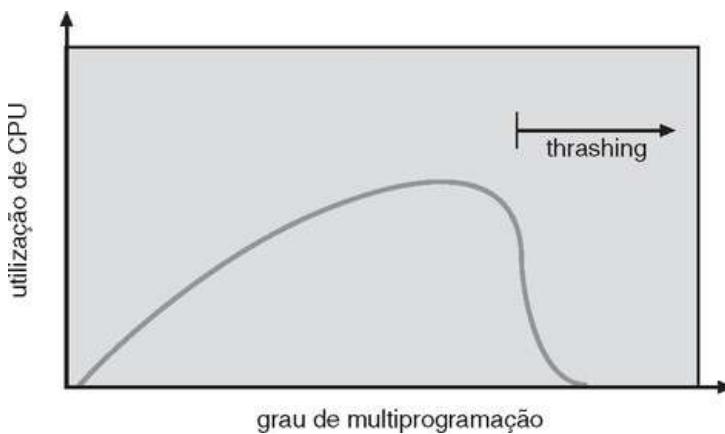


FIGURA 9.18 Thrashing.

Podemos limitar os efeitos do thrashing usando um **algoritmo de substituição local** (ou **algoritmo de substituição por prioridade**). Com a substituição local, se um processo iniciar o thrashing ele não poderá roubar quadros de outro processo e fazer o último também causar o

thrashing. Entretanto, o problema não está solucionado. Se os processos estiverem causando thrashing, eles estarão na fila para o dispositivo de página na maior parte do tempo. O tempo de atendimento médio para uma falha de página aumentará devido à fila média maior para o dispositivo de paginação. Assim, o tempo de acesso efetivo aumentará até mesmo para um processo que não está causando o thrashing.

Para impedir o thrashing, temos de fornecer a um processo tantos quadros quanto precisar. Todavia, como saberemos de quantos quadros ele “precisa”? Uma estratégia começa examinando quantos quadros um processo está usando. Essa técnica define o **modelo de localidade** da execução do processo.

O modelo de localidade afirma que, à medida que um processo é executado, ele passa de uma localidade para outra. Uma localidade é um conjunto de páginas que são usadas ativamente juntas (Figura 9.19). Um programa em geral é composto de várias localidades, que podem se sobrepor.

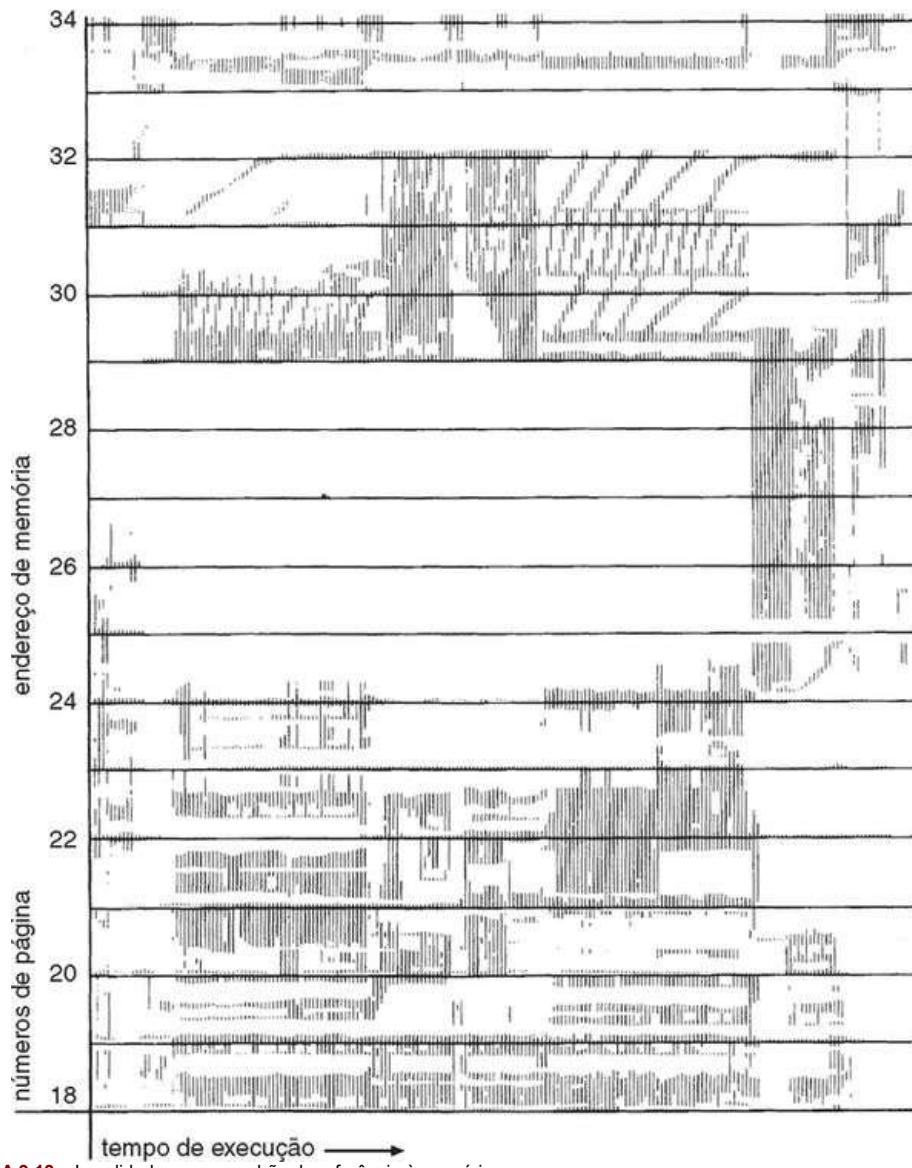


FIGURA 9.19 Localidade em um padrão de referência à memória.

Por exemplo, quando uma sub-rotina é chamada, ela define uma nova localidade. Nessa localidade, as referências à memória são feitas às instruções da chamada de função, suas variáveis locais e um subconjunto das variáveis globais. Quando encerramos a função, o processo deixa essa localidade, pois as variáveis locais e as instruções da função não estão mais em uso ativo. Podemos retornar a essa localidade mais adiante.

Assim, vemos que as localidades são definidas pela estrutura do programa e por suas estruturas de dados. O modelo de localidade afirma que todos os programas exibirão essa estrutura básica de referência da memória. Observe que o modelo de localidade é o princípio não declarado por trás das discussões sobre caching até este ponto no livro. Se os acessos a quaisquer tipos de dados forem aleatórios, sem um padrão, o caching será inútil.

Suponha que aloquemos quadros suficientes a um processo para acomodar sua localidade atual. Ele terá falha de páginas em sua localidade até todas as páginas estarem na memória; depois, ele não terá falha novamente até mudar de localidade. Se não alocarmos quadros suficientes para acomodar o tamanho da localidade atual, o processo falhará, já que não poderá manter na memória todas as páginas que está usando.

9.6.2 Modelo de conjunto de trabalho

O **modelo de conjunto de trabalho** (**working set model**) é baseado na suposição de localidade. Esse modelo utiliza um parâmetro, Δ , para definir a **janela do conjunto de trabalho** (**working set window**). A ideia é examinar as referências da página Δ mais recentes. O conjunto de páginas nas referências de página Δ mais recente é o **conjunto de trabalho** (Figura 9.20). Se uma página estiver em uso ativo, ela estará no conjunto de trabalho. Se não estiver mais sendo usada, ela sairá do conjunto de trabalho Δ unidades de tempo após sua última referência. Assim, o conjunto de trabalho é uma aproximação da localidade do programa.

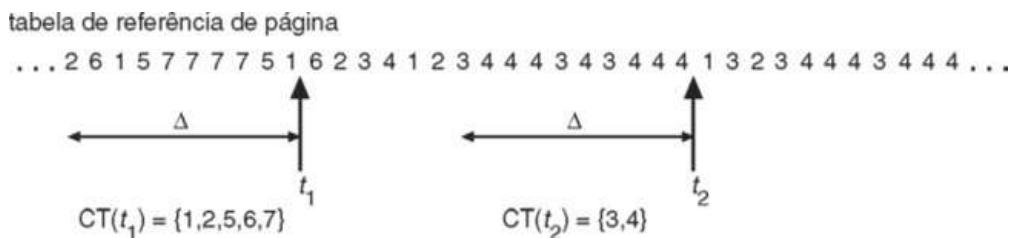


FIGURA 9.20 Modelo do conjunto de trabalho.

Por exemplo, dada a string de referência de memória mostrada na Figura 9.20, se $\Delta = 10$ referências de memória, então o conjunto de trabalho no momento t_1 é $\{1, 2, 5, 6, 7\}$. No momento t_2 , o conjunto de trabalho mudou para $\{3, 4\}$.

A precisão do conjunto de trabalho depende da seleção de Δ . Se Δ for muito pequeno, ele não abrangerá a localidade inteira; se Δ for muito grande, ele poderá sobrepor várias localidades. No extremo, se Δ for infinito, o conjunto de trabalho será o conjunto de todas as páginas referenciadas durante a execução do processo.

A propriedade mais importante do conjunto de trabalho, portanto, é o seu tamanho. Se calcularmos o tamanho do conjunto de trabalho, CTS_i , para cada processo no sistema, poderemos considerar

$$D = \sum CTS_i$$

onde D é a demanda total pelos quadros. Cada processo está usando as páginas em seu conjunto de trabalho. Assim, o processo i precisa de CTS_i quadros. Se a demanda total for maior do que a quantidade total de quadros disponíveis ($D > m$), haverá thrashing, pois alguns processos não terão quadros suficientes.

Quando Δ tiver sido selecionado, o uso do modelo do conjunto de trabalho será simples. O sistema operacional monitora o conjunto de trabalho de cada processo e aloca a esse conjunto de trabalho quadros suficientes para fornecer o tamanho do seu conjunto de trabalho. Se houver quadros extras suficientes, outro processo poderá ser iniciado. Se a soma dos tamanhos de conjunto de trabalho aumentar, ultrapassando a quantidade total de quadros disponíveis, o sistema operacional selecionará um processo para suspender. As páginas do processo são retiradas (swapped) e seus quadros são realocados para outros processos. O processo suspenso pode ser reiniciado mais tarde.

A dificuldade com o modelo de conjunto de trabalho é acompanhar o conjunto de trabalho. A janela do conjunto de trabalho é uma janela móvel. A cada referência à memória, uma nova referência aparece em uma extremidade e a referência mais antiga é descartada pela outra extremidade. Uma página está no conjunto de trabalho se for referenciada em algum lugar na janela do conjunto de trabalho.

Podemos aproximar o modelo do conjunto de trabalho com uma interrupção de temporizador com intervalo fixo e um bit de referência. Por exemplo, suponha que Δ seja igual a 10.000 referências e que podemos causar uma interrupção de temporizador a cada 5.000 referências. Quando obtemos uma interrupção de temporizador, copiamos e apagamos os valores do bit de referência para cada página. Assim, se houver uma falha de página, podemos examinar o bit de referência atual e 2 bits na memória para determinar se uma página foi usada dentro das últimas 10.000 a 15.000

referências. Se tiver sido usada, pelo menos um desses bits estará ligado. Se não tiver sido usada, esses bits estarão desligados. As páginas com pelo menos 1 bit ligado serão consideradas no conjunto de trabalho. Observe que esse arranjo não é preciso, pois não podemos saber onde, dentro do intervalo de 5.000, ocorreu uma referência. Podemos reduzir a incerteza aumentando a quantidade de bits de histórico e a frequência das interrupções (por exemplo, 10 bits e interrupções a cada 1.000 referências). No entanto, o custo para atender a essas interrupções mais frequentes será relativamente maior.

9.6.3 Frequência de falha de página

O modelo do conjunto de trabalho tem sucesso, e o conhecimento do conjunto de trabalho pode ser útil para a pré-paginação ([Seção 9.9.1](#)), mas esse parece ser um modo desajeitado de controlar o thrashing. Uma estratégia que utiliza a **frequência de falha de página (Page-Fault Frequency - PFF)** toma um caminho mais direto.

O problema específico é como prevenir o thrashing. O thrashing possui uma alta taxa de falha de página. Assim, queremos controlar a taxa de falha de página. Quando for muito alta, sabemos que o processo precisa de mais quadros. Ao contrário, se for muito baixa, o processo pode ter muitos quadros. Podemos estabelecer limites superior e inferior para a taxa de falha de página desejada ([Figura 9.21](#)). Se a taxa de falha de página ultrapassar o limite máximo, alocamos outro quadro para esse processo; se ficar abaixo do limite mínimo, removemos um quadro. Assim, podemos medir e controlar a taxa de falha de página para prevenir o thrashing.

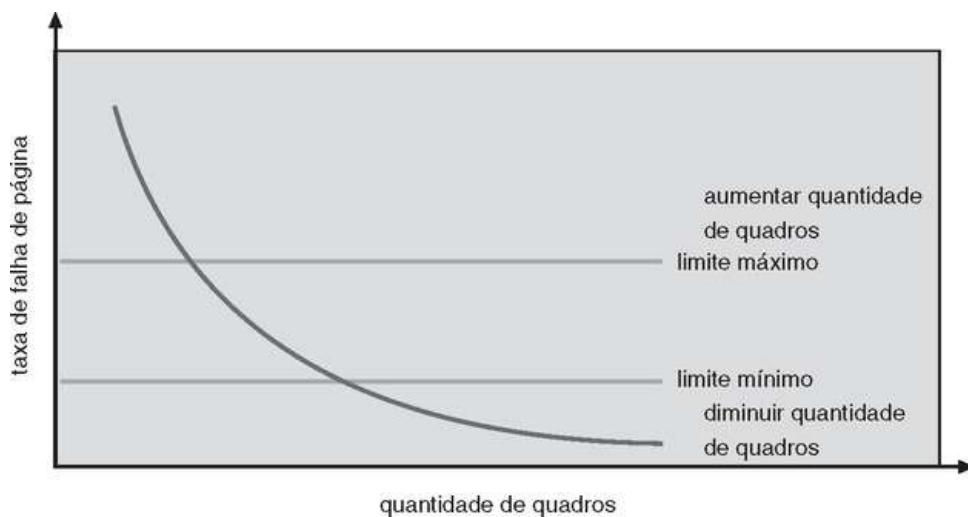


FIGURA 9.21 Frequência de falha de página.

Assim como na estratégia do conjunto de trabalho, podemos ter de suspender um processo. Se a taxa de falha de página aumentar e não houver quadros livres à disposição, temos de selecionar algum processo e suspendê-lo. Os quadros liberados são distribuídos aos processos com altas razões de falha de página.

9.7 Arquivos mapeados na memória

Considere uma leitura sequencial de um arquivo no disco usando as chamadas de sistema-padrão, `open()`, `read()` e `write()`. Cada acesso ao arquivo exige uma chamada de sistema e acesso ao disco. Como alternativa, podemos usar as técnicas de memória virtual discutidas até aqui para tratar da E/S de arquivo como acessos rotineiros à memória. Essa técnica, conhecida como **mapeamento na memória** para um arquivo, permite a uma parte do espaço de endereços virtuais estar associada logicamente ao arquivo.

9.7.1 Mecanismo básico%

O mapeamento de um arquivo na memória é realizado pelo mapeamento de um bloco de disco a uma página (ou páginas) na memória. O acesso inicial ao arquivo prossegue por meio da paginação por demanda normal, resultando em uma falha de página. Entretanto, uma parte do arquivo do tamanho de uma página é lida do sistema de arquivos para a página física (alguns sistemas podem optar por ler mais de um trecho da memória, do tamanho de uma página, de cada vez). As leituras e escritas subsequentes no arquivo são tratadas como acessos rotineiros à memória. Essa prática simplifica assim o acesso e o uso do arquivo, permitindo ao sistema manipular os arquivos por meio da memória, em vez de incorrer no custo adicional de usar as chamadas de sistema `read()` e `write()`.

CONJUNTOS DE TRABALHO E TAXAS DE FALHA DE PÁGINA

Existe um relacionamento direto entre o conjunto de trabalho de um processo e sua taxa de falha de página. Como podemos ver na [Figura 9.20](#), normalmente o conjunto de trabalho de um processo muda com o tempo à medida que as referências a seções de dados e código passam de uma localidade para outra. Considerando que existe memória suficiente para armazenar o conjunto de trabalho de um processo (ou seja, o processo não está causando thrashing), a taxa de falha de página do processo mudará entre picos e vales com o passar do tempo. Esse comportamento geral aparece na [Figura 9.22](#).

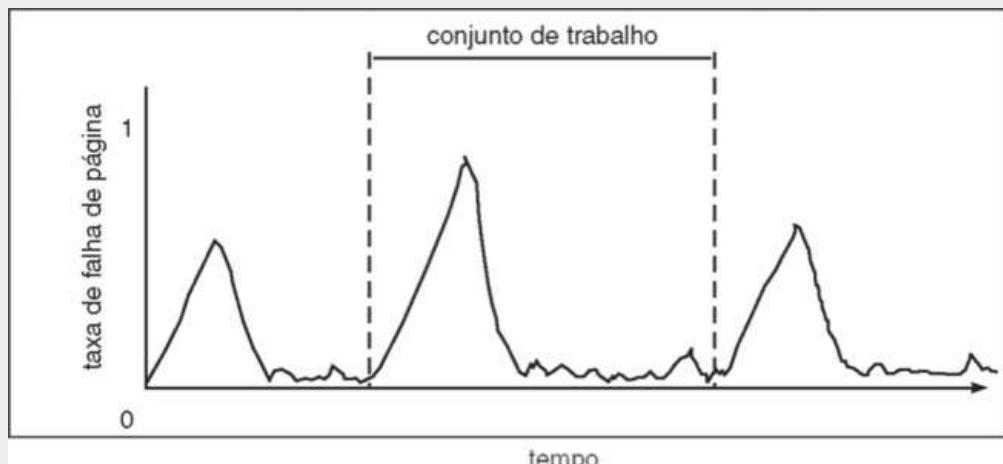


FIGURA 9.22 Taxa de falha de página com o passar do tempo.

Um pico na taxa de falha de página ocorre quando iniciamos a paginação por demanda em uma nova localidade. Contudo, quando o conjunto de trabalho dessa nova localidade está na memória, a taxa de falha de página diminui. Quando o processo passa para um novo conjunto de trabalho, a taxa de falha de página aumenta para um pico mais uma vez, retornando a uma taxa inferior quando o novo conjunto de trabalho é carregado na memória. O espaço de tempo entre o início de um pico e o início do próximo pico ilustra a transição de um conjunto de trabalho para outro.

Observe que as escritas no arquivo mapeado na memória não são necessariamente escritas imediatas (síncronas) ao arquivo em disco. Alguns sistemas podem preferir atualizar o arquivo físico quando o sistema operacional verificar periodicamente se a página na memória foi modificada. Quando o arquivo é fechado, todos os dados mapeados na memória são gravados no disco e removidos da memória virtual do processo.

Alguns sistemas operacionais proveem mapeamento de memória apenas por meio de uma chamada específica do sistema e utilizam as chamadas-padrão do sistema para realizar toda a E/S de arquivo restante. Contudo, alguns sistemas preferem mapear um arquivo na memória, não

importando se o arquivo foi especificado como mapeado na memória. Vamos considerar o Solaris como um exemplo. Se um arquivo for especificado como mapeado na memória (usando a chamada de sistema `mmap()`), o Solaris mapeará o arquivo no espaço de endereços do processo. Se um arquivo for aberto e acessado durante as chamadas de sistema normais, como `open()`, `read()` e `write()`, o Solaris ainda mapeará o arquivo na memória; porém, o arquivo é mapeado no espaço de endereços do kernel. Não importa como o arquivo é aberto; então, o Solaris trata toda a E/S de arquivo como mapeado na memória, permitindo que o acesso ao arquivo ocorra por meio de um subsistema de memória eficiente.

Acessos múltiplos podem ter permissão para mapear o mesmo arquivo simultaneamente, para permitir o compartilhamento dos dados. A escrita por qualquer um dos processos modifica os dados na memória virtual e pode ser vista por todos os outros que mapeiam a mesma seção do arquivo. Dadas nossas discussões anteriores sobre memória virtual, já deve estar claro como é implementado o compartilhamento de seções mapeadas na memória. O mapa da memória virtual para cada processo sendo compartilhado aponta para a mesma página da memória física - a página que mantém uma cópia do bloco de disco. Esse compartilhamento de memória é ilustrado na [Figura 9.23](#). As chamadas de sistema para mapeamento de memória também podem admitir a funcionalidade da cópia na escrita, permitindo que os processos compartilhem um arquivo no modo somente de leitura, mas tenham suas próprias cópias de quaisquer dados que modificam. Para o acesso aos dados compartilhados ser coordenado, os processos envolvidos podem usar um dos mecanismos para conseguir a exclusão mútua descrita no [Capítulo 6](#).

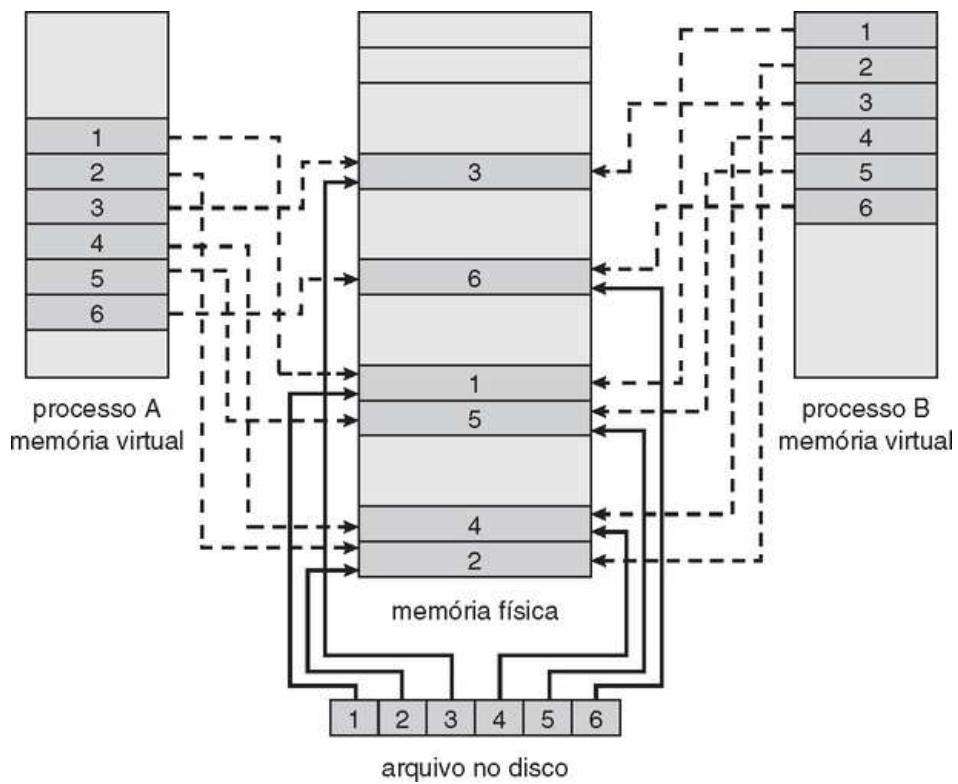


FIGURA 9.23 Arquivos mapeados na memória.

De muitas maneiras, o compartilhamento de arquivos mapeados na memória é semelhante à memória compartilhada, descrita na [Seção 3.4.1](#). Nem todos os sistemas utilizam o mesmo mecanismo para ambos; os sistemas UNIX e Linux, por exemplo, utilizam a chamada de sistema `mmap()` para o mapeamento da memória e as chamadas de sistema `shmget()` e `shmat()` compatíveis com POSIX para a memória compartilhada. Nos sistemas Windows NT, 2000 e XP, porém, a memória compartilhada é realizada por arquivos de mapeamento de memória. Nesses sistemas, os processos podem se comunicar usando a memória compartilhada, fazendo os processos em comunicação mapearem na memória entre o mesmo arquivo e seus espaços de endereços virtuais. O arquivo mapeado na memória serve como a região da memória compartilhada entre os processos em comunicação ([Figura 9.24](#)).

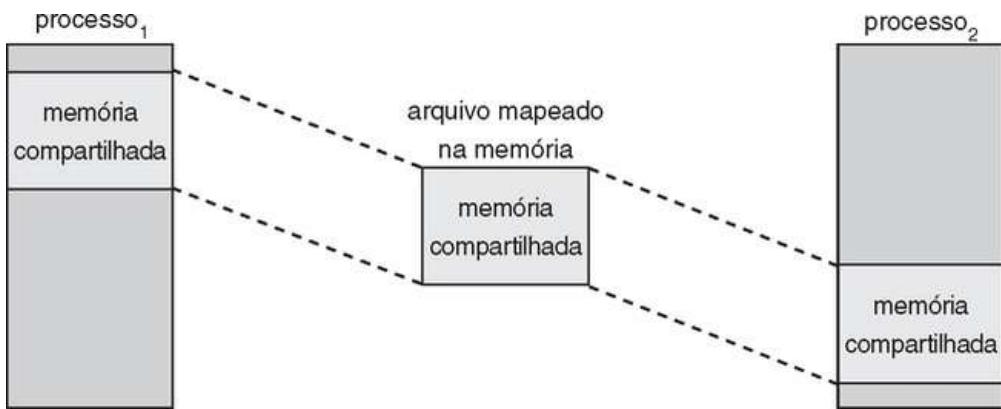


FIGURA 9.24 Memória compartilhada no Windows usando E/S mapeada na memória.

9.7.2 Arquivos mapeados na memória em Java

Em seguida, apresentamos as facilidades na API Java para arquivos mapeados na memória. Para mapear um arquivo na memória, primeiro é preciso abrir o arquivo e depois obter o `FileChannel` para o arquivo aberto. Quando o `FileChannel` for obtido, chamamos o método `map()` sobre esse canal, que mapeia o arquivo na memória. O método `map()` retorna um `MappedByteBuffer`, que é um buffer de bytes mapeado na memória. Isso pode ser visto na [Figura 9.25](#).

```

import java.io.*;
import java.nio.*;
import java.nio.channels.*;

public class MemoryMapReadOnly
{
    // Suponha que o tamanho da página seja 4KB
    public static final int PAGE_SIZE = 4096;

    public static void main(String args[ ]) throws IOException {
        RandomAccessFile inFile = new RandomAccessFile(args[0],"r");

        FileChannel in = inFile.getChannel( );
        MappedByteBuffer mappedBuffer =
            in.map(FileChannel.MapMode.READ_ONLY, 0, in.size( ));
        long numPages = in.size( ) / (long)PAGE_SIZE;
        if (in.size( ) % PAGE_SIZE > 0)
            ++numPages;

        // "tocaremos" no primeiro byte de cada página
        int position = 0;
        for (long i = 0; i < numPages; i++) {
            byte item = mappedBuffer.get(position);
            position += PAGE_SIZE;
        }

        in.close( );
        inFile.close( );
    }
}

```

FIGURA 9.25 Mapeando um arquivo na memória em Java.

A API para o método `map()` é a seguinte:
`map(modo, posição, tamanho)`

O modo refere-se a como ocorre o mapeamento. A [Figura 9.25](#) mapeia o arquivo como READ_ONLY. Os arquivos também podem ser mapeados como READ_WRITE e PRIVATE. Se o arquivo for mapeado como PRIVATE, o mapeamento na memória ocorrerá usando a técnica de cópia na escrita, descrita na [Seção 9.3](#). Quaisquer mudanças no arquivo mapeado resultam apenas em mudanças feitas à instância de objeto do `MappedByteBuffer` realizando o mapeamento.

A posição refere-se à posição do byte em que o mapeamento deve começar, e o tamanho indica quantos bytes devem ser mapeados a partir da posição inicial. A [Figura 9.25](#) mapeia o arquivo inteiro, ou seja, da posição 0 até o tamanho de `FileChannel`, `in.size()`. Também é possível mapear apenas uma parte de um arquivo ou obter vários mapeamentos diferentes do mesmo arquivo.

Na [Figura 9.25](#), consideramos que o tamanho de página mapeando o arquivo é de 4.096 bytes. (Muitos sistemas operacionais proveem uma chamada de sistema para determinar o tamanho da página; porém, esse não é um recurso da API Java.) Depois, determinamos a quantidade de páginas necessárias para mapear o arquivo na memória e acessamos o primeiro byte de cada página usando o método `get()` da classe `MappedByteBuffer`. Isso tem o efeito de paginar por demanda cada página do arquivo para a memória (em sistemas que admitem esse modelo de memória). A API também provê o método `load()` da classe `MappedByteBuffer`, que carrega o arquivo inteiro para a memória usando a paginação por demanda.

Muitos sistemas operacionais proveem uma chamada de sistema que libera (ou desfaz) o mapeamento de um arquivo. Essa ação libera as páginas físicas que mapearam o arquivo na memória. A API Java não provê tais recursos. Um mapeamento existe até o objeto `MappedByteBuffer` passar pela coleta de lixo.

9.7.3 E/S mapeada na memória

No caso da E/S, como mencionamos na [Seção 1.2.1](#), cada controlador de E/S inclui registradores para manter os comandos e os dados que estão sendo transferidos. Normalmente, instruções de E/S especiais permitem transferências de dados entre esses registradores e a memória do sistema. Para permitir o acesso mais conveniente aos dispositivos de E/S, muitas arquiteturas de computador fornecem **E/S mapeada na memória**. Nesse caso, intervalos de endereços de memória são separados e mapeados para os registradores de dispositivos. As leituras e escritas nesses endereços de memória fazem os dados serem transferidos de e para os registradores de dispositivo. Esse método é apropriado para dispositivos que possuem tempos de resposta curtos, como controladores de vídeo. No IBM PC, cada local na tela é mapeado para um local na memória. Exibir texto na tela é quase tão fácil quanto escrever o texto nos locais certos mapeados na memória.

A E/S mapeada na memória também é conveniente para outros dispositivos, como as portas serial e paralela utilizadas para conectar modems e impressoras a um computador. A CPU transfere dados por meio desses tipos de dispositivos lendo e escrevendo alguns registradores de dispositivo, chamados de **porta** de E/S. Para enviar uma longa sequência de bytes por uma porta serial mapeada na memória, a CPU escreve um byte de dados no registrador de dados e marca um bit no registrador de controle para sinalizar que o byte está disponível. O dispositivo apanha o byte de dados e depois apaga o bit no registrador de controle para sinalizar que está pronto para o próximo byte. Depois, a CPU pode transferir o próximo byte. Se a CPU usar o polling para vigiar o bit de controle, realizando um loop contínuo para ver se o dispositivo está pronto, esse método de operação é denominado **E/S programada (PIO - Programmed I/O)**. Se a CPU não sondar o bit de controle, mas em vez disso receber uma interrupção quando o dispositivo estiver pronto para o próximo byte, a transferência de dados é considerada **controlada por interrupção**.

9.8 Alocando memória do kernel

Quando um processo executando no modo usuário requisita memória adicional, as páginas são alocadas a partir da lista de frames de página livres mantidos pelo kernel. Essa lista normalmente é preenchida usando um algoritmo de substituição de página, como aqueles discutidos na [Seção 9.4](#), e provavelmente contém páginas livres espalhadas por toda a memória física, conforme já explicamos. Lembre-se, também, que se um processo do usuário requisitar um único byte de memória, haverá fragmentação interna, pois o processo receberá um quadro de página inteiro.

Contudo, a memória do kernel normalmente é alocada a partir de um banco de memória livre diferente da lista usada para satisfazer os processos comuns do modo usuário. Existem dois motivos principais para isso:

1. O kernel requisita memória para estruturas de dados de tamanhos variáveis, alguns dos quais são menores que uma página em tamanho. Como resultado, o kernel precisa usar a memória com cautela e tentar minimizar o desperdício que acontece com a fragmentação. Isso é especialmente importante porque muitos sistemas operacionais não sujeitam o código ou dados do kernel ao sistema de paginação.
2. As páginas alocadas aos processos no modo usuário não precisam necessariamente estar na memória física contígua. Porém, certos dispositivos de hardware interagem diretamente com a memória física - sem o benefício de uma interface de memória virtual - e, consequentemente, podem exigir que a memória resida em páginas fisicamente contíguas.

Nas próximas seções, examinamos duas estratégias para gerenciar a memória livre que é atribuída aos processos do kernel: o “sistema buddy” e a alocação de slab.

9.8.1 Sistema buddy

O sistema “buddy” aloca memória de um segmento de tamanho fixo, consistindo em páginas fisicamente contíguas. A memória é alocada a partir desse segmento, usando um **alocador na potência de 2**, que satisfaz as requisições em unidades cujo tamanho é uma potência de 2 (4 KB, 8 KB, 16 KB e assim por diante). Uma requisição em unidades com tamanho impróprio é arredondada para a próxima potência de 2 mais alta. Por exemplo, se for feita uma requisição por 11 KB, ela é satisfeita com um segmento de 16 KB.

Vamos considerar um exemplo simples. Suponha que o tamanho de um segmento de memória seja inicialmente 256 KB e o kernel requisite 21 KB de memória. O segmento é inicialmente dividido em dois *buddies* - que chamaremos de A_L e A_R - cada um com 128 KB de tamanho. Um desses buddies é dividido ainda em dois buddies de 64 KB - B_L e B_R . A próxima maior potência de 2 de 21 KB é 32 KB, B_L ou B_R é novamente dividido em dois buddies de 32 KB, C_L e C_R . Um desses amigos é usado para satisfazer a solicitação de 21 KB. Em seguida, ou B_L ou B_R é dividido novamente em dois buddies de 32 KB, C_L e C_R . Dividir novamente resulta em buddies de 16 KB, que seria muito pequeno para satisfazer a requisição de 21 KB. Assim, um dos buddies de 32 KB é usado para satisfazer a requisição. Esse esquema é ilustrado na [Figura 9.26](#), onde C_L é o segmento alocado para a requisição de 21 KB.

páginas fisicamente contíguas

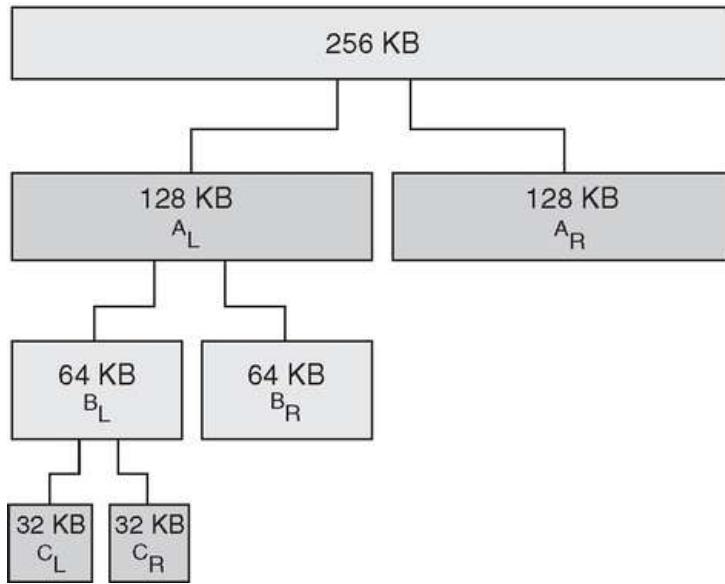


FIGURA 9.26 Alocação de sistema buddy.

Uma vantagem do sistema buddy é a rapidez com que os buddies adjacentes podem ser combinados para formar segmentos maiores usando uma técnica conhecida como **união**. Na [Figura 9.26](#), por exemplo, quando o kernel libera a unidade C_L que estava alocada, o sistema pode unir C_L e C_R em um segmento de 64 KB. Esse segmento, B_L , pode por sua vez ser unido com seu buddy B_R para formar um segmento de 128 KB. Por fim, podemos acabar com o segmento original de 256 KB.

A desvantagem óbvia do sistema buddy é que arredondar para a próxima potência de 2 mais alta provavelmente causará fragmentação dentro dos segmentos alocados. Por exemplo, uma requisição de 33 KB só pode ser satisfeita com um segmento de 64 KB. Na verdade, não podemos garantir que menos de 50% da unidade alocada será desperdiçada devido à fragmentação. Na próxima seção, exploramos um esquema de alocação de memória onde nenhum espaço é perdido devido à fragmentação.

9.8.2 Alocação de slab

Uma segunda técnica para alocar memória do kernel é conhecida como **alocação de slab**. Um **slab** é composto de uma ou mais páginas fisicamente contíguas. Um **cache** consiste em um ou mais slabs. Há um único cache para cada estrutura de dados exclusiva do kernel – por exemplo, um cache separado para a estrutura de dados representando descritores de processo, um cache separado para objetos de arquivo, um cache separado para semáforos, e assim por diante. Cada cache é preenchido com **objetos** que são instanciações da estrutura de dados do kernel que o cache representa. Por exemplo, o cache representando semáforos armazena instâncias de objetos de semáforo, o cache representando descritores de processo armazena instâncias de objetos descritores de processo, e assim por diante. O relacionamento entre slabs, caches e objetos é mostrado na [Figura 9.27](#). A figura mostra dois objetos do kernel de 3 KB de tamanho e três objetos de 7 KB de tamanho. Esses objetos são armazenados em seus respectivos caches.

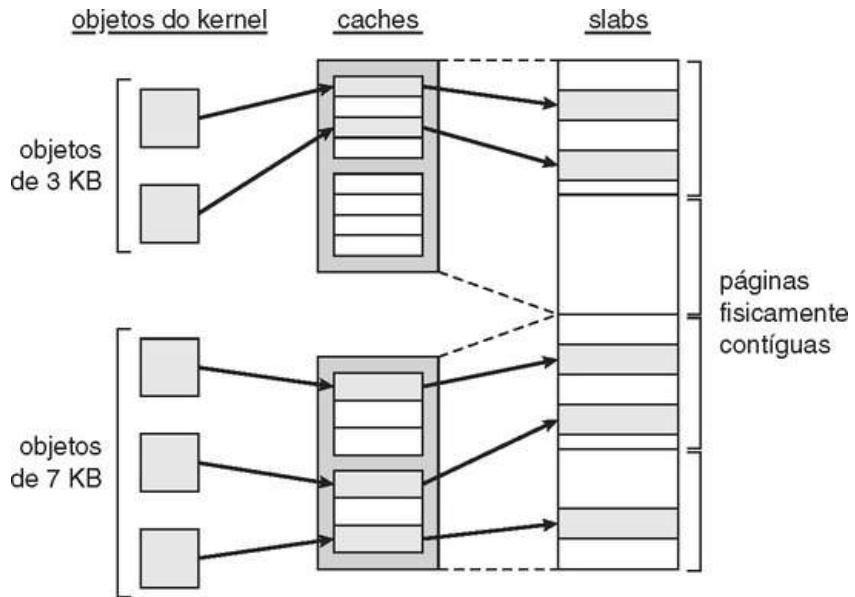


FIGURA 9.27 Alocação de slabs.

O algoritmo de alocação de slab usa caches para armazenar objetos do kernel. Quando um cache é criado, diversos objetos – que inicialmente são marcados como **livres** – são alocados ao cache. A quantidade de objetos no cache depende do tamanho do slab associado. Por exemplo, um slab de 12 KB (composto de três páginas contíguas de 4 KB) poderia armazenar seis objetos de 2 KB. Inicialmente, todos os objetos no cache são marcados como livres. Quando um novo objeto para uma estrutura de dados do kernel é necessário, o alocador pode atribuir qualquer objeto livre do cache para satisfazer a requisição. O objeto atribuído a partir do cache é marcado como **usado**.

Vamos considerar um cenário em que o kernel requisita memória do alocador de slab para um objeto representando um descritor de processo. Nos sistemas Linux, um descritor de processo é do tipo `struct task_struct`, que exige aproximação de 1,7 KB de memória. Quando o kernel do Linux cria uma nova tarefa, ele requisita a memória necessária para o objeto `struct task_struct` a partir do seu cache. O cache atenderá a requisição usando um objeto `struct task_struct` que já foi alocado em um slab e está marcado como livre.

No Linux, um slab pode estar em um dentre três estados possíveis:

1. **Cheio.** Todos os objetos no slab são marcados como usados.
2. **Vazio.** Todos os objetos no slab são marcados como livres.
3. **Parcial.** O slab consiste em objetos usados e livres.

O alocador de slab primeiro tenta satisfazer a requisição com um objeto livre em um slab parcial. Se não houver um, um objeto livre é atribuído a partir de um slab vazio. Se nenhum slab vazio estiver disponível, um novo slab é alocado a partir de páginas físicas contíguas e atribuído a um cache; a memória para o objeto é alocada a partir desse slab.

O alocador de slab fornece dois benefícios principais:

1. A memória é desperdiçada devido à fragmentação. A fragmentação não é problema, porque cada estrutura de dados exclusiva do kernel tem um cache associado, e cada cache é composto de um ou mais slabs que são divididos em pedaços com o tamanho do objeto representado. Assim, quando o kernel requisita memória para um objeto, o alocador de slab retorna a quantidade exata de memória exigida para representar o objeto.
2. As requisições de memória podem ser satisfeitas rapidamente. O esquema de alocação de slab é, assim, particularmente eficaz para gerenciar a memória quando os objetos são frequentemente alocados e dealocados, como normalmente acontece com requisições do kernel. O ato de alocar – e liberar – memória é um processo demorado. Porém, os objetos são criados com antecedência e, portanto, podem ser rapidamente alocados a partir do cache. Além do mais, quando o kernel tiver terminado de usar um objeto e o liberar, ele é marcado como livre e retornado ao seu cache, tornando-se imediatamente disponível para requisições subsequentes do kernel.

O alocador de slab apareceu inicialmente no kernel do Solaris 2.4. Devido à sua natureza de uso geral, o alocador agora também é usado para certas requisições de memória do modo usuário no Solaris. O Linux originalmente usava o sistema buddy; contudo, a partir da versão 2.2, o kernel do Linux adotou o alocador de slab.

9.9 Outras considerações para sistemas de paginação

As principais decisões tomadas para um sistema de página são as seleções de um algoritmo de substituição e uma política de alocação, discutidas anteriormente neste capítulo. Existem também muitas outras considerações, e discutiremos algumas delas.

9.9.1 Pré-paginação

Uma propriedade óbvia da paginação por demanda pura é a grande quantidade de falhas de página que ocorrem quando um processo é iniciado. Essa situação resulta na tentativa de levar a localidade inicial para a memória. A mesma situação pode surgir em outras ocasiões. Por exemplo, quando um processo retirado da memória é reiniciado, todas as suas páginas estão no disco, e cada uma precisa ser trazida por sua própria falha de página. A **pré-paginação** é uma tentativa de evitar esse alto nível de paginação inicial. A estratégia é trazer para a memória de uma só vez todas as páginas necessárias. Alguns sistemas operacionais - em especial, o Solaris - realizam a pré-paginação dos quadros de página para arquivos pequenos.

Em um sistema usando o modelo de conjunto de trabalho, por exemplo, mantemos com cada processo uma lista das páginas no seu conjunto de trabalho. Se tivermos de suspender um processo (devido a uma espera por E/S ou a uma falta de quadros livres), lembramos o conjunto de trabalho para esse processo. Quando o processo tiver de ser retomado (pois a E/S terminou ou quadros livres tornaram-se disponíveis), automaticamente trazemos de volta para a memória seu conjunto de trabalho inteiro, antes de reiniciar o processo.

A pré-paginação pode prover uma vantagem em alguns casos. A questão é se o custo do uso da pré-paginação é menor do que o custo de atender às falhas de página correspondentes. Pode muito bem acontecer de muitas das páginas trazidas para a memória pela pré-paginação não serem utilizadas.

Suponha que s páginas sejam pré-paginadas e uma fração α dessas s páginas seja utilizada ($0 \leq \alpha \leq 1$). A questão é se o custo das falhas de página salvas $s^* \alpha$ é maior ou menor do que o custo de pré-paginar s^* ($1 - \alpha$) páginas desnecessárias. Se α estiver próximo de 0, a pré-paginação perde; se α estiver próximo de 1, a pré-paginação vence.

9.9.2 Tamanho da página

Os projetistas de um sistema operacional para uma máquina existente raramente têm uma escolha em relação ao tamanho da página. No entanto, quando novas máquinas estão sendo projetadas, uma decisão com relação ao melhor tamanho de página precisa ser tomada. Como você poderia imaginar, não existe um único tamanho de página melhor. Em vez disso, um conjunto de fatores deve ser considerado. Os tamanhos de página são potências de 2, geralmente variando de 4.096 (2^{12}) até 4.194.304 (2^{22}) bytes.

Como selecionamos um tamanho de página? Uma preocupação é o tamanho da tabela de página. Para determinado espaço de memória virtual, diminuir o tamanho de página aumenta a quantidade de páginas e, portanto, o tamanho da tabela de página. Para uma memória virtual de 4 MB (2^{22}), por exemplo, haveria 4.096 páginas de 1.024 bytes, mas somente 512 páginas de 8.192 bytes. Como cada processo ativo precisa ter sua própria cópia da tabela de página, um tamanho de página grande é desejável.

Contudo, a memória é mais bem utilizada com páginas menores. Se um processo tiver recebido memória a partir do local 00000 e continuar até ter a quantidade de que precisa, ele não terminará em um limite de página. Assim, uma parte da última página precisa ser alocada (porque as páginas são as unidades de alocação), mas não é utilizada (criando fragmentação interna). Considerando a independência entre tamanho de processo e tamanho de página, podemos esperar que, na média, metade da última página de cada processo será desperdiçada. Essa perda é de apenas 256 bytes para uma página de 512 bytes, mas é de 4.096 bytes para uma página de 8.192 bytes. Para minimizar a fragmentação interna, portanto, precisamos de um tamanho de página pequeno.

Outro problema é o tempo necessário para ler e escrever uma página. O tempo de E/S é composto de tempos de busca, latência e transferência. O tempo de transferência é proporcional à quantidade transferida (ou seja, o tamanho da página) - um fato que poderia argumentar em favor de um tamanho de página pequeno. Entretanto, como veremos na [Seção 12.1.1](#), os tempos de latência e de busca são muito superiores ao tempo de transferência. Com uma taxa de transferência de 2 MB por segundo, são necessários apenas 0,2 milissegundo para transferir 512 bytes. O tempo de latência, no entanto, talvez seja de 8 milissegundos e o tempo de busca de 20 milissegundos. Do tempo total para E/S (28,2 milissegundos), portanto, somente 1% é atribuído à transferência real. Dobrar o tamanho da página aumenta o tempo de E/S para somente 28,4 milissegundos. São necessários 28,4 milissegundos para ler uma única página de 1.024 bytes, mas 56,4 milissegundos para ler a mesma quantidade como duas páginas de 512 bytes cada. Assim, o desejo de reduzir o tempo de E/S é um argumento em favor de um tamanho de página maior.

No entanto, com um tamanho de página menor, a E/S total deverá ser reduzida, pois a localidade será melhorada. Um tamanho de página menor permite que cada página corresponda mais precisamente à localidade do programa. Por exemplo, considere um processo com o tamanho de 200 KB, dos quais somente metade (100 KB) é utilizada em uma execução. Se tivermos apenas uma página grande, teremos de trazer a página inteira, um total de 200 KB transferidos e alocados. Se, em vez disso, tivéssemos páginas de somente 1 byte, então poderíamos trazer apenas os 100 KB que são usados, resultando em apenas 100 KB transferidos e alocados. Com um tamanho de página menor, temos melhor **resolução**, permitindo isolar apenas a memória necessária. Com um tamanho de página maior, temos de alocar e transferir não apenas o necessário, mas também o que estiver na página, sendo necessário ou não. Assim, um tamanho de página menor deverá resultar em menos E/S e menos memória alocada total.

Você notou que, com um tamanho de página de 1 byte, teríamos de realizar a falha de página para *cada* byte? Um processo de 200 KB, usando apenas metade dessa memória, geraria apenas uma falha de página com um tamanho de página de 200 KB, mas 102.400 falhas de página com um tamanho de página de 1 byte. Cada falha de página gera a grande quantidade de custo adicional necessário para processar a interrupção, salvar registradores, substituir a página, enfileirar para o dispositivo de paginação e atualizar as tabelas. Para reduzir a quantidade de falhas de página, precisamos ter um tamanho de página grande.

Outros fatores também precisam ser considerados, como o relacionamento entre o tamanho de página e o tamanho do setor no dispositivo de paginação. O problema não tem uma resposta melhor. Como vimos, alguns fatores (fragmentação interna, localidade) argumentam em favor de um tamanho de página pequeno, enquanto outros (tamanho de tabela, tempo de E/S) argumentam em favor de um tamanho de página grande. Contudo, a tendência histórica é em direção a tamanhos de página maiores. Na realidade, a primeira edição deste livro (1983) usava 4.096 bytes como limite superior para os tamanhos de página, e esse valor era o tamanho de página mais comum em 1990. Os sistemas modernos podem usar tamanhos de página muito maiores, como veremos na próxima seção.

9.9.3 Alcance da TLB

No [Capítulo 8](#), introduzimos a **taxa de acertos** da TLB. Lembre-se de que a taxa de acertos para a TLB refere-se à porcentagem das traduções de endereço virtual resolvidas na TLB, em vez de uma tabela de página. Logicamente, a taxa de acertos está relacionada com a quantidade de entradas na TLB, e a maneira de aumentar a taxa de acertos é aumentando a quantidade de entradas na TLB. Isso, porém, não vem sem um custo, pois a memória associativa usada para construir a TLB é cara e utiliza muita energia.

Relacionada com a taxa de acertos está uma medição semelhante: o **alcance da TLB**. O alcance da TLB refere-se à quantidade de memória acessível da TLB e é a quantidade de entradas multiplicada pelo tamanho da página. O ideal é que o conjunto de trabalho para um processo seja armazenado na TLB. Se não, o processo gastará um tempo considerável resolvendo referências de memória na tabela de página, em vez da TLB. Se dobrarmos a quantidade de entradas na TLB, dobraremos seu alcance. Contudo, para algumas aplicações com uso intenso de memória, isso ainda pode ser insuficiente para armazenar o conjunto de trabalho.

Outra técnica para aumentar o alcance da TLB é aumentar o tamanho da página ou prover vários tamanhos de página. Se aumentarmos o tamanho da página – digamos, de 8 KB para 32 KB –, quadruplicaremos o alcance da TLB. Entretanto, isso pode levar a um aumento na fragmentação para algumas aplicações que não exigem um tamanho de página tão grande quanto 32 KB. Como alternativa, um sistema operacional pode prover vários tamanhos de página diferentes. Por exemplo, o UltraSPARC admite tamanhos de página de 8 KB, 64 KB, 512 KB e 4 MB. Desses tamanhos disponíveis, o Solaris usa os tamanhos de página de 8 KB e 4 MB. E com uma TLB de 64 entradas, o alcance da TLB para o Solaris varia de 512 KB com páginas de 8KB até 256 MB com páginas de 4 MB. Para a maioria das aplicações, o tamanho de página de 8 KB é suficiente, embora o Solaris mapeie os primeiros 4 MB do código do kernel e seus dados com duas páginas de 4 MB. O Solaris também permite que aplicações como bancos de dados tirem proveito do tamanho de página grande de 4 MB.

Prover suporte para várias páginas exige que o sistema operacional – não o hardware – gerencie a TLB. Por exemplo, um dos campos em uma entrada da TLB precisa indicar o tamanho do quadro de página correspondente à entrada da TLB. O gerenciamento da TLB no software e não no hardware possui um custo no desempenho. Entretanto, a taxa de acertos e o alcance da TLB maiores compensam os custos no desempenho. Na realidade, as tendências atuais indicam uma mudança em direção às TLBs gerenciadas pelo software e suporte do sistema operacional para tamanhos de página múltiplos. As arquiteturas UltraSPARC, MIPS e Alpha empregam TLBs gerenciadas pelo software. O PowerPC e o Pentium gerenciam a TLB no hardware.

9.9.4 Tabela de página invertida

A Seção 8.5.3 introduziu o conceito de tabela de página invertida. A finalidade dessa forma de gerenciamento de página era reduzir a quantidade de memória física necessária para acompanhar as traduções de endereços virtuais para físicos. Conseguimos essa economia criando uma tabela que possui uma entrada por página da memória física indexada pelo par <id-processo, número-página>.

Como mantém informações sobre qual página da memória virtual está armazenada em cada quadro físico, as tabelas de página invertidas reduzem a quantidade de memória física necessária para armazenar essa informação. Contudo, a tabela de página invertida não contém mais informações completas sobre o espaço de endereços lógicos de um processo, e essas informações são necessárias se uma página referenciada não estiver na memória. A paginação por demanda exige essa informação para processar falhas de página. Para que as informações estejam disponíveis, uma tabela de página externa (uma por processo) precisa ser mantida. Cada uma dessas tabelas se parece com a tabela de página tradicional, por processo, e contém informações sobre onde está localizada cada página virtual.

Mas será que as tabelas de página externas negam a utilidade das tabelas de página invertidas? Como essas tabelas são referenciadas apenas quando ocorre uma falha de página, elas não precisam estar disponíveis rapidamente. Em vez disso, elas por si só são paginadas para dentro e fora da memória conforme a necessidade. Infelizmente, uma falha de página pode agora fazer o gerenciador de memória virtual causar outra falha de página enquanto entra com a tabela de página externa necessária para localizar a página virtual no armazenamento de apoio. Esse caso especial exige um tratamento cuidadoso no kernel e um atraso no processamento de pesquisa de página.

9.9.5 Estrutura do programa

A paginação por demanda foi projetada para ser transparente ao programa do usuário. Em muitos casos, o usuário está desavisado quanto à natureza paginada da memória. Em outros casos, o desempenho do sistema pode ser melhorado se o usuário (ou compilador) tiver conhecimento da paginação por demanda que está ocorrendo.

Vejamos um exemplo inventado, porém informativo. Suponha que as páginas tenham um tamanho de 128 palavras. Considere um programa Java cuja função é inicializar com 0 cada elemento de um array de 128 por 128. O código a seguir é muito comum:

```
int [128][128] data;

for (int j = 0; j < 128; j++)
    for (int i = 0; i < 128; i++)
        data[i][j] = 0;
```

Observe que o array é armazenado linha por linha, ou seja, o array é armazenado como `data[0][0], data[0][1], ..., data[0][127], data[1][0], data[1][1], ..., data[127][127]`. Para páginas de 128 palavras, cada linha ocupa uma página. Assim, o código mostrado zera uma palavra em cada página, depois outra palavra em cada página, e assim por diante. Se o sistema operacional alocar menos do que 128 quadros para o programa inteiro, então sua execução resultará em $128 \times 128 = 16.384$ falhas de página. Por outro lado, a mudança do código para

```
int [128][128] data;

for (int i = 0; i < 128; i++)
    for (int j = 0; j < 128; j++)
        data[i][j] = 0;
```

zera todas as palavras em uma página antes de iniciar a próxima página, reduzindo a quantidade de falhas de página para 128.

A seleção cuidadosa de estruturas de dados e estruturas de programação pode aumentar a localidade e, portanto, reduzir a taxa de falha de página e o número de páginas no conjunto de trabalho. Por exemplo, uma pilha possui boa localidade, pois o acesso sempre é feito no topo. Uma tabela de hash, ao contrário, é projetada para espalhar referências, produzindo uma localidade ruim. Naturalmente, a localidade da referência é apenas uma medida para a eficiência de uso de uma estrutura de dados. Outros fatores bastante pesados incluem a velocidade de pesquisa, o número total de referências à memória e o número total de páginas tocadas.

Em um estágio posterior, o compilador e o loader podem ter um efeito significativo sobre a

paginação. A separação entre código e dados e a geração de código reentrante significa que as páginas de código podem ser apenas de leitura e, portanto, nunca serão modificadas. Páginas limpas não precisam ser paginadas para fora para serem substituídas. O loader pode evitar a colocação de rotinas além dos limites de página, mantendo cada rotina em uma página. As rotinas que chamam umas às outras por diversas vezes podem ser empacotadas na mesma página. Esse empacotamento é uma variante do problema de empacotamento binário na busca de operações: tente empacotar os segmentos de carga de tamanhos variáveis em páginas de tamanho fixo de modo a reduzir as referências entre páginas. Essa técnica é útil principalmente para grandes tamanhos de página.

A escolha da linguagem de programação também pode afetar a paginação. Por exemplo, C e C++ utilizam ponteiros com frequência, e os ponteiros costumam tornar o acesso à memória aleatório, com o potencial de diminuir a localidade de um processo. Alguns estudos têm mostrado que os programas orientados a objeto também costumam ter uma localidade de referência fraca.

9.9.6 Interlock de E/S

Quando a paginação por demanda é utilizada, às vezes precisamos permitir que algumas das páginas sejam **fechadas (locked)** na memória. Uma situação desse tipo ocorre quando a E/S é feita de ou para a memória do usuário (virtual). A E/S normalmente é implementada por um processo de E/S separado. Por exemplo, um controlador para dispositivo de armazenamento de USB em geral recebe a quantidade de bytes a transferir e um endereço de memória para o buffer ([Figura 9.28](#)). Quando a transferência termina, a CPU é interrompida.

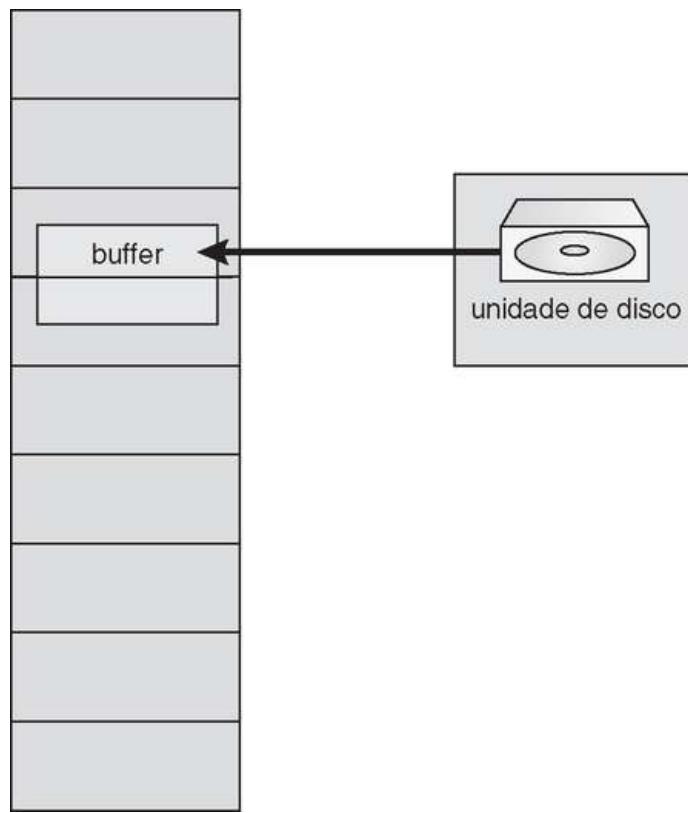


FIGURA 9.28 O motivo pelo qual os quadros usados para E/S precisam estar na memória.

Precisamos ter certeza de que a seguinte sequência de eventos não ocorrerá: um processo emite uma requisição de E/S, colocada em uma fila para esse dispositivo de E/S. Nesse meio-tempo, a CPU recebe outros processos para atuar. Esses processos causam falhas de página, e um deles, usando um algoritmo de substituição global, substitui a página contendo o buffer da memória para o processo que está aguardando. As páginas são removidas da memória. Algum tempo depois, quando a requisição de E/S avança para a cabeça da fila do dispositivo, ocorre a E/S para o endereço especificado. Entretanto, esse quadro agora está sendo usado para uma página diferente, pertencente a outro processo.

Existem duas soluções comuns para esse problema. Uma é nunca executar E/S para a memória do usuário. Em vez disso, os dados sempre são copiados entre a memória do sistema e a memória do usuário. A E/S ocorre somente entre a memória do sistema e o dispositivo de E/S. Para escrever um bloco em fita, primeiro copiamos o bloco para a memória do sistema e depois o escrevemos na fita. Essa cópia extra pode ocasionar um custo adicional inaceitavelmente alto.

Outra solução é permitir que se efetue o lock das páginas na memória. Um bit de lock está associado a cada quadro. Se o quadro estiver com o lock, ele não poderá ser selecionado para substituição. Sob essa técnica, para escrever um bloco em fita, bloqueamos na memória as páginas contendo esse bloco. O sistema, então, pode continuar normalmente. As páginas com lock não poderão ser substituídas. Quando a E/S terminar, removem-se os locks das páginas.

Os bits de lock são usados em diversas situações. Constantemente, parte ou todo o kernel do sistema operacional está sempre presente na memória, pois muitos sistemas operacionais não podem tolerar uma falha de página causada pelo kernel.

Outro uso para um bit de lock envolve a substituição normal de página. Considere a sequência de eventos a seguir. Um processo de baixa prioridade gera uma falha de página. Selecionando um quadro substituto, o sistema de paginação lê a página necessária para a memória. Pronto para continuar, o processo de baixa prioridade entra na fila de espera e aguarda pela CPU. Por ser um processo de baixa prioridade, ele pode não ser selecionado pelo escalonador de CPU por algum tempo. Enquanto o processo de baixa prioridade espera, um processo de alta prioridade gera falhas de página. Procurando um substituto, o sistema de paginação vê uma página na memória, mas que não foi referenciada ou modificada: essa é a página que o processo de baixa prioridade acabou de trazer. Essa página parece ser um substituto perfeito: ela está limpa e não precisará ser escrita fora da memória e aparentemente não foi usada por muito tempo.

Se o processo de alta prioridade conseguirá substituir o processo de baixa prioridade é uma decisão política. Afinal, estamos adiando o processo de baixa prioridade para o benefício do processo de alta prioridade. Todavia, estamos desperdiçando o esforço gasto para trazer a página para o processo de baixa prioridade. Se decidirmos evitar a substituição de uma página recém-trazida até ela poder ser usada pelo menos uma vez, então poderemos usar o bit de lock para implementar esse mecanismo. Quando uma página é selecionada para substituição, seu bit de lock é ligado; ele permanece assim até o processo de falha de página ser novamente despachado.

O uso de um bit de lock pode ser perigoso. Ele pode ser ligado, mas nunca desligado. Se essa situação acontecer (devido a um bug no sistema operacional, por exemplo), o quadro bloqueado se tornará inutilizável. Em um sistema monousuário, a utilização excessiva do lock atrapalhará somente o usuário que o está realizando. Os sistemas multiusuário precisam confiar menos nos usuários. Por exemplo, o Solaris permite “sugestões” de lock, mas está livre para desconsiderá-las se o banco de quadros livres se tornar muito pequeno ou se um processo individual requisitar o lock de muitas páginas na memória.

9.10 Exemplos de sistema operacional

Nesta seção, descrevemos como o Windows XP e o Solaris implementam a memória virtual.

9.10.1 Windows XP

O Windows XP implementa a memória virtual usando a paginação por demanda com **clustering**. O clustering trata de falhas de página trazendo não apenas a página que falta, mas também várias páginas após a página que falta. Quando um processo é criado, ele recebe um mínimo e um máximo do conjunto de trabalho. O **conjunto mínimo de trabalho** é a quantidade mínima de páginas que o processo tem garantias de ter na memória. Se houver memória suficiente disponível, o processo pode receber tantas páginas quanto seu **conjunto máximo de trabalho**. Para a maioria das aplicações, o valor do mínimo e máximo do conjunto de trabalho é de 50 e 345 páginas, respectivamente. (Em algumas circunstâncias, um processo pode ter permissão para exceder seu máximo do conjunto de trabalho.) O gerenciador de memória virtual mantém uma lista de quadros de página livres. Associado a essa lista está o valor de limite usado para indicar se existe memória livre suficiente à disposição. Se houver uma falha de página para um processo que esteja abaixo do seu máximo do conjunto de trabalho, o gerenciador de memória virtual aloca uma página dessa lista de páginas livres. Se um processo estiver no seu máximo do conjunto de trabalho e houver uma falha de página, ele terá de selecionar uma página para substituição usando uma política de substituição de página local.

Quando a quantidade de memória livre estiver abaixo do limite, o gerenciador de memória virtual utiliza uma tática conhecida como **corte automático do conjunto de trabalho** para restaurar o valor acima do limite. O corte automático do conjunto de trabalho funciona avaliando a quantidade de páginas alocadas aos processos. Se um processo tiver recebido mais páginas do que seu mínimo do conjunto de trabalho, o gerenciador de memória virtual removerá as páginas até o processo atingir seu mínimo do conjunto de trabalho. Um processo que está no seu mínimo do conjunto de trabalho pode receber páginas da lista de quadros livres quando houver memória livre suficiente.

O algoritmo utilizado para determinar qual página será removida de um conjunto de trabalho depende do tipo de processador. Em sistemas 80x86 de processador único, o Windows XP utiliza uma variável do algoritmo do *relógio*, discutido na [Seção 9.4.5.2](#). Em sistemas x86 Alpha e de multiprocessadores, apagar o bit de referência pode exibir invalidação da entrada na TLB (Translation Look-aside Buffer) em outros processadores. Em vez de incorrer nesse custo adicional, o Windows XP usa uma variação do algoritmo FIFO, discutido na [Seção 9.4.2](#).

9.10.2 Solaris

No Solaris, quando uma thread incorre em uma falha de página, o kernel atribui uma página à thread causando a falha a partir da lista de páginas livres que ele mantém. Portanto, é imperativo que o kernel mantenha uma quantidade suficiente de memória livre à disposição. Associado a essa lista de páginas livres está um parâmetro - *lotsfree* - que representa um patamar para iniciar a paginação. O parâmetro *lotsfree* é um conjunto de 1/64 do tamanho da memória física. Quatro vezes por segundo, o kernel verifica se a quantidade de memória livre é menor que *lotsfree*.

Se a quantidade de páginas livres ficar abaixo de *lotsfree*, um processo conhecido como **pageout** é iniciado. O processo pageout é semelhante ao algoritmo da segunda chance, descrito na [Seção 9.4.5.2](#), exceto que ele usa dois ponteiros enquanto sonda as páginas, em vez de um. O processo pageout funciona da seguinte maneira: o ponteiro da frente do relógio sonda todas as páginas na memória, definindo o bit de referência como 0. Em outro momento, o ponteiro de trás do relógio examina o bit de referência para as páginas na memória, anexando cada página cujo bit de referência ainda está definido como 0 à lista livre e escrevendo seu conteúdo no disco, se estiver modificado. O Solaris mantém uma lista em cache das páginas que foram “liberadas”, mas ainda não foram sobrescritas. A lista livre contém quadros que possuem conteúdo inválido. As páginas podem ser **reivindicadas** da lista em cache se foram acessadas antes de movidas para a lista livre.

O algoritmo pageout utiliza vários parâmetros para controlar a taxa em que as páginas são sondadas (conhecida como *scanrate*). A *scanrate* é expressa em páginas por segundo e varia de *slowscan* até *fastscan*. Quando a memória livre fica abaixo de *lotsfree*, a sondagem ocorre em *slowscan* páginas por segundo e prossegue até *fastscan*, dependendo da quantidade de memória livre disponível. O valor-padrão de *slowscan* é de 100 páginas por segundo; *fastscan* é definido como o valor (*total de páginas físicas*)/2 páginas por segundo, com um máximo de 8.192 páginas por segundo. Isso é mostrado na [Figura 9.29](#) (com *fastscan* definido no máximo).

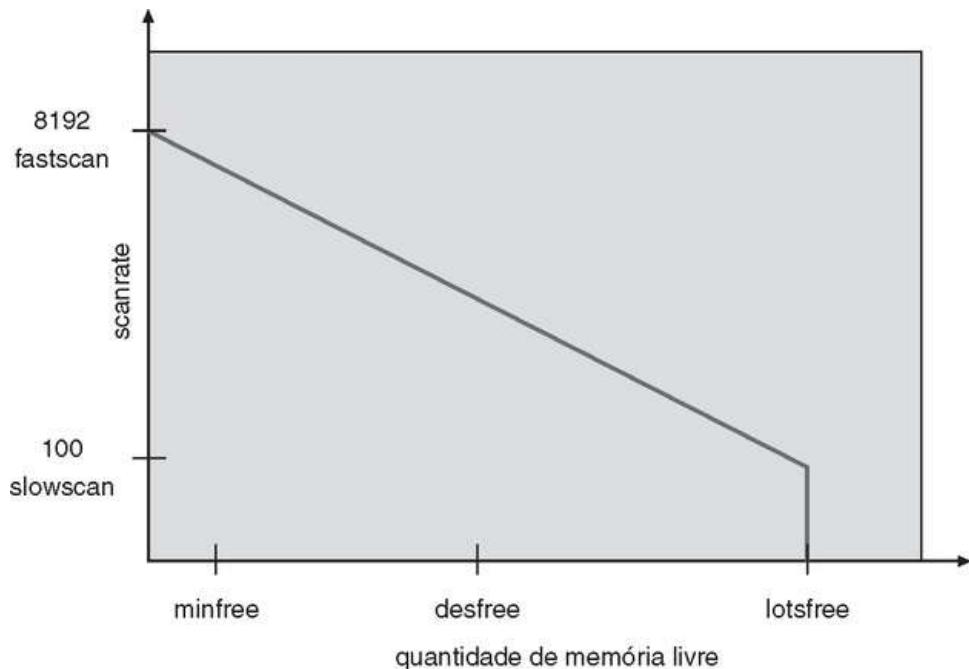


FIGURA 9.29 Verificador de página do Solaris.

A distância (em páginas) entre os ponteiros do relógio é determinada por um parâmetro do sistema, *handspread*. A quantidade de tempo entre o ponteiro da frente apagando um bit e o ponteiro de trás investigando seu valor depende da *scanrate* e da *handspread*. Se a *scanrate* for 100 páginas por segundo e a *handspread* for 1.024 páginas, 10 segundos poderão se passar entre o momento em que um bit é definido pelo ponteiro da frente e o momento em que ele é verificado pelo ponteiro de trás. Contudo, devido às demandas impostas sobre a memória do sistema, uma *scanrate* de vários milhares não é rara. Isso significa que a quantidade de tempo entre apagar e investigar um bit é de alguns segundos.

Como já mencionamos, o processo pageout verifica a memória quatro vezes por segundo. Todavia, se a memória livre estiver abaixo de *desfree* (Figura 9.29), pageout será executada 100 vezes por segundo com a intenção de manter pelo menos *desfree* de memória livre disponível. Se o processo pageout for incapaz de manter a quantidade de memória livre em *desfree* por uma média de 30 segundos, o kernel começará a trocar processos, liberando assim todas as páginas alocadas aos processos trocados. Em geral, o kernel procura processos ociosos por longos períodos. Se o sistema não conseguir manter a quantidade de memória livre em *minfree*, o processo pageout será chamado para cada requisição de uma nova página.

Versões recentes do kernel do Solaris forneceram melhorias do algoritmo de paginação. Uma melhoria desse tipo envolve o reconhecimento de páginas de bibliotecas compartilhadas. As páginas pertencentes a bibliotecas compartilhadas por vários processos – mesmo que possam ser reivindicadas pelo analisador – são puladas durante o processo de sondagem de página. Outra melhoria refere-se à distinção de páginas que foram alocadas a processos de páginas alocadas a arquivos normais. Isso é conhecido como **paginação por prioridade** e é explicado na [Seção 11.6.2](#).

9.11 Resumo

É desejável poder executar um processo cujo espaço de endereços lógicos seja maior do que o espaço disponível para endereços físicos. A memória virtual é uma técnica que nos possibilita mapear um grande espaço de endereços lógico em uma memória física menor. A memória virtual permite a execução de processos extremamente grandes, além de aumentar o grau de multiprogramação, aumentando a utilização de CPU. Além do mais, isso evita que os programadores de aplicação se preocupem com a disponibilidade de memória. Além disso, com a memória virtual, vários processos podem compartilhar bibliotecas de sistema e memória. A memória virtual também nos permite usar um tipo eficiente de criação de processo, conhecido como cópia na escrita, no qual os processos pai e filho compartilham páginas reais da memória.

A memória virtual é implementada pela paginação por demanda. A paginação por demanda pura nunca traz uma página até essa página ser referenciada. A primeira referência causa uma falha de página para o sistema operacional. O kernel do sistema operacional consulta uma tabela interna para determinar onde a página está localizada no armazenamento de apoio. Depois, ele encontra um quadro livre e lê a página do armazenamento de apoio. A tabela de página é atualizada para refletir essa mudança, e a instrução que causou a falha de página é reiniciada. Essa técnica permite que um processo seja executado mesmo que sua imagem de memória inteira não esteja na memória principal ao mesmo tempo. Desde que a taxa de falha de página seja baixa, o desempenho será aceitável.

Podemos usar a paginação por demanda para reduzir a quantidade de quadros alocados a um processo. Esse arranjo pode aumentar o grau de multiprogramação (permitindo que mais processos estejam disponíveis para execução de uma só vez) e - pelo menos, teoricamente - a utilização de CPU pelo sistema. Isso também permite aos processos serem executados, embora seus requisitos de memória ultrapassem a memória física total disponível. Esses processos são executados na memória virtual.

Se os requisitos totais de memória ultrapassarem a capacidade da memória física, pode ser necessário substituir as páginas da memória para quadros livres por novas páginas. Diversos algoritmos de substituição de página são utilizados. A substituição de página FIFO é fácil de programar, mas sofre com a anomalia de Belady. A substituição de página ótima exige conhecimento futuro. A substituição LRU é uma aproximação da substituição de página ótima, mas até mesmo ela pode ser difícil de implementar. A maioria dos algoritmos de substituição de página, como o algoritmo da segunda chance, é uma aproximação da substituição LRU.

Além do algoritmo de substituição de página, uma política de alocação de quadros é necessária. A alocação pode ser fixa, sugerindo a substituição de página local, ou dinâmica, sugerindo a substituição global. O modelo de conjunto de trabalho considera que os processos são executados em localidades. O conjunto de trabalho é o conjunto de páginas na localidade atual. De acordo com isso, cada processo deverá receber quadros suficientes para o seu conjunto de trabalho atual. Se um processo não tiver memória suficiente para o seu conjunto de trabalho, ele causará o thrashing. Fornecer quadros suficientes para cada processo, a fim de evitar o thrashing, pode exigir troca e escalonamento de processos.

Muitos sistemas operacionais proveem recursos para arquivos de mapeamento de memória, permitindo assim que a E/S de arquivo seja tratada como acesso à memória de rotina. A API Win 32 implementa a memória compartilhada por meio do mapeamento de memória.

Os processos do kernel normalmente exigem que a memória seja alocada usando páginas que sejam fisicamente contíguas. O sistema buddy aloca memória aos processos do kernel em unidades com tamanhos de acordo com uma potência de 2, que normalmente resulta em fragmentação. Alocadores slab atribuem estruturas de dados do kernel aos caches associados a slabs, que são compostas de uma ou mais páginas fisicamente contíguas. Com a alocação de slab, nenhuma memória é desperdiçada devido à fragmentação, e as requisições de memória podem ser satisfeitas rapidamente.

Além de exigir que solucionemos os principais problemas da substituição de páginas e alocação de quadros, o projeto apropriado de um sistema de paginação exige considerarmos pré-paginação, o tamanho da página, alcance de TLB, tabelas de página invertidas, estrutura do program, interlock de E/S e outras questões.

Exercícios práticos

- 9.1. Sob quais circunstâncias ocorrem as falhas de página? Descreva as ações tomadas pelo sistema operacional quando ocorre uma falha de página.
- 9.2. Suponha que você tenha uma string de referência de página para um processo com m quadros (inicialmente vazios). A string de referência de página possui comprimento p ; n números de página distintos ocorrem nela. Responda a estas perguntas para qualquer algoritmo de substituição de página:
- Qual é o limite inferior no número de falhas de página?
 - Qual é o limite superior no número de falhas de página?
- 9.3. Quais das seguintes técnicas de programação e estruturas são “boas” para o ambiente paginado por demanda? Quais “não são boas”? Explique suas respostas.
- Pilha
 - Tabela de símbolos de hash
 - Busca sequencial
 - Busca binária
 - Código puro
 - Operações de vetor
 - Indireção
- 9.4. Considere os seguintes algoritmos de substituição de página. Avalie esses algoritmos em uma escala de cinco pontos, de “ruim” a “perfeito”, de acordo com sua taxa de falha de página. Separe aqueles algoritmos que sofrem da anomalia de Belady daqueles que não sofrem.
- Substituição LRU
 - Substituição FIFO
 - Substituição ótima
 - Substituição da segunda chance
- 9.5. Quando a memória virtual é implementada em um sistema de computação, existem certos custos associados com a técnica e certos benefícios. Liste os custos e os benefícios. É possível que os custos sejam maiores que os benefícios? Se for, que medidas podem ser tomadas para garantir que isso não aconteça?
- 9.6. Um sistema operacional admite uma memória virtual paginada, usando um processador central com um tempo de ciclo de 1 microsegundo. É necessário 1 microsegundo adicional para acessar uma página diferente da atual. As páginas possuem 1.000 words, e o dispositivo de paginação é um tambor que gira a 3.000 rotações por minuto e transfere 1 milhão de words por segundo. As seguintes medições estatísticas foram obtidas do sistema:
- Um por cento de todas as instruções executadas acessaram uma página diferente da página atual.
 - Das instruções que acessaram outra página, 80% acessaram uma página já contida na memória.
 - Quando uma nova página era exigida, a página substituída era modificada em 50% do tempo. Calcule o tempo de instrução efetivo nesse sistema, supondo que o sistema esteja executando um processo apenas e que o processador esteja ocioso durante as transferências de tambor.
- 9.7. Considere o array bidimensional A:
- ```
int A[][] = new int[100][100];
```
- onde  $A[0][0]$  está no local 200 em um sistema de memória paginada com páginas com tamanho 200. Um pequeno processo que manipula a matriz reside na página 0 (locais 0 a 199). Assim, cada busca de instrução será a partir da página 0.
- Para três quadros de página, quantas falhas de página são geradas pelas seguintes malhas de inicialização de array, usando a substituição LRU e supondo que o quadro de [página 1](#) contenha o processo e os outros dois estejam inicialmente vazios?
- for (int j = 0; j < 100; j++)  
for (int i = 0; i < 100; i++)  
 $A[i][j] = 0;$
  - for (int i = 0; i < 100; i++)  
for (int j = 0; j < 100; j++)  
 $A[i][j] = 0;$
- 9.8. Considere a seguinte string de referência de página:

1, 2, 3, 4, 2, 1, 5, 6, 2, 1, 2, 3, 7, 6, 3, 2, 1, 2, 3, 6

Quantas falhas de página ocorreriam para os seguintes algoritmos de substituição, considerando um, dois, três, quatro, cinco, seis e sete quadros? Lembre-se de que todos os quadros estão inicialmente vazios, de modo que suas primeiras páginas exclusivas custarão uma falha cada.

- Substituição LRU

- Substituição FIFO
- Substituição ótima

- 9.9. Suponha que você queira usar um algoritmo de paginação que exija um bit de referência (como uma substituição da segunda chance ou o modelo de conjunto de trabalho), mas o hardware não ofereça um. Esboce como você poderia simular um bit de referência mesmo que não haja um fornecido pelo hardware, ou explique por que não é possível fazer isso. Se possível, calcule qual seria o custo.
- 9.10. Você imaginou um novo algoritmo de substituição de página que você acredita ser ótimo. Em alguns casos de teste contorcidos, ocorre anomalia de Belady. O novo algoritmo é ótimo? Explique sua resposta.
- 9.11. A segmentação é semelhante à paginação, mas usa “páginas” de tamanho variável. Defina dois algoritmos de substituição de segmento com base nos esquemas de substituição de página FIFO e LRU. Lembre-se de que, como os segmentos não têm o mesmo tamanho, o segmento escolhido para ser substituído pode não ser grande o bastante para deixar locais consecutivos suficientes para o segmento necessário. Considere as estratégias para sistemas onde os segmentos não possam ser relocados e as estratégias para sistemas onde eles possam.
- 9.12. Considere um sistema de computação paginado por demanda onde o grau de multiprogramação seja atualmente fixado em quatro. O sistema foi medido recentemente para determinar a utilização da CPU e o disco de paginação. Os resultados são uma das seguintes alternativas. Para cada caso, o que está acontecendo? O grau de multiprogramação pode ser aumentado para aumentar a utilização de CPU? A paginação está ajudando?
- Utilização de CPU 13%; utilização de disco 97%
  - Utilização de CPU 87%; utilização de disco 3%
  - Utilização de CPU 13%; utilização de disco 3%
- 9.13. Temos um sistema operacional para uma máquina que usa registradores de base e limite, mas modificamos a máquina para fornecer uma tabela de página. As tabelas de página podem ser preparadas para simular registradores de base e limite? Como elas podem ser preparadas, ou por que não podem?

## Exercícios

- 9.14. Suponha que um programa tenha acabado de referenciar um endereço na memória virtual. Descreva um cenário em que possa haver cada uma das seguintes ocorrências. (Se tal cenário puder ocorrer, explique o motivo.)
- Falta de TLB sem falha de página
  - Falta de TLB e falha de página
  - Acerto de TLB sem falha de página
  - Acerto de TLB e falha de página
- 9.15. Suponha que haja três estados de thread: *pronto*, *executando* e *bloqueado*, e uma thread pronta e esperando para ser escalonada, está executando no processador ou está bloqueada (por exemplo, esperando por E/S). Essa visão simplificada é ilustrada na [Figura 9.30](#). Supondo que uma thread esteja no estado *executando*, responda às seguintes perguntas. (Explique sua resposta.)
- ```
graph TD; Pronto((Pronto)) --> Executando((Executando)); Executando --> Bloqueado((Bloqueado)); Bloqueado --> Pronto; Pronto --> Executando;
```
- FIGURA 9.30** Diagrama de estado de thread para o [Exercício 9.15](#).
- a. A thread mudará de estado se ela incorrer em uma falha de página? Nesse caso, para qual novo estado?
 - b. A thread mudará de estado se ela gerar uma falta de TLB que é resolvida na tabela de página? Nesse caso, para qual novo estado?
 - c. A thread mudará de estado se uma referência de endereço for resolvida na tabela de página? Nesse caso, para qual novo estado?
- 9.16. Considere um sistema que utiliza a paginação por demanda pura.
- a. Quando um processo inicia sua execução, como você caracterizaria a taxa de falha de página?
 - b. Uma vez que o conjunto de trabalho para um processo é carregado na memória, como você caracterizaria a taxa de falha de página?
 - c. Suponha que um processo mude sua locação e o tamanho do novo conjunto de trabalho seja muito grande para ser armazenado na memória livre disponível. Identifique algumas opções que os projetistas de sistemas poderiam ter para escolher no tratamento dessa situação.
- 9.17. Dê um exemplo que ilustre o problema com o reinício da instrução MVC (move character) no IBM 360/370 quando as regiões de origem e destino forem superpostas.
- 9.18. Discuta o suporte de hardware exigido para aceitar a paginação por demanda.
- 9.19. O que é o recurso de cópia na escrita, e sob quais circunstâncias é benéfico usar esse recurso? Qual é o suporte de hardware exigido para implementar esse recurso?
- 9.20. Certo computador fornece aos usuários um espaço de memória virtual de 2^{32} bytes. O computador possui 2^{18} bytes de memória física. A memória virtual é implementada pela paginação, e o tamanho de página é de 4.096 bytes. Um processo do usuário gera o endereço virtual 11123456. Explique como o sistema estabelece a locação física correspondente. Faça a distinção entre as operações de software e de hardware.
- 9.21. Suponha que tenhamos uma memória paginada por demanda. A tabela de página é mantida em registradores. São necessários 8 milissegundos para atender a uma falha de página se um quadro vazio estiver disponível ou se a página substituída não estiver modificada, e 20 milissegundos se a página substituída estiver modificada. O tempo de acesso à memória é de 100 nanossegundos.
- Suponha que a página a ser trocada seja modificada 70% do tempo. Qual é a taxa de falha de página aceitável para um tempo de acesso eficaz de mais de 200 nanossegundos?
- 9.22. Quando ocorre uma falha de página, o processo solicitando a página precisa bloquear enquanto espera que a página seja trazida do disco para a memória física. Suponha que haja um processo com cinco threads em nível de usuário e que o mapeamento das threads do usuário para threads do kernel seja muitos para um. Se uma thread do usuário incorrer em uma falha de página enquanto acessa sua pilha, as outras threads do usuário pertencentes ao mesmo processo também serão afetadas pela falha de página – ou seja, elas também terão que esperar

que a página que falhou seja trazida para a memória? Explique.

- 9.23. Considere a tabela de página mostrada na [Figura 9.31](#) para um sistema com endereços virtuais e físicos de 12 bits e páginas de 256 bytes. A lista de quadros de página livres é *D, E, F* (ou seja, *D* está no início da lista e *F* no final).

Página	Quadro de página
0	—
1	2
2	C
3	A
4	—
5	4
6	3
7	—
8	B
9	0

FIGURA 9.31 Diagrama de estado de thread para o [Exercício 9.23](#).

Converta os endereços virtuais a seguir em endereços físicos equivalentes em hexadecimal. Todos os números são indicados em hexadecimal. (Um traço para um quadro de página indica que a página não está na memória.)

- 9EF
- 111
- 700
- OFF

- 9.24. Suponha que você esteja monitorando a velocidade em que o ponteiro no algoritmo de relógio (que indica a página candidata para substituição) se move. O que você pode dizer sobre o sistema se observar o seguinte comportamento:

- O ponteiro está se movendo rapidamente.
- O ponteiro está se movendo lentamente.

- 9.25. Discuta situações sob as quais o algoritmo de substituição da página usada menos frequentemente gera menos falhas de página do que o algoritmo de substituição da página usada menos recentemente. Discuta também sob que circunstância acontece o oposto.

- 9.26. Discuta situações sob as quais o algoritmo de substituição de página usada mais frequentemente gera menos falhas de página do que o algoritmo de substituição de página usada com menos frequência. Discuta também sob que circunstância acontece o oposto.

- 9.27. O sistema VAX/VMS utiliza um algoritmo FIFO para páginas residentes e um banco de quadros livres das páginas usadas recentemente. Suponha que o banco de quadros livres seja controlado com o uso da política de substituição de uso menos recente. Responda às seguintes perguntas:

- Se ocorrer uma falha de página e se a página não existir no banco de quadros livres, como o espaço livre é gerado para a página recém-requisitada?
- Se ocorrer uma falha de página e se a página existir no banco de quadros livres, como a página residente é definida e o banco de quadros livres é gerenciado para criar espaço para a página requisitada?
- Para que o sistema se degenera, se o número de páginas residentes for definido como um?
- Para que o sistema se degenera, se o número de páginas no banco de quadros livres for zero?

- 9.28. Considere um sistema de paginação por demanda com as seguintes utilizações medidas no tempo:

Utilização de CPU	20%
Disco de paginação	97,7%
Outros dispositivos de E/S	5%

Para cada um dos seguintes, diga se a utilização de CPU melhorará (ou se isso é provável). Explique suas respostas.

- a. Instalar uma CPU mais rápida.
 - b. Instalar um disco de paginação maior.
 - c. Aumentar o grau de multiprogramação.
 - d. Diminuir o grau de multiprogramação.
 - e. Instalar mais memória principal.
 - f. Instalar um disco rígido mais rápido ou múltiplos controladores com múltiplos discos rígidos.
 - g. Acrescentar a pré-paginação aos algoritmos de busca de página.
 - h. Aumentar o tamanho da página.
- 9.29. Suponha que uma máquina ofereça instruções que podem acessar locais da memória usando o esquema de endereçamento indireto em um nível. Qual é a sequência de falhas de página que acontece quando todas as páginas de um programa atualmente não estão residentes e a primeira instrução do programa é uma operação de carga de memória indireta? O que acontece quando o sistema operacional está usando uma técnica de alocação de quadros por processo e somente duas páginas são alocadas para esse processo?
- 9.30. Suponha que sua política de substituição (em um sistema paginado) seja examinar cada página regularmente e descartá-la se não tiver sido usada desde o último exame. O que você ganharia e o que você perderia usando essa política em vez da substituição LRU ou da segunda chance?
- 9.31. Um algoritmo de substituição de página deveria minimizar a quantidade de falhas de página. Podemos conseguir essa redução distribuindo as páginas muito usadas uniformemente por toda a memória, em vez de deixar que disputem uma pequena quantidade de quadros de página. Podemos associar a cada quadro de página um contador do número de páginas associadas a esse quadro. Depois, para substituir uma página, podemos procurar o quadro de página com o menor contador.
- Defina um algoritmo de substituição de página usando essa ideia básica. Especificamente, resolva estes problemas:
 - Qual é o valor inicial dos contadores?
 - Quando os contadores são aumentados?
 - Quando os contadores são diminuídos?
 - Como é selecionada a página a ser substituída?
 - Quantas falhas de página ocorrem para o seu algoritmo para a seguinte string de referência, com quatro quadros de página?

1, 2, 3, 4, 5, 3, 4, 1, 6, 7, 8, 7, 8, 9, 7, 8, 9, 5, 4, 5, 4, 2

- c. Qual é a quantidade mínima de falhas de página para uma estratégia de substituição de página ótima, para a string de referência na parte b, com quatro quadros de página?
- 9.32. Considere um sistema de paginação por demanda com um disco de paginação que possui um tempo médio de acesso e transferência de 20 milissegundos. Os endereços são traduzidos por meio de uma tabela de página na memória principal, com um tempo de acesso de 1 microsegundo por acesso à memória. Assim, cada referência à memória no decorrer da tabela de página exige dois acessos. Para melhorar esse tempo, acrescentamos uma memória associativa, que reduz o tempo de acesso a uma referência à memória se a entrada da tabela de página estiver na memória associativa.
- Considere que 80% dos acessos estão na memória associativa e que, do restante, 10% (ou 2% do total) causam falhas de página. Qual é o tempo efetivo de acesso à memória?
- 9.33. Qual é a causa do thrashing? Como o sistema detecta o thrashing? Ao detectar o thrashing, o que o sistema pode fazer para eliminar esse problema?
- 9.34. É possível que um processo tenha dois conjuntos de trabalho, um representando dados e outro representando código? Explique.
- 9.35. Considere o parâmetro Δ usado para definir a janela do conjunto de trabalho no modelo de conjunto de trabalho. Qual é o efeito de definir Δ com um valor pequeno na frequência de falha de página e o número de processos ativos (não suspensos) atualmente em execução no sistema? Qual é o efeito quando Δ é definido, por exemplo, com um valor muito alto?
- 9.36. Suponha que haja um segmento inicial de 1.024 KB onde a memória é alocada usando o sistema buddy. Usando a [Figura 9.26](#) como guia, desenhe a árvore que ilustra como as seguintes requisições de memória são alocadas:
- requisitar 240 bytes
 - requisitar 120 bytes
 - requisitar 60 bytes
 - requisitar 130 bytes
- Em seguida, modifique a árvore para as seguintes liberações de memória. Faça a união sempre

que for possível:

- liberar 240 bytes
- liberar 60 bytes
- liberar 120 bytes

- 9.37. Considere um sistema que oferece suporte para threads em nível de usuário e em nível de kernel. O mapeamento nesse sistema é um para um (há uma thread de kernel correspondente para cada thread de usuário). Um processo com multithreading consiste em (a) um conjunto de trabalho para o processo inteiro ou (b) um conjunto de trabalho para cada thread? Explique.
- 9.38. O algoritmo de alocação de slab usa um cache separado para cada tipo de objeto diferente. Supondo que haja um cache por tipo de objeto, explique por que isso não pode ser aumentado para CPUs múltiplas. O que poderia ser feito para resolver essa questão de escalabilidade?
- 9.39. Considere um sistema que aloca páginas de diferentes tamanhos aos seus processos. Quais são as vantagens desse esquema de paginação? Que modificações no sistema de memória virtual fornecem essa funcionalidade?

Problemas de programação

9.40. Escreva um programa que implemente os algoritmos de substituição de página FIFO e LRU apresentados neste capítulo. Primeiro, gere uma string de referência de página aleatória, na qual os números de página variam de 0 a 9. Aplique a string de referência de página aleatória a cada algoritmo e registre a quantidade de falhas de página incorridas em cada algoritmo. Implemente os algoritmos de substituição de modo que a quantidade de quadros de página possa variar também. Suponha que a paginação por demanda seja utilizada. Seus algoritmos serão baseados na classe abstrata representada na [Figura 9.32](#).

```
public abstract class ReplacementAlgorithm
{
    // o número de falhas de página
    protected int pageFaultCount;

    // o número do quadro de página físico
    protected int pageFrameCount;

    // pageFrameCount - o número de quadros de página físicos
    public ReplacementAlgorithm(int pageFrameCount) {
        if (pageFrameCount < 0)
            throw new IllegalArgumentException();

        this.pageFrameCount = pageFrameCount;
        pageFaultCount = 0;
    }

    // return - o número de falhas de página que ocorreram.
    public int getPageFaultCount() {
        return pageFaultCount;
    }

    // int pageNumber = número da página a ser inserida
    public abstract void insert(int pageNumber);
}
```

FIGURA 9.32 Classe abstrata ReplacementAlgorithm.

Crie e implemente duas classes – LRU e FIFO – que estendem `ReplacementAlgorithm`. Cada uma dessas classes implementará o método `insert()`, uma classe usando o algoritmo de substituição de página LRU e a outra usando o algoritmo FIFO.

Quando você tiver implementado os algoritmos FIFO e LRU, experimente um número diferente de quadros de página para determinada string de referência e registre o número de falhas de página. Um algoritmo funciona melhor do que o outro? Para determinado tamanho de string de referência, qual é o número ideal de quadros de página?

Projetos de programação

Projeto de um gerenciador de memória virtual

Esse projeto consiste em escrever um programa Java que traduz endereços lógicos para físicos para um espaço de endereços virtuais com tamanho de $2^{16} = 65.536$ bytes. O programa lerá de um arquivo que contém endereços lógicos e, usando uma TLB e uma tabela de página, traduzirá cada endereço lógico em seu endereço físico correspondente, gerando o valor do byte armazenado no endereço físico traduzido.

Seu programa lerá um arquivo contendo vários números inteiros de 32 bits que representam os endereços lógicos. Entretanto, você só precisa se preocupar com endereços de 16 bits, de modo que precisa mascarar os 16 bits mais à direita de cada endereço lógico. Esses 16 bits são divididos em (1) um número de página de 8 bits; e (2) um deslocamento de página de 8 bits. Logo, os endereços são estruturados conforme mostra a [Figura 9.33](#).



FIGURA 9.33 Estrutura de endereços.

Outros detalhes são:

- 2^8 entradas na tabela de página
- Tamanho de página = 2^8 bytes
- 16 entradas na TLB
- Tamanho do quadro = 2^8 bytes
- 256 quadros
- Memória física = 65536 bytes (256 quadros \times tamanho de quadro de 256 bytes)

Além disso, seu programa só precisa se preocupar com a leitura de endereços lógicos e a tradução para seus endereços físicos correspondentes. Você não precisa dar suporte para a escrita no espaço de endereços lógicos.

Tradução de endereços

Seu programa traduzirá endereços de lógicos para físicos usando uma TLB e tabela de página, conforme esboçado na [Seção 8.4](#). Primeiro, o número de página é extraído do endereço lógico, e a TLB é consultada. No caso de um acerto de TLB, o número do quadro é obtido pela TLB. No caso de uma falta, a tabela de página deverá ser consultada. Se o número do quadro não puder ser obtido pela tabela de página, haverá uma falha de página. Mostramos uma representação do processo de tradução na [Figura 9.34](#).

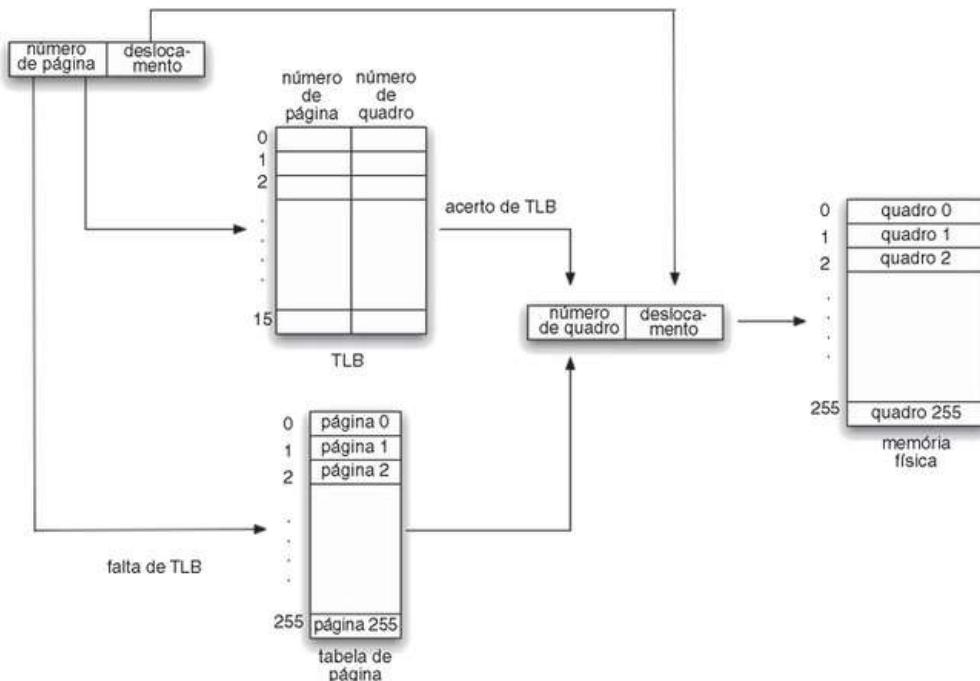


FIGURA 9.34 Representação do processo de tradução de endereços.

Tratamento das falhas de página

Seu programa implementará a paginação por demanda conforme descrita na [Seção 9.2](#). O armazenamento de apoio é representado pelo arquivo BACKING_STORE, um arquivo binário com 65.536 bytes. Quando houver uma falha de página, você lerá uma página de 256 bytes do arquivo BACKING_STORE e a armazenará em um quadro de página disponível na memória física. Por exemplo, se o endereço lógico com o número de [página 15](#) resultar em uma falha de página, seu programa lerá a [página 15](#) de BACKING_STORE (lembre-se de que as páginas começam em 0 e possuem um comprimento de 256 bytes) e a armazenará em um quadro de página na memória física. Quando esse quadro estiver armazenado (e a tabela de página e a TLB forem atualizadas), os próximos acessos à [página 15](#) serão resolvidos ou pela TLB ou pela tabela de página.

Você precisará tratar BACKING_STORE como um arquivo de acesso aleatório, permitindo a busca em certas posições no arquivo para leitura. Finalmente, cuide para que, ao usar BACKING_STORE, o arquivo seja aberto somente para leitura, pois ele só precisará ser lido, e você não desejará modificá-lo:

```
fileName = new File("BACKING_STORE");
backingStore = new RandomAccessFile(fileName, "r");
```

O tamanho da memória física é o mesmo que o tamanho do espaço de endereço virtual - 65.536 bytes -, você não precisa se preocupar com as substituições de página durante uma falha de página. Mais adiante descreveremos uma modificação no projeto usando uma quantidade menor de memória física. Essa modificação exige uma estratégia de substituição de página.

Arquivo de teste

O arquivo `InputFile.txt` contém valores inteiros que representam os endereços lógicos que variam de 0 até 65535 (o tamanho do espaço de endereços virtuais). Seu programa abrirá esse arquivo, lerá cada endereço lógico e o traduzirá para o seu endereço físico correspondente, enviando o valor do byte com sinal existente no endereço físico.

Como começar

Escreva um programa simples, que extraia o número de página e o deslocamento dos números inteiros a seguir (com base na [Figura 9.33](#)):

1, 256, 32768, 32769, 128, 65534, 33153

Talvez o modo mais fácil de fazer isso seja usando os operadores Java para mascarar e deslocar bits. Uma vez que você consiga estabelecer corretamente o número de página e o deslocamento a partir de um número inteiro, estará pronto para começar.

Inicialmente, sugerimos que você contorne a TLB e use uma tabela de página, integrando a TLB apenas depois que sua tabela de página estiver funcionando corretamente. Lembre-se de que a tradução de endereços pode funcionar sem uma TLB; a TLB serve apenas para torná-la mais rápida. Quando você estiver pronto para implementar a TLB, lembre-se de que ela tem apenas 16 entradas, de modo que você precisará utilizar uma estratégia de substituição de página quando tiver que atualizar uma TLB inteira. Você poderá usar uma política FIFO ou LRU para atualizar sua TLB.

Como executar seu programa

Seu programa deverá ser executado da seguinte forma:

```
java TradutorEndereco InputFile.txt
```

Seu programa lerá o arquivo `InputFile.txt`, que contém 1000 endereços lógicos, variando de 0 a 65535. Seu programa deverá traduzir cada endereço lógico em um endereço físico e determinar o conteúdo do byte sinalizado armazenado no endereço físico correto.

Seu programa deverá informar os seguintes valores:

1. O endereço lógico que está sendo traduzido (o valor inteiro que está sendo lido de `InputFile.txt`).
2. O endereço físico correspondente (a tradução do endereço lógico, feita pelo programa).
3. O valor do byte com sinal armazenado no endereço físico traduzido.

Também on-line oferecemos o arquivo `correct.txt`, que contém os valores de saída corretos para o arquivo `InputFile.txt`. Use esse arquivo para determinar se o seu programa está traduzindo endereços lógicos em endereços físicos corretamente.

Estatísticas

Depois que concluir, seu programa deverá informar as seguintes estatísticas:

1. Taxa de falha de página: o percentual de referências de endereço que resultaram em falhas de página.
2. Taxa de acerto da TLB: o percentual de referências de endereço que foram resolvidos na TLB.
Visto que os endereços lógicos em `InputFile.txt` são gerados aleatoriamente e não refletem qualquer locação de acesso à memória, não espere ter uma taxa de acerto da TLB muito alta.

Modificações

Esse projeto considera que a memória física tem o mesmo tamanho do espaço de endereços virtuais. Na prática, a memória física normalmente é muito menor do que um espaço de endereços virtuais. Modifique seu programa para usar um espaço de endereços físicos menor. Ao invés de usar 256 quadros de página, recomendamos usar 128. Isso exigirá que seu programa registre os quadros de página livres, além de implementar uma política de substituição de página usando políticas FIFO ou LRU ([Seção 9.4](#)).

Notas bibliográficas

A paginação por demanda foi usada no sistema Atlas, implementado no computador MUSE da Universidade de Manchester por volta de 1960 ([Kilburn e outros \[1961\]](#)). Outro antigo sistema de paginação por demanda foi o MULTICS, implementado no sistema GE 645 ([Organick \[1972\]](#)).

[Belady e outros \[1969\]](#) foram os primeiros pesquisadores a observar que a estratégia de substituição FIFO pode produzir a anomalia que recebe o nome de Belady. [Mattson e outros \[1970\]](#) demonstraram que os algoritmos de pilha não estão sujeitos à anomalia de Belady.

O algoritmo de substituição ótima foi apresentado por [Belady \[1966\]](#). [Mattson e outros \[1970\]](#) provaram que ele seria o ideal. O algoritmo ótimo de Belady é para uma alocação fixa; [Prieve e Fabry \[1976\]](#) têm um algoritmo ótimo para situações em que a alocação pode variar.

O algoritmo do relógio melhorado foi discutido por [Carr e Hennessy \[1981\]](#).

O modelo de conjunto de trabalho foi desenvolvido por [Denning \[1968\]](#). As discussões referentes ao modelo do conjunto de trabalho foram apresentadas por [Denning \[1980\]](#).

O esquema para monitorar a taxa de falha de página foi desenvolvido por [Wulf \[1969\]](#), que aplicou essa técnica com sucesso ao sistema computadorizado Burroughs B5500.

[Wilson e outros \[1995\]](#) apresentaram vários algoritmos para alocação dinâmica de memória. [Johnstone e Wilson \[1998\]](#) descreveram diversas questões de fragmentação de memória. Os alocadores de memória de sistema buddy foram descritos em [Knowlton \[1965\]](#), [Peterson e Norman \[1977\]](#) e [Purdom e Stigler \[1970\]](#). [Bonwick \[1994\]](#) discutiu o alocador de slab, e [Bonwick e Adams \[2001\]](#) estenderam a discussão para processadores múltiplos. Outros algoritmos de ajuste da memória podem ser encontrados em [Stephenson \[1983\]](#), [Bays \[1977\]](#) e [Brent \[1989\]](#). Um estudo das estratégias de alocação de memória pode ser encontrado em [Wilson e outros \[1995\]](#).

[Solomon e Russinovich \[2000\]](#) e [Russinovich e Solomon \[2005\]](#) descrevem como o Windows 2000 implementa a memória virtual. Mauro e McDougall [2001] discutem a memória virtual no Solaris. As técnicas de memória virtual no Linux e no BSD são descritas por [Bovet e Cesati \[2002\]](#), e [McKusick e outros \[1996\]](#), respectivamente. [Ganapathy e Schimmel \[1998\]](#) e [Navarro e outros \[2002\]](#) discutem o suporte do sistema operacional para múltiplos tamanhos de página. [Ortix \[2001\]](#) discute a memória virtual utilizada em um sistema operacional embutido de tempo real.

[Jacob e Mudge \[1998b\]](#) comparam implementações da memória virtual nas arquiteturas MIPS, PowerPC e Pentium. Um artigo acompanhante ([Jacob e Mudge \[1998a\]](#)) descreve o suporte de hardware necessário para a implementação da memória virtual em seis arquiteturas diferentes, incluindo UltraSPARC.

PARTE IV

GERÊNCIA DE ARMAZENAMENTO

ESBOÇO

- [Capítulo 13: Introdução a Gerência de armazenamento](#)
- [Capítulo 14: Interface do sistema de arquivos](#)
- [Capítulo 15: Implementação do sistema de arquivos](#)
- [Capítulo 16: Estrutura de armazenamento em massa](#)
- [Capítulo 17: Sistemas de E/S](#)

Introdução a Gerência de armazenamento

Como a memória principal costuma ser muito pequena para acomodar todos os dados e programas permanentemente, o sistema computadorizado precisa prover armazenamento secundário para dar suporte à memória principal. Os sistemas computadorizados modernos utilizam discos como principal meio de armazenamento on-line para as informações (programas e dados). O sistema de arquivos fornece o mecanismo para o armazenamento on-line e o acesso aos dados e programas que residem nos discos. Um arquivo é uma coleção de informações relacionadas, definidas por seu criador. Os arquivos são associados pelo sistema operacional a dispositivos físicos. Os arquivos são organizados em diretórios, para facilitar sua utilização.

Os dispositivos que se conectam a um computador variam em muitos aspectos. Alguns dispositivos transferem um caractere ou um bloco de caracteres de cada vez. Alguns só podem ser acessados sequencialmente; outros, aleatoriamente. Alguns transferem dados de forma síncrona, enquanto outros os transferem de forma assíncrona. Alguns são dedicados, alguns compartilhados. Eles podem ser apenas de leitura ou de leitura e escrita. Eles variam bastante em velocidade. De muitas formas, eles também são o componente principal mais lento do computador.

Devido a toda essa variação de dispositivos, o sistema operacional precisa fornecer muita funcionalidade às aplicações, para permitir que eles controlem todos os aspectos dos dispositivos. Um objeto importante do subsistema de E/S de um sistema operacional é fornecer a interface mais simples possível para o restante do sistema. Como os dispositivos são um gargalo no desempenho, outra chave é otimizar a E/S para o máximo de concorrência.

CAPÍTULO 10

Interface do sistema de arquivos

Para a maioria dos usuários, o sistema de arquivos é o aspecto mais visível de um sistema operacional. Ele provê o mecanismo para o armazenamento on-line e o acesso a dados e programas do sistema operacional e de todos os usuários do computador. O sistema de arquivos consiste em duas partes distintas: uma coleção de *arquivos*, cada um armazenando dados relacionados, e uma *estrutura de diretório*, que organiza e fornece informações sobre todos os arquivos no sistema. Os sistemas de arquivos residem nos dispositivos, os quais exploraremos por completo nos próximos capítulos, mas que são introduzidos aqui. Neste capítulo, consideramos os diversos aspectos dos arquivos e das principais estruturas de diretório. Também discutimos a semântica do compartilhamento de arquivos entre vários processos, usuários e computadores. Finalmente, discutimos as maneiras de lidar com a *proteção de arquivos*, necessárias quando temos vários usuários e queremos controlar quem pode acessar os arquivos e como os arquivos podem ser acessados.

OBJETIVOS DO CAPÍTULO

- Explicar a função dos sistemas de arquivos.
- Descrever as interfaces dos sistemas de arquivos.
- Discutir as escolhas de projeto do sistema de arquivos, incluindo os métodos de acesso, compartilhamento de arquivos, lock de arquivo e estruturas de diretório.
- Explorar a proteção do sistema de arquivos.

10.1 Conceito de arquivo

Os computadores podem armazenar informações em diversos meios de armazenamento, como discos magnéticos, fitas magnéticas e discos ópticos. Para o computador ser conveniente para uso, o sistema operacional provê uma visão lógica uniforme do armazenamento de informações. O sistema operacional se afasta das propriedades físicas dos dispositivos de armazenamento para definir uma unidade de armazenamento lógica, o **arquivo**. Os arquivos são mapeados pelo sistema operacional em dispositivos físicos. Esses dispositivos de armazenamento não são voláteis, de modo que o conteúdo persiste mesmo em casos de falta de energia e reinicialização do sistema.

Um arquivo é uma coleção nomeada de informações relacionadas, registradas no armazenamento secundário. Do ponto de vista de um usuário, um arquivo é a menor alocação de armazenamento secundário, ou seja, os dados não podem ser escritos no armazenamento secundário a menos que estejam dentro de um arquivo. Em geral, os arquivos representam programas (nas formas de fonte e objeto) e dados. Os arquivos de dados podem ser numéricos, alfabeticos, alfanuméricos ou binários. Os arquivos podem ser de forma livre, como arquivos de texto, ou podem ser formatados rigidamente. Um arquivo é uma sequência de bits, bytes, linhas ou registros, sendo que seu significado é definido pelo criador e usuário do arquivo. O conceito de arquivo, portanto, é bastante geral.

As informações em um arquivo são definidas por seu criador. Muitos tipos diferentes de informação podem ser armazenados em um arquivo - programas-fonte, programas objeto, programas executáveis, dados numéricos, texto, registros de folha de pagamento, imagens gráficas, gravações de som, e assim por diante. Um arquivo possui **estrutura** definida, que depende do seu tipo. Um arquivo de *texto* é uma sequência de caracteres organizados em linhas (e, possivelmente, páginas). Um arquivo-*fonte* é uma sequência de sub-rotinas e funções, cada uma organizada ainda mais como declarações seguidas por instruções executáveis. Um arquivo *objeto* é uma sequência de bytes organizados em blocos que podem ser entendidos pelo linker do sistema. Um arquivo *executável* é uma série de seções de código que o loader pode trazer para a memória e executar.

10.1.1 Atributos de arquivo

Um arquivo tem um nome, para a conveniência dos usuários humanos, e é referenciado por esse nome. Um nome costuma ser uma sequência de caracteres, como *exemplo.c*. Alguns sistemas diferenciam entre caracteres maiúsculos e minúsculos nos nomes, enquanto outros sistemas não os diferenciam. Quando um arquivo recebe um nome, ele se torna independente do processo, do usuário e até mesmo do sistema que o criou. Por exemplo, um usuário poderia criar o arquivo *exemplo.c*, e outro usuário poderia editar esse arquivo especificando seu nome. O proprietário do arquivo poderia gravar o arquivo em um disquete, enviá-lo em uma mensagem de correio eletrônico ou copiá-lo por uma rede, e ele ainda poderia se chamar *exemplo.c* no sistema de destino.

Além do nome, os arquivos possuem vários outros atributos. Os atributos do arquivo variam de um sistema operacional para outro, mas tipicamente consistem em:

- **Nome.** O nome de arquivo simbólico é a única informação armazenada no formato legível para o ser humano.
- **Identificador.** Essa marca exclusiva, normalmente um número, identifica o arquivo dentro do sistema de arquivos; ele é o nome do arquivo que não é legível para o ser humano.
- **Tipo.** Essa informação é necessária para os sistemas que admitem diferentes tipos de arquivos.
- **Local.** Essa informação é um ponteiro para um dispositivo e para o local do arquivo nesse dispositivo.
- **Tamanho.** O tamanho atual do arquivo (em bytes, palavras ou blocos) e possivelmente o tamanho máximo permitido são incluídos nesse atributo.
- **Proteção.** Informações de controle de acesso determinam quem pode realizar leitura, escrita, execução e assim por diante.
- **Hora, data e identificação do usuário.** Essas informações podem ser mantidas para criação, última modificação e último uso. Esses dados podem ser úteis para proteção, segurança e monitoramento de uso.

As informações sobre todos os arquivos são mantidas na estrutura de diretório, que também reside no armazenamento secundário. Em geral, uma entrada de diretório consiste no nome do arquivo e seu identificador exclusivo. O identificador, por sua vez, localiza os outros atributos do arquivo. Pode ser necessário mais de 1 kilobyte para registrar essas informações para cada arquivo. Em um sistema com muitos arquivos, o tamanho do próprio diretório pode ser de megabytes. Como os diretórios, assim como os arquivos, precisam ser não voláteis, eles precisam ser armazenados no dispositivo e trazidos para a memória aos poucos, conforme a necessidade.

10.1.2 Operações de arquivo

Um arquivo é um **tipo de dado abstrato**. Para definir um arquivo corretamente, precisamos

considerar as operações que podem ser realizadas sobre os arquivos. O sistema operacional pode fornecer chamadas de sistema para criar, escrever, ler, reposicionar, excluir e truncar arquivos. Vamos examinar o que o sistema operacional precisa fazer para realizar cada uma dessas seis operações básicas com arquivo. Depois, deverá ser fácil ver como outras operações semelhantes, como a operação para renomear um arquivo, podem ser implementadas.

■ **Criar um arquivo.** Duas etapas são necessárias para criar um arquivo. Na primeira, é preciso encontrar um espaço no sistema de arquivos para esse arquivo. Discutimos como alocar espaço para o arquivo no [Capítulo 11](#). Na segunda, uma entrada para o novo arquivo precisa ser feita no diretório.

■ **Escrever em um arquivo.** Para escrever um arquivo, podemos fazer uma chamada de sistema especificando o nome do arquivo e as informações a serem escritas no arquivo. Dado o nome do arquivo, o sistema pesquisa o diretório para encontrar o local do arquivo. O sistema precisa manter um ponteiro de *escrita* para o local no arquivo onde acontecerá a próxima escrita. O ponteiro de escrita precisa ser atualizado sempre que houver uma escrita.

■ **Ler de um arquivo.** Para ler de um arquivo, usamos uma chamada de sistema que especifica o nome do arquivo e onde (na memória) o próximo bloco do arquivo deve ser colocado. Novamente, o diretório é pesquisado em busca da entrada associada, e o sistema precisa manter um ponteiro de *leitura* para o local no arquivo onde a próxima leitura deverá ocorrer. Quando a leitura tiver terminado, o ponteiro de leitura será atualizado. Como um processo ou está lendo ou escrevendo em um arquivo, o local da operação atual pode ser mantido como um **ponteiro de posição atual do arquivo** por processo. As operações de leitura e escrita utilizam esse mesmo ponteiro, economizando espaço e diminuindo a complexidade do sistema.

■ **Reposicionar dentro de um arquivo.** O diretório é pesquisado em busca da entrada correta, e o ponteiro da posição atual do arquivo é reposicionado para determinado valor. O reposicionamento dentro de um arquivo não precisa envolver qualquer E/S real. Essa operação de arquivo também é conhecida como *busca* de arquivo.

■ **Excluir um arquivo.** Para excluir um arquivo, procuramos o arquivo nomeado no diretório. Ao encontrar a entrada de diretório associada, liberamos todo o espaço do arquivo, para poder ser reutilizado por outros arquivos, e apagamos a entrada do diretório.

■ **Truncar um arquivo.** O usuário pode querer apagar o conteúdo de um arquivo, mas manter seus atributos. Em vez de forçar o usuário a excluir o arquivo e depois recriá-lo, essa função permite a todos os atributos permanecerem inalterados - exceto pelo tamanho do arquivo -, mas permite ao arquivo ser reiniciado para o tamanho zero e seu espaço ser liberado.

Essas seis operações básicas compreendem o conjunto mínimo das operações exigidas sobre o arquivo. Outras operações comuns são *acréscimo* de novas informações ao final de um arquivo existente e *renomeação* de um arquivo existente. Essas operações primitivas podem, então, ser combinadas para realizar outras operações com arquivos. Por exemplo, podemos criar uma *cópia* de um arquivo ou copiar o arquivo para outro dispositivo de E/S, como uma impressora ou um monitor de vídeo, com a criação de um novo arquivo e depois com a leitura do antigo e escrita no novo. Também podemos ter operações que permitem a um usuário obter e definir os diversos atributos de um arquivo. Por exemplo, podemos querer ter operações que permitem a um usuário determinar o status de um arquivo, como o tamanho do arquivo, e definir os atributos do arquivo, como o proprietário do arquivo.

A maioria das operações de arquivo mencionadas envolve a pesquisa, no diretório, da entrada associada ao arquivo nomeado. Para evitar essa pesquisa constante, muitos sistemas exigem o uso de uma chamada de sistema `open()` antes de esse arquivo ser usado ativamente pela primeira vez. O sistema operacional mantém uma pequena tabela, chamada **tabela de arquivos abertos**, contendo informações sobre todos os arquivos abertos. Quando uma operação de arquivo é requisitada, o arquivo é especificado por meio de um índice para essa tabela, de modo que nenhuma pesquisa é necessária. Quando o arquivo não está mais sendo usado, ele é fechado pelo processo, e o sistema operacional remove sua entrada da tabela de arquivos abertos. `create` e `delete` são chamadas de sistema que trabalham com arquivos fechados e não abertos.

Alguns sistemas abrem um arquivo implicitamente quando é feita a primeira referência a ele. O arquivo é fechado de forma automática quando a tarefa ou programa que abriu o arquivo termina. A maioria dos sistemas, porém, exige que o programador abra um arquivo explicitamente com a chamada de sistema `open()` antes de esse arquivo poder ser usado. A operação `open()` apanha um nome de arquivo e pesquisa o diretório, copiando a entrada do diretório para a tabela de arquivos abertos. A chamada `open()` também pode aceitar informações sobre o modo de acesso - criar, somente leitura, leitura/escrita, somente acréscimo, e assim por diante. Esse modo é comparado com as permissões do arquivo. Se o modo de requisição for permitido, o arquivo será aberto para o processo. A chamada de sistema `open()` costuma retornar um ponteiro para a entrada na tabela de arquivos abertos. Esse ponteiro, e não o nome de arquivo real, é usado em todas as operações de E/S, evitando qualquer pesquisa mais demorada e simplificando a interface da chamada de sistema.

A implementação das operações `open()` e `close()` é mais complicada em um ambiente no qual vários processos podem abrir o arquivo simultaneamente. Isso pode ocorrer em um sistema no qual várias aplicações diferentes podem abrir o mesmo arquivo ao mesmo tempo. Em geral, o sistema

operacional utiliza dois níveis de tabelas internas: uma tabela por processo e uma tabela para todo o sistema. A tabela por processo acompanha todos os arquivos abertos por um processo. Nessa tabela, encontram-se armazenadas informações referentes ao uso do arquivo pelo processo. Por exemplo, o ponteiro de arquivo atual para cada arquivo pode ser encontrado aqui. Os direitos de acesso ao arquivo e informações contábeis também podem estar incluídos.

Cada entrada na tabela por processo, por sua vez, aponta para uma tabela de arquivos abertos para todo o sistema. A tabela para todo o sistema contém informações independentes de processo, como local do arquivo no disco, datas de acesso e tamanho de arquivo. Quando um arquivo tiver sido aberto por um processo, a tabela para todo o sistema incluirá uma entrada para o arquivo. Quando outro processo executar uma chamada `open()`, uma nova entrada será acrescentada à tabela de arquivos abertos do processo, apontando para a entrada apropriada na tabela para todo o sistema. Normalmente, a tabela de arquivos abertos também possui um *contador de abertura* associado a cada arquivo, para indicar quantos processos abriram o arquivo. Cada `close()` diminui esse *contador de abertura*, e quando o *contador de abertura* chegar a zero o arquivo não estará mais em uso, e a entrada do arquivo será removida da tabela de arquivos abertos.

Resumindo, várias informações estão associadas a cada arquivo aberto.

■ **Ponteiro de arquivo.** Em sistemas que não incluem um deslocamento de arquivo como parte das chamadas de sistema `read()` e `write()`, o sistema precisa rastrear o último local de leitura-escrita como um ponteiro para a posição atual no arquivo. Esse ponteiro é exclusivo a cada processo atuando sobre o arquivo e, portanto, precisa ser mantido separado dos atributos do arquivo em disco.

■ **Contador de abertura de arquivo.** À medida que os arquivos são fechados, o sistema operacional precisa reutilizar suas entradas da tabela de arquivos abertos ou poderá ficar sem espaço na tabela. Como vários processos podem ter aberto um arquivo, o sistema precisa esperar o último arquivo se fechar antes de remover a entrada da tabela de arquivos abertos. O contador de abertura de arquivo acompanha a quantidade de aberturas e fechamentos e atinge zero no último fechamento. O sistema pode, então, remover a entrada.

■ **Local do arquivo no disco.** A maior parte das operações de arquivo exige que o sistema modifique os dados dentro do arquivo. As informações necessárias para localizar o arquivo no disco são mantidas na memória, para o sistema não precisar lê-las do disco para cada operação.

■ **Direitos de acesso.** Cada processo abre um arquivo em um modo de acesso. Essa informação é armazenada na tabela por processo, para o sistema operacional poder permitir ou negar requisições de E/S subsequentes.

Alguns sistemas operacionais fornecem facilidades para efetuar o lock de um arquivo aberto (ou seções de um arquivo). Os locks de arquivo permitem que um processo restrinja o acesso a um arquivo e impeça que outros processos tenham acesso a ele. Os locks de arquivo são úteis para arquivos compartilhados por vários processos – por exemplo, um arquivo de log do sistema que pode ser acessado e modificado por diversos processos no sistema.

Os locks de arquivo fornecem funcionalidade semelhante aos locks de leitores-escritores, discutidos na Seção 6.6.2. Um **lock compartilhado** é semelhante a um lock de leitor, em que vários processos podem obter o lock ao mesmo tempo. Um **lock exclusivo** se comporta como um lock de escritor; apenas um processo por vez pode obter esse lock. É importante observar que nem todos os sistemas operacionais proveem os dois tipos de locks; alguns sistemas só proveem o lock exclusivo ao arquivo.

Além do mais, os sistemas operacionais podem fornecer mecanismos de lock de arquivo **obrigatório** ou **consultivo**. Se um lock for obrigatório, então, quando um processo obtiver um lock exclusivo, o sistema operacional impedirá que qualquer outro processo acesse o arquivo com o lock. Por exemplo, suponha que um processo adquira um lock exclusivo sobre o arquivo `system.log`. Se tentarmos abrir `system.log` a partir de outro processo – por exemplo, um editor de textos –, o sistema operacional impedirá o acesso até o lock exclusivo ser liberado. Isso acontece mesmo que o editor de textos não seja escrito explicitamente para adquirir o lock. Como alternativa, se o lock for de consulta, o sistema operacional não impedirá que o editor de textos adquira o acesso a `system.log`. Em vez disso, o editor de textos deve ser escrito de modo que obtenha o lock manualmente antes de acessar o arquivo. Em outras palavras, se o esquema de lock for obrigatório, o sistema operacional garante a integridade do lock. Para o lock de consulta, fica a critério dos desenvolvedores do software garantir que os locks sejam obtidos e liberados de forma apropriada. Como regra, os sistemas operacionais Windows adotam o lock obrigatório, e os sistemas UNIX empregam locks de consulta.

O uso de lock de arquivo exige as mesmas precauções do sincronismo de processo comum. Por exemplo, os programadores precisam ter cuidado para manter locks de arquivo exclusivos somente enquanto estão acessando o arquivo; caso contrário, eles impedirão que outros processos também acessem o arquivo. Além do mais, algumas medidas precisam ser tomadas para garantir que dois ou mais processos não se envolvam em um deadlock enquanto tentam obter locks.

LOCK DE ARQUIVO EM JAVA

Na API Java, a aquisição de um lock primeiro exige obter o `FileChannel` para o arquivo em que

deverá ser efetuado o lock. O método `lock()` do `FileChannel` é usado para obter o lock. A API do método `lock()` é `FileLock lock(long begin, long end, boolean shared)`

Onde `begin` e `end` são as posições inicial e final da região sendo bloqueada. A definição de `shared` como `true` é para locks compartilhados; a definição de `shared` como `false` adquire o lock exclusivamente. O lock é liberado pela chamada de `release()` do `FileLock` retornado pela operação `lock()`.

O programa na [Figura 10.1](#) ilustra o lock de arquivo em Java. Esse programa obtém dois locks para o arquivo `file.txt`. A primeira metade do arquivo é obtida com um lock exclusivo; o lock para a segunda metade é um lock compartilhado.

```
import java.io.*;
import java.nio.channels.*;

public class LockingExample {
    public static final boolean EXCLUSIVE = false;
    public static final boolean SHARED = true;

    public static void main(String args[ ]) throws IOException {
        FileLock sharedLock = null;
        FileLock exclusiveLock = null;

        try {
            RandomAccessFile raf = new RandomAccessFile("file.txt", "rw");

            // apanha o canal para o arquivo
            FileChannel ch = raf.getChannel();

            // efetua o lock da primeira metade do arquivo – exclusivo
            exclusiveLock = ch.lock(0, raf.length( )/2, EXCLUSIVE);

            /** Agora modifica os dados... */

            // libera o lock
            exclusiveLock.release( );

            // efetua o lock a segunda metade do arquivo – compartilhado
            sharedLock = ch.lock(raf.length( )/2 + 1, raf.length( ), SHARED);

            /** Agora lê os dados... */

            // libera o lock
            sharedLock.release( );
        } catch (java.io.IOException ioe) {
            System.err.println(ioe);
        }
        finally {
            if (exclusiveLock != null)
                exclusiveLock.release( );
            if (sharedLock != null)
                sharedLock.release( );
        }
    }
}
```

FIGURA 10.1 Exemplo de lock de arquivo em Java.

10.1.3 Tipos de arquivo

Quando projetamos um sistema de arquivos – na realidade, um sistema operacional inteiro –, sempre consideramos se o sistema operacional deve reconhecer e aceitar os tipos de arquivo. Se um sistema operacional reconhecer o tipo de um arquivo, ele poderá, então, operar sobre o arquivo de maneiras razoáveis. Por exemplo, um erro comum ocorre quando um usuário tenta imprimir um programa na forma de um objeto binário. Essa tentativa normalmente produz lixo; porém, a tentativa pode ter sucesso se o sistema operacional tiver sido informado de que o arquivo é um programa de objeto binário.

Uma técnica comum para implementar os tipos de arquivo é incluir o tipo como parte do nome do arquivo. Desse modo, o usuário e o sistema operacional podem saber, apenas pelo nome, qual o tipo de cada arquivo. O nome é dividido em duas partes – um nome e uma *extensão* ([Figura 10.2](#)). Por

exemplo, a maioria dos sistemas operacionais permite que os usuários especifiquem o nome de arquivo como uma sequência de caracteres seguidos por um ponto e terminados por uma extensão de caracteres extras. Exemplos de nome de arquivo incluem *resume.doc*, *Server.java* e *ReaderThread.c*.

tipo de arquivo	extensão normal	função
executável	exe, com, bin ou nenhuma	programa em linguagem de máquina pronto para ser executado
objeto	obj, o	programa compilado, em linguagem de máquina, não vinculado
código-fonte	c, cc, Java, pas, asm, a	código-fonte nas diversas linguagens
lote	bat, sh	comandos ao interpretador de comandos
texto	txt, doc	dados textuais, documentos
processador de textos	wp, tex, rtf, doc	diversos formatos de processador de textos
biblioteca	lib, a, so, dll	bibliotecas de rotinas para programadores
impressão ou exibição	ps, pdf, jpg	arquivo ASCII ou binário em um formato para impressão ou exibição
arquivamento	arc, zip, tar	arquivos relacionados, agrupados em um arquivo, às vezes compactado, para arquivamento ou armazenamento
multimídia	mpeg, mov, rm, mp3, avi	arquivo binário contendo informações de áudio e/ou vídeo

FIGURA 10.2 Tipos de arquivo comuns.

O sistema utiliza a extensão para indicar o tipo do arquivo e o tipo das operações que podem ser feitas sobre esse arquivo. Somente um arquivo com a extensão *.com*, *.exe* ou *.bat* pode ser *executado*, por exemplo. Os arquivos *.com* e *.exe* são duas formas de arquivos executáveis binários, enquanto um arquivo *.bat* é um **arquivo batch** contendo, em formato ASCII, comandos para o sistema operacional. O MS-DOS reconhece somente algumas extensões, mas os programas de aplicação também utilizam extensões para indicar os tipos de arquivo em que estão interessados. Por exemplo, os assemblers esperam que os arquivos-fonte tenham uma extensão *.asm*, e o processador de textos Microsoft Word espera que seu arquivo termine com uma extensão *.doc*. Essas extensões não são obrigatórias, de modo que um usuário pode especificar um arquivo sem a extensão (para poupar digitação), e a aplicação procura um arquivo com o nome indicado e com a extensão esperada. Como essas extensões não são definidas pelo sistema operacional, elas podem ser consideradas “dicas” para as aplicações que operam sobre elas.

Outro exemplo da utilidade dos tipos de arquivo vem do sistema operacional TOPS-20. Se o usuário tentar executar um programa objeto cujo arquivo-fonte tiver sido modificado (ou editado) desde que o arquivo objeto foi produzido, o arquivo-fonte será recompilado automaticamente. Essa função garante que o usuário sempre executará um arquivo objeto atualizado. Caso contrário, o usuário poderá desperdiçar um tempo significativo executando o arquivo objeto antigo. Para essa função ser possível, o sistema operacional precisa ser capaz de diferenciar o arquivo-fonte do arquivo objeto, verificar a hora em que cada arquivo foi criado ou modificado pela última vez e determinar a linguagem do programa-fonte (a fim de usar o compilador correto).

Considere o sistema operacional do Mac OS X. Nesse sistema, cada arquivo possui um tipo, como *TEXT* (para o arquivo de texto) ou *APPL* (para aplicações). Cada arquivo também possui um atributo de criador, contendo o nome do programa que o criou. Esse atributo é definido pelo sistema operacional durante a chamada a *create()*, de modo que seu uso é imposto e admitido pelo sistema. Por exemplo, um arquivo produzido por um processador de textos tem o nome do processador de textos como seu criador. Quando o usuário abre esse arquivo, clicando duas vezes com o mouse no ícone que representa o arquivo, o processador de textos é chamado automaticamente, e o arquivo é carregado, pronto para ser editado.

O sistema UNIX usa um **número mágico** primitivo, armazenado no início de alguns arquivos, para indicar aproximadamente o tipo do arquivo – programa executável, arquivo batch (ou **shell script**), arquivo PostScript, e assim por diante. Nem todos os arquivos possuem números mágicos, de modo que os recursos do sistema não podem ser baseados unicamente nesse tipo de informação. O UNIX também não registra o nome do programa criador. O UNIX permite dicas de extensão do nome de arquivo, mas essas extensões não são impostas nem dependem do sistema operacional; elas servem principalmente para auxiliar os usuários a determinar o tipo do conteúdo do arquivo. As extensões podem ser usadas ou ignoradas por determinada aplicação, mas isso fica a critério do programador da aplicação.

10.1.4 Estrutura do arquivo

Os tipos de arquivo também podem ser usados para indicar a estrutura interna do arquivo. Como dissemos na [Seção 10.1.3](#), os arquivos-fonte e objeto possuem estruturas que correspondem às expectativas dos programas que os leem. Além do mais, certos arquivos precisam estar em conformidade com uma estrutura exigida, entendida pelo sistema operacional. Por exemplo, o sistema operacional exige que um arquivo executável tenha uma estrutura específica, para poder determinar onde o arquivo será carregado na memória e qual é o local da sua primeira instrução. Alguns sistemas operacionais estendem essa ideia para um conjunto de estruturas de arquivo aceitas pelo sistema, com conjuntos de operações especiais para manipular arquivos com essas estruturas. Por exemplo, o sistema operacional VMS dos computadores DEC possui um sistema de arquivos que aceita três estruturas de arquivo definidas.

Esse ponto nos leva a uma das desvantagens de ter um sistema operacional com suporte para múltiplas estruturas de arquivo: o sistema operacional resultante fica desajeitado. Se o sistema operacional definir cinco estruturas de arquivo diferentes, ele precisará conter o código para dar suporte a essas cinco estruturas. Além disso, cada arquivo pode precisar ser definido como um dos tipos de arquivo aceitos pelo sistema operacional. Quando novas aplicações exigirem informações estruturadas de maneiras não aceitas pelo sistema operacional, poderá haver muitos problemas.

Por exemplo, suponha que um sistema aceite dois tipos de arquivos: arquivos de texto (compostos de caracteres ASCII separados por um par carriage return/line feed) e arquivos binários executáveis. Agora, se nós (como usuários) quisermos definir um arquivo criptografado para proteger nosso conteúdo contra leitura por pessoas não autorizadas, veremos que nenhum dos dois tipos é apropriado. O arquivo criptografado não é composto de linhas de texto ASCII, mas sim de bits (aparentemente) aleatórios. Embora podendo parecer ser um arquivo binário, ele não é executável. Como resultado, teremos de evitar ou usar de forma errada o mecanismo de tipos de arquivo do sistema operacional ou, então, abandonar nosso esquema de criptografia.

Alguns sistemas operacionais impõem (e admitem) uma quantidade mínima de estruturas de arquivo. Essa técnica foi adotada no UNIX, MS-DOS e outros. O UNIX considera cada arquivo uma sequência de bytes de 8 bits; nenhuma interpretação desses bits é feita pelo sistema operacional. Esse esquema provê o máximo de flexibilidade, mas pouco suporte. Cada programa de aplicação precisa incluir seu próprio código para interpretar um arquivo de entrada conforme a estrutura apropriada. Entretanto, todos os sistemas operacionais precisam aceitar pelo menos uma estrutura - a de um arquivo executável -, para que o sistema consiga carregar e executar programas.

O sistema operacional do Macintosh também aceita uma quantidade mínima de estruturas de arquivo. Ele espera que os arquivos contenham duas partes: um **fork de recursos** e um **fork de dados**. O fork de recursos contém informações de interesse do usuário. Por exemplo, ele mantém os nomes que aparecem em quaisquer botões exibidos pelo programa. Um usuário que utiliza outro idioma pode modificar esses botões para o seu idioma, e o sistema operacional do Macintosh contém ferramentas para permitir a modificação dos dados no fork de recursos. O fork de dados contém código de programa ou dados - o conteúdo tradicional do arquivo. Para conseguir a mesma tarefa em um sistema UNIX ou MS-DOS, o programador precisará modificar e recompilar o código-fonte, a menos que crie seu próprio arquivo de dados modificável pelo usuário. Logicamente, é útil que um sistema operacional aceite estruturas que serão usadas com frequência e que evitarião um esforço substancial da parte do programador. Muito poucas estruturas tornam a programação inconveniente, enquanto muitas estruturas aumentam o sistema operacional e a confusão para o programador.

10.1.5 Estrutura interna do arquivo

Internamente, localizar um deslocamento dentro de um arquivo pode ser complicado para o sistema operacional. Os sistemas de disco possuem um tamanho de bloco bem definido, determinado pelo tamanho de um setor. Toda a E/S de disco é realizada em unidades de um bloco (registro físico), e todos os blocos têm o mesmo tamanho. É pouco provável que o tamanho do registro físico combine exatamente com o tamanho do registro lógico desejado. Os registros lógicos podem até mesmo variar no tamanho. **Empacotar** diversos registros lógicos nos blocos físicos é uma solução comum para esse problema.

Por exemplo, o sistema operacional UNIX define todos os arquivos para serem fluxos de bytes. Cada byte é endereçável individualmente por seu deslocamento do início (ou fim) do arquivo. Nesse caso, o registro lógico é de 1 byte. O sistema de arquivos empacota e desempacota os bytes automaticamente em blocos físicos do disco - digamos, 512 bytes por bloco -, conforme a necessidade.

O tamanho do registro lógico, o tamanho do bloco físico e a técnica de empacotamento determinam quantos registros lógicos existem em cada bloco físico. O empacotamento pode ser feito pelo programa de aplicação do usuário ou pelo sistema operacional. De qualquer forma, o arquivo pode ser considerado uma sequência de blocos. Todas as funções básicas de E/S operam em termos de blocos. A conversão de registros lógicos para blocos físicos é um problema de software

relativamente simples.

Como o espaço em disco sempre é alocado em blocos, alguma parte do último bloco de cada arquivo em geral é desperdiçada. Se cada bloco tivesse 512 bytes, por exemplo, então um arquivo com 1.949 bytes receberia quatro blocos (2.048 bytes); os últimos 99 bytes seriam desperdiçados. O desperdício que acontece por mantermos tudo em unidades de blocos (em vez de bytes) é a **fragmentação interna**. Todos os sistemas de arquivos sofrem com a fragmentação interna; quanto maior o tamanho do bloco, maior a fragmentação interna.

10.2 Métodos de acesso

Os arquivos armazenam informações. Quando usadas, essas informações precisam ser acessadas e lidas para a memória do computador. As informações no arquivo podem ser acessadas de várias maneiras. Alguns sistemas oferecem apenas um método de acesso para os arquivos. Outros sistemas, como os da IBM, admitem muitos métodos de acesso, e a escolha do método correto para determinada aplicação é uma questão de projeto importante.

10.2.1 Acesso sequencial

O método de acesso mais simples é o **acesso sequencial**. As informações no arquivo são processadas em ordem, um registro após o outro. Esse modo de acesso é, de longe, o mais comum; por exemplo, os editores e compiladores normalmente acessam os arquivos dessa maneira.

As leituras e escritas compõem a maior parte das operações sobre um arquivo. Uma operação de leitura - *leia próximo* - lê a próxima parte do arquivo e avança um ponteiro de arquivo, que acompanha o local da E/S. De modo semelhante, uma operação de escrita - *escreva próximo* - acrescenta ao final do arquivo e avança para o final do material que acabou de ser escrito (o novo final do arquivo). Esse arquivo pode ser retornado ao início e, em alguns sistemas, um programa pode ser capaz de pular n registros para frente ou para trás, para algum n inteiro - talvez somente para $n = 1$. O acesso sequencial, representado na [Figura 10.3](#), é baseado em um modelo de fita de um arquivo, e funciona tanto em dispositivos de acesso sequencial quanto de acesso aleatório.

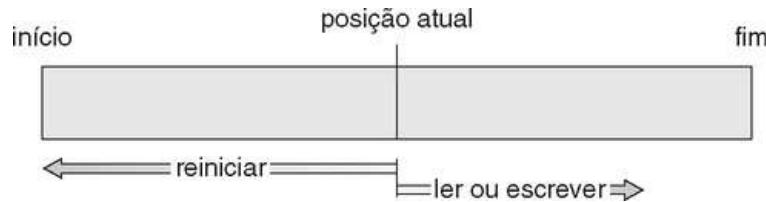


FIGURA 10.3 Arquivo de acesso sequencial.

10.2.2 Acesso direto

Outro método é o **acesso direto** (ou **acesso relativo**). Um arquivo é composto de **registros lógicos** de tamanho fixo, permitindo que os programas leiam ou escrevam registros rapidamente sem qualquer ordem específica. O método de acesso direto é baseado em um modelo de disco de um arquivo, pois os discos permitem o acesso aleatório a qualquer bloco do arquivo. Para o acesso direto, o arquivo é visto como uma sequência numerada de blocos ou registros. Assim, podemos ler o bloco 14, depois ler o bloco 53 e depois escrever no bloco 7. Não existem restrições sobre a ordem de leitura ou escrita para um arquivo de acesso direto.

Os arquivos de acesso direto são muito utilizados para o acesso imediato a grandes quantidades de informação. Os bancos de dados normalmente são desse tipo. Quando chega uma consulta referente a determinado assunto, calculamos qual bloco contém a resposta e depois lemos esse bloco diretamente, para prover a informação desejada.

Como um exemplo simples, em um sistema de reservas de passagens aéreas, poderíamos armazenar todas as informações sobre determinado voo (por exemplo, o voo 713) no bloco identificado pelo número do voo. Assim, a quantidade de assentos disponíveis para o voo 713 está armazenada no bloco 713 do arquivo de reserva. Para armazenar informações sobre um conjunto maior, como pessoas, poderíamos calcular uma função de hash sobre os nomes das pessoas ou pesquisar um pequeno índice na memória para determinar um bloco para ler e consultar.

Para o método de acesso direto, as operações de arquivo precisam ser modificadas para incluir o número do bloco como um parâmetro. Assim, temos *leia n*, onde n é o número do bloco, em vez de *leia próximo*, e *escreva n*, em vez de *escreva próximo*. Uma técnica alternativa é manter *leia próximo* e *escreva próximo*, como no acesso sequencial, e acrescentar uma operação *posicione arquivo para n*, onde n é o número do bloco. Depois, para efetuar um *leia n*, usariamos *posicione para n* e depois *leia próximo*.

O número de bloco fornecido pelo usuário ao sistema operacional normalmente é um **número de bloco relativo**. Um número de bloco relativo é um índice relativo ao início do arquivo. Assim, o primeiro bloco relativo do arquivo é 0, o próximo é 1, e assim por diante, embora o endereço de disco absoluto real do bloco possa ser 14703 para o primeiro bloco e 3192 para o segundo. O uso de números de bloco relativos permite que o sistema operacional decida onde o arquivo deve ser colocado (chamado *problema de alocação*, conforme discutimos no [Capítulo 11](#)) e ajuda a evitar que o usuário acesse partes do sistema de arquivos que podem não fazer parte do seu arquivo. Alguns

sistemas iniciam seus números de bloco relativos em 0; outros iniciam em 1.

Como o sistema satisfaz a requisição para o registro N ? Supondo que temos um tamanho de registro lógico L , uma requisição para o registro N é transformada em uma requisição de E/S para L bytes a partir do local $L * (N)$ dentro do arquivo (supondo que o primeiro registro seja $N = 0$). Como os registros lógicos são de um tamanho fixo, também é fácil ler, escrever ou excluir um registro.

Nem todos os sistemas operacionais admitem o acesso sequencial e direto para arquivos. Alguns sistemas só permitem o acesso sequencial ao arquivo; outros só permitem o acesso direto. Alguns sistemas exigem que um arquivo seja definido como sequencial ou direto quando é criado; tal arquivo só pode ser acessado de maneira coerente com sua declaração. Podemos simular o acesso sequencial em um arquivo de acesso direto mantendo uma variável cp que define nossa posição atual, como mostra a [Figura 10.4](#). Entretanto, a simulação de um arquivo de acesso direto em um arquivo de acesso sequencial é bastante ineficaz e desajeitada.

acesso sequencial	implementação para acesso direto
<i>reset</i>	$cp = 0;$
<i>read next</i>	$read cp;$ $cp = cp + 1;$
<i>write next</i>	$write cp;$ $cp = cp + 1;$

FIGURA 10.4 Simulação do acesso sequencial em um arquivo de acesso direto.

10.2.3 Outros métodos de acesso

Outros métodos de acesso podem ser montados em cima de um método de acesso direto. Esses métodos em geral envolvem a construção de um **índice** para o arquivo. O índice, como um índice no final de um livro, contém ponteiros para os diversos blocos. Para encontrar um registro no arquivo, primeiro pesquisamos o índice e depois usamos o ponteiro para acessar o arquivo diretamente e encontrar o registro desejado.

Por exemplo, um arquivo de preços de revenda poderia listar os códigos universais de produto (UPCs) para os itens, com os preços associados. Cada registro consiste em um UPC de 10 dígitos e um preço de 6 dígitos, formando um registro de 16 bytes. Se o nosso disco possui 1.024 bytes por bloco, podemos armazenar 64 registros por bloco. Um arquivo de 120.000 registros ocuparia cerca de 2.000 blocos (2 milhões de bytes). Mantendo o arquivo classificado por UPC, podemos definir um índice consistindo no primeiro UPC em cada bloco. Esse índice teria 2.000 entradas de 10 dígitos cada, ou 20.000 bytes, e poderia ser mantido na memória. Para encontrar o preço de determinado item, podemos fazer uma pesquisa binária do índice. Nessa pesquisa, descobrimos qual bloco contém o registro desejado e acessamos esse bloco. Essa estrutura nos permite consultar um arquivo grande realizando pouca E/S.

Com arquivos grandes, o próprio arquivo de índice pode se tornar muito grande para ser mantido na memória. Uma solução é criar um índice para o arquivo de índice. O arquivo de índice primário teria ponteiros para os arquivos de índice secundários, que apontariam para os itens de dados reais.

Por exemplo, o método de acesso sequencial indexado (ISAM) da IBM utiliza um pequeno índice mestre que aponta para os blocos de disco de um índice secundário. Os blocos do índice secundário apontam para os blocos de arquivo reais. O arquivo é mantido classificado sobre uma chave definida. Para encontrar determinado item, primeiro fazemos uma busca binária do índice mestre, que provê o número de bloco do índice secundário. Esse bloco é lido e novamente uma pesquisa binária é usada para encontrar o bloco contendo o registro desejado. Por fim, esse bloco é pesquisado sequencialmente. Desse modo, qualquer registro pode ser localizado de sua chave por, no máximo, duas leituras de acesso direto. A [Figura 10.5](#) mostra uma situação semelhante, implementada em arquivos de índice e relativos no VMS.

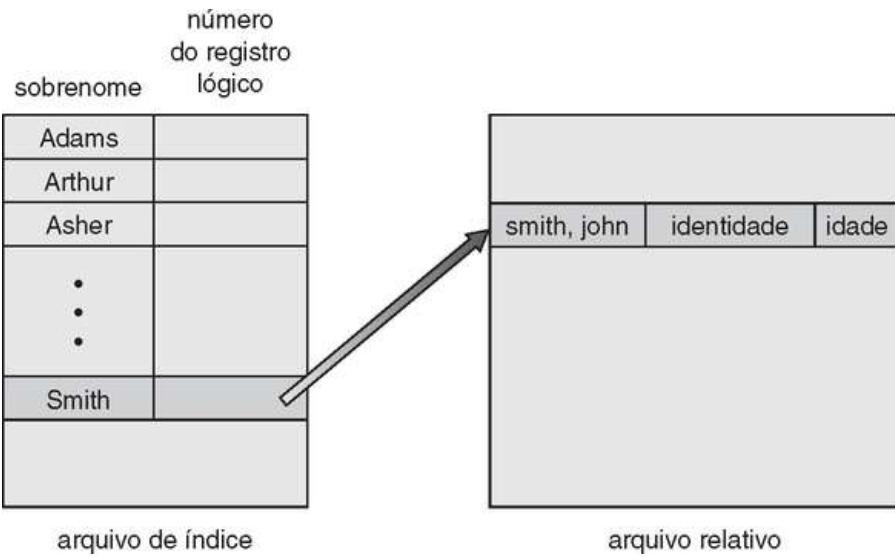


FIGURA 10.5 Exemplo de arquivos de índice e relativo.

10.3 Estrutura de diretório e disco

Em seguida, veremos como armazenar arquivos. Certamente, nenhum sistema operacional armazena apenas um arquivo. Geralmente, existem milhares, milhões e até mesmo bilhões de arquivos dentro de um computador. Os arquivos são armazenados em dispositivos de armazenamento de acesso aleatório, como discos rígidos, discos óticos e discos em estado sólido (baseados em memória).

Um dispositivo de armazenamento pode ser usado em sua totalidade para um sistema de arquivos. Ele também pode ser subdividido para que haja um controle mais minucioso. Por exemplo, um disco pode ser **particionado** em quatro, e cada quarta parte pode manter um sistema de arquivos. Os dispositivos de armazenamento também podem ser reunidos em volumes RAID, que oferecem proteção contra falha de um único disco (conforme descrevemos na [Seção 12.7](#)). Às vezes, os discos são subdivididos e também coletados em volumes RAID.

O particionamento é útil para limitar os tamanhos dos sistemas de arquivos individuais, colocando-se vários tipos de sistemas de arquivos em um dispositivo ou deixando parte do dispositivo disponível para outras coisas, como um espaço de swap ou espaço em disco não formatado (**raw**). Essas partes são conhecidas como **partições**, **slices** ou (no mundo IBM) **minidiscos**. Um sistema de arquivos pode ser criado em cada uma dessas partes do disco. Uma entidade contendo um sistema de arquivos geralmente é conhecida como **volume**. O volume pode ser um subconjunto de um dispositivo, um dispositivo inteiro ou vários dispositivos ligados em um conjunto RAID. Cada volume pode ser considerado um disco virtual. Os volumes também podem armazenar vários sistemas operacionais, permitindo a um sistema inicializar e executar mais de um deles.

Cada volume que contém um sistema de arquivos também deve conter informações sobre os arquivos dentro dele. Essas informações são mantidas em entradas no **diretório do dispositivo** ou **sumário do volume**. O **diretório** do dispositivo (mais conhecido como diretório) registra informações – como nome, local, tamanho e tipo – para todos os arquivos nesse volume. A [Figura 10.6](#) mostra a organização típica do sistema de arquivos.

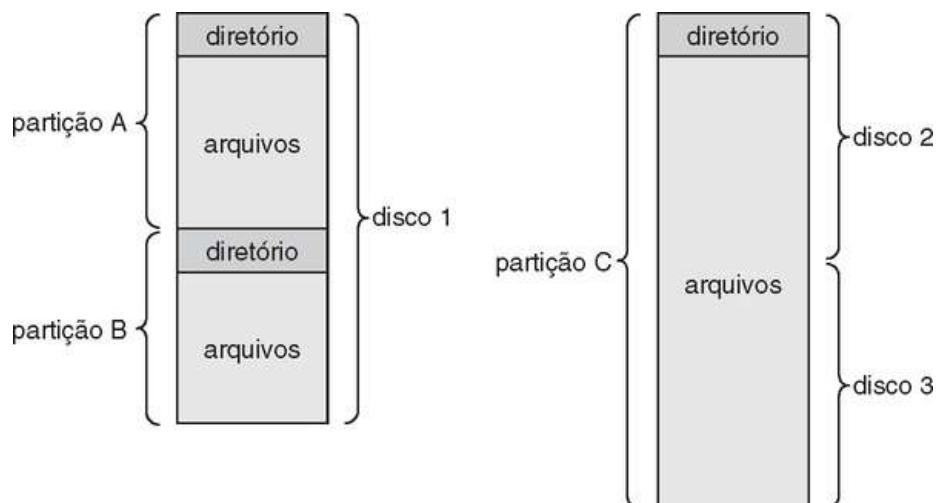


FIGURA 10.6 Organização típica do sistema de arquivos.

10.3.1 Estrutura de armazenamento

Até este ponto, vimos que um sistema de computação de uso geral possui vários dispositivos de armazenamento, e esses dispositivos podem ser divididos em volumes que mantêm sistemas de arquivos. Os sistemas de computação podem ter zero ou mais sistemas de arquivos, e os sistemas de arquivos podem ser de tipos variados. Por exemplo, um sistema Solaris típico pode ter dezenas de sistemas de arquivos de dezenas de tipos diferentes, como mostra a lista do sistema de arquivos da [Figura 10.7](#).

/	ufs
/devices	devfs
/dev	dev
/system/contract	ctfs
/proc	proc
/etc/mnttab	mntfs
/etc/svc/volatile	tmpfs
/system/object	objfs
/lib/libc.so.1	lofs
/dev/fd	fd
/var	ufs
/tmp	tmpfs
/var/run	tmpfs
/opt	ufs
/zpbge	zfs
/zpbge/backup	zfs
/export/home	zfs
/var/mail	zfs
/var/spool/mqueue	zfs
/zpbg	zfs
/zpbg/zones	zfs

FIGURA 10.7 Sistema de arquivos do Solaris.

Neste livro, consideramos apenas os sistemas de arquivos de uso geral. No entanto, vale a pena observar que existem muitos sistemas de arquivos de uso especial. Considere os tipos de sistemas de arquivos no exemplo do Solaris mencionado anteriormente:

- **tmpfs** - um sistema de arquivos “temporário”, que é criado na memória principal volátil e tem seu conteúdo apagado se o sistema for reinicializado ou se falhar.
- **objfs** - um sistema de arquivos “virtual” (basicamente, uma interface para o kernel, semelhante a um sistema de arquivos) que oferece acesso aos símbolos do kernel para os depuradores.
- **ctfs** - um sistema de arquivos virtual que mantém informações de “contrato” para gerenciar quais processos começam quando o sistema é inicializado e devem continuar a ser executados durante a operação.
- **lofs** - um sistema de arquivos de “loop back”, que permite que um sistema de arquivos seja acessado no lugar de outro.
- **procfs** - um sistema de arquivos virtual, que apresenta informações sobre todos os processos como um sistema de arquivos.
- **ufs, zfs** - sistemas de arquivos de uso geral.

Logo, os sistemas de arquivos dos computadores podem ser extensos. Mesmo dentro de um sistema de arquivos, é útil segregar os arquivos em grupos e gerenciar e atuar sobre esses grupos. Essa organização envolve o uso de diretórios. No restante desta seção, exploramos o tópico de estrutura de diretório.

10.3.2 Conceitos básicos do diretório

O diretório pode ser visto como uma tabela de símbolos, que traduz nomes de arquivo nas entradas de diretório. Se considerarmos essa visão, podemos ver que o próprio diretório pode ser organizado de várias maneiras. Queremos ser capazes de inserir entradas, excluir entradas, procurar uma entrada nomeada e listar todas as entradas no diretório. Nesta seção, examinamos vários esquemas para definir a estrutura lógica do sistema de diretórios.

Ao considerar uma estrutura de diretório em particular, precisamos ter em mente as operações que devem ser realizadas sobre um diretório:

- **Procurar um arquivo.** Precisamos ser capazes de pesquisar uma estrutura de diretório para procurar a entrada para determinado arquivo. Como os arquivos possuem nomes simbólicos, e nomes semelhantes podem indicar um relacionamento entre os arquivos, também podemos querer encontrar todos os arquivos cujos nomes combinem com determinado padrão.

- **Criar um arquivo.** Novos arquivos precisam ser criados e acrescentados ao diretório.
- **Excluir um arquivo.** Quando um arquivo não é mais necessário, queremos ser capazes de removê-lo do diretório.
- **Listar um diretório.** Precisamos ser capazes de listar os arquivos em um diretório e o conteúdo da entrada de diretório para cada arquivo na lista.
- **Renomear um arquivo.** Como o nome de um arquivo representa seu conteúdo para os usuários, precisamos ser capazes de alterar o nome quando o conteúdo ou o uso do arquivo mudar. A troca do nome de um arquivo também pode permitir a mudança da sua posição dentro da estrutura do diretório.
- **Atravessar o sistema de arquivos.** Podemos querer acessar cada diretório e cada arquivo dentro de uma estrutura de diretórios. Por confiabilidade, é uma boa ideia salvar o conteúdo e a estrutura do sistema de arquivos inteiro em intervalos regulares. Normalmente, fazemos isso copiando todos os arquivos para fita magnética. Essa técnica provê uma cópia de backup para o caso de uma falha no sistema. Além disso, se um arquivo não estiver mais em uso, ele pode ser copiado para fita e o espaço em disco desse arquivo liberado para reutilização por outro arquivo. Nas seções seguintes, descrevemos os esquemas mais comuns para a definição da estrutura lógica de um diretório.

10.3.3 Diretório de único nível

A estrutura de diretório mais simples é a estrutura de único nível. Todos os arquivos estão contidos no mesmo diretório, o que é fácil de manter e entender ([Figura 10.8](#)).

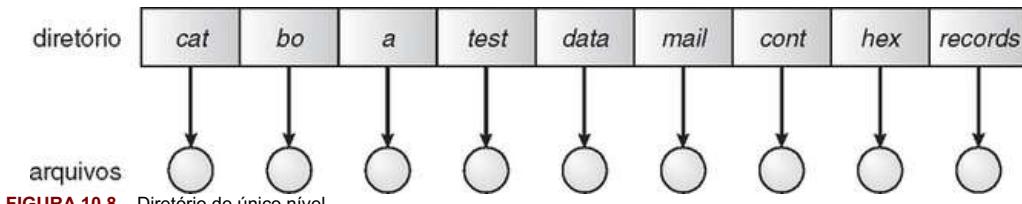


FIGURA 10.8 Diretório de único nível.

No entanto, um diretório de único nível possui limitações significativas quando o número de arquivos aumenta ou quando o sistema possui mais de um usuário. Como todos os arquivos estão no mesmo diretório, eles precisam ter nomes exclusivos. Se dois usuários chamarem seu arquivo de dados de *teste*, então a regra do nome exclusivo será violada. Por exemplo, em uma aula de programação, 23 alunos chamaram sua segunda tarefa de programação de *prog2*; outros 11 a chamaram de *tarefa2*. Embora os nomes de arquivo sejam selecionados para refletir o conteúdo do arquivo, eles costumam ser limitados em tamanho, complicando a tarefa de tornar os nomes de arquivo exclusivos. O sistema operacional MS-DOS permite nomes de arquivo com apenas 11 caracteres; o UNIX permite 255 caracteres.

Até mesmo um único usuário em um diretório com nível único poderá achar difícil se lembrar dos nomes de todos os arquivos quando a quantidade de arquivos aumentar. Não é raro que um usuário tenha centenas de arquivos em um computador e uma quantidade igual de arquivos adicionais em outro. Lembrar tantos nomes de arquivos é uma tarefa assustadora.

10.3.4 Diretório de dois níveis

Como vimos, um diretório de único nível normalmente ocasiona confusão de nomes de arquivo entre diferentes usuários. A solução-padrão é criar um diretório *separado* para cada usuário.

Na estrutura de diretório de dois níveis, cada usuário possui seu próprio **diretório de arquivos do usuário (User File Directory - UFD)**. Os UFDs possuem estruturas semelhantes, mas cada um lista apenas os arquivos de um único usuário. Quando a tarefa de um usuário é iniciada ou quando um usuário efetua o login, o **diretório de arquivos mestre (Master File Directory - MFD)** é pesquisado. O MFD é indexado por nome de usuário ou número de conta, e cada entrada aponta para o UFD desse usuário ([Figura 10.9](#)).

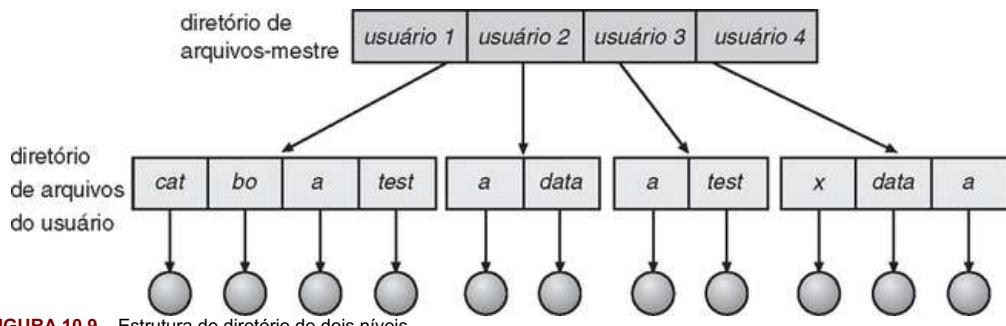


FIGURA 10.9 Estrutura de diretório de dois níveis.

Quando um usuário se refere a determinado arquivo, somente seu próprio UFD é pesquisado. Assim, diferentes usuários podem ter arquivos com o mesmo nome, desde que todos os nomes de arquivo dentro de cada UFD sejam exclusivos. Para criar um arquivo para um usuário, o sistema operacional pesquisa apenas o UFD desse usuário para verificar se existe outro arquivo com esse nome. Para excluir um arquivo, o sistema operacional confina sua busca ao UFD local; assim, ele não pode excluir acidentalmente o arquivo de outro usuário, com o mesmo nome.

Os próprios diretórios do usuário precisam ser criados e excluídos conforme a necessidade. Um programa especial do sistema é executado com as informações apropriadas de nome de usuário e conta. O programa cria um novo UFD e acrescenta uma entrada no MFD. A execução desse programa poderia ser restrita a administradores do sistema. A alocação do espaço em disco para os diretórios do usuário pode ser tratada com as técnicas discutidas no [Capítulo 11](#) para os próprios arquivos.

Embora a estrutura de diretório de dois níveis solucione o problema de colisão de nomes, ela ainda possui desvantagens. Essa estrutura isola um usuário do outro. O isolamento é uma vantagem quando os usuários são independentes, mas é uma desvantagem quando os usuários *querem* cooperar em alguma tarefa e acessar os arquivos uns dos outros. Alguns sistemas não permitem que os arquivos do usuário local sejam acessados por outros usuários.

Se o acesso tiver de ser permitido, um usuário precisa ter a capacidade de citar um arquivo no diretório de outro usuário. Para citar um arquivo específico em um diretório de dois níveis, temos de dar o nome do usuário e o nome do arquivo. Um diretório de dois níveis pode ser imaginado como uma árvore, ou uma árvore invertida, de altura 2. A raiz da árvore é o MFD. Seus descendentes diretos são os UFDs. Os descendentes dos UFDs são os próprios arquivos. Os arquivos são as folhas da árvore. A especificação de um nome de usuário e um nome de arquivo define um caminho na árvore desde a raiz (o MFD) até a folha (o arquivo especificado). Assim, um nome de usuário e um nome de arquivo definem um *nome de caminho*. Cada arquivo no sistema possui um nome de caminho. Para citar apenas um arquivo, um usuário precisa saber o nome de caminho do arquivo desejado.

Por exemplo, se o usuário A deseja acessar seu próprio arquivo de teste, chamado *teste*, ele pode se referir a *teste*. Todavia, para acessar o arquivo chamado *teste* do usuário B (com o nome da entrada de diretório *userb*), ele poderá se referir a */userb/teste*. Cada sistema possui sua própria sintaxe para citar os arquivos nos diretórios diferentes do próprio diretório do usuário.

Uma sintaxe adicional é necessária para especificar o volume de um arquivo. Por exemplo, no MS-DOS, um volume é especificado por uma letra seguida pelo sinal dois-pontos. Assim, uma especificação de arquivo poderia ser *C:\userb\teste*. Alguns sistemas vão ainda mais longe e separam as partes do volume, do nome do diretório e do nome do arquivo da especificação. Por exemplo, no VMS, o arquivo *login.com* poderia ser especificado como *u:[sst.jdeck]login.com;1*, onde *u* é o nome do volume, *sst* é o nome do diretório, *jdeck* é o nome do subdiretório e *1* é o número da versão. Outros sistemas tratam o nome do volume como parte do nome do diretório. O primeiro nome dado é o do volume, e o restante é o diretório e o arquivo. Por exemplo, */u/pbg/teste* poderia especificar o volume *u*, diretório *pbg* e arquivo *teste*.

Um caso especial dessa situação ocorre com os sistemas de arquivo. Os programas fornecidos como parte do sistema - loaders, assemblers, compiladores, rotinas utilitárias, bibliotecas, e assim por diante - em geral são definidos como arquivos. Quando os comandos apropriados são dados ao sistema operacional, esses arquivos são lidos pelo loader e executados. Muitos interpretadores de comandos tratam o comando como o nome de um arquivo para carregar e executar. Conforme o sistema de diretório está definido atualmente, esse nome de arquivo é pesquisado no UFD atual. Uma solução seria copiar os arquivos do sistema para cada UFD. Entretanto, a cópia de todos os arquivos do sistema desperdiçaria um espaço enorme. (Se os arquivos do sistema exigem 5 MB, então o suporte para 12 usuários exigiria $5 \times 12 = 60$ MB só para as cópias dos arquivos do sistema.)

A solução-padrão é complicar um pouco o procedimento de busca. Um diretório de usuário especial é definido para conter os arquivos do sistema (por exemplo, o usuário 0). Sempre que um nome de arquivo é indicado para ser carregado, o sistema operacional primeiro procura no UFD

local. Se o arquivo for encontrado, ele é usado. Se não, o sistema procura automaticamente no diretório do usuário especial, que contém os arquivos do sistema. A sequência de diretórios pesquisados quando um arquivo é citado é chamada de **caminho de busca**. O caminho de busca pode ser estendido para conter uma lista ilimitada de diretórios para pesquisar quando um nome de comando é indicado. Esse método é o mais utilizado no UNIX e no MS-DOS. Os sistemas também podem ser projetados de modo que cada usuário tenha seu próprio caminho de busca.

10.3.5 Diretórios estruturados em árvore

Depois de termos visto como visualizar um diretório de dois níveis como uma árvore de dois níveis, a generalização natural é estender a estrutura de diretórios para uma árvore sem uma altura determinada (Figura 10.10). Essa generalização permite aos usuários criarem seus próprios subdiretórios e organizarem seus arquivos de modo conveniente. Uma árvore é a estrutura de diretório mais comum. A árvore possui um diretório raiz, e cada arquivo no sistema possui um nome de caminho exclusivo.

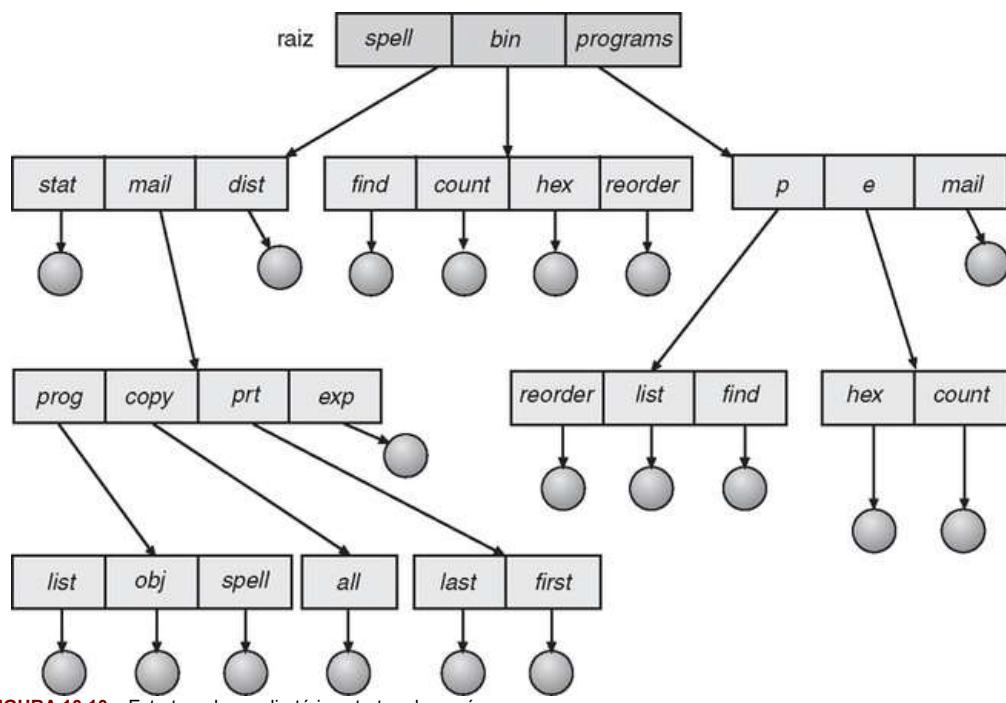


FIGURA 10.10 Estrutura de um diretório estruturado em árvore.

Um diretório (ou subdiretório) contém um conjunto de arquivos ou subdiretórios. Um diretório é outro arquivo, mas é tratado de uma forma especial. Todos os diretórios possuem o mesmo formato interno. Um bit em cada entrada de diretório define a entrada como um arquivo (0) ou como um subdiretório (1). Chamadas de sistema especiais são usadas para criar e excluir diretórios.

Em uso normal, cada usuário possui um **diretório atual**. O diretório atual deverá conter a maioria dos arquivos do interesse atual do processo. Quando é feita uma referência a um arquivo, o diretório atual é pesquisado. Se um arquivo for necessário e não estiver no diretório atual, o usuário precisa especificar um nome de caminho ou mudar o diretório atual para ser o diretório com esse arquivo. Para mudar de diretório, uma chamada de sistema é utilizada; ela apanha um nome de diretório como parâmetro e o utiliza para redefinir o diretório atual. Assim, o usuário pode mudar seu diretório atual sempre que desejar. De uma chamada de sistema para mudança de diretório para outra, todas as chamadas de sistema pesquisam o diretório atual em busca do arquivo especificado. Observe que o caminho de busca pode ou não conter uma entrada especial que significa "diretório atual".

O diretório atual inicial do shell login de um usuário é designado quando a tarefa do usuário é iniciada ou quando o usuário efetua o login. O sistema operacional pesquisa o arquivo de contabilidade (ou algum outro local predefinido) para encontrar uma entrada para esse usuário (para fins de contabilidade). No arquivo de contabilidade, existe um ponteiro para o diretório (ou o nome do diretório) inicial do usuário. Esse ponteiro é copiado para uma variável local para esse usuário, que especifica o diretório atual inicial do usuário. A partir desse shell, outros processos podem ser gerados. O diretório atual de qualquer subprocesso normalmente é o diretório atual do pai quando ele foi gerado.

Os nomes de caminho podem ser de dois tipos: *absolutos* ou *relativos*. Um **nome de caminho**

absoluto começa na raiz e segue um caminho até o arquivo especificado, incluindo os nomes de diretório no caminho. Um **nome de caminho relativo** define um caminho a partir do diretório atual. Por exemplo, no sistema de arquivos estruturado em árvore da [Figura 10.10](#), se o diretório atual for *raiz/spell/mail*, então o nome de caminho relativo *prt/first* refere-se ao mesmo arquivo do nome de caminho absoluto *raiz/spell/mail/prt/first*.

Consentir que o usuário defina seus próprios subdiretórios permite a ele impor uma estrutura sobre seus arquivos. Essa estrutura poderia resultar em diretórios separados para arquivos associados a diferentes tópicos (por exemplo, um subdiretório foi criado para manter o texto deste livro) ou diferentes formas de informação (por exemplo, o diretório *programas* pode conter programas-fonte; o diretório *bin* pode armazenar todos os binários).

Uma decisão política interessante no diretório em forma de árvore refere-se a como tratar da exclusão de um diretório. Se um diretório estiver vazio, sua entrada no diretório que o contém pode ser excluída. Entretanto, suponha que o diretório a ser excluído não esteja vazio, mas contenha vários arquivos ou subdiretórios: duas técnicas podem ser utilizadas. Alguns sistemas, como o MS-DOS, não excluirão um diretório se ele não estiver vazio. Assim, para excluir um diretório, o usuário primeiro precisa excluir todos os arquivos nesse diretório. Se houver algum subdiretório, esse procedimento deverá ser executado recursivamente a eles, de modo que também possam ser excluídos. Essa técnica pode resultar em um trabalho substancial. Uma técnica alternativa, como aquela usada pelo comando *rm* do UNIX, é prover uma opção: quando for requisitada a exclusão de um diretório, todos os arquivos e subdiretórios desse diretório também serão excluídos. As duas técnicas são muito fáceis de implementar; a escolha é apenas uma questão de política. A segunda opção é mais conveniente, mas também é mais perigosa, pois uma estrutura de diretório inteira pode ser removida com um comando. Se esse comando for emitido com erro, uma grande quantidade de arquivos e diretórios precisará ser restaurada (supondo que exista backup).

Com um sistema de diretórios estruturado em forma de árvore, os usuários podem ter permissão de acesso, além de a seus arquivos, aos arquivos de outros usuários. Por exemplo, o usuário B pode acessar os arquivos do usuário A especificando seus nomes de caminho. O usuário B pode especificar um nome de caminho absoluto ou relativo. Como alternativa, o usuário B pode mudar seu diretório atual para ser o diretório do usuário A e acessar os arquivos por seus nomes de arquivo.

Um caminho até um arquivo em um diretório estruturado em árvore não pode ser maior do que aquele em um diretório de dois níveis. Para permitir aos usuários acessar programas sem ter de se lembrar desses caminhos longos, o sistema operacional Macintosh automatiza a busca para os programas executáveis. Um método que ele utiliza é para manter um arquivo, chamado *arquivo desktop*, contendo o código de metadados e os nomes e os locais de todos os programas executáveis que já viu. Quando um novo disco rígido é acrescentado ao sistema, ou quando a rede é acessada, o sistema operacional atravessa a estrutura de diretórios, procurando programas executáveis no dispositivo e registrando as informações pertinentes. Esse mecanismo admite a funcionalidade da execução por clique duplo, descrita anteriormente. Um clique duplo em um arquivo faz seus dados de atributo de criador ser lido e o *arquivo desktop* ser procurado em busca de uma combinação. Quando uma combinação é encontrada, o programa executável apropriado é iniciado, e o arquivo clicado é usado como entrada.

10.3.6 Diretórios em grafo acíclico

Considere dois programadores trabalhando em um projeto conjunto. Os arquivos associados a esse projeto podem ser armazenados em um subdiretório, separando-os dos outros projetos e arquivos de cada um dos programadores. Contudo, como os dois programadores são responsáveis pelo projeto, ambos desejam que o subdiretório esteja em seus próprios diretórios. O subdiretório comum deverá ser *compartilhado*. Um diretório ou arquivo compartilhado existirá no sistema de arquivos dos dois (ou mais) lugares ao mesmo tempo.

Uma estrutura de árvore proíbe o compartilhamento de arquivos ou diretórios. Um grafo acíclico – ou seja, um grafo sem ciclos – permite aos diretórios compartilharem subdiretórios e arquivos ([Figura 10.11](#)). O mesmo arquivo ou subdiretório pode estar em dois diretórios diferentes. O grafo acíclico é uma generalização natural do esquema de diretório estruturado em árvore.

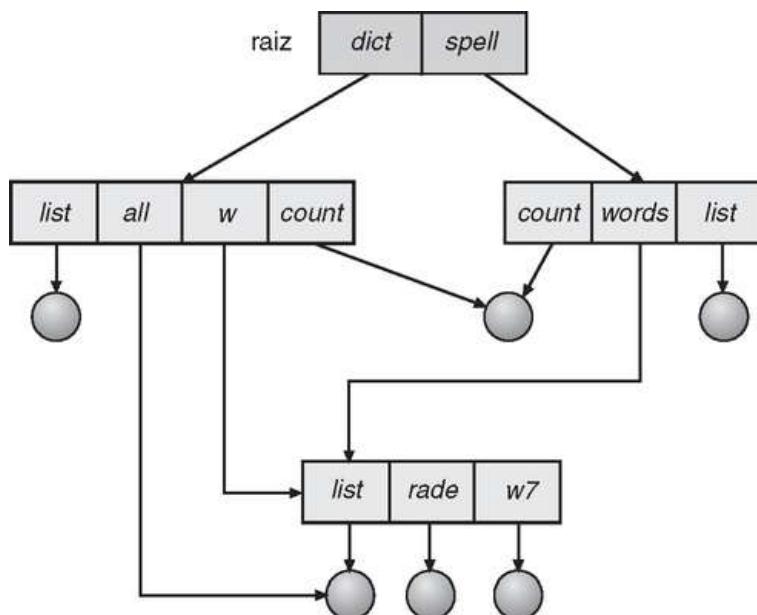


FIGURA 10.11 Estrutura de diretórios com grafo acíclico.

É importante observar que um arquivo (ou diretório) compartilhado não é o mesmo que duas cópias do arquivo. Com duas cópias, cada programador pode ver a cópia, em vez do original, mas se um programador alterar o arquivo as mudanças não aparecerão na outra cópia. Com um arquivo compartilhado, só existe *um* arquivo real, de modo que quaisquer mudanças feitas por uma pessoa são visíveis imediatamente à outra. O compartilhamento é importante para subdiretórios; um novo arquivo criado por uma pessoa aparecerá automaticamente em todos os subdiretórios compartilhados.

Quando as pessoas estão trabalhando como uma equipe, todos os arquivos que querem compartilhar podem ser colocados em um diretório. O UFD de cada um dos membros da equipe terá esse diretório de arquivos compartilhados como um subdiretório. Mesmo no caso de um único usuário, o compartilhamento de arquivo é útil quando a organização dos arquivos do usuário exige que algum arquivo seja colocado em múltiplos subdiretórios. Por exemplo, um programa escrito para determinado projeto deverá estar tanto no diretório de todos os programas quanto no diretório desse projeto.

Os arquivos e subdiretórios compartilhados podem ser implementados de várias maneiras. Uma maneira comum, exemplificada por muitos dos sistemas UNIX, é criar uma nova entrada de diretório, chamada de **link**. Um link é um ponteiro para outro arquivo ou subdiretório. Por exemplo, um link pode ser implementado como um nome de caminho absoluto ou relativo. Quando é feita uma referência a um arquivo, pesquisamos o diretório. Se a entrada do diretório for marcada como um link, então o nome do arquivo real é incluído nas informações do link. **Resolvemos** o link usando o nome de caminho para localizar o arquivo real. Os links são identificados por seu formato na entrada do diretório (ou por um tipo especial em sistemas que admitem tipos) e são chamados de ponteiros indiretos. O sistema operacional ignora esses links quando atravessa árvores de diretório para preservar a estrutura acíclica do sistema.

Outra técnica comum para implementar arquivos compartilhados é duplicar todas as informações sobre eles nos diretórios que os compartilham. Assim, as duas entradas são idênticas e iguais. Considere a diferença entre esta técnica e a criação de um link. Um link é bem diferente da entrada de diretório original; assim, os dois não são iguais. Entretanto, entradas de diretório duplicadas tornam o original e a cópia indistinguíveis. Um problema importante com as entradas de diretório duplicadas é manter a coerência quando um arquivo é modificado.

Uma estrutura de diretórios com grafo acíclico é mais flexível do que uma estrutura de árvore simples, mas também é mais complexa. Vários problemas precisam ser considerados com cuidado. Um arquivo agora pode ter vários nomes de arquivo absolutos. Como consequência, nomes de arquivo distintos podem se referir ao mesmo arquivo. Essa situação é semelhante ao problema de aliasing para as linguagens de programação. Se estivermos tentando atravessar o sistema de arquivos inteiro – para localizar um arquivo, acumular estatísticas sobre todos os arquivos ou copiar todos os arquivos para o armazenamento de backup –, esse problema torna-se significativo, pois não queremos atravessar estruturas compartilhadas mais de uma vez.

Outro problema envolve a exclusão. Quando o espaço alocado a um arquivo compartilhado pode ser desalocado e reutilizado? Uma possibilidade é remover o arquivo sempre que alguém o excluir, mas essa ação pode deixar ponteiros pendentes para o arquivo agora inexistente. Pior ainda, se os ponteiros restantes tiverem endereços de disco reais, e o espaço mais tarde for reutilizado para outros arquivos, esses ponteiros pendentes podem apontar para o meio de outros arquivos.

Em um sistema em que o compartilhamento é implementado por links simbólicos, essa situação é mais fácil de ser resolvida. A exclusão de um link não precisa afetar o arquivo original; somente o link é removido. Se a própria entrada do arquivo for excluída, o espaço para o arquivo é desalocado, deixando os links pendentes. Podemos procurar esses links e removê-los também, mas, a menos que uma lista dos links associados seja mantida com cada arquivo, essa pesquisa pode ser muito dispendiosa. Como alternativa, podemos manter os links até que se tente usá-los. Nesse momento, podemos determinar que o arquivo com o nome indicado pelo link não existe mais e não atende a requisição; o acesso é tratado como qualquer outro nome de arquivo ilegal. (Nesse caso, o projetista do sistema deverá considerar cuidadosamente o que fazer quando um arquivo é excluído e outro arquivo com o mesmo nome é criado.) No caso do UNIX, os links são mantidos quando um arquivo é excluído, e fica a critério do usuário observar se o arquivo original não existe mais ou foi substituído. O Microsoft Windows (todas as versões) utiliza a mesma técnica.

Outra técnica para a exclusão é preservar o arquivo até todas as referências a ele serem excluídas. Para implementar essa técnica, precisamos ter algum mecanismo para determinar que a última referência ao arquivo foi excluída. Poderíamos manter uma lista de todas as referências a um arquivo (entradas de diretório ou links). Quando um link ou uma cópia da entrada de diretório for estabelecido, uma nova entrada será acrescentada à lista de referência de arquivo. Quando um link ou uma entrada de diretório for excluído, removeremos sua entrada na lista. O arquivo será excluído quando sua lista de referência de arquivo estiver vazia.

O problema com essa técnica é o tamanho variável e potencialmente grande da lista de referência de arquivo. No entanto, não precisamos manter a lista inteira; precisamos manter apenas uma contagem do *número* de referências. A adição de um novo link ou entrada de diretório incrementa a contagem de referência; a exclusão de um link ou entrada decrementa a contagem. Quando a contagem chegar a 0, o arquivo poderá ser excluído; não existem mais referências a ele. O sistema operacional UNIX utiliza essa técnica para os links não simbólicos (ou **hard links**), mantendo uma contagem de referência no bloco de informações do arquivo (ou *inode*). Efetivamente proibindo múltiplas referências a diretórios, mantemos uma estrutura de grafo acíclico.

Para evitar problemas como aqueles que acabamos de discutir, alguns sistemas não permitem diretórios compartilhados ou links. Por exemplo, no MS-DOS, a estrutura de diretório é uma estrutura de árvore, em vez de um grafo acíclico.

10.3.7 Diretório geral do grafo

Um problema sério com o uso de uma estrutura de grafo acíclico é garantir que não existam ciclos. Se começarmos com um diretório de dois níveis e permitirmos que os usuários criem subdiretórios, o resultado será um diretório estruturado em árvore. Deverá ser muito fácil ver que a simples inclusão de novos arquivos e subdiretórios a um diretório estruturado em árvore existente preserva a natureza estruturada em árvore. No entanto, quando acrescentamos links, a estrutura de árvore é destruída, resultando em uma estrutura de grafo simples (Figura 10.12).

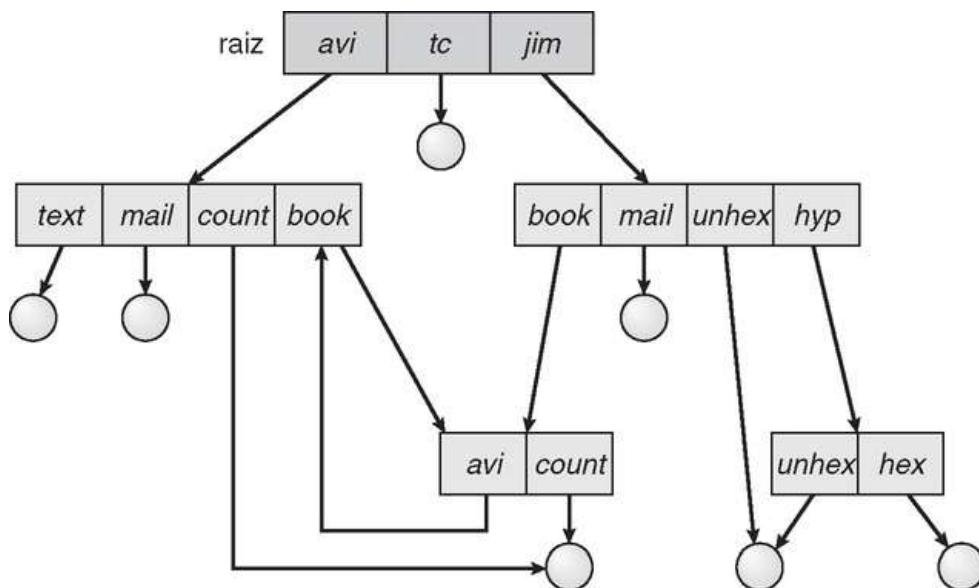


FIGURA 10.12 Diretório de grafo geral.

Ao considerarmos essa estrutura de grafo simples, precisamos levar em conta que a principal vantagem de um grafo acíclico é a relativa simplicidade dos algoritmos para atravessar o grafo e determinar quando não existem mais referências a um arquivo. Queremos evitar a travessia de

seções compartilhadas de um grafo acíclico duas vezes, principalmente por motivos de desempenho. Se acabamos de pesquisar um subdiretório compartilhado principal em busca de determinado arquivo, sem encontrá-lo, não queremos pesquisar novamente nesse subdiretório; a segunda pesquisa seria uma perda de tempo.

Se os ciclos tiverem permissão para existirem no diretório, da mesma forma queremos evitar a pesquisa em qualquer componente duas vezes, por motivos de exatidão e também de desempenho. Um algoritmo mal projetado poderia resultar em um loop sem fim, pesquisando de forma contínua pelo ciclo e nunca terminando. Uma solução é limitar arbitrariamente a quantidade de diretórios acessados durante uma pesquisa.

Existe um problema semelhante quando tentamos determinar quando um arquivo pode ser excluído. Com estruturas de diretório com grafo acíclico, um valor 0 na contagem de referência significa que não existem mais referências ao arquivo ou diretório, e o arquivo pode ser excluído. Todavia, quando existem ciclos, a contagem de referência pode ser diferente de 0, mesmo quando não for mais possível se referir a um diretório ou arquivo. Essa anomalia é resultante da possibilidade de autorreferência (ou um ciclo) na estrutura de diretórios. Nesse caso, em geral precisamos usar um esquema de coleta de lixo (garbage-collection) para determinar quando a última referência foi excluída e o espaço em disco pôde ser realocado. A coleta de lixo envolve a travessia do sistema de arquivos inteiro, marcando tudo o que pode ser acessado. Depois, uma segunda passada coleta tudo o que não está marcado na lista de espaço livre. (Um procedimento de marcação semelhante pode ser usado para garantir que uma travessia ou pesquisa cobrirá tudo no sistema de arquivos apenas uma vez.) No entanto, a coleta de lixo para um sistema de arquivos baseado em disco é muito demorada e, portanto, raramente é experimentada.

A coleta de lixo só é necessária por causa dos possíveis ciclos no grafo. Assim, uma estrutura com grafo acíclico é muito mais fácil de trabalhar. A dificuldade é evitar ciclos à medida que novos links são acrescentados à estrutura. Como sabemos quando um novo link completará um ciclo? Existem algoritmos para detectar ciclos nos grafos; porém, são muito dispendiosos em termos de computação exigida, em especial quando o grafo está armazenado em disco. Um algoritmo mais simples no caso especial dos diretórios e links é evitar links durante a travessia do diretório. Os ciclos são evitados, e nenhum custo adicional é contraído.

10.4 Montagem do sistema de arquivos

Assim como um arquivo precisa ser *aberto* antes de ser usado, um sistema de arquivos precisa ser *montado* antes de poder estar disponível para os processos no sistema. Mais especificamente, a estrutura de diretórios pode ser montada de múltiplos volumes, que precisam estar montados para estarem disponíveis dentro do espaço de nomes do sistema de arquivos.

O procedimento de montagem é simples. O sistema operacional recebe o nome do dispositivo e do **ponto de montagem** – o local dentro da estrutura de arquivos onde o sistema de arquivos deve ser anexado. Alguns sistemas operacionais exigem que um tipo de sistema de arquivos seja fornecido, enquanto outros inspecionam as estruturas do dispositivo e determinam o tipo do sistema de arquivos. Normalmente, um ponto de montagem é um diretório vazio. Por exemplo, em um sistema UNIX, um sistema de arquivos contendo o diretório home de um usuário poderia ser montado como */home*; depois, para acessar a estrutura de diretórios dentro desse sistema de arquivos, poderíamos preceder os nomes de diretório com */home*, como em */home/jane*. Montar esse sistema de arquivos sob */users* resultaria no nome de caminho */users/jane*, que poderíamos usar para alcançar o mesmo diretório.

Em seguida, o sistema operacional verifica se o dispositivo contém um sistema de arquivos válido. Ele faz isso pedindo ao driver de dispositivo para ler o diretório do dispositivo e verificando se o diretório possui o formato esperado. Por fim, o sistema operacional observa, em sua estrutura de diretórios, que o sistema de arquivos está montado no ponto de montagem especificado. Esse esquema permite ao sistema operacional atravessar sua estrutura de diretórios, alternando entre os sistemas de arquivos, até mesmo sistemas de arquivos de tipos variados, conforme a necessidade.

Para ilustrar a montagem de arquivos, considere o sistema de arquivos representado na [Figura 10.13](#), onde os triângulos representam subárvores dos diretórios que são do nosso interesse. A [Figura 10.13\(a\)](#) mostra um sistema de arquivos existente, enquanto a [Figura 10.13\(b\)](#) mostra um volume não montado residindo em */device/dsk*. Nesse ponto, somente os arquivos no sistema de arquivos existente podem ser acessados. A [Figura 10.14](#) mostra os efeitos da montagem do volume residindo em */device/dsk* sobre */users*. Se o volume não estiver montado, o sistema de arquivos será restaurado para a situação representada na [Figura 10.13](#).

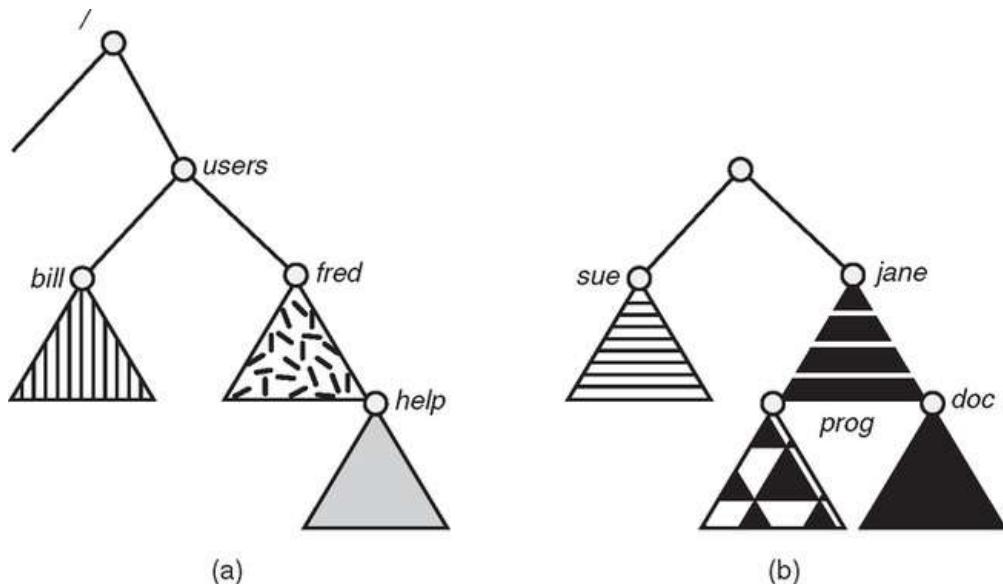


FIGURA 10.13 Sistema de arquivos. (a) Existente e (b) volume não montado.

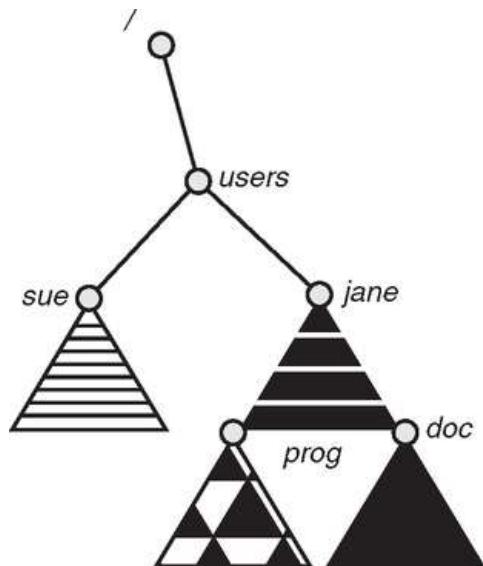


FIGURA 10.14 Ponto de montagem.

Os sistemas impõem semântica para esclarecer a funcionalidade. Por exemplo, um sistema pode proibir uma montagem sobre um diretório que contém arquivos ou, então, pode tornar o sistema de arquivos montado disponível nesse diretório e ocultar os arquivos existentes no diretório até o sistema de arquivos ser desmontado, terminando com o uso do sistema de arquivos e permitindo o acesso aos arquivos originais nesse diretório. Como outro exemplo, um sistema pode permitir ao mesmo sistema de arquivos ser montado repetidamente, em diferentes pontos de montagem ou, então, pode permitir apenas uma montagem por sistema de arquivos.

Considere as ações do sistema operacional do Macintosh clássico. Sempre que o sistema encontra um disco pela primeira vez (discos rígidos são encontrados no momento do boot, e discos óticos são vistos quando inseridos na unidade), o sistema operacional do Macintosh procura um sistema de arquivos no dispositivo. Se encontrar um, ele automaticamente monta o sistema de arquivos no nível de raiz, acrescentando um ícone de pasta na tela, rotulado com o nome do sistema de arquivos (conforme armazenado no diretório do dispositivo). O usuário, então, pode clicar no ícone e, assim, exibir o sistema de arquivos recém-montado. O Mac OS X se comporta de modo semelhante ao BSD UNIX, no qual ele é baseado. Todos os sistemas de arquivos são montados sob o diretório /Volumes. A GUI do Mac OS X oculta esse fato, mostrando os sistemas de arquivos como se fossem todos montados no nível de raiz.

A família de sistemas operacionais Microsoft Windows (95, 98, NT, 2000, 2003, XP, Vista) mantém uma estrutura de diretório de dois níveis estendida, com dispositivos e volumes recebendo uma letra de unidade. Os volumes possuem uma estrutura de diretório com grafo geral associada à letra da unidade. O caminho até um arquivo específico tem a forma de *letra-unidade:\caminho\até\arquivo*. As versões mais recentes do Windows permitem que um sistema de arquivos seja montado em qualquer lugar na árvore de diretórios, assim como no UNIX. Os sistemas operacionais do Windows descobrem automaticamente todos os dispositivos e montam todos os sistemas de arquivos localizados no momento do boot. Em alguns sistemas, como o UNIX, os comandos de montagem são explícitos. Um arquivo de configuração de sistema contém uma lista de dispositivos e pontos de montagem para a montagem automática no momento do boot, mas outras montagens podem ser executadas manualmente.

Aspectos sobre a montagem do sistema de arquivos são discutidos com mais detalhes na [Seção 11.2.2](#).

10.5 Compartilhamento de arquivos

Nas seções anteriores, exploramos a motivação para o compartilhamento de arquivos e algumas das dificuldades envolvidas em permitir que os usuários compartilhem arquivos. Esse compartilhamento de arquivos é bastante desejável para usuários que desejam colaborar e reduzir o esforço exigido para alcançar um objetivo de computação. Portanto, os sistemas operacionais orientados ao usuário precisam acomodar a necessidade de compartilhar arquivos apesar das dificuldades inerentes.

Nesta seção, examinamos mais aspectos do compartilhamento de arquivos. Começamos com a discussão de questões gerais que surgem quando múltiplos usuários compartilham arquivos. Quando múltiplos usuários têm permissão para compartilhar arquivos, o desafio é estender o compartilhamento para múltiplos sistemas de arquivos, inclusive sistemas de arquivos remotos, e discutimos esse desafio também. Finalmente, consideramos o que fazer sobre ações em conflito que ocorrem em dispositivos compartilhados. Por exemplo, se vários usuários estiverem escrevendo em um arquivo, todas as operações de escrita deverão ser permitidas ou o sistema operacional deverá proteger as ações do usuário umas das outras?

10.5.1 Múltiplos usuários

Quando um sistema operacional acomoda múltiplos usuários, as questões de compartilhamento de arquivos, nomeação de arquivos e proteção de arquivos tornam-se mais importantes. Dada uma estrutura de diretório que permita que os arquivos sejam compartilhados pelos usuários, o sistema precisa mediar o compartilhamento de arquivos. O sistema pode permitir a um usuário acessar os arquivos de outros usuários como padrão ou exigir que um usuário conceda acesso especificamente aos arquivos. Essas são as questões de controle de acesso e proteção, abordadas na [Seção 10.6](#).

Para implementar o compartilhamento e a proteção, o sistema precisa manter mais atributos de arquivo e diretório do que são necessários em um sistema monousuário. Embora muitas técnicas tenham sido utilizadas para cumprir esse requisito, a maioria dos sistemas evoluiu para utilizar os conceitos de *proprietário* (ou *usuário*) e *grupo* de arquivo/diretório. O proprietário é o usuário que pode alterar os atributos e conceder acesso, e quem tem mais controle sobre o arquivo ou diretório. O atributo de grupo define um subconjunto de usuários que podem compartilhar acesso ao arquivo. Por exemplo, o proprietário de um arquivo em um sistema UNIX pode emitir todas as operações sobre um arquivo, enquanto os membros do grupo do arquivo podem executar um subconjunto dessas operações, e todos os outros usuários podem executar outro subconjunto das operações. Exatamente quais operações podem ser executadas pelos membros do grupo e outros usuários é algo que pode ser definido pelo proprietário do arquivo. Outros detalhes sobre atributos de permissão estão incluídos na próxima seção.

Os IDs de proprietário e grupo de determinado arquivo (ou diretório) são armazenados com os outros atributos do arquivo. Quando um usuário requisita uma operação sobre um arquivo, o ID do usuário pode ser comparado com o atributo do proprietário para determinar se o usuário requisitante é o proprietário do arquivo. De modo semelhante, os IDs de grupo podem ser comparados. O resultado indica quais permissões se aplicam. O sistema, então, aplica essas permissões à operação requisitada e a concede ou recusa.

Muitos sistemas possuem múltiplos sistemas de arquivos locais, incluindo volumes de um único disco ou múltiplos volumes sobre múltiplos discos anexados. Nesses casos, a verificação de ID e a combinação de permissões são simples, uma vez que o sistema de arquivos esteja montado.

10.5.2 Sistemas de arquivo remotos

Com o advento das redes ([Capítulo 16](#)), a comunicação entre os computadores remotos tornou-se possível. As redes permitem o compartilhamento de recursos espalhados por um campus ou até mesmo no mundo inteiro. Um recurso óbvio a ser compartilhado são os dados, na forma de arquivos.

Com a evolução da tecnologia de rede e arquivo, os métodos de compartilhamento de arquivos remotos mudaram. O primeiro método implementado envolve a transferência manual de arquivos entre as máquinas, por meio de programas como `ftp`. O segundo método principal utiliza um **sistema de arquivos distribuído (Distributed File System - DFS)**, em que os diretórios remotos são visíveis da máquina local. De algumas maneiras, o terceiro método, a **World Wide Web**, é uma reversão para o primeiro. Um navegador é necessário para obter acesso aos arquivos remotos, e operações separadas (basicamente, um embrulho para o `ftp`) são usadas para transferir arquivos.

O `ftp` é usado para o **acesso anônimo** e autenticado. O acesso anônimo permite a um usuário transferir arquivos sem ter uma conta no sistema remoto. A World Wide Web utiliza a troca de arquivos anônima quase exclusivamente. O DFS envolve uma integração muito mais estreita entre a máquina que está acessando os arquivos remotos e a máquina que fornece os arquivos. Essa integração aumenta a complexidade, que descrevemos nesta seção.

10.5.2.1 O modelo cliente-servidor

Os sistemas de arquivos remotos permitem a um computador montar um ou mais sistemas de arquivos a partir de uma ou mais máquinas remotas. Nesse caso, a máquina contendo os arquivos é o *servidor*, e a máquina buscando acesso aos arquivos é o *cliente*. O relacionamento cliente-servidor é comum em máquinas em rede. Em geral, o servidor declara que um recurso está disponível aos clientes e especifica exatamente qual recurso (nesse caso, quais arquivos) e exatamente quais clientes. Um servidor pode atender a múltiplos clientes, e um cliente pode usar múltiplos servidores, dependendo dos detalhes de implementação de determinada facilidade cliente-servidor.

O servidor em geral especifica os arquivos disponíveis em nível de volume ou subdiretório. A identificação do cliente é mais difícil. Um cliente pode ser especificado por um nome de rede ou por outro identificador, como *endereço IP*, mas pode ser **falsificado** ou imitado. Como resultado da falsificação, um cliente não autorizado poderá ter acesso permitido ao servidor. Soluções mais seguras incluem a autenticação segura do cliente por meio de chaves criptografadas. Infelizmente, com a segurança surgem muitos desafios, inclusive garantir a compatibilidade entre o cliente e o servidor (eles precisam usar os mesmos algoritmos de criptografia) e trocas de chave seguras (chaves interceptadas poderiam novamente permitir o acesso não autorizado). Em razão da dificuldade de resolver esses problemas, os métodos de autenticação desprotegidos são mais usados.

No caso do UNIX e seu sistema de arquivos de rede (Network File System - NFS), a autenticação ocorre por meio das informações de rede do cliente, como padrão. Nesse esquema, os IDs do usuário no cliente e no servidor devem combinar. Se não combinarem, o servidor não poderá determinar os direitos de acesso aos arquivos. Considere o exemplo de um usuário que tem o ID 1000 no cliente e 2000 no servidor. Uma requisição do cliente para o servidor, a fim de obter um arquivo específico, não será tratada corretamente, já que o servidor determinará se o usuário 1000 tem acesso ao arquivo, em vez de basear a determinação no ID de usuário *real*, que é 2000. O servidor precisa confiar que o cliente apresentará o ID de usuário correto. Observe que os protocolos do NFS permitem relacionamentos muitos para muitos, ou seja, muitos servidores podem fornecer arquivos a muitos clientes. De fato, determinada máquina pode ser um servidor para alguns clientes NFS e um cliente de outros servidores NFS.

Quando o sistema de arquivos remoto está montado, as requisições para operações de arquivo são enviadas em favor do usuário do outro lado da rede ao servidor por meio do protocolo DFS. Normalmente, uma requisição de abertura de arquivo é enviada junto com o ID do usuário requisitante. O servidor, então, aplica as verificações de acesso-padrão para determinar se o usuário possui credenciais para acessar o arquivo no modo requisitado. A requisição é permitida ou negada. Se for permitida, um descritor de arquivo (file handle) é retornado à aplicação cliente, e a aplicação pode então realizar operações de leitura, escrita e outras sobre o arquivo. O cliente fecha o arquivo quando o acesso estiver terminado. O sistema operacional pode aplicar uma semântica semelhante à usada para a montagem do sistema de arquivos local ou pode usar uma semântica diferente.

10.5.2.2 Sistemas de informação distribuídos

Para tornar os sistemas cliente-servidor mais fáceis de gerenciar, os **sistemas de informação distribuídos**, também conhecidos como **serviços de nomes distribuídos**, proveem acesso unificado às informações necessárias à computação remota. Em tal sistema, o **sistema de nomes de domínio (Domain Name System - DNS)** provê traduções de nome de host para endereço de rede para toda a Internet (incluindo a World Wide Web). Antes que o DNS fosse usado de modo generalizado, os arquivos contendo as mesmas informações eram enviados por correio eletrônico ou *ftp* entre todos os hosts em rede. Essa metodologia não pôde acompanhar o crescimento. O DNS é discutido com mais detalhes na [Seção 16.5.1](#).

Outros sistemas de informações distribuídas proveem espaço de *nome de usuário/senha/ID de usuário/ID de grupo* para uma facilidade distribuída. Os sistemas UNIX empregaram uma grande variedade de métodos de informações distribuídas. A Sun Microsystems introduziu as *páginas amarelas* (que passou a ser o **serviço de informações de rede**, ou **Network Information Service - NIS**) e a maioria do setor adotou seu uso. Isso centraliza o armazenamento de nomes de usuário, nomes de host, informações de impressora e coisas desse tipo. Infelizmente, também usa métodos de autenticação insecuros, incluindo o envio de senhas do usuário não criptografadas (em *texto limpo*) e identificação de hosts por endereço IP. O NIS+ da Sun é um substituto muito mais seguro para o NIS, mas também é muito mais complicado e ainda não foi adotado para uso geral.

No caso do **common internet file system (CIFS)** da Microsoft, as informações da rede são usadas em conjunto com a autenticação do usuário (nome de usuário e senha) para criar um **login de rede** que o servidor usa para decidir se permitirá ou negará acesso a um sistema de arquivos requisitado. Para essa autenticação ser válida, os nomes de usuário precisam combinar de uma máquina para outra (assim como o NFS). A Microsoft usa duas estruturas de nomeação distribuídas para prover um único espaço de nomes para os usuários. A tecnologia de nomes mais antiga utiliza **domínios**. A tecnologia mais recente, disponível no Windows XP e no Windows 2000, é o **diretório ativo**. Uma vez estabelecida, a facilidade de nomes distribuídos é usada por todos os clientes e servidores para autenticar usuários.

O setor está passando para o **Lightweight Directory Access Protocol (LDAP)** como um

mecanismo seguro para nomes distribuídos. Na verdade, o diretório ativo é baseado em LDAP. O Sun Microsystems inclui o LDAP com o sistema operacional e permite que ele seja utilizado para a autenticação do usuário, bem como a recuperação de informações de todo o sistema, como a disponibilidade de impressoras. Teoricamente, um diretório LDAP distribuído poderia ser usado por uma organização para armazenar todas as informações de usuários e recursos para todos os computadores dentro dessa organização. O resultado seria a **autenticação única e segura** para os usuários, que entrariam com suas informações de autenticação uma vez para acessar todos os computadores da organização. Isso também facilitaria os esforços de administração de sistemas combinando, em um local, informações que atualmente estão espalhadas em vários arquivos em cada sistema ou em diferentes serviços de informação distribuídos.

10.5.2.3 Modos de falha

Os sistemas de arquivos locais podem falhar por diversos motivos, incluindo falha do disco que contém o sistema de arquivos, adulteração da estrutura de diretórios ou outras informações de gerenciamento de disco (coletivamente chamados de **metadados**), falha no controlador de disco, falha no cabo ou falha no adaptador do host. A falha do usuário ou do administrador de sistemas também pode fazer os arquivos se perderem ou diretórios ou volumes inteiros serem excluídos. Muitas dessas falhas causariam a falha de um host e uma condição de erro seria exibida, quando a intervenção humana seria exigida para reparar o dano.

Os sistemas de arquivos remotos possuem ainda mais modos de falha. Devido à complexidade dos sistemas de rede e das interações exigidas entre as máquinas remotas, muito mais problemas podem interferir com a operação correta dos sistemas de arquivos remotos. No caso das redes, a rede pode ser interrompida entre os dois hosts. Essas interrupções podem resultar de falha do hardware, má configuração do hardware ou problemas de implementação de rede. Embora algumas redes tenham elasticidade embutida, incluindo vários caminhos entre os hosts, muitas não têm. Qualquer falha isolada pode, portanto, interromper o fluxo de comandos do DFS.

Considere um cliente no meio do uso de um sistema de arquivos remoto. Ele tem arquivos abertos a partir do host remoto; entre outras atividades, ele pode estar realizando pesquisas de diretório para os arquivos abertos, lendo ou escrevendo dados em arquivos e fechando arquivos. Agora, considere um particionamento da rede, uma falha do servidor ou mesmo um encerramento planejado do servidor. De repente, o sistema de arquivos remoto não pode mais ser alcançado. Esse cenário é muito comum, de modo que não seria apropriado para um sistema cliente atuar como o faria se um sistema de arquivos local estivesse perdido. Em vez disso, o sistema pode terminar todas as operações com o servidor perdido ou atrasar operações até o servidor poder ser alcançado. Essas semânticas de falha são definidas e implementadas como parte do protocolo do sistema de arquivos remoto. O término de todas as operações pode resultar em usuários perdendo dados - e paciência. Assim, a maioria dos protocolos do DFS impõe ou permite o atraso das operações do sistema de arquivos para hosts remotos, com a esperança de que o host remoto se torne novamente disponível.

Para implementar esse tipo de recuperação de falhas, algum tipo de **informação de estado** pode ser mantido no cliente e no servidor. Se o servidor e o cliente mantiverem conhecimento de suas atividades atuais e arquivos abertos, então poderão se recuperar de uma falha de modo transparente. Na situação em que o servidor falha, mas precisa reconhecer que exportou sistemas de arquivos e abriu certos arquivos, o NFS utiliza uma técnica simples, implementando um DFS **stateless (sem manutenção de estado)**. Em essência, ele considera que uma requisição do cliente para uma leitura ou escrita de arquivo não teria ocorrido a menos que o sistema de arquivos tivesse sido montado de maneira remota e o arquivo tivesse sido previamente aberto. O protocolo NFS transporta todas as informações necessárias para localizar o arquivo apropriado e realizar a operação requisitada sobre um arquivo. De modo semelhante, ele não acompanha quais clientes têm seus volumes exportados montados, novamente supondo que se uma requisição chega ela precisa também ser legítima. Embora essa técnica sem estado torne o NFS elástico e relativamente fácil de implementar, ela o torna inseguro. Por exemplo, requisições de leitura ou escrita forjadas poderiam ser permitidas por um servidor NFS embora os requisitos de requisição de montagem e verificação de permissão não tenham ocorrido. Essas questões são resolvidas no padrão NFS versão 4, em que o NFS possui informações de estado para melhorar sua segurança, desempenho e funcionalidade.

10.5.3 Semântica de consistência

A **semântica de consistência** representa um critério importante para avaliar qualquer sistema de arquivos que admita compartilhamento de arquivos. Essa semântica especifica como os diversos usuários de um sistema devem acessar um arquivo compartilhado simultaneamente. Em particular, eles especificam quando as modificações nos dados por parte de um usuário serão observáveis por outros usuários. Essa semântica é implementada como código com o sistema de arquivos.

A semântica de consistência está relacionada diretamente com os algoritmos de sincronismo de processo do [Capítulo 6](#). Contudo, os algoritmos complexos daquele capítulo costumam não ser implementados no caso da E/S de arquivo, devido às grandes latências e baixas razões de transferência de discos e redes. Por exemplo, a realização de uma transação atômica para um disco

remoto poderia envolver diversas comunicações na rede, muitas leituras e escritas em disco ou ambos. Os sistemas que tentam esse conjunto completo de funcionalidades costumam funcionar mal. Uma implementação bem-sucedida da semântica de compartilhamento complexa pode ser encontrada no sistema de arquivos Andrew.

Para a discussão a seguir, consideraremos que uma série de acessos a arquivo (ou seja, leituras e escritas) tentados por um usuário ao mesmo arquivo sempre está cercada por operações `open()` e `close()`. A série de acessos entre as operações `open()` e `close()` é uma **sessão de arquivo**. Para ilustrar o conceito, esboçamos vários exemplos importantes de semântica de consistência.

10.5.3.1 Semântica do UNIX

O sistema de arquivos do UNIX ([Capítulo 17](#)) utiliza a seguinte semântica de consistência:

- Escritas em um arquivo aberto por um usuário são visíveis imediatamente a outros usuários que têm esse arquivo aberto.
- Um modo de compartilhamento permite aos usuários compartilharem o ponteiro do local atual no arquivo. Assim, o avanço do ponteiro por um usuário afeta todos os usuários compartilhando. Aqui, um arquivo tem uma única imagem que intercala todos os acessos, não importando sua origem.

Na semântica do UNIX, um arquivo está associado a uma única imagem física, acessada como um recurso exclusivo. A disputa por essa única imagem causa atrasos nos processos do usuário.

10.5.3.2 Semântica da sessão

O Andrew File System – AFS ([Capítulo 17](#)) utiliza a seguinte semântica de consistência:

- As escritas em um arquivo aberto feitas por um usuário não são visíveis a outros usuários que possuem o mesmo arquivo aberto.
- Quando um arquivo é fechado, as mudanças feitas a ele são visíveis apenas em sessões começando mais tarde. Instâncias do arquivo já abertas não refletem essas mudanças.

De acordo com essa semântica, um arquivo pode estar associado temporariamente a várias imagens (possivelmente diferentes) ao mesmo tempo. Como consequência, vários usuários têm permissão de realizar acessos para leitura e escrita simultâneos em suas imagens do arquivo, sem atraso. Quase nenhuma restrição é imposta sobre os acessos por escalonamento.

10.5.3.3 Semântica de arquivos compartilhados imutáveis

Uma técnica exclusiva é a de arquivos compartilhados imutáveis. Quando um arquivo é declarado como *compartilhado* por seu criador, ele não pode ser modificado. Um arquivo imutável possui duas propriedades fundamentais: seu nome não pode ser reutilizado e seu conteúdo não pode ser alterado. Assim, o nome de um arquivo imutável significa que o conteúdo do arquivo é fixo. A implementação dessa semântica em um sistema distribuído ([Capítulo 17](#)) é simples, pois o compartilhamento é disciplinado (somente de leitura).

10.6 Proteção

Quando informações são armazenadas em um sistema computadorizado, queremos mantê-las protegidas contra dano físico (o problema da *confiabilidade*) e acesso impróprio (o problema da *proteção*).

A confiabilidade é fornecida por cópias duplicadas dos arquivos. Muitos computadores possuem programas do sistema que automaticamente (ou por intervenção de um operador) copiam arquivos de disco para fita em intervalos regulares (uma vez por dia, ou semana, ou mês) para manter uma cópia, caso o sistema de arquivos seja acidentalmente destruído. Os sistemas de arquivo podem ser danificados por problemas de hardware (como erros na leitura ou escrita), surtos ou faltas de energia, falha na cabeça de leitura/escrita, sujeira, temperaturas extremas e vandalismo. Os arquivos podem ser excluídos acidentalmente. Bugs no software do sistema de arquivos também podem fazer o conteúdo do arquivo ser perdido. A confiabilidade é abordada com mais detalhes no [Capítulo 12](#).

A proteção pode ser fornecida de muitas maneiras. Para um pequeno sistema monousuário, poderíamos prover proteção removendo os disquetes fisicamente e trancando-os em uma gaveta de mesa ou arquivo de aço. Contudo, em um sistema multiusuário, outros mecanismos são necessários.

10.6.1 Tipos de acesso

A necessidade de proteger arquivos é um resultado direto da capacidade de acessar arquivos. Os sistemas que não permitem o acesso aos arquivos de outros usuários não precisam de proteção. Assim, poderíamos prover proteção completa proibindo o acesso. Como alternativa, poderíamos prover acesso livre sem proteção. As duas técnicas são muito extremas para uso geral. Precisamos é de um **acesso controlado**.

Os mecanismos de proteção proveem acesso controlado limitando os tipos de acesso a arquivos que podem ser feitos. O acesso é permitido ou negado dependendo de vários fatores, sendo que um deles é o tipo de acesso requisitado. Vários tipos de operações diferentes podem ser controlados:

- **Ler.** Ler do arquivo.
- **Escrever.** Escrever ou reescrever o arquivo.
- **Executar.** Carregar o arquivo para a memória e executá-lo.
- **Acrescentar.** Escrever novas informações no final do arquivo.
- **Excluir.** Excluir o arquivo e liberar seu espaço para possível reutilização.
- **Listar.** Listar o nome e os atributos do arquivo.

Outras operações, como renomear, copiar e editar o arquivo, também podem ser controladas. Entretanto, para muitos sistemas, essas funções de nível mais alto podem ser implementadas por um programa do sistema que utiliza chamadas de sistema de nível mais baixo. A proteção é fornecida somente nesse nível. Por exemplo, a cópia de um arquivo pode ser implementada por uma sequência de operações de leitura. Nesse caso, um usuário com acesso para leitura também pode fazer um arquivo ser copiado, impresso, e assim por diante.

Muitos mecanismos de proteção foram propostos. Cada esquema possui vantagens e desvantagens e precisa ser apropriado para a aplicação desejada. Um sistema computadorizado pequeno, usado apenas por alguns membros de um grupo de pesquisa, por exemplo, pode não precisar dos mesmos tipos de proteção de um computador corporativo grande, usado para pesquisa, finanças e operações pessoais. Discutimos algumas técnicas de proteção nas próximas seções e apresentamos um tratamento mais completo no [Capítulo 14](#).

10.6.2 Controle de acesso

A técnica mais comum para o problema de proteção é tornar o acesso dependente da identidade do usuário. Diferentes usuários podem precisar de diferentes tipos de acesso a um arquivo ou diretório. O esquema mais geral de implementar o acesso dependente da identidade é associar a cada arquivo e diretório uma **lista de controle de acesso (Access Control List - ACL)**, especificando os nomes de usuário e os tipos de acesso permitidos para cada usuário. Quando um usuário requisita acesso a determinado arquivo, o sistema operacional verifica a lista de acesso associada a esse arquivo. Se esse usuário estiver listado para o acesso requisitado, o acesso será permitido. Caso contrário, haverá uma violação de proteção, e a tarefa do usuário terá acesso negado ao arquivo.

Essa técnica possui a vantagem de permitir metodologias de acesso complexas. O problema principal com as listas de acesso é seu tamanho. Se quisermos permitir que todos leiam um arquivo, temos de listar todos os usuários com acesso de leitura. Essa técnica possui duas consequências indesejáveis:

- A construção de tal lista pode ser uma tarefa tediosa e não gratificante, em especial se não soubermos com antecipação a lista de usuários no sistema.
- A entrada do diretório, anteriormente de tamanho fixo, agora precisa ser de tamanho variável, resultando em um gerenciamento de espaço mais complicado.

Esses problemas podem ser resolvidos com o uso de uma versão condensada da lista de acesso.

Para condensar o tamanho da lista de controle de acesso, muitos sistemas reconhecem três classificações de usuários em conjunto com cada arquivo:

- **Proprietário.** O usuário que criou o arquivo é o proprietário.
- **Grupo.** Um conjunto de usuários que estão compartilhando o arquivo e precisam de acesso semelhante é um grupo ou grupo de trabalho.
- **Universo.** Todos os outros usuários no sistema constituem o universo.

A técnica recente mais comum é combinar as listas de controle de acesso com o esquema de proprietário, grupo e universo, mais genérico (e mais fácil de implementar) que acabamos de descrever. Por exemplo, o Solaris, a partir da versão 2.6, utiliza três categorias de acesso como padrão, mas permite o acréscimo de listas de controle de acesso a arquivos e diretórios específicos quando for desejado um controle de acesso mais detalhado.

Para ilustrar, considere uma pessoa, Sara, que está escrevendo um novo livro. Ela contratou três alunos formados (Jim, Dawn e Jill) para ajudar no projeto. O texto do livro é mantido em um arquivo chamado *livro*. A proteção associada a esse arquivo é a seguinte:

- Sara precisa ser capaz de chamar todas as operações sobre o arquivo.
- Jim, Dawn e Jill devem ser capazes apenas de ler e escrever no arquivo; eles não podem ter permissão para excluir o arquivo.
- Todos os outros usuários poderão ler, mas não escrever no arquivo. (Sara está interessada em permitir que o máximo de pessoas possível leia o texto, para obter um feedback apropriado.)

Para conseguir tal proteção, temos de criar um novo grupo - digamos, *texto* - com os membros Jim, Dawn e Jill. O nome do grupo, *texto*, precisa então estar associado ao arquivo *livro*, e os direitos de acesso precisam ser definidos de acordo com a política esboçada.

Agora, considere um visitante para quem Sara gostaria de conceder acesso temporário ao [Capítulo 1](#). O visitante não pode ser adicionado ao grupo *texto*, pois isso lhe daria acesso a todos os capítulos. Como os arquivos só podem estar em um grupo, Sara não pode acrescentar outro grupo ao [Capítulo 1](#). Com a inclusão da funcionalidade de lista de controle de acesso, o visitante pode ser acrescentado à lista de controle de acesso do [Capítulo 1](#).

Para que esse esquema funcione corretamente, as permissões e as listas de acesso precisam ser controladas de perto. Esse controle pode ser realizado de várias maneiras. Por exemplo, no sistema UNIX, os grupos só podem ser criados e modificados pelo gerente da instalação (ou por qualquer superusuário). Assim, o controle é obtido pela interação humana. No sistema VMS, o proprietário do arquivo pode criar e modificar a lista de controle de acesso. As listas de acesso são discutidas com mais detalhes na [Seção 14.5.2](#).

Com a classificação de proteção mais limitada, somente três campos são necessários para definir a proteção. Cada campo é uma coleção de bits, cada um deles permitindo ou impedindo o acesso associado a ele. Por exemplo, o sistema UNIX define três campos de 3 bits cada: rwx, onde r controla o acesso para leitura, w controla o acesso para escrita e x controla a execução. Existe um campo separado para o proprietário do arquivo, para o grupo do arquivo e para todos os outros usuários. Nesse esquema, 9 bits por arquivo são necessários para registrar a informação de proteção. Assim, para o nosso exemplo, os campos de proteção para o arquivo *livro* são os seguintes: para a proprietária Sara, todos os bits estão marcados; para o grupo *texto*, os bits r e w estão marcados; e para o universo, somente o bit r está marcado.

Uma dificuldade na combinação de técnicas surge na interface com o usuário. Os usuários precisam ser capazes de saber quando as permissões opcionais da ACL estão marcadas em um arquivo. No exemplo do Solaris, um "+" anexa as permissões regulares, como em:

```
19 -rw-r-r+ 1 jim staff 130 May 25 22:13 arquivo1
```

Um conjunto de comandos separado, *setfacl* e *getfacl*, é usado para gerenciar as ACLs.

Os usuários do Windows XP normalmente gerenciam as listas de controle de acesso por meio da GUI. A [Figura 10.15](#) mostra uma janela de permissão de arquivo no sistema de arquivos NTFS do Windows XP. Neste exemplo, o usuário "guest" tem acesso negado especificamente ao arquivo *10.tex*.

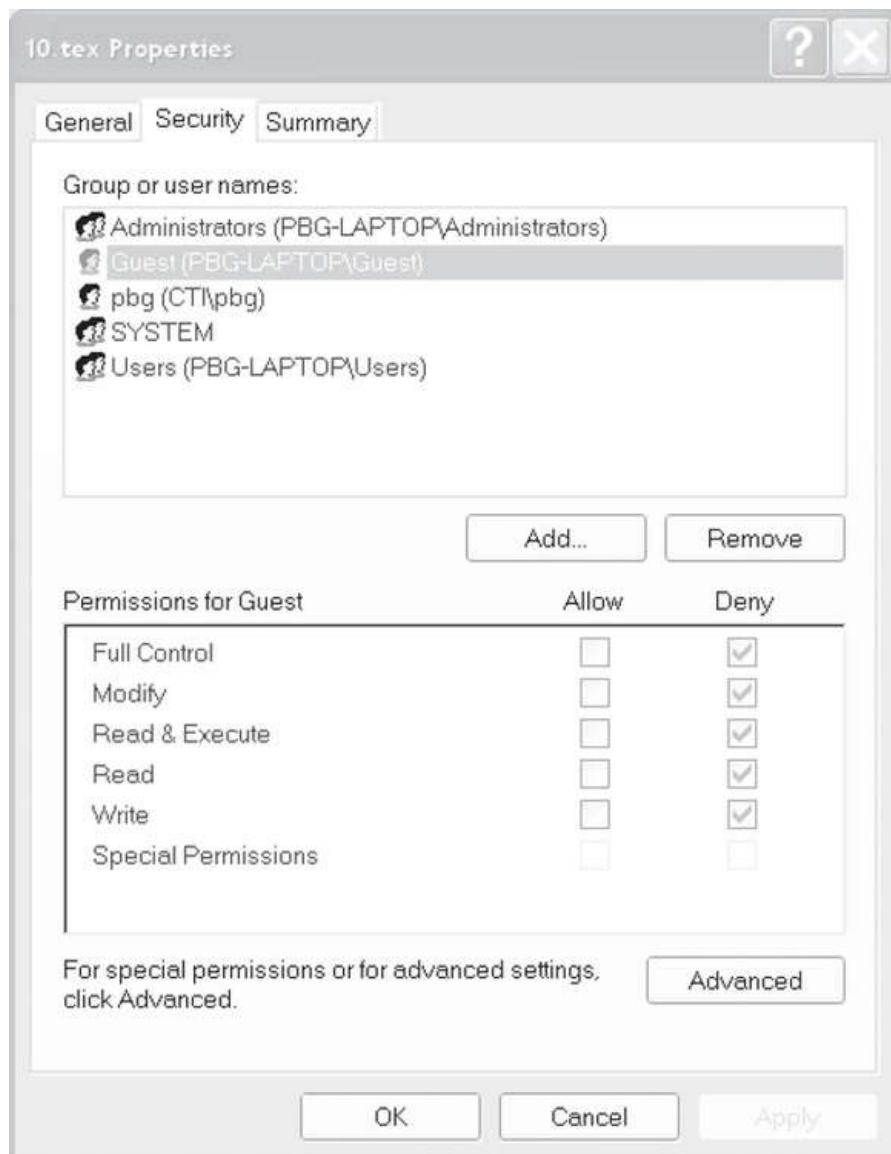


FIGURA 10.15 Gerenciamento da lista de controle de acesso no Windows XP.

Outra dificuldade é a atribuição de precedência quando a permissão e as ACLs entram em conflito. Por exemplo, suponha que Joe esteja no grupo do arquivo, que tem permissão para leitura, mas o arquivo tem uma ACL concedendo a Joe permissões para leitura e escrita. Uma tentativa de escrita por Joe deve ser concedida ou negada? O Solaris dá a permissão das ACLs (pois são mais detalhadas e não são atribuídas como padrão). Isso segue a regra geral de que a especificidade deve ter prioridade.

10.6.3 Outras técnicas de proteção

Outra técnica para o problema de proteção é associar uma senha a cada arquivo. Assim como o acesso ao sistema computadorizado é controlado por uma senha, o acesso a cada arquivo pode ser controlado por uma senha. Se as senhas forem escolhidas aleatoriamente e alteradas com frequência, esse esquema pode ser eficaz na limitação do acesso a um arquivo. O uso de senha, porém, possui algumas desvantagens. Primeira, a quantidade de senhas de que o usuário precisa se lembrar pode se tornar grande, tornando o esquema impraticável. Segunda, se apenas uma senha for usada para todos os arquivos, então, uma vez descoberta, todos os arquivos estarão acessíveis; a proteção é com base no tudo ou nada. Alguns sistemas (por exemplo, o TOPS-20) permitem a um usuário associar uma senha a um subdiretório, em vez de um arquivo individual, para lidar com esse problema. O sistema operacional VM/CMS da IBM permite três senhas para um minidisco – uma para acesso para leitura, outra para escrita e outra para multiescrita.

Alguns sistemas monousuário – como o MS-DOS e versões do Macintosh antes do Mac OS X – apresentam pouca proteção. Em cenários nos quais esses antigos sistemas agora estão sendo colocados em redes onde o compartilhamento de arquivos e a comunicação são necessários, os mecanismos de proteção precisam ser **encaixados**. O projeto de um recurso para um sistema

operacional novo quase sempre é mais fácil do que acrescentar um recurso a um sistema existente. Essas atualizações são menos eficazes e não são transparentes.

Em uma estrutura de diretório multinível, precisamos proteger não apenas os arquivos individuais, mas também as coleções de arquivos em subdiretórios, ou seja, precisamos fornecer um mecanismo para a proteção do diretório. As operações com diretório que precisam ser protegidas são diferentes das operações de arquivo. Queremos controlar a criação e a exclusão de arquivos em um diretório. Além disso, provavelmente queremos controlar se um usuário pode determinar a existência de um arquivo em um diretório. Às vezes, o conhecimento da existência e nome de um arquivo pode ser significativo por si só. Assim, a listagem do conteúdo de um diretório precisa ser uma operação protegida. Da mesma maneira, se um nome de caminho se refere a um arquivo em um diretório, o usuário precisa ter acesso concedido para o diretório e o arquivo. Em sistemas em que os arquivos podem ter diversos nomes de caminho (como nos grafos acíclico ou geral), um usuário qualquer pode ter direitos de acesso diferentes para um arquivo específico, dependendo do nome de caminho utilizado.

PERMISSÕES EM UM SISTEMA UNIX

No sistema UNIX, a proteção do diretório e a proteção do arquivo são tratadas de modo semelhante. Ou seja, associado a cada subdiretório existem três campos - owner, grupo e universo -, cada um consistindo em três bits rwx. Assim, um usuário só pode listar o conteúdo de um subdiretório se o bit r estiver marcado no campo apropriado. De modo semelhante, um usuário pode alterar seu diretório atual para outro diretório atual (digamos, foo) somente se o bit x associado ao subdiretório foo estiver marcado no campo apropriado.

Um exemplo de listagem de diretório de um ambiente UNIX aparece na [Figura 10.16](#). O primeiro campo descreve a proteção do arquivo ou diretório. Um d como primeiro caractere indica um subdiretório. Também aparece o número de links para o arquivo, o nome do owner, o nome do grupo, o tamanho do arquivo em bytes, a data da última modificação e finalmente o nome do arquivo (com a extensão opcional).

-r--r--r--	1	pbg	staff	31200	Sep 3 08:30	intro.ps
drwx----	5	pbg	staff	512	Jul 8 09:33	private/
drwxrwxr-x	2	pbg	staff	512	Jul 8 09:35	doc/
drwxrwx---	2	pbg	student	512	Aug 3 14:13	student-proj/
-r--r--r--	1	pbg	staff	9423	Feb 24 2003	program.c
-rwxr-xr-x	1	pbg	staff	20471	Feb 24 2003	program
drwx--x--x	4	pbg	faculty	512	Jul 31 10:31	lib/
drwx----	3	pbg	staff	1024	Aug 29 06:52	mail/
drwxrwxrwx	3	pbg	staff	512	Jul 8 09:35	test/

FIGURA 10.16 Um exemplo de listagem de diretório.

10.7 Resumo

Um arquivo é um tipo de dado abstrato, definido e implementado pelo sistema operacional. Essa é uma sequência de registros lógicos. Um registro lógico pode ser um byte, uma linha (de tamanho fixo ou variável) ou um item de dados mais complexo. O sistema operacional pode aceitar especificamente vários tipos de registro ou pode deixar esse suporte para o programa de aplicação.

A tarefa principal para o sistema operacional é mapear o conceito de arquivo lógico nos dispositivos físicos de armazenamento, como fita magnética ou disco. Como o tamanho do registro físico do dispositivo pode não ser o mesmo que o tamanho do registro lógico, pode ser preciso ordenar os registros lógicos nos registros físicos. Novamente, essa tarefa pode ter o suporte do sistema operacional ou pode ficar para o programa de aplicação.

Cada dispositivo em um sistema de arquivos mantém um sumário do volume, ou diretório do dispositivo, listando o local dos arquivos no dispositivo. Além disso, é útil criar diretórios, para permitir que os arquivos sejam organizados. Um diretório de único nível em um sistema multiusuário pode causar problemas, pois cada arquivo precisa ter um nome exclusivo. Um diretório de dois níveis resolve esse problema, criando um diretório separado para os arquivos de cada usuário. O diretório lista os arquivos por nome e inclui o local do arquivo no disco, tamanho, tipo, proprietário, hora da criação, hora da última modificação, e assim por diante, de cada arquivo.

A generalização natural de um diretório de dois níveis é um diretório estruturado em árvore. Um diretório estruturado em árvore permite a um usuário criar subdiretórios para organizar seus arquivos. As estruturas de diretório com grafo acíclico possibilitam ao usuário compartilhar subdiretórios e arquivos, mas complicam a pesquisa e a exclusão. Uma estrutura de grafo geral permite a flexibilidade completa no compartilhamento de arquivos e diretórios, mas às vezes exige a coleta de lixo (garbage collection) para recuperar o espaço em disco não utilizado.

Os discos são segmentados em um ou mais volumes, cada um contendo um sistema de arquivos, ou permanecem “brutos”. Os sistemas de arquivos são montados nas estruturas de nomes do sistema para torná-los disponíveis. O esquema de nomes depende do sistema operacional. Uma vez montados, os arquivos dentro da partição estão disponíveis para uso. Os sistemas de arquivo podem ser desmontados para desativar o acesso ou para manutenção.

O compartilhamento de arquivos depende da semântica fornecida pelo sistema. Os arquivos podem ter vários leitores, vários escritores ou limites no compartilhamento. Os sistemas de arquivos distribuídos permitem que os hosts clientes montem volumes ou diretórios a partir dos servidores, desde que possam acessar um ao outro por meio de uma rede. Os sistemas de arquivos remotos apresentam desafios na confiabilidade, desempenho e segurança. Os sistemas de informação distribuídos mantêm informações de usuário, host e acesso, para que clientes e servidores possam compartilhar informações de estado para gerenciar o uso e o acesso.

Como os arquivos são o principal mecanismo de armazenamento de informações na maioria dos sistemas computadorizados, a proteção do arquivo é necessária. O acesso aos arquivos pode ser controlado separadamente para cada tipo de acesso - leitura, escrita, execução, acréscimo, exclusão, listagem de diretório, e assim por diante. A proteção do arquivo pode ser fornecida por listas de acesso, por senhas ou por outras técnicas.

Exercícios práticos

- 10.1. Alguns sistemas excluem automaticamente todos os arquivos do usuário quando um usuário faz o logoff ou um job termina, a menos que o usuário solicite explicitamente que eles devem ser mantidos; outros sistemas mantêm todos os arquivos, a menos que o usuário os exclua explicitamente. Discuta as vantagens relativas de cada uma dessas técnicas.
- 10.2. Por que alguns sistemas registram o tipo de um arquivo, enquanto outros deixam isso para o usuário e outros não implementam tipos de arquivos múltiplos? Qual sistema é "melhor"?
- 10.3. De modo semelhante, alguns sistemas admitem muitos tipos de estruturas para os dados de um arquivo, enquanto outros admitem um fluxo de bytes. Quais são as vantagens e as desvantagens de cada técnica?
- 10.4. Você conseguiria simular uma estrutura de diretório multinível com uma estrutura de diretório de único nível, na qual podem ser usados nomes de qualquer tamanho? Se a sua resposta for sim, explique como você pode fazer isso e compare esse esquema com o esquema de diretório multinível. Se a resposta for não, explique o que impede o sucesso da sua simulação. Como a sua resposta mudaria se os nomes de arquivo fossem limitados a sete caracteres?
- 10.5. Explique a finalidade das operações `open()` e `close()`.
- 10.6. Dê um exemplo de uma aplicação na qual os dados em um arquivo devem ser acessados na seguinte ordem:
 - a. Sequencialmente
 - b. Aleatoriamente
- 10.7. Em alguns sistemas, um subdiretório pode ser lido e gravado por um usuário autorizado, assim como os arquivos normais.
 - a. Descreva os problemas de proteção que poderiam surgir.
 - b. Sugira um esquema para lidar com cada um desses problemas de proteção.
- 10.8. Considere um sistema que admite 5.000 usuários. Suponha que você queira permitir que 4.990 desses usuários sejam capazes de acessar um arquivo.
 - a. Como você especificaria esse esquema de proteção em UNIX?
 - b. Você consegue sugerir outro esquema de proteção que possa ser usado de modo mais eficaz para esse propósito do que o esquema fornecido pelo UNIX?
- 10.9. Os pesquisadores têm sugerido que, em vez de ter uma lista de acesso associada a cada arquivo (especificando quais usuários podem acessar o arquivo, e como), deveríamos ter uma *lista de controle de usuário* com cada usuário (especificando quais arquivos um usuário pode acessar, e como). Discuta as vantagens relativas desses dois esquemas.

Exercícios

- 10.10. Considere um sistema de arquivos em que um arquivo pode ser excluído e seu espaço em disco retomado enquanto os links para esse arquivo ainda existem. Que problemas poderão ocorrer se um novo arquivo for criado na mesma área de armazenamento ou com o mesmo nome de caminho absoluto? Como esses problemas podem ser evitados?
- 10.11. A tabela de arquivos abertos é usada para manter informações sobre arquivos que estão atualmente abertos. O sistema operacional deve manter uma tabela separada para cada usuário ou manter apenas uma tabela que contém referências aos arquivos que estão sendo acessados por todos os usuários ao mesmo tempo? Se o mesmo arquivo estiver sendo acessado por dois programas ou usuários diferentes, deverá haver duas entradas separadas na tabela de arquivos abertos?
- 10.12. Quais são as vantagens e as desvantagens de fornecer locks obrigatórios em vez de locks de consulta, cujo uso fica a critério dos usuários?
- 10.13. Quais são as vantagens e as desvantagens de registrar o nome do programa criador com os atributos do arquivo (como acontece no sistema operacional do Macintosh)?
- 10.14. Alguns sistemas abrem automaticamente um arquivo quando ele é referenciado pela primeira vez e fecham o arquivo quando a tarefa termina. Discuta as vantagens e desvantagens desse esquema em comparação com o mais tradicional, em que o usuário precisa abrir e fechar o arquivo explicitamente.
- 10.15. Se o sistema operacional soubesse que determinada aplicação teria acesso a dados de arquivo de uma maneira sequencial, como ele poderia explorar essa informação para melhorar o desempenho?
- 10.16. Dê um exemplo de uma aplicação que pudesse se beneficiar com o suporte do sistema operacional para acesso aleatório a arquivos indexados.
- 10.17. Discuta as vantagens e desvantagens de dar suporte a links para arquivos que cruzam pontos de montagem (ou seja, o link do arquivo refere-se a um arquivo que é armazenado em um volume diferente).
- 10.18. Alguns sistemas provedem compartilhamento de arquivos mantendo uma única cópia de um arquivo; outros sistemas mantêm várias cópias, uma para cada um dos usuários que compartilham o arquivo. Discuta os méritos relativos de cada abordagem.
- 10.19. Discuta as vantagens e desvantagens de associar a sistemas de arquivos remotos (armazenados em servidores de arquivos) um conjunto de semânticas de falha diferente do conjunto associado a sistemas de arquivos locais.
- 10.20. Quais são as implicações de dar suporte à semântica de coerência do UNIX para o acesso compartilhado aos arquivos armazenados em sistemas de arquivos remotos?

Notas bibliográficas

As discussões gerais referentes aos sistemas de arquivos foram oferecidas por Grosshans [1986], Golden e Pechura [1986] descreveram a estrutura dos sistemas de arquivos para microcomputadores. Os sistemas de banco de dados e suas estruturas de arquivo foram descritos com detalhes em Silberschatz e outros [2001].

Uma estrutura de diretórios multinível foi implementada inicialmente no sistema MULTICS (Organick [1972]). A maioria dos sistemas operacionais agora implementa estruturas de diretório multinível. Entre eles estão Linux (Bovet e Cesati [2002]), Mac OS X (<http://www.apple.com/macosx/>), Solaris (McDougall e Mauro [2007]) e todas as versões do Windows (Russinovich e Solomon [2005]).

O Network File System (NFS), criado pela Sun Microsystems, permite que estruturas de diretório sejam distribuídas pelos computadores em rede. O NFS é descrito com detalhes no Capítulo 17. O NFS versão 4 é descrito na RFC3530 (<http://www.ietf.org/rfc/rfc3530.txt>). A discussão geral dos sistemas de arquivos do Solaris pode ser encontrada no *System Administration Guide: Devices and File Systems* da Sun (<http://docs.sun.com/app/docs/doc/817-5093>).

O DNS foi proposto inicialmente por Su [1982] e passou por várias revisões desde então, com Mockapetris [1987] acrescentando vários recursos importantes. Eastlake [1999] propôs extensões de segurança para permitir que o DNS mantenha chaves de segurança.

O LDAP, também conhecido como X.509, é um subconjunto derivado do protocolo de diretório distribuído X.500. Ele foi definido por Yeong e outros [1995] e tem sido implementado em muitos sistemas operacionais.

Uma pesquisa interessante está sendo feita na área de interfaces do sistema de arquivos - em particular, sobre questões relativas à nomeação e atributos de arquivo. Por exemplo, o sistema operacional Plan 9, da Bell Laboratories (Lucent Technology), faz todos os objetos se parecerem com sistemas de arquivos. Assim, para exibir uma lista de processos em um sistema, um usuário lista o conteúdo do diretório */proc*. De modo semelhante, para exibir a hora, um usuário só precisa digitar o arquivo */dev/time*.

CAPÍTULO 11

Implementação do sistema de arquivos

Como vimos no [Capítulo 10](#), o sistema de arquivos fornece o mecanismo para armazenamento online e acesso ao conteúdo do arquivo, incluindo dados e programas. O sistema de arquivos reside no *armazenamento secundário*, projetado para manter uma grande quantidade de dados permanentemente. Este capítulo trata em especial de questões em torno de armazenamento e acesso a arquivos no meio de armazenamento secundário mais comum, o disco. Exploramos as maneiras de estruturar o uso do arquivo, alocar espaço em disco, recuperar o espaço liberado, rastrear os locais dos dados e fazer a interface de outras partes do sistema operacional com o armazenamento secundário. As questões de desempenho são consideradas no decorrer do capítulo.

OBJETIVOS DO CAPÍTULO

- Descrever os detalhes da implementação de sistemas de arquivos e estruturas de diretórios locais.
- Descrever a implementação dos sistemas de arquivos remotos.
- Discutir os algoritmos e as escolhas entre alocação com blocos e sem blocos.

11.1 Estrutura do sistema de arquivos

Os discos proveem o kernel do armazenamento secundário em que um sistema de arquivos é mantido. Eles possuem duas características que os tornam um meio conveniente para armazenar muitos arquivos:

1. O disco pode ser alterado no local; é possível ler um bloco do disco, modificá-lo e escrevê-lo de volta no mesmo lugar.
2. O disco pode acessar diretamente qualquer bloco de informações que contém. Assim, é simples acessar qualquer arquivo em sequência ou de forma aleatória, e passar de um arquivo para outro exige apenas o movimento das cabeças de leitura/escrita e esperar pela rotação do disco.

Discutiremos a estrutura do disco com todos os detalhes no [Capítulo 12](#).

Para melhorar a eficiência da E/S, as transferências de E/S entre a memória e o disco são realizadas em unidades de *blocos*. Cada bloco possui um ou mais setores. Dependendo da unidade de disco, os tamanhos dos setores variam de 32 bytes a 4.096 bytes; em geral, são de 512 bytes.

Sistemas de arquivos fornecem um acesso eficaz e conveniente ao disco, para permitir aos dados serem armazenados, localizados e recuperados com facilidade. Um sistema de arquivos impõe dois problemas de projeto muito diferentes. O primeiro problema é definir como o sistema de arquivos deverá aparecer para o usuário. Essa tarefa envolve definir um arquivo e seus atributos, as operações permitidas sobre um arquivo e a estrutura de diretórios para organizar os arquivos. O segundo problema é criar algoritmos e estruturas de dados para mapear o sistema de arquivos lógico nos dispositivos físicos de armazenamento secundário.

O sistema de arquivos propriamente dito costuma ser composto de muitos níveis diferentes. A estrutura mostrada na [Figura 11.1](#) é um exemplo de um projeto em camadas. Cada nível no projeto utiliza os recursos dos níveis mais baixos para criar novos recursos para serem usados por níveis mais altos.

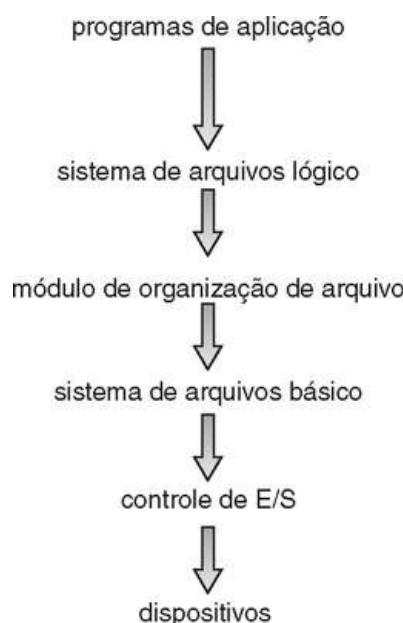


FIGURA 11.1 Sistema de arquivos em camadas.

O nível mais baixo, o *controle de E/S*, consiste em **drivers de dispositivo** e tratadores de interrupção para transferir informações entre a memória principal e o sistema de disco. Um driver de dispositivo pode ser considerado um tradutor. Sua entrada consiste em comandos de alto nível, como "apanhar o bloco 123". Sua saída consiste em instruções de baixo nível, específicas do hardware, usadas pelo controlador do hardware, o qual liga o dispositivo de E/S ao restante do sistema. O driver de dispositivo escreve padrões de bits específicos em locais especiais na memória do controlador de E/S, para dizer em que local do dispositivo ele deve atuar e que ações devem ser realizadas. Os detalhes dos drivers de dispositivo e da infraestrutura de E/S são explicados no [Capítulo 13](#).

O **sistema de arquivos básico** só precisa emitir comandos genéricos para o driver de dispositivo apropriado para poder ler e escrever blocos físicos no disco. Cada bloco físico é identificado por seu endereço numérico no disco (por exemplo, unidade 1, cilindro 73, trilha 2, setor 10). Essa camada também gerencia os buffers de memória e os caches que mantêm diversos blocos do sistema de arquivo, de diretório e de dados. Um bloco no buffer é alocado antes que possa haver transferência

de um bloco de disco. Quando o buffer está cheio, o gerenciador de buffer precisa encontrar mais memória de buffer ou liberar espaço no buffer para permitir que a E/S requisitada seja concluída. Caches são usadas para manter metadados do sistema de arquivos usados com frequência, a fim de melhorar o desempenho, de modo que a gerência do seu conteúdo é fundamental para o desempenho ótimo do sistema.

O **módulo de organização de arquivo** sabe a respeito dos arquivos e seus blocos lógicos, bem como os blocos físicos. Sabendo o tipo de alocação de arquivo utilizado e o local do arquivo, o módulo de organização de arquivo pode traduzir os endereços de blocos lógicos em endereços de blocos físicos, para que o sistema de arquivos básico transfira. Os blocos lógicos de cada arquivo são numerados de 0 (ou 1) até N . Como os blocos físicos contendo os dados não combinam com os números lógicos, é preciso traduzir entre os dois para localizar cada bloco. O módulo de organização de arquivo também inclui o gerenciador de espaço livre, que acompanha blocos desalocados e fornece esses blocos ao módulo de organização de arquivo, quando requisitado.

Por fim, o **sistema de arquivos lógico** gerencia as informações dos metadados. Os metadados incluem toda a estrutura do sistema de arquivos, excluindo os *dados* reais (ou conteúdo dos arquivos). O sistema de arquivos lógico gerencia a estrutura de diretórios para fornecer ao módulo de organização de arquivo as informações de que precisa, dado um nome de arquivo simbólico. Ele mantém a estrutura do arquivo por meio de blocos de controle de arquivo. Um **bloco de controle de arquivo (File Control Block - FCB)**, ou **inode** na maioria dos sistemas de arquivo do UNIX, contém informações sobre o arquivo, incluindo propriedade, permissões e local do conteúdo. O sistema de arquivos lógico também é responsável pela proteção e segurança, conforme discutimos nos [Capítulos 10 e 14](#).

Quando uma estrutura em camadas é usada para a implementação do sistema de arquivos, a duplicação de código é reduzida. O código do controle de E/S, e às vezes o código básico do sistema de arquivos, pode ser usado por múltiplos sistemas de arquivos. Cada sistema de arquivo pode, então, ter seu próprio sistema de arquivos lógico e módulos de organização de arquivos. Infelizmente, a estrutura em camadas pode introduzir mais overhead ao sistema operacional, o que pode resultar em desempenho degradado. O uso da estrutura em camadas, incluindo a decisão de quantas camadas usar e o que cada camada deverá fazer, é um desafio importante no projeto de novos sistemas.

Atualmente, existem muitos sistemas de arquivos. A maioria dos sistemas operacionais admite mais de um. Por exemplo, a maioria dos CD-ROMs é gravada no formato ISO 9660, o formato-padrão combinado pelos fabricantes de CD-ROM. Além dos sistemas de arquivos de mídia removível, cada sistema operacional possui um sistema de arquivos (ou mais) baseado em disco. O UNIX utiliza o **UNIX File System (UFS)**, que é baseado no Berkeley Fast File System (FFS). Windows NT, 2000 e XP aceitam os formatos do sistema de arquivos de disco FAT, FAT32 e NTFS (ou Windows NT File System), além dos formatos do sistema de arquivos de CD-ROM, DVD e disquete. Embora o Linux admita mais de 40 sistemas de arquivos diferentes, o sistema de arquivos padrão do Linux é conhecido como **sistema de arquivo estendido**, com as versões mais comuns sendo ext2 e ext3. Há também sistemas de arquivos distribuídos em que um sistema de arquivos em um servidor é montado por um ou mais computadores clientes por uma rede.

A pesquisa em sistemas de arquivos continua sendo uma área ativa no projeto e implementação de sistemas operacionais. A Google criou seu próprio sistema de arquivos para atender às necessidades específicas de armazenamento e recuperação da empresa. Outro projeto interessante é o sistema de arquivos FUSE, que oferece flexibilidade no uso do sistema de arquivos, implementando e executando sistemas de arquivos em nível de usuário, ao invés de código em nível de kernel. Usando FUSE, um usuário pode acrescentar um novo sistema de arquivos a diversos sistemas operacionais e pode usar esse sistema de arquivos para gerenciar seus arquivos.

11.2 Implementação do sistema de arquivos

Conforme descrevemos na [Seção 10.1.2](#), os sistemas operacionais implementam as chamadas de sistema `open()` e `close()` para os processos requisitarem acesso ao conteúdo do arquivo. Nesta seção, mergulhamos nas estruturas e operações usadas para implementar as operações do sistema de arquivos.

11.2.1 Visão geral

Diversas estruturas no disco e na memória são utilizadas para implementar um sistema de arquivos. Elas variam dependendo do sistema operacional e do sistema de arquivos, mas alguns princípios gerais se aplicam.

No disco, o sistema de arquivos pode conter informações sobre como dar boot em um sistema operacional armazenado nele, a quantidade total de blocos, a quantidade e o local dos blocos livres, a estrutura de diretórios e arquivos individuais. Muitas dessas estruturas estão detalhadas no restante deste capítulo, e a seguir as descreveremos brevemente.

- Um **boot control block** (por volume) pode conter informações necessárias para o sistema carregar um sistema operacional desse volume. Se o disco não tiver um sistema operacional, esse bloco poderá estar vazio. Esse costuma ser o primeiro bloco de um volume. No UFS, ele é chamado **boot block**; no NTFS, ele é o **partition boot sector**.
- Um **volume control block** (por volume) contém os detalhes do volume (ou partição), como a quantidade de blocos, o tamanho dos blocos, o contador de blocos livres e os ponteiros dos blocos livres, e o contador de FCBs livres e ponteiros de FCBs. No UFS, esse bloco é chamado **superbloco**; no NTFS, ele é a **Master File Table** (MFT).
- Uma estrutura de diretórios (por sistema de arquivo) é usada para organizar os arquivos. No UFS, ela inclui nomes de arquivo e números de **inode** associados; no NTFS, ele é a master file table.
- Um FCB por arquivo contém muitos detalhes do arquivo. Ele possui um número identificador exclusivo para permitir a associação com uma entrada do diretório. No NTFS, essa informação na realidade fica armazenada dentro da MFT, que usa uma estrutura de banco de dados relacional, com uma linha por arquivo.

As informações na memória são usadas para o gerenciamento do sistema de arquivos e para a melhoria do desempenho via técnica de cache. Os dados são carregados no momento da montagem, atualizados durante as operações do sistema de arquivos e descartados na desmontagem. As estruturas podem incluir os itens descritos a seguir:

- Uma **tabela de partição** na memória, contendo informações sobre cada volume montado.
- Uma cache de estrutura de diretórios na memória, que mantém as informações dos diretórios acessados recentemente. (Para os diretórios em que os volumes são montados, ela pode conter um ponteiro para a tabela de volume.)
- A **tabela de arquivos abertos em todo o sistema** contém uma cópia do FCB de cada arquivo aberto, bem como outras informações.
- A **tabela de arquivos abertos por processo** contém um ponteiro para a entrada apropriada na tabela de arquivos abertos em todo o sistema, além de outras informações.
- Buffers mantêm blocos do sistema de arquivos quando eles estão sendo lidos do disco ou escritos no disco.

Para criar um novo arquivo, um programa de aplicação chama o sistema de arquivos lógico. O sistema de arquivos lógico conhece o formato das estruturas de diretório. Para criar um novo arquivo, ele aloca um novo FCB. (Como alternativa, se a implementação do sistema de arquivos criar todos os FCBs na hora da criação do sistema de arquivos, um FCB será alocado a partir do conjunto de três FCBs.) O sistema lê o diretório apropriado para a memória, atualiza-o com o novo nome de arquivo e FCB, e escreve-o de volta no disco. Um FCB típico aparece na [Figura 11.2](#).

permissões de arquivo
datas do arquivo (criação, acesso, escrita)
proprietário do arquivo, grupo, ACL
tamanho do arquivo
blocos de dados do arquivo ou ponteiros para blocos de dados do arquivo

FIGURA 11.2 Um bloco de controle de arquivo típico.

Alguns sistemas operacionais, incluindo o UNIX, tratam um diretório exatamente como um arquivo – com um campo de tipo indicando que ele é um diretório. Outros sistemas operacionais, incluindo o Windows NT, implementam chamadas de sistema separadas para arquivos e diretórios e tratam os diretórios como entidades separadas dos arquivos. Independentemente das questões estruturais, o sistema de arquivos lógico pode chamar o módulo de organização de arquivo para mapear a E/S de diretório para números de bloco de disco, que são passados adiante para o sistema de arquivos básico e para o sistema de controle de E/S.

Agora que um arquivo foi criado, ele pode ser usado para E/S. Porém, primeiro, ele precisa ser *aberto*. A chamada `open()` passa um nome de arquivo ao sistema de arquivos lógico. A chamada de sistema `open()` primeiro pesquisa a tabela de arquivos abertos em todo o sistema para ver se o arquivo já está em uso por outro processo. Se estiver, uma entrada na tabela de arquivos abertos por processo é criada, apontando para a tabela de arquivos abertos em todo o sistema já existente. Esse algoritmo pode economizar muito trabalho adicional na abertura de arquivos que já estão abertos. Se o arquivo ainda não estiver aberto, a estrutura do diretório é pesquisada em busca de determinado nome de arquivo. As partes da estrutura do diretório são colocadas em cache na memória, para agilizar as operações de diretório. Quando o arquivo é localizado, o FCB é copiado para uma tabela de arquivos abertos em todo o sistema, na memória. Essa tabela não apenas armazena o FCB, mas também rastreia um contador da quantidade de processos que abriram o arquivo.

Em seguida, é feita uma entrada na tabela de arquivos abertos por processo, com um ponteiro para a entrada na tabela de arquivos abertos e alguns outros arquivos. Esses outros arquivos podem incluir um ponteiro para a localização atual no arquivo (para a próxima operação de leitura ou escrita) e o modo de acesso em que o arquivo está aberto. A chamada `open()` retorna um ponteiro para a entrada apropriada na tabela de arquivos abertos por processo. Todas as operações de arquivo, em seguida, são realizadas usando esse ponteiro. O nome do arquivo pode não fazer parte da tabela de arquivos abertos, pois o sistema não tem mais uso para ele depois da localização no disco do FCB apropriado. Ele poderia estar em cache, para ganhar tempo em aberturas subsequentes do mesmo arquivo. O nome dado à entrada varia. Os sistemas UNIX se referem a ela como **descriptor de arquivo**; o Windows, como **handle de arquivo** (ou descriptor de arquivo).

Quando um processo fecha o arquivo, a entrada na tabela por processo é removida e o contador de abertura na entrada da tabela do sistema é decrementado. Quando todos os usuários que abriram o arquivo o tiverem fechado, qualquer metadado atualizado é copiado para a estrutura de diretório baseada em disco e a entrada na tabela de arquivos abertos em todo o sistema é removida.

Alguns sistemas complicam esse esquema ainda mais, usando o sistema de arquivos como uma interface para outros aspectos do sistema, como as redes. Por exemplo, no UFD, a tabela de arquivos abertos em todo o sistema mantém os inodes e outras informações para arquivos e diretórios. Ela também mantém informações semelhantes para conexões de rede e dispositivos. Desse modo, um mecanismo pode ser usado para diversos objetivos.

Os aspectos de caching dessas estruturas não devem ser esquecidos. Muitos sistemas mantêm todas as informações sobre um arquivo aberto, exceto por seus blocos de dados reais, que estão na memória. O sistema BSD UNIX é típico no seu uso de caches onde quer que a E/S de disco possa ser salva. Sua taxa média de acertos no cache é de 85%, mostrando que essas técnicas merecem ser implementadas.

As estruturas operacionais da implementação de um sistema de arquivos são resumidas na [Figura 11.3](#).

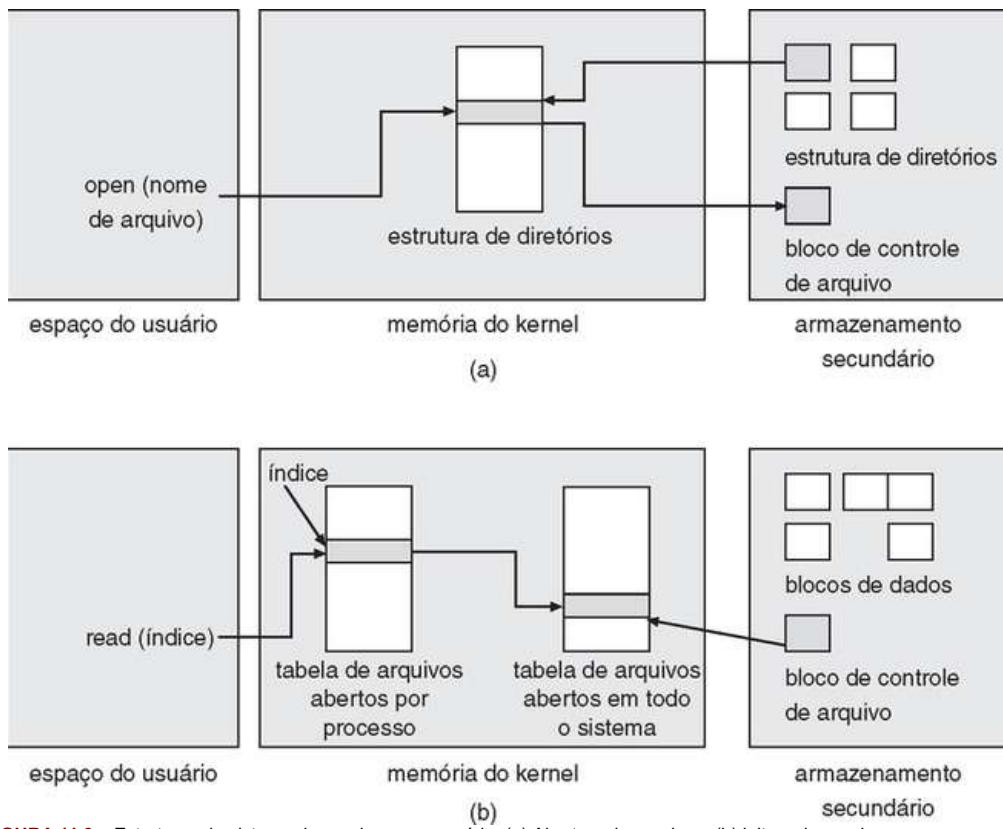


FIGURA 11.3 Estruturas do sistema de arquivos na memória. (a) Abertura de arquivo e (b) leitura de arquivo.

11.2.2 Partições e montagem

O layout de um disco pode ter muitas variações, dependendo do sistema operacional. Um disco pode ser dividido em várias partições ou um volume pode espalhar várias partições por vários discos. Discutiremos aqui o primeiro layout, enquanto o segundo é considerado mais corretamente uma forma de RAID, algo que será discutido na [Seção 12.7](#).

Cada partição pode ser “bruta” (raw), não contendo um sistema de arquivos, ou “processada” (cooked), contendo um sistema de arquivos. Um **raw disk** é usado quando nenhum sistema de arquivos for apropriado. Por exemplo, o swap do UNIX utiliza uma partição raw, pois usa seu próprio formato no disco e não utiliza um sistema de arquivos. De modo semelhante, alguns bancos de dados utilizam um disco bruto e formatam os dados de acordo com suas necessidades. O disco bruto também pode manter informações necessárias pelos sistemas de disco RAID, como mapas de bits indicando quais blocos estão espelhados e quais foram alterados e precisam ser espelhados. De modo semelhante, o raw disk pode conter um banco de dados em miniatura, contendo informações de configuração RAID, como quais discos são membros de cada conjunto RAID. O uso do disco bruto é discutido com mais detalhes na [Seção 12.5.1](#).

As informações de boot podem ser armazenadas em uma partição separada. Mais uma vez, ela tem seu próprio formato, pois, no momento do boot, o sistema ainda não carregou o código do sistema de arquivos e, por isso, não pode interpretar o formato do sistema de arquivos. Em vez disso, a informação de boot normalmente é uma sequência de blocos, carregados como uma imagem para a memória. A execução da imagem começa em um local predefinido, como o primeiro byte. Esse **carregador de boot**, por sua vez, conhece o suficiente sobre a estrutura do sistema de arquivos para poder encontrar e carregar o kernel e iniciar sua execução. Ele pode conter mais do que as instruções de como dar boot em um sistema operacional específico. Por exemplo, os PCs e outros sistemas podem realizar o **boot dual**. Vários sistemas operacionais podem estar instalados nesse sistema. Como o sistema sabe qual deverá dar boot? Um boot loader que entende vários sistemas de arquivos e vários sistemas operacionais pode ocupar o espaço de boot. Uma vez carregado, ele pode dar boot em um dos sistemas operacionais disponíveis no disco. O disco pode ter várias partições, cada uma contendo um tipo diferente de sistema de arquivos e um sistema operacional diferente.

A **partição raiz (root partition)**, que contém o kernel do sistema operacional e às vezes outros arquivos do sistema, é montada no momento do boot. Outras partições podem ser montadas automaticamente no boot ou montadas manualmente depois, dependendo do sistema operacional. Como parte da montagem bem-sucedida, o sistema operacional verifica se o dispositivo contém um sistema de arquivos válido. Ele faz isso pedindo ao driver de dispositivo para ler o diretório do

dispositivo e verificando se o diretório possui o formato esperado. Se o formato for inválido, a partição precisará ter sua coerência verificada e possivelmente corrigida, com ou sem intervenção do usuário. Por fim, o sistema operacional observa em sua estrutura de **tabela de montagem (mount table)** na memória que um sistema de arquivos está montado e o tipo do sistema de arquivos. Os detalhes dessa função dependem do sistema operacional. Os sistemas baseados no Microsoft Windows montam cada volume em um espaço de nomes separado, indicado por uma letra e um sinal de dois-pontos. Para registrar que um sistema de arquivos está montado em F:, por exemplo, o sistema operacional coloca um ponteiro para o sistema de arquivos em um campo da estrutura de dispositivos correspondente a F:. Quando um processo especifica uma letra de unidade, o sistema operacional encontra o ponteiro do sistema de arquivos apropriado e atravessa as estruturas de diretório nesse dispositivo para encontrar o arquivo ou diretório especificado. Versões mais recentes do Windows podem montar um sistema de arquivos em qualquer ponto dentro da estrutura de diretórios existente.

No UNIX, o sistema de arquivos pode ser montado em qualquer diretório. A montagem é implementada pela definição de um sinalizador na cópia da memória do inode para esse diretório. O sinalizador indica que o diretório é um ponto de montagem. Um campo, então, aponta para uma entrada na tabela de montagem, indicando qual dispositivo está montado lá. A entrada da tabela de montagem contém um ponteiro para o superbloco do sistema de arquivos nesse dispositivo. Esse esquema permite ao sistema operacional atravessar sua estrutura de diretório, alternando de forma transparente entre os sistemas de arquivos conforme a necessidade.

11.2.3 Sistemas de arquivos virtuais

A seção anterior deixa claro que os sistemas operacionais modernos precisam prover suporte a vários tipos simultâneos de sistemas de arquivos. Mas como um sistema operacional permite que vários tipos de sistemas de arquivos sejam integrados em uma estrutura de diretório? Como os usuários podem se mover de forma transparente entre os tipos de sistema de arquivo enquanto navegam pelo espaço do sistema de arquivos? Agora precisamos discutir alguns detalhes de implementação.

Um método óbvio, mas não o ideal, para a implementação de vários tipos de sistemas de arquivos é escrever rotinas de diretório e arquivo para cada tipo. Em vez disso, a maioria dos sistemas operacionais, incluindo o UNIX, utiliza técnicas orientadas a objeto para simplificar, organizar e modularizar a implementação. O uso desses métodos permite que tipos de sistemas de arquivos muito diferentes sejam implementados dentro da mesma estrutura, incluindo sistemas de arquivos de rede, como o NFS. Os usuários podem acessar arquivos contidos dentro de vários sistemas de arquivos no disco local ou ainda pela rede.

As estruturas e procedimentos de dados são usados para isolar a funcionalidade básica de chamada de sistema dos detalhes da implementação. Assim, a implementação do sistema de arquivos consiste em três camadas principais; ela é representada esquematicamente na [Figura 11.4](#). A primeira camada é a interface do sistema de arquivos, baseada nas chamadas de sistema open(), read(), write() e close(), e em descritores de arquivo.

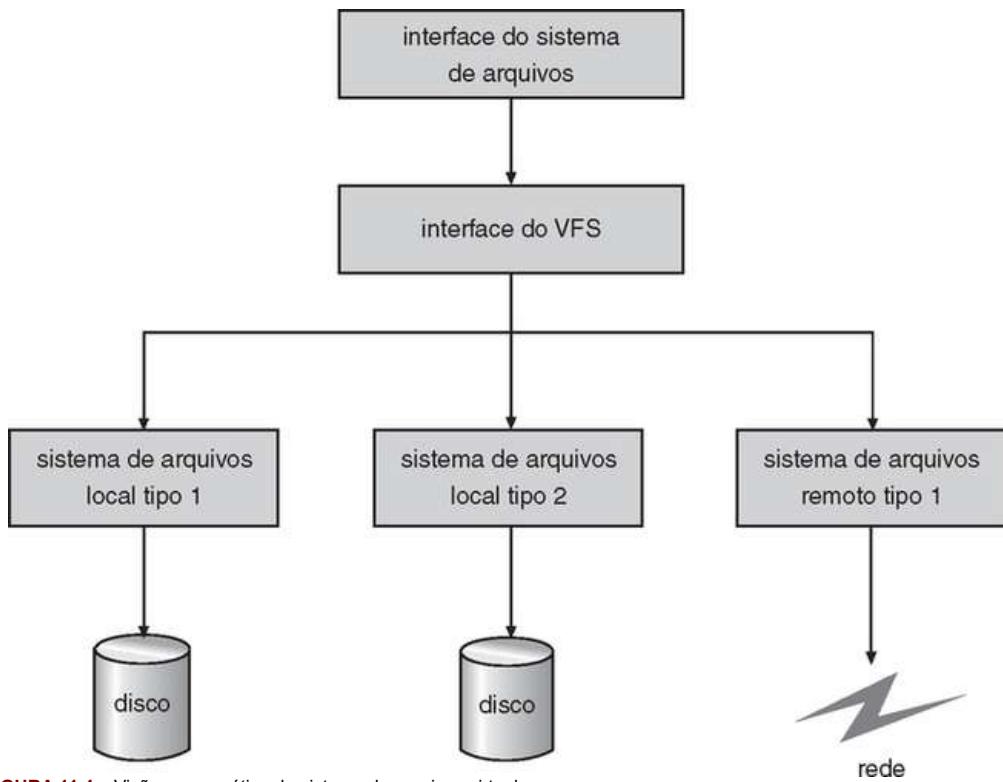


FIGURA 11.4 Visão esquemática do sistema de arquivos virtual.

A segunda camada é chamada **sistema de arquivos virtual (Virtual File System - VFS)**. A camada VFS possui duas funções importantes:

1. Ela separa as operações genéricas do sistema de arquivos de sua implementação, definindo uma interface VFS limpa. Várias implementações para a interface VFS poderão coexistir na mesma máquina, permitindo o acesso transparente a diferentes tipos de sistemas de arquivos montados localmente.
2. O VFS provê um mecanismo para representar exclusivamente um arquivo por toda a rede. O VFS é baseado em uma estrutura de representação de arquivo, chamada **vnode**, que contém um designador numérico para um arquivo exclusivo em toda a rede. (Os inodes do UNIX são exclusivos apenas dentro de um único sistema de arquivos.) Essa exclusividade em toda a rede é obrigatória para o suporte a sistemas de arquivos de rede. O kernel mantém uma estrutura de vnode para cada nó ativo (arquivo ou diretório).

Assim, o VFS distingue os arquivos locais dos remotos, e os arquivos locais são distinguidos ainda mais de acordo com seus tipos de sistema de arquivos.

O VFS ativa operações específicas do sistema de arquivos para lidar com requisições locais, de acordo com seus tipos de sistema de arquivos e até mesmo chama os procedimentos do protocolo NFS para requisições remotas. Os descritores de arquivo são construídos a partir dos vnodes relevantes e são passados como argumentos para esses procedimentos. A camada que implementa o tipo do sistema de arquivos, ou protocolo do sistema de arquivos remoto, é a terceira camada da arquitetura.

Vamos examinar rapidamente a arquitetura do VFS no Linux. Os quatro principais tipos de objetos definidos pelo VFS do Linux são:

- O **objeto inode**, que representa um arquivo individual.
- O **objeto file**, que representa um arquivo aberto.
- O **objeto superblock**, que representa um sistema de arquivos inteiro.
- O **objeto dentry**, que representa uma entrada de diretório individual.

Para cada um desses quatro tipos de objetos, o VFS define um conjunto de operações que precisam ser implementadas. Cada objeto de um desses tipos contém um ponteiro para uma tabela de função. A tabela de função lista os endereços das funções reais que implementam as operações definidas para esse objeto em particular. Por exemplo, uma API abreviada para algumas das operações para o objeto file inclui:

- int open(...) - Abre um arquivo.
- ssize_t read(...) - Lê de um arquivo.
- ssize_t write(...) - Escreve em um arquivo.
- int mmap(...) - Mapeia um arquivo na memória.

Uma implementação do objeto file para um tipo de arquivo específico é exigida para implementar cada função especificada na definição do objeto file. (A definição completa do objeto file é especificada na struct `file_operations`, que está localizada no arquivo `/usr/include/linux/fs.h`.)

Assim, a camada de software do VFS pode realizar uma operação em um desses objetos chamando a função apropriada a partir da tabela de função do objeto, sem ter de saber com antecedência exatamente com que tipo de objeto está lidando. O VFS não sabe, ou não se importa em saber, se um inode representa um arquivo de disco, um arquivo de diretório ou um arquivo remoto. A função apropriada para a operação `read()` desse arquivo sempre estará no mesmo lugar em sua tabela de função, e a camada de software do VFS chamará essa função sem se importar em como os dados são realmente lidos.

11.3 Implementação do diretório

A seleção de algoritmos de alocação de diretório e gerenciamento de diretório afeta a eficiência, o desempenho e a confiabilidade do sistema de arquivos. Nesta seção discutimos as compensações envolvidas na escolha desses algoritmos.

11.3.1 Lista linear

O método mais simples de implementar um diretório é usar uma lista linear de nomes de arquivo com ponteiros para os blocos de dados. Esse método é simples de programar, mas demorado de executar. Para criar um novo arquivo, primeiro temos de pesquisar o diretório para ter certeza de que não existe um arquivo com o mesmo nome. Depois, incluímos uma nova entrada no final do diretório. Para excluir um arquivo, pesquisamos seu nome no diretório, depois liberamos o espaço alocado a ele. Para reutilizar a entrada do diretório, podemos fazer várias coisas. Podemos marcar a entrada como não usada (atribuindo um nome especial, como um nome com espaços em branco ou com um bit usado-não usado em cada entrada) ou então podemos acrescentá-la a uma lista de entradas de diretório livres. Uma terceira alternativa é copiar a última entrada no diretório para o local liberado e diminuir o tamanho do diretório. Uma lista interligada também pode ser usada para diminuir o tempo de exclusão de um arquivo.

A desvantagem real de uma lista linear de entradas de diretório é que encontrar um arquivo requer uma busca linear. As informações do diretório são usadas com frequência, e os usuários observarão se o acesso é lento. De fato, muitos sistemas operacionais implementam um cache de software para armazenar as informações de diretório usadas mais recentemente. Um acerto do cache evita a nova leitura constante das informações em disco. Uma lista classificada permite uma pesquisa binária e diminui o tempo médio da pesquisa. Contudo, o requisito de que a lista precisa ser mantida classificada pode complicar a criação e a exclusão de arquivos, pois podemos ter de mover quantidades substanciais de informação de diretório para manter um diretório classificado. Uma estrutura de dados em árvore mais sofisticada, como uma árvore B (B-tree), poderia ajudar nesse caso. Uma vantagem da lista classificada é que uma listagem de diretório classificada pode ser produzida sem uma etapa de classificação separada.

11.3.2 Tabela de hash

Outra estrutura de dados usada para um diretório de arquivo é uma **tabela de hash**. Aqui, uma lista linear armazena as entradas de diretório, mas uma estrutura de dados de hash também é usada. A tabela de hash apanha um valor calculado do nome do arquivo e retorna um ponteiro ao nome do arquivo na lista linear. Portanto, isso pode diminuir bastante o tempo de busca do diretório. A inserção e a exclusão também são bastante simples, embora seja necessário prever as **colisões** - situações em que dois nomes de arquivo são traduzidos para o mesmo local.

As principais dificuldades com uma tabela de hash são seu tamanho, em geral fixo, e a dependência da função de hash sobre esse tamanho. Por exemplo, suponha uma tabela de hash com sondagem linear, para manter 64 entradas. A função de hash converte os nomes de arquivo em inteiros de 0 a 63, por exemplo, usando o resto de uma divisão por 64. Se, mais tarde, tentarmos criar um 65º arquivo, teremos de ampliar a tabela de hash do diretório - digamos, para 128 entradas. Como resultado, precisamos de uma nova função de hash a fim de mapear nomes de arquivo para o intervalo de 0 a 127 e reorganizar as entradas de diretório existentes para refletir seus novos valores da função de hash.

Como alternativa, uma tabela de hash com estouro encadeado pode ser utilizada. Cada entrada de hash pode ser uma lista interligada, em vez de um valor individual, e podemos resolver as colisões acrescentando a nova entrada à lista interligada. As pesquisas podem ser um pouco mais lentas, pois a procura de um nome pode exigir a busca em uma lista interligada de entradas de tabela em colisão. Esse método provavelmente será mais rápido do que uma pesquisa linear pelo diretório inteiro.

11.4 Métodos de alocação

A natureza de acesso direto dos discos dá flexibilidade na implementação dos arquivos. Em quase todos os casos, muitos arquivos serão armazenados no mesmo disco. O problema principal é como alocar espaço para esses arquivos de modo que o espaço no disco seja utilizado de forma eficaz e os arquivos possam ser acessados com rapidez. Três métodos principais de alocação do espaço em disco estão em uso comum: **alocação contígua**, interligada e indexada. Cada método possui vantagens e desvantagens. Alguns sistemas (como o RDOS da Data General, para sua linha de computadores Nova) admitem todos esses métodos. Em geral, um sistema usará um método em particular para todos os arquivos.

11.4.1 Alociação contígua

A alocação contígua requer que cada arquivo ocupe um conjunto de blocos contíguos no disco. Os endereços de disco definem uma ordenação linear no disco. Com essa ordenação, supondo que somente uma tarefa esteja acessando o disco, o acesso ao bloco $b + 1$ após o bloco b não exige movimentação da cabeça de leitura/escrita. Quando o movimento da cabeça é necessário (do último setor de um cilindro para o primeiro setor do cilindro seguinte), a cabeça precisa apenas mover de uma trilha para a próxima. Assim, a quantidade de buscas de disco exigidas para acessar arquivos alocados consecutivamente é mínima, assim como o tempo de busca quando uma busca é necessária. O sistema operacional VM/CMS da IBM utiliza a alocação contígua porque ela fornece um bom desempenho.

A alocação contígua de um arquivo é definida pelo endereço e tamanho de disco (em unidades de bloco) do primeiro bloco. Se o arquivo tiver n blocos de extensão e começar no local b , ele ocupará os blocos $b, b + 1, b + 2, \dots, b + n - 1$. A entrada de diretório para cada arquivo indica o endereço do bloco inicial e o tamanho da área alocada para esse arquivo ([Figura 11.5](#)).

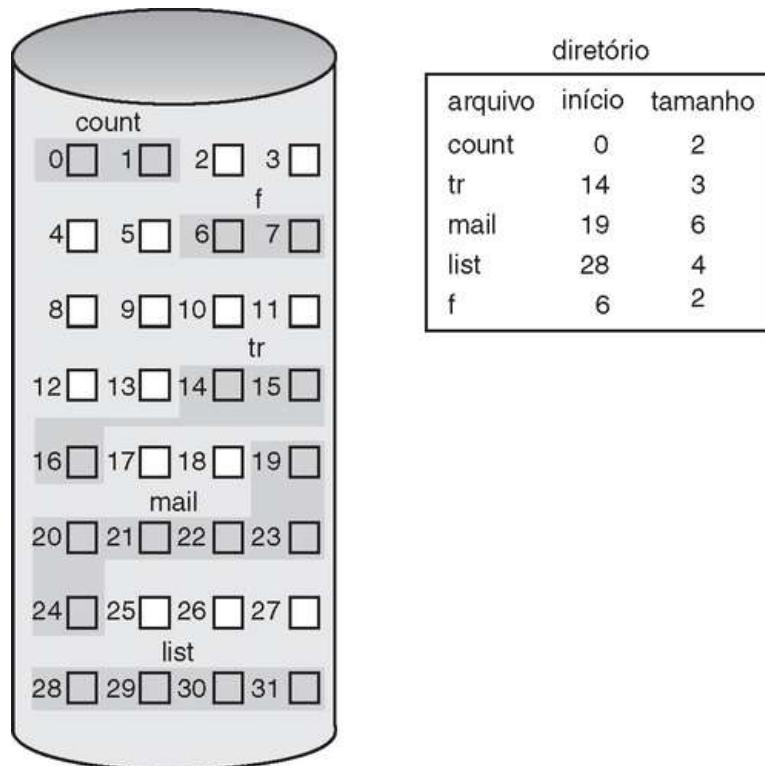


FIGURA 11.5 Alociação contígua do espaço em disco.

O acesso consecutivo a um arquivo é fácil. Para o acesso sequencial, o sistema de arquivos se lembra do endereço de disco do último bloco referenciado e, quando necessário, lê o bloco seguinte. Para o acesso direto ao bloco i de um arquivo que começa no bloco b , podemos acessar imediatamente o bloco $b + i$. Assim, os acessos sequencial e direto podem ser admitidos pela alocação contígua.

Contudo, a alocação contígua possui alguns problemas. Uma dificuldade é encontrar espaço para um novo arquivo. O sistema escolhido para gerenciar o espaço livre determina como essa tarefa é realizada (esses sistemas de gerenciamento são discutidos na [Seção 11.5](#)). Qualquer sistema de

gerenciamento pode ser usado, mas alguns são mais lentos do que outros.

O problema da alocação contígua de espaço em disco pode ser visto como uma aplicação em particular do problema da **alocação dinâmica de armazenamento**, discutido na [Seção 8.3](#), que envolve como satisfazer uma requisição de tamanho n a partir de uma lista de buracos livres. O first-fit e o best-fit são as estratégias mais comuns usadas para requisitar um buraco livre do conjunto de buracos disponíveis. As simulações têm mostrado que o first-fit e o best-fit são mais eficientes do que o worst-fit em termos do tempo e da utilização do armazenamento. Nem o first-fit nem o best-fit é particularmente melhor em termos de utilização de armazenamento, mas o first-fit é, em geral, mais rápido.

Todos esses algoritmos sofrem com o problema da **fragmentação externa**. À medida que os arquivos são alocados e excluídos, o espaço livre em disco é dividido em pequenos pedaços. A fragmentação externa existe sempre que o espaço livre é repartido em pedaços. Isso se torna um problema quando o maior trecho contíguo é insuficiente para uma requisição; o armazenamento é fragmentado em uma série de buracos, nenhum deles grande o suficiente para armazenar os dados. Dependendo da quantidade total de armazenamento de disco e do tamanho médio do arquivo, a fragmentação externa pode ser um problema pequeno ou grande.

Uma estratégia para evitar a perda de quantidades significativas de espaço em disco para a fragmentação externa é copiar um sistema de arquivos inteiro para outro disco ou para uma fita. O disco original é, então, liberado, criando um grande espaço livre contíguo. Depois copiamos os arquivos de volta para o disco original, alocando um espaço contíguo desse grande buraco. Esse esquema efetivamente **compacta** todo o espaço livre para um espaço contíguo, solucionando o problema da fragmentação. No entanto, o custo dessa compactação é o tempo, e isso pode ser particularmente prejudicial em discos rígidos grandes, que utilizam a alocação contígua, nos quais a compactação de todo o espaço pode levar horas e ter de ser feita semanalmente. Alguns sistemas exigem que essa função seja feita **off-line**, com o sistema de arquivos desmontado. Durante esse **tempo de paralisação** (ou **down time**), a operação normal do sistema não pode ser permitida, de modo que tal compactação é evitada a todo custo nas máquinas de produção. A maioria dos sistemas modernos que precisam de desfragmentação pode realizá-la **on-line** durante as operações normais do sistema, mas a penalidade do desempenho pode ser substancial.

Outro problema com a alocação contígua é determinar quanto espaço é necessário para um arquivo. Quando o arquivo é criado, a quantidade total de espaço de que ele precisará deve ser encontrada e alocada. Como o criador (programa ou pessoa) sabe o tamanho do arquivo a ser criado? Em alguns casos, essa determinação pode ser muito simples (copiando um arquivo existente, por exemplo); contudo, em geral, o tamanho de um arquivo de saída pode ser difícil de estimar.

Se alocarmos muito pouco espaço para um arquivo, podemos descobrir que ele não pode ser estendido. Especialmente com uma estratégia de alocação de best-fit, o espaço nos dois lados do arquivo pode estar em uso. Logo, não podemos tornar esse arquivo maior no local. Portanto, existem duas possibilidades. Primeiro, o programa do usuário pode ser terminado, com uma mensagem de erro apropriada. O usuário precisa, então, alocar mais espaço e executar o programa novamente. Essas execuções repetidas podem ser dispendiosas. Para evitá-las, o usuário pode estimar por cima a quantidade de espaço necessária, resultando em um considerável espaço desperdiçado. A outra possibilidade é encontrar um buraco maior, copiar o conteúdo do arquivo para o novo espaço e liberar o espaço anterior. Essa série de ações pode ser repetida, desde que exista espaço, embora isso possa ser demorado. Entretanto, o usuário nunca precisa ser informado explicitamente sobre o que está acontecendo; o sistema continua apesar do problema, embora cada vez mais lento.

Mesmo que a quantidade total de espaço necessária para um arquivo seja conhecida com antecedência, a pré-alocação pode ser ineficaz. Um arquivo que cresce lentamente por um período longo (meses ou anos) precisa receber espaço suficiente para o seu tamanho final, embora grande parte desse espaço possa não ser usada por um longo tempo. Portanto, o arquivo possui grande quantidade de fragmentação interna.

Para reduzir essas desvantagens, alguns sistemas operacionais utilizam um esquema modificado de alocação contígua. Aqui um trecho contíguo é alocado inicialmente e, depois, se esse espaço não for grande o bastante, outro trecho de espaço contíguo (uma **extensão**) é adicionado. O local dos blocos de um arquivo é então registrado como um local e um contador de bloco, com um link para o primeiro bloco da próxima extensão. Em alguns sistemas, o proprietário do arquivo pode definir o tamanho da extensão, mas essa configuração resulta em ineficiências se o proprietário estiver errado. A fragmentação interna ainda pode ser um problema se as extensões forem muito grandes, e a fragmentação externa pode ser um problema quando extensões de tamanhos variados forem alocadas e desalocadas. O Veritas File System comercial utiliza extensões para otimizar o desempenho. Ele é um substituto de alto desempenho para o UFS padrão do Unix.

11.4.2 Alocação interligada

A **alocação interligada** soluciona todos os problemas da alocação contígua. Com a alocação interligada, cada arquivo é uma lista interligada de blocos de disco; os blocos de disco podem estar espalhados em qualquer lugar no disco. O diretório contém um ponteiro para o primeiro e último

blocos do arquivo. Por exemplo, um arquivo com cinco blocos poderia começar no bloco 9, continuar no bloco 16, depois no bloco 1, bloco 10 e, finalmente, no bloco 25 (Figura 11.6). Cada bloco contém um ponteiro para o próximo bloco. Esses ponteiros não estão disponíveis para o usuário. Assim, se cada bloco tiver 512 bytes de tamanho e um endereço de disco (o ponteiro) exigir 4 bytes, então o usuário verá blocos de 508 bytes.

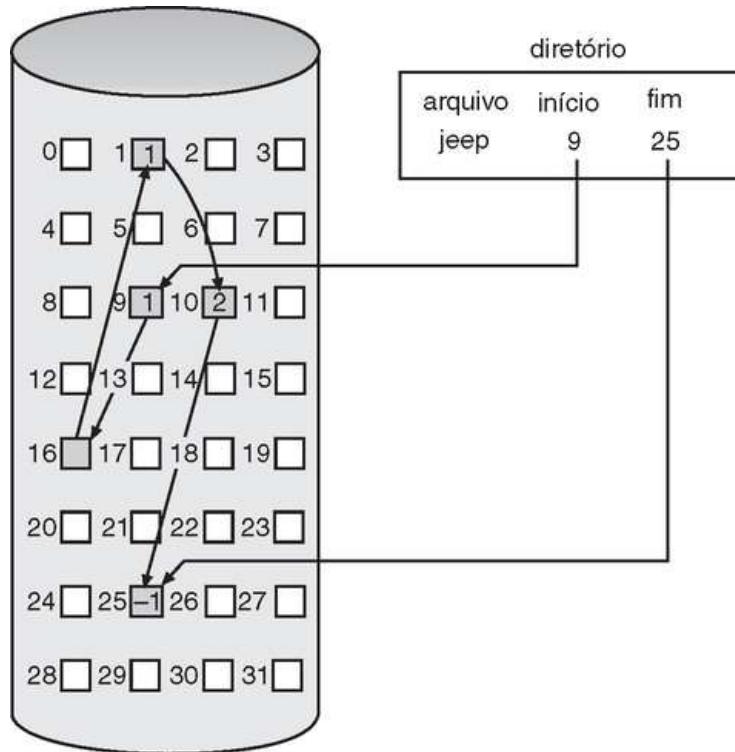


FIGURA 11.6 Alocação interligada do espaço em disco.

Para criar um novo arquivo, criamos uma nova entrada no diretório. Com a alocação interligada, cada entrada possui um ponteiro para o primeiro bloco de disco do arquivo. Esse ponteiro é inicializado com *nil* (o valor do ponteiro de fim de lista) para indicar um arquivo vazio. O campo de tamanho também é definido como 0. Uma escrita no arquivo faz o sistema de gerenciamento de espaço livre encontrar um bloco livre, e esse novo bloco é escrito e vinculado ao final do arquivo. Para ler um arquivo, lemos blocos seguindo os ponteiros de um bloco para outro. Não existe fragmentação externa com a alocação interligada, e qualquer bloco livre na lista de espaço livre pode ser usado para satisfazer uma requisição. O tamanho de um arquivo não precisa ser declarado quando ele é criado. Um arquivo pode crescer enquanto os blocos livres estão disponíveis. Como resultado, nunca é necessário compactar o espaço em disco.

Contudo, a alocação interligada possui desvantagens. O maior problema é que ela só pode ser usada de modo eficiente para os arquivos de acesso sequencial. Para encontrar o bloco de posição i de um arquivo, temos de começar no início desse arquivo e seguir os ponteiros até chegarmos ao bloco na posição i . Cada acesso a um ponteiro requer uma leitura de disco e, às vezes, uma busca de disco. Como consequência, não é eficiente prover suporte à capacidade de acesso direto para arquivos com alocação interligada.

Outra desvantagem é o espaço exigido para os ponteiros. Se um ponteiro precisa de 4 bytes de um bloco de 512 bytes, então 0,78% do disco está sendo usado para ponteiros, em vez de informações. Cada arquivo exige um pouco mais de espaço do que exigiria de outra forma.

A solução normal para esse problema é coletar blocos em múltiplos, chamados **clusters**, e alojar os clusters em vez de blocos. Por exemplo, o sistema de arquivos pode definir um cluster como quatro blocos e operar sobre o disco apenas em unidades de cluster. Os ponteiros, então, utilizam uma porcentagem muito menor do espaço em disco ocupado pelo arquivo. Esse método permite que o mapeamento de bloco lógico para físico permaneça simples, mas melhora o throughput do disco (pois menos buscas da cabeça do disco são necessárias) e diminui o espaço necessário para a alocação de bloco e gerenciamento da lista de espaço livre. O custo dessa técnica é um aumento na fragmentação interna, pois mais espaço é desperdiçado quando um cluster está parcialmente cheio do que quando um bloco está parcialmente cheio. Os clusters também podem ser usados para melhorar o tempo de acesso ao disco para muitos outros algoritmos, de modo que são usados na maioria dos sistemas operacionais.

Outro problema da alocação interligada é a confiabilidade. Lembre-se de que os arquivos estão

interligados por ponteiros espalhados por todo o disco, e considere o que aconteceria se um ponteiro estivesse perdido ou danificado. Um bug no software do sistema operacional ou uma falha no hardware do disco poderia resultar na escolha do ponteiro errado. Esse erro, por sua vez, poderia resultar na ligação com a lista de espaço livre com outro arquivo. Uma solução parcial é usar listas duplamente interligadas e outra é armazenar o nome do arquivo e o número relativo do bloco em cada bloco; porém, esses esquemas exigem ainda mais custo adicional para cada arquivo.

Uma variação importante da alocação interligada é o uso de uma **tabela de alocação de arquivos (File-Allocation Table - FAT)**. Esse método simples de alocação de espaço em disco, porém eficiente, é usado pelos sistemas operacionais MS-DOS e OS/2. Uma seção do disco, no início de cada volume, é separada para conter a tabela. A tabela possui uma entrada para cada bloco de disco e é indexada por número de bloco. A FAT é usada de modo bastante semelhante a uma lista interligada. A entrada do diretório contém o número do primeiro bloco do arquivo. A entrada da tabela indexada por esse número de bloco contém o número do próximo bloco no arquivo. Essa cadeia continua até alcançar o último bloco, que possui um valor especial de fim de arquivo como entrada na tabela. Um bloco não usado é indicado por um valor de tabela 0. A alocação de um novo bloco a um arquivo é uma simples questão de localizar a primeira entrada da tabela com valor 0 e substituir o valor anterior de fim de arquivo pelo endereço do novo bloco. O valor 0, em seguida, é substituído pelo valor de fim de arquivo. Um exemplo ilustrativo é a estrutura de FAT da [Figura 11.7](#), para um arquivo consistindo nos blocos 217, 618 e 339.

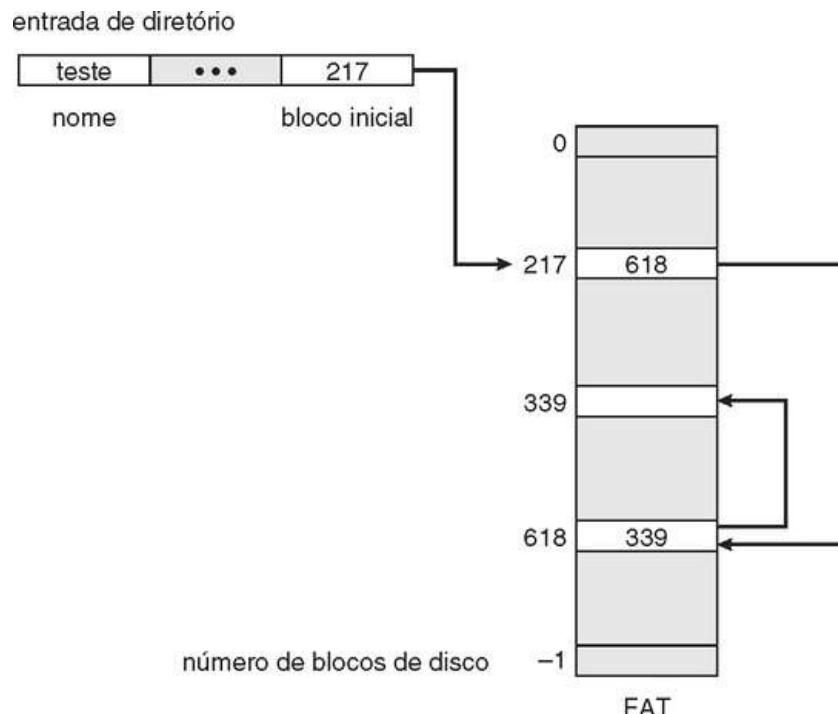


FIGURA 11.7 Tabela de alocação de arquivo.

O esquema de alocação da FAT pode resultar em uma quantidade significativa de buscas da cabeça do disco, a menos que a FAT esteja em cache. A cabeça do disco precisa se mover para o início da partição para ler a FAT e encontrar o local do bloco em questão, depois se mover para o local do bloco propriamente dito. Na pior das hipóteses, os dois movimentos ocorrem para cada um dos blocos. Um benefício é que o tempo de acesso aleatório é melhorado, pois a cabeça do disco pode encontrar o local de qualquer bloco lendo a informação na FAT.

11.4.3 Alocação indexada

A alocação interligada resolve os problemas de fragmentação externa e declaração de tamanho da alocação contígua. Todavia, na ausência de uma FAT, a alocação interligada não pode admitir um acesso direto eficiente, pois os ponteiros para os blocos estão espalhados com os blocos propriamente ditos espalhados por todo o disco, e precisam ser apanhados em ordem. A **alocação indexada** resolve esse problema reunindo todos os ponteiros em um único local: o **bloco de índice**.

Cada arquivo possui seu próprio bloco de índice, que é um array de endereços de bloco do disco. A entrada i do bloco de índice aponta para o bloco i do arquivo. O diretório contém o endereço do bloco de índice ([Figura 11.8](#)). Para encontrar e ler o bloco i , usamos o ponteiro para a entrada de bloco de índice i . Esse esquema é semelhante ao esquema de paginação descrito na [Seção 8.4](#).

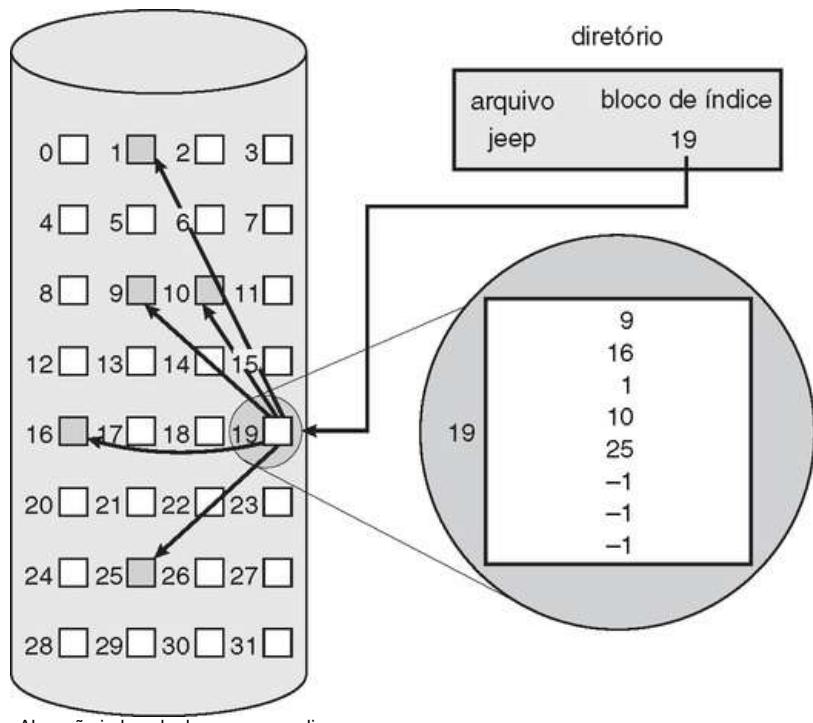


FIGURA 11.8 Alocação indexada do espaço em disco.

Quando o arquivo é criado, todos os ponteiros no bloco de índice são definidos como *nil*. Quando o bloco i é escrito inicialmente, um bloco é obtido do gerenciador de espaço livre e seu endereço é colocado na entrada de bloco de índice na posição i .

A alocação indexada admite acesso direto, sem sofrer com a fragmentação externa, pois qualquer bloco livre no disco pode satisfazer a uma requisição de mais espaço. Contudo, a alocação indexada sofre com o espaço desperdiçado. O custo adicional de ponteiro do bloco de índice em geral é maior do que o custo adicional de ponteiro da alocação interligada. Considere um caso comum em que temos um arquivo apenas com um ou dois blocos. Com a alocação interligada, perdemos o espaço somente de um ponteiro por bloco. Com a alocação indexada, um bloco de índice inteiro precisa ser alocado, mesmo que apenas um ou dois ponteiros sejam diferentes de *nil*.

Esse ponto levanta a questão do tamanho que o bloco de índice precisa ter. Cada arquivo precisa ter um bloco de índice, de modo que queremos que o bloco de índice seja o menor possível. No entanto, se o bloco de índice for muito pequeno, ele não poderá manter ponteiros suficientes para um arquivo grande e será preciso um mecanismo para lidar com essa questão. Os mecanismos para esse objetivo incluem:

■ **Esquema interligado (linked scheme).** Um bloco de índice normalmente é um bloco de disco.

Assim, ele pode ser lido e escrito por si só. Para levar em conta os arquivos grandes, podemos vincular vários blocos de índice. Por exemplo, um bloco de índice poderia conter um pequeno cabeçalho indicando o nome do arquivo e um conjunto dos 100 primeiros endereços de bloco de disco. O endereço seguinte (a última palavra no bloco de índice) é *nil* (para um arquivo pequeno) ou é um ponteiro para outro bloco de índice (para um arquivo grande).

■ **Índice multinível (multilevel index).** Uma variante da representação interligada é usar um bloco de índice de primeiro nível para apontar para um conjunto de blocos de índice de segundo nível, que por sua vez aponta para os blocos de arquivo. Para acessar um bloco, o sistema operacional utiliza o índice de primeiro nível para encontrar um bloco de índice de segundo nível e, então, usa esse bloco para encontrar o bloco de dados desejado. Essa técnica poderia ser continuada para um terceiro ou quarto nível, dependendo do tamanho máximo de arquivo desejado. Com blocos de 4.096 bytes, poderíamos armazenar 1.024 ponteiros de 4 bytes em um bloco de índice. Dois níveis de índice permitem 1.048.576 blocos de dados, dando lugar a um arquivo de até 4 GB.

■ **Esquema combinado (combined scheme).** Outra alternativa usada no UFS é manter, digamos, os primeiros 15 ponteiros do bloco de índice no inode do arquivo. Os 12 primeiros desses ponteiros apontam para **blocos diretos**, ou seja, eles contêm endereços de blocos que contêm dados do arquivo. Assim, os dados para arquivos pequenos (não mais do que 12 blocos) não precisam de um bloco de índice separado. Se um tamanho de bloco for de 4 KB, então até 48 KB de dados podem ser acessados diretamente. Os três ponteiros seguintes apontam para **blocos indiretos**. O primeiro ponteiro do bloco indireto é o endereço de um **bloco indireto único**, que é um bloco de índice, sem dados, mas com os endereços dos blocos que contêm dados. O segundo aponta para um **bloco indireto duplo**, que contém o endereço de um bloco que contém os endereços dos blocos que contêm ponteiros para os blocos de dados reais. O último ponteiro tem

o endereço de um **bloco indireto triplo**. Sob esse método, a quantidade de blocos que podem ser alocados a um arquivo ultrapassa a quantidade de espaço endereçável pelos ponteiros de arquivo em 4 bytes, usados por muitos sistemas operacionais. Um ponteiro de arquivo de 32 bits alcança apenas 2^{32} bytes ou 4 GB. Muitas implementações do UNIX, incluindo Solaris e AIX da IBM, agora admitem ponteiros de arquivo de até 64 bits. Os ponteiros desse tamanho permitem que arquivos e sistemas de arquivos tenham terabytes de tamanho. Um inode aparece na [Figura 11.9](#).

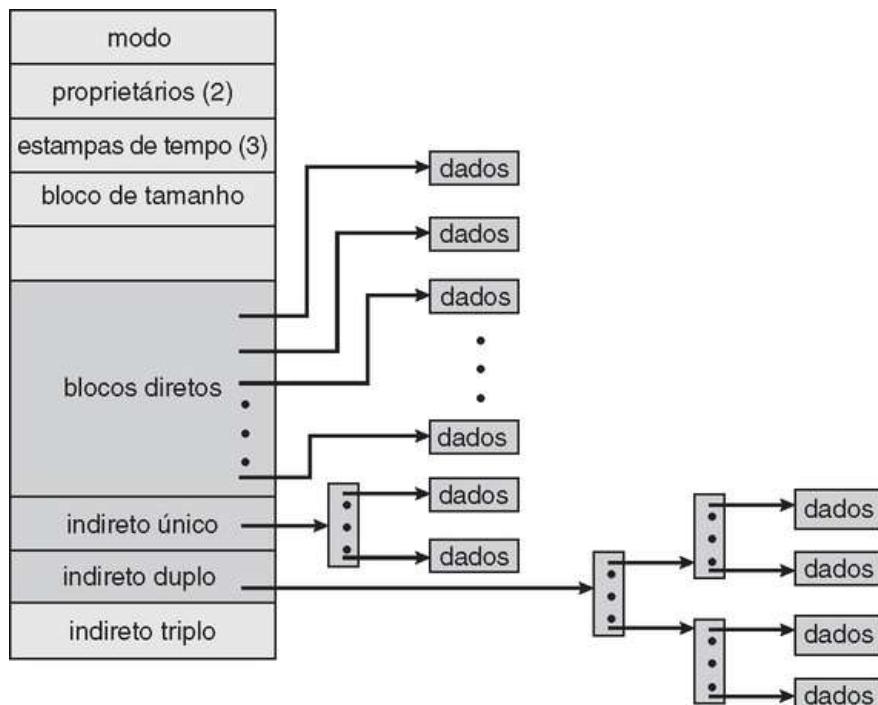


FIGURA 11.9 O inode do UNIX.

Os esquemas de alocação indexada sofrem dos mesmos problemas de desempenho da alocação interligada. Especificamente, os blocos de índice podem ser colocados em cache na memória, mas os blocos de dados podem estar espalhados por todo um volume.

11.4.4 Desempenho

Os métodos de alocação discutidos variam em sua eficiência de armazenamento e tempos de acesso ao bloco de dados. Ambos são critérios importantes na seleção do método ou dos métodos apropriados para um sistema operacional implementar.

Antes de selecionarmos um método de alocação, precisamos determinar como os sistemas serão usados. Um sistema com acesso principalmente sequencial não deve utilizar o mesmo método que um sistema com acesso principalmente aleatório.

Para qualquer tipo de acesso, a alocação contígua requer apenas um acesso para obter um bloco de disco. Como podemos manter o endereço inicial do arquivo na memória, podemos calcular o endereço do disco do bloco na posição i (ou o bloco seguinte) e lê-lo diretamente.

Para a alocação interligada, também podemos manter o endereço do próximo bloco na memória e lê-lo diretamente. Esse método serve para o acesso sequencial; porém, para o acesso direto, um acesso ao bloco i poderia exigir i leituras do disco. Esse problema indica por que a alocação interligada não deverá ser usada para uma aplicação que exige o acesso direto.

Como resultado, alguns sistemas admitem os arquivos de acesso direto usando a alocação contígua e arquivos de acesso sequencial pela alocação interligada. Para esses sistemas, o tipo de acesso a ser feito precisa ser declarado quando o arquivo é criado. Um arquivo criado para acesso sequencial será vinculado e não poderá ser usado para acesso direto. Um arquivo criado para acesso direto será contíguo e poderá admitir o acesso direto e o acesso sequencial, mas seu tamanho máximo terá de ser declarado quando criado. Nesse caso, o sistema operacional precisa ter estruturas de dados e algoritmos apropriados para admitir *ambos* os métodos de alocação. Os arquivos podem ser convertidos de um tipo para outro pela criação de um novo arquivo do tipo desejado, para o qual o conteúdo do arquivo antigo é copiado. O arquivo antigo pode, então, ser excluído, e o arquivo novo, renomeado.

A alocação indexada é mais complexa. Se o bloco de índice já estiver na memória, então o acesso

pode ser feito diretamente. Entretanto, a manutenção do bloco de índice na memória exige um espaço considerável. Se esse espaço de memória não estiver disponível, pode ser preciso ler o bloco de índice e depois o bloco de dados desejado. Para um índice de dois níveis, duas leituras de bloco de índice podem ser necessárias. Para um arquivo extremamente grande, o acesso a um bloco perto do final do arquivo exigiria a leitura de todos os blocos de índice antes de o bloco de dados necessário ser lido. Assim, o desempenho da alocação indexada depende da estrutura do índice, do tamanho do arquivo e da posição do bloco desejado.

Alguns sistemas combinam a alocação contígua com a alocação indexada, usando a alocação contígua para arquivos pequenos (até três ou quatro blocos) e passando automaticamente para uma alocação indexada se o arquivo crescer. Como a maioria dos arquivos tem tamanho pequeno e a alocação contígua é eficiente para esse tipo de arquivo, o desempenho médio pode ser muito bom.

Por exemplo, a versão do sistema operacional UNIX da Sun Microsystems foi alterada em 1991 para melhorar o desempenho no algoritmo de alocação do sistema de arquivos. As medidas de desempenho indicavam que o throughput máximo do disco em uma estação de trabalho típica (12-MIPS SPARCstation1) ocupava 50% da CPU e produzia uma largura de banda de disco de apenas 1,5 MB por segundo. Para melhorar o desempenho, a Sun fez mudanças para alocar espaço em clusters com tamanho de 56 KB sempre que possível (56 KB era o tamanho máximo de uma transferência do DMA na Sun, naquela época). Essa alocação reduziu a fragmentação externa e, assim, os tempos de busca e latência. Além disso, as rotinas de leitura de disco foram otimizadas para ler nesses clusters grandes. A estrutura de inode ficou inalterada. Como resultado dessas mudanças, mais o uso de read-ahead e free-behind (discutidos na [Seção 11.6.2](#)), 25% menos da CPU foi usado e o throughput foi substancialmente melhorado.

Muitas outras otimizações estão em uso. Dada a disparidade entre velocidade de CPU e disco, é válido acrescentar milhares de instruções extras ao sistema operacional para economizar apenas alguns movimentos na cabeça de leitura/escrita do disco. Além do mais, a disparidade está aumentando com o tempo, a ponto de centenas de milhares de instruções poderem ser usadas para otimizar os movimentos da cabeça.

11.5 Gerenciamento do espaço livre

Como o espaço em disco é limitado, precisamos reutilizar o espaço dos arquivos excluídos para os novos arquivos, se possível. (Os discos ótimos de única escrita só permitem uma escrita em qualquer setor e, por isso, essa reutilização não é fisicamente possível.) Para acompanhar o espaço livre no disco, o sistema mantém uma **lista de espaço livre**. Essa lista registra todos os blocos de disco *livres* – aqueles não alocados para algum arquivo ou diretório. Para criar um arquivo, pesquisamos a lista de espaço livre em busca da quantidade de espaço exigida e alocamos esse espaço para o novo arquivo. Esse espaço, então, é removido da lista de espaço livre. Quando o arquivo é excluído, seu espaço em disco é acrescentado à lista de espaço livre. A lista de espaço livre, apesar do nome, pode não ser implementada como uma lista, conforme discutiremos.

11.5.1 Vetor de bits

Frequentemente, a lista de espaço livre é implementada como um **mapa de bits** ou **vetor de bits**. Cada bloco é representado por 1 bit. Se o bloco estiver livre, o bit é 1; se o bloco estiver alocado, o bit é 0.

Por exemplo, considere um disco em que os blocos 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 e 27 estão livres e o restante dos blocos está alocado. O mapa de bits do espaço livre seria
00111100111110001100000011100000...

A principal vantagem dessa técnica é sua relativa simplicidade e eficiência na localização do primeiro bloco livre ou n blocos livres consecutivos no disco. Na realidade, muitos computadores fornecem instruções de manipulação de bits que podem ser usadas para essa finalidade. Por exemplo, a família Intel, a partir do 80386, e a família Motorola, a partir do 68020, possuem instruções que retornam o deslocamento em uma palavra do primeiro bit com o valor 1 (esses processadores possuem PCs poderosos e sistemas Macintosh, respectivamente). Uma técnica para localizar o primeiro bloco livre no sistema que usa o vetor de bits para alocar o espaço do disco é verificar sequencialmente cada palavra no mapa de bits para ver se esse valor não é 0, pois uma palavra de valor 0 possui todos os bits 0 e representa um conjunto de blocos alocados. A primeira palavra diferente de 0 é pesquisada em busca do primeiro bit 1, que é o local do primeiro bloco livre. O cálculo do número do bloco é

(número de bits por palavra) × (número de palavras com valor 0) + deslocamento do primeiro bit 1.

Mais uma vez, vemos recursos do hardware controlando a funcionalidade do software. Infelizmente, os vetores de bits somente serão eficientes se o vetor inteiro for mantido na memória principal (e gravado em disco ocasionalmente, para as necessidades de recuperação). Manter isso na memória principal é possível para discos menores, mas não necessariamente para os maiores. Um disco de 1,3 GB com blocos de 512 bytes precisaria de um mapa de bits com mais de 332 KB para acompanhar seus blocos livres, embora o agrupamento dos blocos em grupos de quatro reduza esse número para mais de 83 KB por disco. Um disco de 1 TB com blocos de 4 KB requer 32 MB para armazenar o mapa de bits. Visto que esse tamanho de disco aumenta constantemente, o problema com os vetores de bits continuará a escalar. Um disco de 1 PB exigiria um mapa de bits de 32 GB somente para gerenciar seu espaço livre.

11.5.2 Lista interligada

Outra técnica para o gerenciamento do espaço livre é interligar todos os blocos de disco livres, mantendo um ponteiro para o primeiro bloco livre em um local especial no disco e colocando-o em cache na memória. Esse primeiro bloco contém um ponteiro para o próximo bloco de disco livre, e assim por diante. Lembre-se de nosso exemplo ([Seção 11.5.1](#)), no qual os blocos 2, 3, 4, 5, 8, 9, 10, 11, 12, 13, 17, 18, 25, 26 e 27 estavam livres e o restante dos blocos estava alocado. Nessa situação, manteríamos um ponteiro para o bloco 2 como primeiro bloco livre. O bloco 2 teria um ponteiro para o bloco 3, que apontaria para o bloco 4, que apontaria para o bloco 5, que apontaria para o bloco 8, e assim por diante ([Figura 11.10](#)). Esse esquema não é eficiente; para atravessar a lista, temos de ler cada bloco, o que requer um tempo de E/S substancial. Felizmente, a travessia da lista de espaço livre não é uma ação frequente. Em geral, o sistema operacional precisa de um bloco livre para poder alocar esse bloco a um arquivo, de modo que utiliza o primeiro bloco na lista de espaço livre. O método FAT incorpora a contabilidade dos blocos livres à estrutura de dados de alocação. Não é preciso usar qualquer método separado.



FIGURA 11.10 Lista interligada de espaço livre no disco.

11.5.3 Agrupamento

Uma modificação da técnica de espaço livre é armazenar os endereços de n blocos livres no primeiro bloco livre. Os primeiros $n - 1$ desses blocos estão livres. O último bloco contém os endereços de outros n blocos livres, e assim por diante. Os endereços de uma grande quantidade de blocos livres podem ser localizados rapidamente, ao contrário da situação na qual a técnica-padrão de lista encadeada é usada.

11.5.4 Contagem

Outra técnica é tirar proveito do fato de que, em geral, vários blocos contíguos podem ser alocados ou liberados simultaneamente, em especial quando o espaço é alocado com o algoritmo de alocação contígua ou por meio do agrupamento. Assim, em vez de manter uma lista de n endereços de disco livres, podemos manter o endereço do primeiro bloco livre e o número (n) de blocos contíguos livres existentes após o primeiro bloco. Cada entrada na lista de espaço livre consiste em um endereço de disco e um contador. Embora cada entrada exija mais espaço do que um endereço de disco simples, a lista geral será mais curta, desde que o contador seja maior que 1. Observe que esse método de rastrear o espaço livre é semelhante ao método estendido de alocação de blocos. Essas entradas podem ser armazenadas em uma árvore B, em vez de uma lista interligada, para que haja pesquisa, inserção e exclusão eficientes.

11.5.5 Mapas de espaço

O sistema de arquivos ZFS da Sun foi projetado para compreender grandes quantidades de arquivos, diretórios e até mesmo sistemas de arquivos (no ZFS, podemos criar hierarquias de sistema de arquivos). As estruturas de dados resultantes poderiam ter sido grandes e ineficazes se não tivessem sido devidamente projetadas e implementadas. Nessas escalas, a E/S de metadados pode ter um grande impacto no desempenho. Considere, por exemplo, que se a lista de espaços livres for implementada como um mapa de bits, os mapas de bits precisam ser modificados tanto quando os blocos são alocados quanto quando são liberados. Liberar 1 GB de dados em um disco de 1 TB poderia fazer milhares de blocos de mapas de bits serem atualizados, pois esses blocos de dados poderiam estar espalhados pelo disco inteiro.

O ZFS usa uma combinação de técnicas em seu algoritmo de gerenciamento do espaço livre para controlar o tamanho das estruturas de dados e minimizar a E/S necessária para gerenciar essas estruturas. Primeiro, o ZFS usa **metaslabs** para dividir o espaço no dispositivo em pedaços com um tamanho de fácil manejo. Determinado volume pode conter centenas de metaslabs. Cada metaslab possui um mapa de espaços associado. O ZFS usa o algoritmo de contagem para armazenar

informações sobre blocos livres. Em vez de escrever estruturas de contagem em disco, ele usa técnicas de sistema de arquivos estruturadas em log para registrá-las. O mapa de espaço é um log de toda a atividade de bloco (alocando e liberando), em ordem cronológica e em formato de contagem. Quando o ZFS decide alocar ou liberar espaço de um metaslab, ele carrega o mapa de espaço associado na memória, em uma estrutura de árvore balanceada (para uma operação muito eficiente), indexada por deslocamento, e reproduz o log nessa estrutura. O mapa de espaço na memória é, então, uma representação precisa do espaço alocado e livre no metaslab. O ZFS também condensa o mapa ao máximo possível, combinando blocos livres contíguos em uma única entrada. Por fim, a lista de espaço livre é atualizada no disco como parte das operações orientadas para a transação do ZFS. Durante a fase de coleta e armazenamento, as solicitações de bloco ainda poderão ocorrer, e o ZFS satisfaz essas solicitações a partir do log. Basicamente, o log mais a árvore balanceada é a lista livre.

11.6 Eficiência e desempenho

Agora que já discutimos a respeito das opções de alocação de bloco e gerenciamento de diretórios, podemos considerar melhor seu efeito sobre o desempenho e o uso eficiente do disco. Os discos costumam ser um grande gargalo no desempenho do sistema, pois constituem o mais lento dos componentes principais do computador. Nesta seção, discutimos uma série de técnicas utilizadas para melhorar a eficiência e o desempenho do armazenamento secundário.

11.6.1 Eficiência

O uso eficiente do espaço em disco depende bastante dos algoritmos usados para alocação de disco e diretório. Por exemplo, os inodes do UNIX são pré-alocados em um volume. Até mesmo um disco “vazio” possui uma porcentagem do seu espaço perdida para os inodes. Contudo, pré-alocando os inodes e espalhando-os pelo volume, melhoramos o desempenho do sistema de arquivos. Esse desempenho melhorado resulta da alocação do UNIX e dos algoritmos de espaço livre, que tentam manter os blocos de dados do arquivo perto do bloco de inode desse arquivo, para reduzir o tempo de busca.

Como outro exemplo, vamos reconsiderar o esquema de agrupamento discutido na [Seção 11.4](#), que ajuda no desempenho da busca e transferência de arquivos, à custa da fragmentação interna. Para reduzir essa fragmentação, o BSD UNIX varia o tamanho do cluster à medida que um arquivo cresce. Grandes clusters são usados onde possam ser preenchidos, e pequenos clusters são usados para pequenos arquivos e para o último cluster de um arquivo.

Os tipos de dados mantidos na entrada do diretório de um arquivo (ou inode) também exigem consideração. Em geral, a “data da última escrita” é registrada para prestar informações ao usuário e determinar se o arquivo precisa de um backup. Alguns sistemas também mantêm a “data do último acesso”, para o usuário poder determinar quando o arquivo foi lido pela última vez. O resultado de manter essas informações é que, sempre que o arquivo for lido, um campo na estrutura de diretórios precisa ser escrito. Isso significa que o bloco seja lido para a memória, uma seção seja alterada e o bloco seja escrito de volta no disco, pois as operações sobre os discos ocorrem apenas em pedaços de bloco (ou cluster). Assim, sempre que um arquivo for aberto para leitura, sua entrada de diretório também precisa ser lida e escrita. Esse requisito pode ser ineficaz para arquivos acessados com frequência, de modo que precisamos pesar seu benefício contra seu custo de desempenho ao projetar um sistema de arquivos. Em geral, *cada* item de dados associado a um arquivo precisa ser considerado por seu efeito na eficiência e no desempenho.

Como exemplo, considere como a eficiência é afetada pelo tamanho dos ponteiros usados para acessar dados. A maioria dos sistemas utiliza ponteiros de 16 ou 32 bits por meio do sistema operacional. Esses tamanhos de ponteiro limitam o tamanho de um arquivo a 2^{16} (64 KB) ou 2^{32} bytes (4 GB). Alguns sistemas implementam ponteiros de 64 bits para aumentar esse limite para 2^{64} bytes, que é um número muito grande. No entanto, ponteiros de 64 bits ocupam mais espaço para armazenar e, por sua vez, fazem os métodos de alocação e gerenciamento de espaço livre (listas interligadas, índices, e assim por diante) utilizarem mais espaço em disco.

Uma das dificuldades na escolha de um tamanho de ponteiro ou, na verdade, de qualquer tamanho de alocação fixo dentro de um sistema operacional, é planejar os efeitos da mudança da tecnologia. Considere que o IBM PC XT tivesse um disco rígido de 10 MB e um sistema de arquivos do MS-DOS que podia admitir apenas 32 MB. (Cada entrada da FAT era composta por 12 bits, apontando para um cluster de 8 KB.) À medida que as capacidades de disco aumentavam, discos maiores tiveram de ser divididos em partições de 32 MB, porque o sistema de arquivos não podia acompanhar blocos além de 32 MB. Quando os discos rígidos com capacidade de mais de 100 MB se tornaram comuns, as estruturas de dados do disco e seus algoritmos no MS-DOS tiveram de ser modificados para permitir sistemas de arquivos maiores. (Cada entrada de FAT foi expandida para 16 bits e, mais tarde, para 32 bits.) As decisões iniciais do sistema de arquivos foram feitas por motivos de eficiência; porém, com o advento do MS-DOS versão 4, milhões de usuários de computador sofreram muitos transtornos quando tiveram de passar para o novo e maior sistema de arquivos. O sistema de arquivos ZFS da Sun utiliza ponteiros de 128 bits, que teoricamente nunca deveriam precisar ser estendidos. (A massa mínima de um dispositivo capaz de armazenar 2^{128} bytes usando o armazenamento em nível atômico seria de cerca de 272 trilhões de quilos.)

Como outro exemplo, considere a evolução do sistema operacional Solaris, da Sun. Originalmente, muitas estruturas de dados eram de tamanhos fixos, alocados no boot do sistema. Essas estruturas incluíam a tabela de processos e a tabela de arquivos abertos. Quando a tabela de processos se tornava cheia, outros processos não poderiam ser criados. Quando a tabela de arquivos se tornava cheia, outros arquivos não poderiam ser abertos. O sistema deixaria de fornecer serviços aos usuários. Esses tamanhos de tabela só poderiam ser aumentados pela recompilação do kernel e reinicialização do sistema. Desde o lançamento do Solaris 2, quase todas as estruturas do kernel são alocadas de maneira dinâmica, eliminando esses limites artificiais sobre o desempenho do sistema. Naturalmente, os algoritmos que manipulam essas tabelas são mais complicados, e o sistema

operacional é um pouco mais lento porque precisa alocar e desalocar entradas de tabela dinamicamente, mas esse é o preço comum para haver uma funcionalidade mais geral.

11.6.2 Desempenho

Desde que os algoritmos básicos do sistema de arquivos foram selecionados, ainda podemos melhorar o desempenho de várias maneiras. Como observaremos no [Capítulo 13](#), a maioria dos controladores de disco inclui memória local para formar um **cache** na placa, suficientemente grande para armazenar trilhas inteiras de cada vez. Quando uma busca é realizada, a trilha é lida para o cache de disco, começando no setor sob a cabeça do disco (reduzindo o tempo de latência). O controlador de disco, em seguida, transfere quaisquer requisições de setor para o sistema operacional. Quando os blocos passam do controlador de disco para a memória principal, o sistema operacional pode colocar os blocos no cache da memória.

Alguns sistemas mantêm uma seção separada da memória principal para um **cache do buffer**, na qual os blocos são mantidos sob a suposição de que serão usados novamente em breve. Outros sistemas colocam dados de arquivo em cache usando um **cache de página**. O cache de página usa técnicas de memória virtual para colocar dados de arquivo em cache como páginas, em vez de blocos orientados ao sistema de arquivos. O caching de dados de arquivo usando endereços virtuais é muito mais eficiente do que o caching de blocos de disco físicos, pois os acessos são ligados à memória virtual, em vez de ao sistema de arquivos. Vários sistemas, incluindo Solaris, Linux, Windows NT, 2000 e XP, utilizam caching de página para o cache de páginas de processo e de página, o que é conhecido como **memória virtual unificada**.

Algumas versões do UNIX proveem um **buffer cache unificado**. Para ilustrar os benefícios do buffer cache unificado, considere as duas alternativas para abrir e acessar um arquivo. Uma técnica é usar o mapeamento de memória ([Seção 9.7](#)), a segunda é usar as chamadas-padrão do sistema, `read()` e `write()`. Sem um buffer cache unificado, temos uma situação semelhante à vista na [Figura 11.11](#). Aqui, as chamadas de sistema `read()` e `write()` passam pelo buffer cache. A chamada de mapeamento de memória exige o uso de dois caches - o cache de página e o buffer cache. Um mapeamento de memória prossegue lendo blocos de disco do sistema de arquivos e armazenando-os no buffer cache. Como o sistema de memória virtual não pode realizar a interface com o buffer cache, o conteúdo do arquivo encontrado nele precisa ser copiado para o cache de página. Essa situação é conhecida como **duplo caching** e exige o caching dos dados do sistema de arquivos duas vezes. Isso não apenas desperdiça memória, mas também ciclos significativos de CPU e E/S, devido ao movimento de dados extra dentro da memória do sistema. Além disso, as incoerências entre os dois caches podem resultar na adulteração de arquivos. Quando um buffer cache unificado é oferecido, o mapeamento de memória e as chamadas de sistema `read()` e `write()` utilizam o mesmo cache de página. Isso tem o benefício de evitar o duplo caching e permite ao sistema de memória virtual gerenciar os dados do sistema de arquivos. O buffer cache unificado pode ser visto na [Figura 11.12](#).

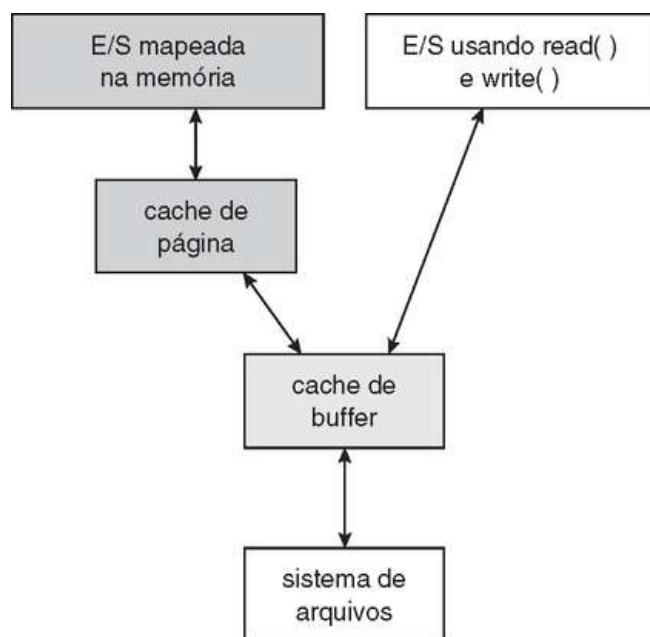


FIGURA 11.11 E/S sem um buffer cache unificado.

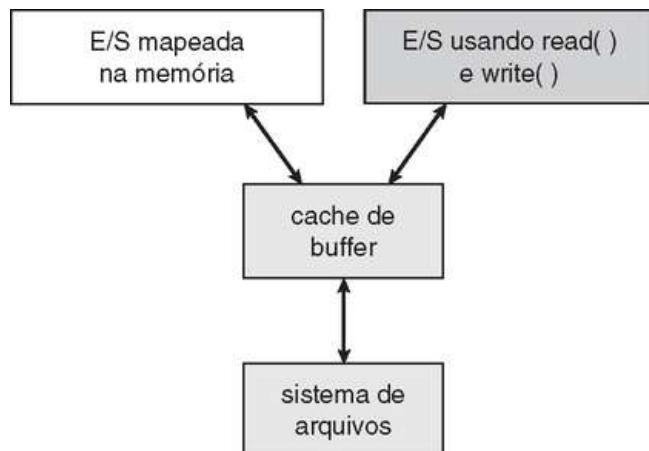


FIGURA 11.12 E/S usando um buffer cache unificado.

Não importa se estamos colocando blocos de disco ou páginas em cache (ou ambos), a LRU ([Seção 9.4.4](#)) parece ser um algoritmo de uso geral razoável para a substituição de bloco ou página. Contudo, a escolha de um algoritmo apropriado é difícil, como podemos observar ao considerarmos a evolução dos algoritmos de caching de página do Solaris. O Solaris permite que processos e o cache de página compartilhem memória não usada. Versões anteriores do Solaris 2.5.1 não faziam distinção entre alocar páginas a um processo ou ao cache de páginas. Como resultado, um sistema realizando muitas operações de E/S utiliza a maior parte da memória disponível para o caching de páginas. Devido às altas taxas de E/S, o verificador de páginas ([Seção 9.10.2](#)) recupera as páginas dos processos – em vez do cache de páginas – quando a memória livre está baixa. O Solaris 2.6 e o Solaris 7, opcionalmente, implementaram a *paginação por prioridade*, em que o verificador de páginas dá prioridade às páginas do processo em relação ao cache de páginas. O Solaris 8 acrescentou um limite fixo para as páginas de processo e o cache de páginas do sistema de arquivos, impedindo que um deles force o outro para fora da memória. O Solaris 9 e o 10 novamente mudaram os algoritmos para maximizar o uso da memória e minimizar o thrashing.

Outra questão que pode afetar o desempenho da E/S é se as escritas no sistema de arquivos ocorrem de forma síncrona ou assíncrona. As **escritas síncronas** ocorrem na ordem em que o subsistema de disco as recebe, e as escritas não são colocadas em buffer. Assim, a rotina de chamada precisa esperar até os dados alcançarem a unidade de disco antes de poder prosseguir. Em uma **escrita assíncrona**, os dados são armazenados no cache, e o controle retorna a quem chamou. Escritas assíncronas são realizadas na maior parte do tempo. No entanto, escritas de metadados, entre outras, podem ser síncronas. Os sistemas operacionais normalmente incluem um sinalizador na chamada de sistema `open` para permitir que um processo requisite que as escritas sejam realizadas de forma síncrona. Por exemplo, os bancos de dados utilizam esse recurso para transações indivisíveis, para garantir que os dados alcancem armazenamento estável na ordem exigida.

Alguns sistemas otimizam seu cache de página usando diferentes algoritmos de substituição, dependendo do tipo de acesso do arquivo. Um arquivo lido ou escrito sequencialmente não deve ter suas páginas substituídas na ordem LRU, pois a página usada mais recentemente será usada por último ou, talvez, nunca mais seja usada. Em vez disso, o acesso sequencial pode ser otimizado por técnicas conhecidas como **free-behind** e **read-ahead**. A free-behind remove uma página do buffer assim que a página seguinte é requisitada. As páginas já usadas provavelmente não serão mais usadas, e assim desperdiçam espaço no buffer. Com a read-ahead (**leitura antecipada**), uma página requisitada e várias páginas subsequentes são lidas e mantidas em cache. Essas páginas podem ser requisitadas após a página atual ser processada. A leitura desses dados do disco em uma transferência e sua manutenção em cache economizam um tempo considerável. Poderíamos pensar que um cache de trilha no controlador não eliminaria a necessidade da read-ahead sobre um sistema multiprogramado, no entanto, devido à alta latência e custo adicional envolvidos na realização de muitas transferências pequenas do cache de trilha para a memória principal, a realização de uma read-ahead continua sendo benéfica.

O cache de página, o sistema de arquivos e os drivers de disco possuem algumas interações interessantes. Quando os dados são escritos em um arquivo em disco, as páginas são colocadas em buffer no cache e o driver de disco classifica sua fila de saída de acordo com o endereço no disco. Essas duas ações permitem ao driver de disco reduzir as buscas da cabeça do disco e escrever dados em instantes otimizados conforme a rotação do disco. A menos que sejam necessárias escritas síncronas, um processo que esteja escrevendo no disco escreve no cache, e o sistema escreve os dados no disco de forma assíncrona, quando conveniente. O processo do usuário percebe escritas muito rápidas. Quando os dados são lidos de um arquivo em disco, o sistema de E/S em bloco realiza alguma leitura antecipada; porém, as escritas são muito mais próximas do assíncrono do que as leituras. Assim, a saída para o disco por meio do sistema de arquivos normalmente é mais rápida do

que a entrada para transferências grandes, ao contrário da intuição natural.

11.7 Recuperação

Os arquivos e os diretórios são mantidos na memória principal e no disco, devendo-se ter o cuidado de garantir que uma falha no sistema não resulte em perda de dados ou em incoerência nos dados. Tratamos dessas questões nesta seção, além de como um sistema pode recuperar-se desse tipo de falha.

Uma falha no sistema pode causar inconsistências entre as estruturas de dados do sistema de arquivos no disco, como as estruturas de diretório, os ponteiros de blocos livres e os ponteiros FCB livres. Em muitos sistemas de arquivos, as mudanças feitas aplicavam-se a essas estruturas no local. Uma operação típica, como a criação de arquivo, pode envolver muitas mudanças estruturais dentro do sistema de arquivos no disco. Estruturas de diretório são modificadas, FCBs e blocos de dados são alocados e a contagem do espaço livre para todos esses blocos é diminuída. Essas mudanças podem ser interrompidas por uma falha e ocorrer inconsistência nas estruturas. Por exemplo, o contador de FCB livre poderia indicar que um FCB foi alocado, mas a estrutura de diretórios poderia não apontar para o FCB. Para aumentar esse problema, existe o caching que os sistemas operacionais realizam para otimizar o desempenho da E/S. Se as mudanças em cache não alcançarem o disco antes que haja uma falha, poderá haver mais dados corrompidos.

Além das falhas, os bugs na implementação do sistema de arquivos, os controladores de discos e até mesmo as aplicações do usuário podem adulterar um sistema de arquivos. Os sistemas de arquivos possuem diversos métodos para lidar com esse problema, dependendo das estruturas de dados e dos algoritmos do sistema de arquivos. Trataremos dessas questões em seguida.

11.7.1 Verificação da consistência

Qualquer que seja a causa da adulteração, um sistema de arquivos precisa primeiro detectar os problemas e depois corrigi-los. Para a detecção, uma varredura de todos os metadados em cada sistema de arquivos pode confirmar ou negar a consistência do sistema. Infelizmente, essa varredura pode levar minutos ou horas, e deverá ocorrer toda vez que o sistema for inicializado. Como alternativa, um sistema de arquivos pode registrar seu estado dentro dos metadados do sistema de arquivos. No início de qualquer alteração de metadados, um bit de status é marcado para indicar que os metadados estão em andamento. Se todas as atualizações aos metadados forem realizadas com sucesso, o sistema de arquivos poderá limpar esse bit. Porém, se o bit de status permanecer marcado, um **verificador de consistência** é executado.

O verificador de consistência - um programa de sistemas como o fsck no UNIX ou o chkdsk no Windows - compara os dados na estrutura de diretório com os blocos de dados no disco e tenta reparar quaisquer inconsistências encontradas. Os algoritmos de gerenciamento de alocação e espaço livre ditam os tipos de problemas que o verificador pode encontrar e o sucesso que terá no reparo. Por exemplo, se a alocação interligada for usada e houver um vínculo de qualquer bloco para seu bloco seguinte, então o arquivo inteiro poderá ser reconstruído a partir dos blocos de dados, e a estrutura de diretório poderá ser recriada. Ao contrário, a perda de uma entrada de diretório em um sistema de alocação indexada poderia ser desastrosa, pois os blocos de dados não possuem conhecimento um do outro. Por esse motivo, o UNIX coloca as entradas de diretório em cache para as leituras, mas qualquer escrita que resulta em alocação de espaço ou outras mudanças nos metadados devem ser feitas de forma síncrona, antes de os blocos de dados correspondentes serem escritos. Naturalmente, ainda podem ocorrer problemas se uma escrita síncrona for interrompida por uma falha.

11.7.2 Sistemas de arquivo estruturados em log

Constantemente, os cientistas de computação constatam que as tecnologias e os algoritmos originalmente usados em uma área são também úteis em outras áreas de aplicação. Isso acontece com os algoritmos de recuperação baseados em log de banco de dados, descritos na [Seção 6.10.2](#). Esses algoritmos de log foram aplicados com sucesso ao problema da verificação de consistência. As implementações resultantes são conhecidas como sistemas de arquivos **orientados a transação baseados em log** (ou **diários**).

Observe que, com a técnica de verificação de consistência discutida na seção anterior, basicamente permitimos que as estruturas se rompam para repará-las na recuperação. Porém, existem vários problemas com essa técnica. Um problema é que a inconsistência pode ser irreparável. A verificação de consistência pode não ser capaz de recuperar as estruturas, resultando em perda de arquivos e até mesmo diretórios inteiros. Essa verificação pode exigir intervenção humana para resolver conflitos, e isso é inconveniente se nenhum ser humano estiver disponível. O sistema pode permanecer indisponível até um ser humano dizer ao sistema como prosseguir. Além disso, ela também toma tempo do sistema e do relógio. A verificação de terabytes de dados pode exigir horas.

A solução desse problema é aplicar as técnicas de recuperação baseadas em log para as

atualizações de metadados do sistema de arquivos. O NTFS e o sistema de arquivos Veritas utilizam esse método, e essa é uma opção para o UFS no Solaris. De fato, ele está se tornando comum em muitos sistemas operacionais.

Fundamentalmente, todas as mudanças de metadados são escritas em sequência em um log. Cada conjunto de operações que realiza uma tarefa específica é uma **transação**. Quando as mudanças são escritas nesse log, elas são consideradas confirmadas, e a chamada de sistema pode retornar ao processo do usuário, permitindo que ele continue a execução. Nesse meio-tempo, essas entradas de log são reproduzidas pelas estruturas reais do sistema de arquivos. Enquanto as mudanças são feitas, um ponteiro é atualizado para indicar quais ações foram concluídas e quais ainda estão incompletas. Quando uma transação confirmada inteira é concluída, ela é removida do arquivo de log, que, na realidade, é um **buffer circular**. Um buffer circular escreve no final do seu espaço e depois continua no início, escrevendo sobre os valores antigos enquanto prossegue. Não gostaríamos que o buffer escrevesse sobre os dados que ainda não foram salvos, de modo que esse cenário é evitado. O log pode estar em uma seção separada do sistema de arquivos ou em um eixo de disco separado. É mais eficiente, porém mais complexo, que o log esteja sob cabeças de leitura/escrita separadas, diminuindo assim a disputa pela cabeça e os tempos de busca.

Se um sistema falhar, o arquivo de log conterá zero ou mais transações. Quaisquer transações contidas nunca foram completadas para o sistema de arquivos, embora tenham sido confirmadas pelo sistema operacional, de modo que agora precisam ser concluídas. As transações podem ser executadas do ponteiro até o trabalho estar completo, de modo que as estruturas do sistema de arquivos permanecem consistentes. O único problema ocorre quando uma transação tiver sido abortada - ou seja, ela não foi confirmada antes de o sistema falhar. Quaisquer mudanças provenientes das transações aplicadas ao sistema de arquivos precisam ser desfeitas, preservando a consistência do sistema de arquivos. Essa recuperação é tudo o que é necessário após uma falha, eliminando todos os problemas com a verificação de consistência.

Um efeito colateral benéfico do uso do logging sobre as atualizações dos metadados de disco é que essas atualizações prosseguem muito mais rapidamente do que quando são aplicadas às estruturas de dados no disco. O motivo para essa melhoria é encontrado na vantagem do desempenho da E/S sequencial em relação à E/S aleatória. As dispendiosas escritas síncronas dos metadados aleatórios são transformadas em escritas sequenciais muito menos dispendiosas feitas à área de log dos sistemas de arquivos estruturados em log. Essas mudanças, por sua vez, são reproduzidas de forma assíncrona por meio de escritas aleatórias às estruturas apropriadas. O resultado geral é um ganho significativo no desempenho de operações orientadas a metadados, como criação e exclusão de arquivos.

11.7.3 Outras soluções

Outra alternativa para a verificação de consistência é empregada pelo sistema de arquivos WAFL da Network Appliance e pelo sistema de arquivos ZFS da Sun. Esses sistemas nunca sobrescrevem blocos com novos dados. Em vez disso, uma transação escreve todas as mudanças nos dados e metadados em novos blocos. Quando a transação é concluída, as estruturas de metadados que apontavam para as versões antigas desses blocos são atualizadas para que apontem para os novos blocos. O sistema de arquivos, então, pode remover os ponteiros antigos e os blocos antigos, tornando-os disponíveis para reutilização. Se os ponteiros e blocos antigos forem mantidos, um **snapshot** será criado; o snapshot é uma visão do sistema de arquivos antes que ocorresse a última atualização. Essa solução não deverá exibir verificação de consistência se a atualização do ponteiro for feita de forma indivisível. No entanto, WAFL possui um verificador de consistência, de modo que alguns cenários de falha ainda podem causar adulteração de metadados. (Veja, na [Seção 11.9](#), os detalhes do sistema de arquivos WAFL.)

O ZFS da Sun utiliza uma técnica ainda mais inovadora para a consistência do disco. Ele nunca sobrescreve blocos, como acontece com o WAFL. No entanto, o ZFS vai além disso e oferece soma de verificação de todos os blocos de metadados e dados. Essa solução (quando combinada com RAID) garante que os dados sempre estarão corretos. O ZFS, portanto, não possui verificador de consistência. (Outros detalhes sobre o ZFS podem ser encontrados na [Seção 12.7.6](#).)

11.7.4 Backup e restauração

Os discos magnéticos às vezes falham, devendo-se ter o cuidado de garantir que os dados perdidos nessa falha não o sejam para sempre. Para essa finalidade, os programas do sistema podem ser usados para o **backup** dos dados, do disco para outro dispositivo de armazenamento, como um disquete, fita magnética, disco óptico ou outro disco rígido. A recuperação da perda de um arquivo individual ou de um disco inteiro pode ser uma questão de **restauração** dos dados do backup.

Para reduzir a cópia necessária, podemos usar informações da entrada de diretório de cada arquivo. Por exemplo, se o programa de backup souber quando o último backup de um arquivo foi feito, e a data da última escrita de um arquivo, no diretório, indicar que o arquivo não foi alterado desde essa data, o arquivo não precisa ser copiado novamente. Uma agenda de backup típica

poderia ser a seguinte:

- **Dia 1.** Copiar para um meio de backup todos os arquivos do disco. Isso é chamado de **backup completo**.
- **Dia 2.** Copiar para outro meio todos os arquivos alterados desde o dia 1. Esse é um **backup incremental**.
- **Dia 3.** Copiar para outro meio todos os arquivos alterados desde o dia 2.
- ⋮
- ⋮

- **Dia N.** Copiar para outro meio todos os arquivos alterados desde o dia $N - 1$. Depois, voltar ao Dia 1.

O novo ciclo pode ter seu backup escrito em cima do anterior ou então em um novo conjunto de meios de backup. Dessa maneira, podemos restaurar um disco inteiro iniciando as restaurações com o backup completo e continuando em cada um dos backups incrementais. Naturalmente, quanto maior for N , maior o número de mídias que precisam ser lidas para uma restauração completa. Uma vantagem adicional desse ciclo de backup é que podemos restaurar qualquer arquivo excluído acidentalmente durante o ciclo, recuperando o arquivo excluído por meio do backup do dia anterior. O tamanho do ciclo é um meio-termo entre a quantidade de meios de backup necessária e a quantidade de dias desde quando uma restauração poderá ser feita. Para diminuir o número de fitas que precisam ser lidas para realizar uma restauração, uma opção é realizar um backup completo e depois, a cada dia, fazer o backup de todos os arquivos que foram alterados desde o último backup completo. Desse modo, uma restauração pode ser feita por meio do backup incremental mais recente e o backup completo, sem que quaisquer outros backups incrementais sejam necessários. O problema é que mais arquivos serão modificados a cada dia, de modo que cada backup incremental sucessivo envolve mais arquivos e mais mídia de backup.

Um usuário pode observar que determinado arquivo está faltando ou que foi adulterado muito tempo depois de o dano ter sido causado. Por esse motivo, normalmente planejamos realizar um backup completo de vez em quando, que será salvo "para sempre". É uma boa ideia armazenar esses backups permanentes longe dos backups regulares, para proteger contra danos, como um incêndio que destrua o computador e todos os backups também. E se o ciclo de backup reutilizar o meio de armazenamento, é preciso ter cuidado para não reutilizar o meio muitas vezes - se ele se desgastar, pode não ser possível restaurar dado algum dos backups.

11.8 NFS

Sistemas de arquivos de rede são muito comuns. Eles costumam se integrar à estrutura de diretórios gerais e realizar a interface com o sistema cliente. O NFS é um bom exemplo de um sistema de arquivos de rede cliente-servidor bastante usado e bem implementado. Aqui, vamos usá-lo como exemplo para explorar os detalhes de implementação dos sistemas de arquivos de rede.

O NFS é uma implementação e uma especificação de um sistema de software para acessar arquivos remotos por meio de LANs (ou até mesmo WANs). O NFS faz parte do ONC+, para o qual a maioria dos fornecedores de UNIX e alguns sistemas operacionais para PC dão suporte. A implementação descrita aqui faz parte do sistema operacional Solaris, que é uma versão modificada do UNIX SVR4, executando em estações de trabalho Sun e outros tipos de hardware. Ela utiliza o protocolo TCP ou UDP/IP (dependendo da rede de interconexão). A especificação e a implementação estão inter-relacionadas em nossa descrição do NFS. Sempre que um detalhe for necessário, vamos nos referir à implementação da Sun; sempre que a descrição for genérica, ela se aplica também à especificação.

Existem várias versões do NFS, sendo a última a versão 4. Aqui, descrevemos a versão 3, pois esta é a mais utilizada.

11.8.1 Visão geral

O NFS vê um conjunto de estações de trabalho interligadas como um conjunto de máquinas independentes, com sistemas de arquivos independentes. O objetivo é permitir algum grau de compartilhamento entre esses sistemas de arquivos (ou requisição explícita) de uma maneira transparente. O compartilhamento é baseado em um relacionamento cliente-servidor. Uma máquina pode ser, e normalmente é, tanto cliente quanto servidor. O compartilhamento é permitido entre qualquer par de máquinas. Para garantir a independência da máquina, o compartilhamento de um sistema de arquivos remoto afeta apenas a máquina cliente e nenhuma outra máquina.

Para que um diretório remoto seja acessível de uma máquina de maneira transparente - digamos, da M_1 -, um cliente dessa máquina precisa primeiro executar uma operação de montagem. A semântica da operação envolve a montagem de um diretório remoto em cima de um diretório de um sistema de arquivos local. Quando a operação de montagem estiver completada, o diretório montado se parecerá com uma subárvore integrante do sistema de arquivos local, substituindo a subárvore descendente do diretório local. O diretório local torna-se o nome da raiz do diretório recém-montado. A especificação do diretório remoto como um argumento para a operação de montagem não é feita de maneira transparente; o local (ou nome de host) do diretório remoto precisa ser fornecido. Contudo, desse ponto em diante, os usuários na máquina M_1 podem acessar os arquivos no diretório remoto de maneira totalmente transparente.

Para ilustrar a montagem de arquivos, considere o sistema de arquivos representado na [Figura 11.13](#), na qual os triângulos representam subárvores de diretórios que nos interessam. A figura mostra três sistemas de arquivos independentes de máquinas chamadas U , S_1 e S_2 . Nesse ponto, em cada máquina, somente os arquivos locais podem ser acessados. Na [Figura 11.14\(a\)](#), são mostrados os efeitos da montagem de $S_1:/usr/shared$ em cima de $U:/usr/local$. Essa figura representa a visão que os usuários em U têm do seu sistema de arquivos. Observe que depois que a montagem terminar, eles podem acessar qualquer arquivo dentro do diretório $dir1$ usando o prefixo $/usr/local/dir1$. O diretório original $/usr/local$ nessa máquina não estará mais visível.

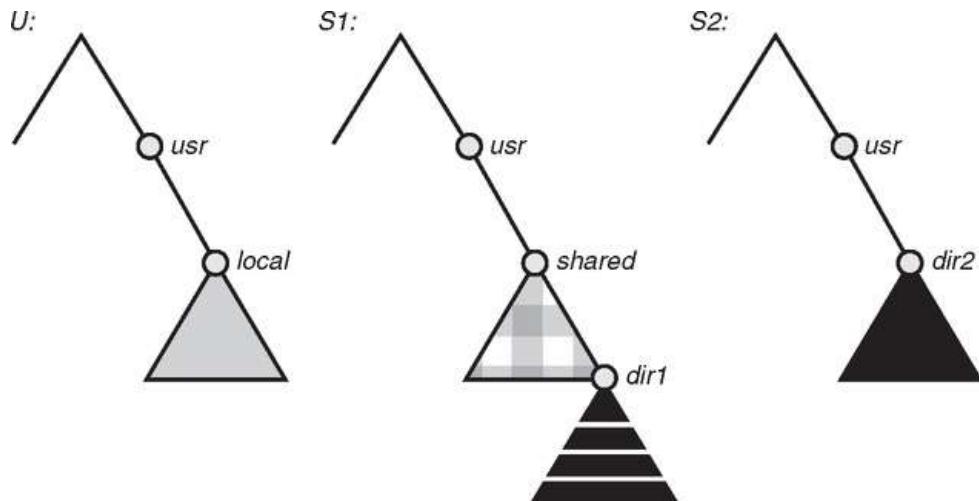


FIGURA 11.13 Três sistemas de arquivos independentes.

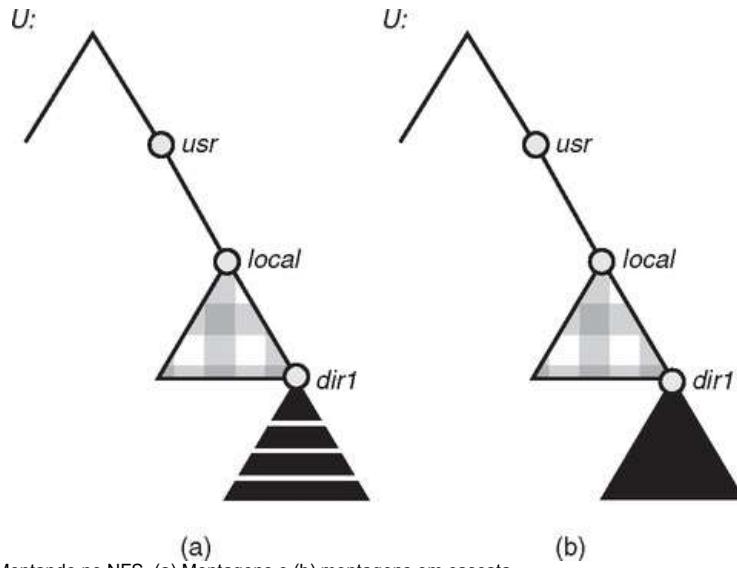


FIGURA 11.14 Montando no NFS. (a) Montagens e (b) montagens em cascata.

Sujeito à aprovação dos direitos de acesso, qualquer sistema de arquivos ou qualquer diretório dentro de um sistema de arquivos pode ser montado de forma remota em cima de qualquer diretório local. Estações de trabalho sem disco podem ainda montar suas próprias raízes a partir dos servidores. As montagens em cascata também são permitidas em algumas implementações NFS, ou seja, um sistema de arquivos pode ser montado em cima de outro sistema de arquivos montado remotamente, e não no local. Uma máquina é afetada somente pelas montagens invocadas por ela. Montar um sistema de arquivos remoto não dá ao cliente acesso a outros sistemas de arquivos que, por acaso, foram montados em cima do primeiro sistema de arquivos. Assim, o mecanismo de montagem não exibe uma propriedade de transitividade.

Na [Figura 11.14\(b\)](#), ilustramos a montagens em cascata, continuando nosso exemplo anterior. A figura mostra o resultado da montagem de $S2:/usr/dir2$ em cima de $U:/usr/local/dir1$, que já está montada remotamente de $S1$. Os usuários podem acessar arquivos dentro de $dir2$ em U usando o prefixo $/usr/local/dir1$. Se um sistema de arquivos compartilhado for montado em cima dos diretórios home de um usuário em todas as máquinas de uma rede, um usuário pode se conectar a qualquer estação de trabalho e obter seu ambiente home. Essa propriedade permite a **mobilidade do usuário**.

Um dos objetivos de projeto do NFS foi operar em um ambiente heterogêneo, com diferentes máquinas, sistemas operacionais e arquiteturas de rede. A especificação NFS é independente desse tipo de mídia e, portanto, incentiva outras implementações. Essa independência é alcançada por meio do uso de primitivas RPC montadas em cima de um protocolo External Data Representation (XDR) usado entre duas interfaces independentes de implementação. Em virtude disso, se o sistema consiste em máquinas e sistemas de arquivos heterogêneos, com a devida interface com o NFS, os sistemas de arquivos de diferentes tipos podem ser montados local e remotamente.

A especificação NFS distingue entre os serviços fornecidos por um mecanismo de montagem e os

serviços reais de acesso a arquivo remoto. Como resultado, dois protocolos separados são especificados para esses serviços: um protocolo de montagem e um protocolo para acessos a arquivos remotos, o **protocolo NFS**. Os protocolos são especificados como conjuntos de RPCs. Essas RPCs são os blocos de montagem usados para implementar o acesso transparente ao arquivo remoto.

11.8.2 O protocolo de montagem

O **protocolo de montagem** estabelece a conexão lógica inicial entre um servidor e um cliente. Na implementação da Sun, cada máquina possui um processo servidor, fora do kernel, realizando as funções do protocolo.

Uma operação de montagem inclui o nome do diretório remoto a ser montado e o nome da máquina servidora que o armazena. A requisição de montagem é mapeada para a RPC correspondente e é encaminhada ao servidor de montagem rodando na máquina servidora específica. O servidor mantém uma **lista de exportação** que especifica os sistemas de arquivos locais que ela exporta para montagem, junto com os nomes das máquinas que têm permissão para montá-los. (No Solaris, essa lista é /etc/dfs/dfstab, que só pode ser editada por um superusuário.) A especificação também pode incluir direitos de acesso, como "somente de leitura". Para simplificar a manutenção da lista de exportação e tabelas de montagem, um esquema de nomeação distribuído pode ser usado para manter essa informação e torná-la disponível aos clientes apropriados.

Lembre-se de que qualquer diretório dentro de um sistema de arquivos exportado pode ser montado remotamente por uma máquina autorizada. Uma unidade componente é um diretório desse tipo. Quando o servidor recebe uma requisição de montagem que esteja em conformidade com sua lista de exportação, ele retorna ao cliente um descritor de arquivo que serve como chave para outros acessos aos arquivos dentro do sistema de arquivos montado. O descritor de arquivo contém todas as informações de que o servidor precisa para distinguir um arquivo individual que armazena. Em termos de UNIX, o handle de arquivo consiste em um identificador do sistema de arquivos e um número de inode para identificar o exato diretório montado dentro do sistema de arquivos exportado.

O servidor também mantém uma lista de máquinas cliente e os diretórios correspondentes montados. Essa lista é usada principalmente para fins administrativos - por exemplo, para notificar a todos os clientes de que o servidor está parado. Apenas por meio da inclusão e exclusão de entradas nessa lista o estado do servidor pode ser afetado pelo protocolo de montagem.

Em geral, um sistema possui uma pré-configuração de montagem estática, estabelecida no momento do boot (/etc/vfstab no Solaris); porém, esse layout pode ser modificado. Além do procedimento de montagem propriamente dito, o protocolo de montagem inclui vários outros procedimentos, como desmontar e retornar lista de exportação.

11.8.3 O protocolo NFS

O protocolo NFS provê um conjunto de RPCs para operações com arquivo remoto. Os procedimentos admitem as seguintes operações:

- Procurar um arquivo dentro de um diretório.
- Ler um conjunto de entradas de diretório.
- Manipular links e diretórios.
- Acessar atributos de arquivo.
- Ler e escrever arquivos.

Esses procedimentos só podem ser invocados depois de ter sido estabelecido um descritor de arquivo para o diretório montado remotamente.

A omissão das operações open() e close() é intencional. Um recurso importante dos servidores NFS é que eles são *stateless* (*sem manutenção de estado*). Os servidores não mantêm informações sobre seus clientes de um acesso para outro. Não existem paralelos com a tabela de arquivos abertos ou estruturas de arquivo do UNIX no lado do servidor. Consequentemente, cada requisição precisa prover um conjunto completo de argumentos, incluindo um identificador de arquivo exclusivo e um deslocamento absoluto dentro do arquivo para as operações apropriadas. O projeto resultante é robusto; não são necessárias medidas especiais para recuperar um servidor após uma falha. As operações de arquivo precisam ser coerentes com essa finalidade. Cada requisição no NFS possui um número de sequência, permitindo ao servidor determinar se uma requisição é duplicata ou se existem requisições faltando.

A manutenção da lista de clientes, que mencionamos, parece violar a natureza stateless do servidor. Contudo, essa lista não é essencial para a operação correta do cliente ou do servidor e, por isso, não precisa ser restaurada após uma falha do servidor. Como resultado, poderia incluir dados inconsistentes e ser tratada apenas como palpite.

Outra implicação da filosofia de servidor stateless e um resultado da sincronia de uma RPC é que os dados modificados (incluindo blocos de indireção e status) precisam ser confirmados ao disco do servidor antes de os resultados retornarem ao cliente, ou seja, um cliente pode colocar blocos de

escrita em cache, mas, quando ele os esvaziar para o servidor, assumirá que alcançaram os discos do servidor. O servidor precisa escrever todos os dados do NFS de forma síncrona. Assim, uma falha e uma recuperação no servidor serão invisíveis a um cliente; todos os blocos que o servidor está gerenciando para o cliente estarão intactos. A consequente penalidade no desempenho pode ser grande, pois as vantagens do caching são perdidas. O desempenho pode ser aumentado por meio do armazenamento com seu próprio cache não volátil (normalmente, memória com bateria própria). O controlador de disco confirma a escrita no disco quando ela for armazenada no cache não volátil. Em essência, o host vê uma escrita síncrona muito veloz. Esses blocos permanecem intactos, mesmo depois de uma falha no sistema, e são transmitidos periodicamente desse armazenamento estável para o disco.

Uma única chamada de procedimento para escrita no NFS tem garantia de ser indivisível e também não se mistura com outras chamadas de escrita para o mesmo arquivo. Contudo, o protocolo NFS não provê mecanismos de controle de concorrência. Uma chamada de sistema write() pode ser dividida em várias escritas RPC, pois cada chamada de escrita ou leitura NFS pode conter até 8 KB de dados, e os pacotes UDP são limitados a 1.500 bytes. Como resultado, dois usuários escrevendo no mesmo arquivo remoto podem ter seus dados misturados. A reivindicação é que, como o gerenciamento de lock é inherentemente com estado, um serviço fora do NFS deverá prover o lock (e o Solaris fornece). Os usuários são aconselhados a coordenar o acesso aos arquivos compartilhados usando mecanismos fora do escopo do NFS.

O NFS é integrado ao sistema operacional por meio de um VFS. Como ilustração da arquitetura, vamos analisar como uma operação sobre um arquivo remoto já aberto é tratada (sigue o exemplo da Figura 11.15). O cliente inicia a operação por uma chamada de sistema regular. A camada do sistema operacional mapeia essa chamada a uma operação VFS sobre o vnode apropriado. A camada do VFS identifica o arquivo como sendo remoto e invoca o procedimento NFS apropriado. Uma chamada de RPC é feita à camada do serviço NFS no servidor remoto. Essa chamada é retornada à camada do VFS no sistema remoto, que descobre que ela é local e invoca a operação apropriada do sistema de arquivos. Esse caminho é rastreado para retornar o resultado. Uma vantagem dessa arquitetura é que o cliente e o servidor são idênticos; assim, uma máquina pode ser cliente, servidor ou ambos. O serviço real em cada servidor é realizado por várias threads do kernel.

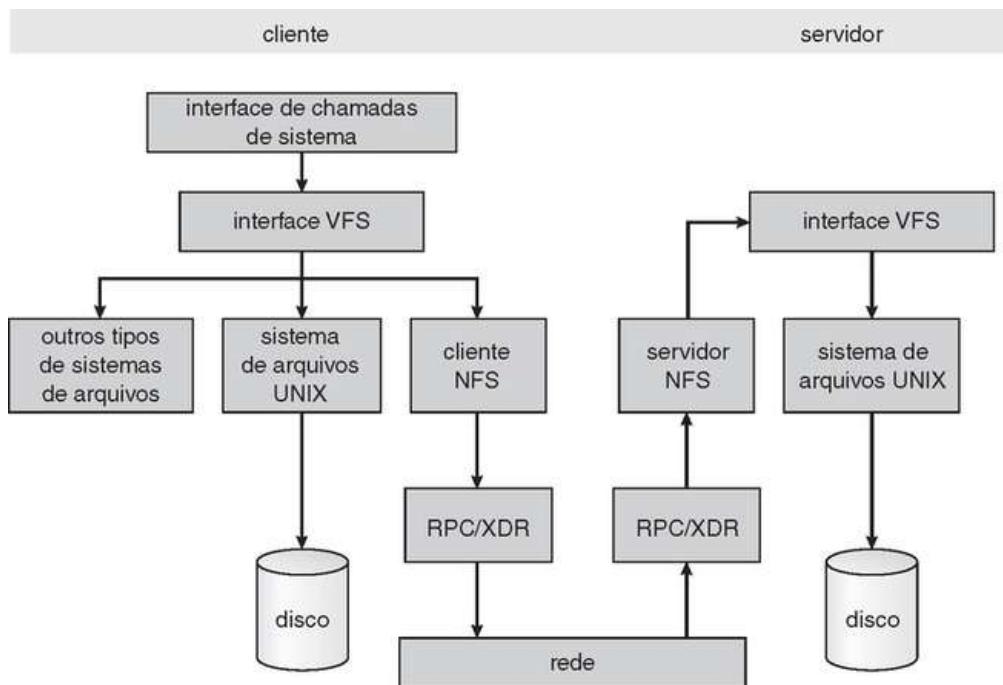


FIGURA 11.15 Visão esquemática da arquitetura do NFS.

11.8.4 Tradução de nome de caminho

A **tradução de nome de caminho** no NFS envolve o desmembramento do nome do caminho em entradas de diretório ou em componentes separados. Por exemplo, desmembramos o nome de caminho /usr/local/dir1/file.txt nos componentes (1) usr, (2) local e (3) dir1. Depois é realizada uma chamada lookup do NFS separada para cada par de nome de componente e vnode de diretório. Quando um ponto de montagem é atravessado, a pesquisa de cada componente causa uma RPC separada ao servidor. Esse esquema dispendioso de travessia de nome de caminho é necessário, pois

o layout de cada espaço de nomes lógico do cliente é exclusivo, ditado pelas montagens que o cliente realizou. Teria sido muito mais eficiente entregar um nome de caminho a um servidor e receber um vnode de destino quando um ponto de montagem fosse encontrado. Contudo, em qualquer ponto, pode haver outro ponto de montagem para o cliente específico, do qual o servidor stateless não está ciente.

Para a pesquisa ser rápida, um cache de pesquisa de nome de diretório no cliente mantém os vnodes para os nomes de diretório remotos. Esse cache agiliza as referências aos arquivos com o mesmo nome de caminho virtual. O cache de diretório é descartado quando os atributos retornados do servidor não combinam com os atributos do vnode em cache.

Lembre-se de que a montagem de um sistema de arquivos remoto em cima de outro sistema de arquivos remoto montado (montagens em cascata) é permitida em algumas implementações do NFS. Contudo, um servidor não pode atuar como um intermediário entre um cliente e outro servidor. Em vez disso, um cliente precisa estabelecer uma conexão cliente-servidor direta com o segundo servidor, montando o diretório desejado. Quando um cliente tem uma montagem em cascata, mais de um servidor pode estar envolvido na travessia do nome de caminho. Entretanto, a pesquisa de cada componente é realizada entre o cliente original e algum servidor. Portanto, quando um cliente realiza uma pesquisa sobre um diretório em que o servidor montou um sistema de arquivos, o cliente vê o diretório básico, em vez do diretório montado.

11.8.5 Operações remotas

Com a exceção da abertura e fechamento de arquivos, há uma correspondência quase biunívoca entre as chamadas de sistema regulares do UNIX para operações de arquivo e as RPCs do protocolo NFS. Assim, uma operação de arquivo remoto pode ser traduzida diretamente para a RPC correspondente. Em teoria, o NFS adere ao paradigma de serviço remoto, mas, na prática, as técnicas que utilizam buffers e caches são empregadas por questão de desempenho. Não existe uma correspondência direta entre uma operação remota e uma RPC. Em vez disso, os blocos de arquivo e os atributos de arquivo são apanhados pelas RPCs e são mantidos em cache local. As operações remotas futuras utilizam os dados em cache, sujeitos a restrições de coerência.

Existem dois caches: o cache de atributo de arquivo (informação de inode) e o cache de blocos de arquivo. Quando um arquivo é aberto, o kernel verifica com o servidor remoto para determinar se irá buscar ou revalidar os atributos em cache. Os blocos de arquivo em cache são usados apenas se os atributos em cache correspondentes estiverem atualizados. O cache de atributos é atualizado sempre que novos atributos chegam do servidor. Os atributos em cache, como padrão, são descartados após 60 segundos. As técnicas de leitura antecipada (read-ahead) e escrita adiada (delayed-write) são usadas entre o servidor e o cliente. Os clientes não liberam os blocos de escrita adiada até o servidor confirmar se os dados foram escritos em disco. A escrita adiada é retida mesmo quando um arquivo é aberto simultaneamente, com modos em conflito. Logo, a semântica do UNIX ([Seção 10.5.3.1](#)) não é preservada.

O ajuste do sistema para o desempenho torna difícil caracterizar a semântica de consistência do NFS. Novos arquivos criados em uma máquina podem não ser visíveis em outro lugar por 30 segundos. Além disso, escrever em um arquivo em um local pode ou não ser visível em outros locais que estão com esse arquivo aberto para leitura. Novas aberturas de um arquivo observam apenas as mudanças já submetidas ao servidor. Assim, o NFS não fornece nem simulação estrita da semântica do UNIX nem a semântica de sessão do Andrew ([Seção 10.5.3.2](#)). Apesar dessas desvantagens, a utilidade e o alto desempenho do mecanismo o tornam o mais usado sistema distribuído em múltiplos fornecedores em operação.

11.9 Exemplo: O sistema de arquivos WAFL

A E/S de disco possui grande impacto no desempenho do sistema. Como resultado, o projeto e a implementação do sistema de arquivos exigem muita atenção dos projetistas do sistema. Alguns sistemas de arquivos são de uso geral, pois podem fornecer desempenho e funcionalidade razoáveis para uma grande variedade de tamanhos de arquivo, tipos de arquivo e cargas de E/S. Outros são otimizados para tarefas específicas numa tentativa de fornecer melhor desempenho nessas áreas do que os sistemas de uso geral. O sistema de arquivos WAFL, da Network Appliance, é um exemplo desse tipo de otimização. O WAFL, *write-anywhere file layout*, é um sistema de arquivos poderoso e elegante, otimizado para escritas aleatórias.

O WAFL é usado exclusivamente em servidores de arquivos de rede produzidos pela Network Appliance, e foi criado para uso como um sistema de arquivos distribuído. Ele pode fornecer arquivos aos clientes por meio dos protocolos NFS, CIFS, ftp e http, embora tenha sido projetado apenas para o NFS e o CIFS. Quando muitos clientes utilizam esses protocolos para falar com um servidor de arquivos, o servidor pode ver uma demanda muito grande por leituras aleatórias e até mesmo uma demanda maior por escritas aleatórias. Os protocolos NFS e CIFS colocam em cache os dados das operações de leitura, de modo que as escritas normalmente são a maior preocupação para os criadores de servidor de arquivos.

O WAFL é usado em servidores de arquivos que incluem um cache NVRAM para escritas. Os projetistas do WAFL tiraram proveito da execução em uma arquitetura específica para otimizar o sistema de arquivos para E/S aleatória, com cache de armazenamento estável na frente. A facilidade de uso é um dos exemplos de orientação do WAFL, pois é projetado para ser usado em uma appliance. Seus criadores também o projetaram para incluir uma nova funcionalidade de snapshot, que cria múltiplas cópias somente de leitura do sistema de arquivos em diferentes pontos no tempo, conforme veremos.

O sistema de arquivos é semelhante ao Berkeley Fast File System, com muitas modificações. Ele é baseado em bloco e usa inodes para descrever arquivos. Cada inode contém 16 ponteiros para blocos (ou blocos indiretos) pertencentes ao arquivo descrito pelo inode. Cada sistema de arquivos possui um inode raiz. Todos os metadados residem em arquivos: todos os inodes estão em um arquivo, o mapa de blocos livres em outro, e o mapa de inodes livres em um terceiro, como mostra a [Figura 11.16](#). Como eles são arquivos-padrão, os blocos de dados não estão limitados no local e podem ser colocados em qualquer lugar. Se um sistema de arquivos for expandido pela inclusão de discos, os tamanhos desses arquivos de metadados são automaticamente expandidos pelo sistema de arquivos.

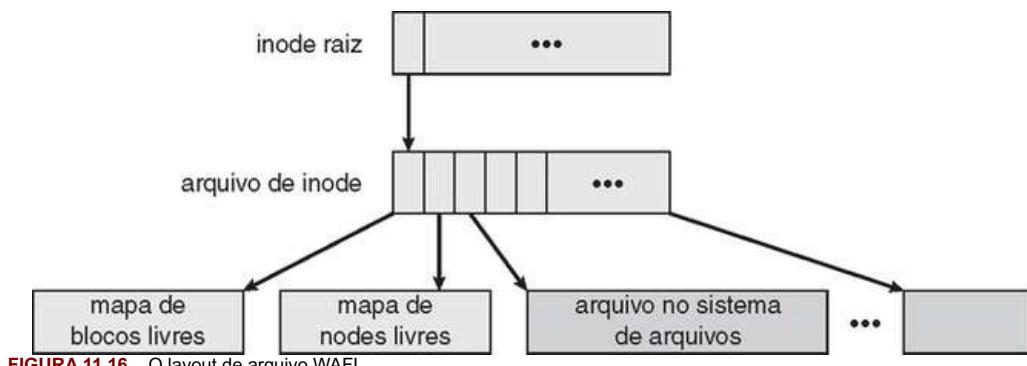


FIGURA 11.16 O layout de arquivo WAFL.

Assim, um sistema de arquivos WAFL é uma árvore de blocos com o inode raiz como sua base. Para tirar um snapshot, o WAFL cria uma cópia do inode raiz. Quaisquer atualizações de arquivos ou metadados depois disso vão para blocos novos, em vez de escrever sobre seus blocos existentes. O novo inode raiz aponta para metadados e dados alterados como resultado dessas escritas. Enquanto isso, o snapshot (o antigo inode raiz) ainda aponta para os blocos antigos, que não foram atualizados. Portanto, ele fornece acesso ao sistema de arquivos, como no instante em que o snapshot foi criado - e é preciso muito pouco espaço em disco para fazer isso! Basicamente, o espaço em disco extra ocupado por um snapshot consiste apenas nos blocos que foram modificados desde que o snapshot foi tirado.

Uma mudança importante dos sistemas de arquivos mais padronizados é que o mapa de blocos livres tem mais de um bit por bloco. Ele é um mapa de bits com um bit marcado para cada snapshot que está usando o bloco. Quando todos os snapshots que estiverem usando o bloco forem excluídos, o mapa de bits para esse bloco será composto de zeros e o bloco estará livre para ser reutilizado. Os blocos usados nunca são reescritos, de modo que as escritas são muito rápidas, pois uma escrita

pode ocorrer no bloco livre mais próximo do local atual da cabeça. Também existem muitas outras otimizações de desempenho no WAFL.

Muitos snapshots podem existir simultaneamente, de modo que pode ser tirado a qualquer hora do dia e a cada dia do mês. Um usuário com acesso a esses snapshots pode acessar arquivos conforme existiam em qualquer uma das horas em que os snapshots foram tirados. A facilidade de snapshot também é útil para backups, teste, versão, e assim por diante. A facilidade de snapshot do WAFL é muito eficiente porque nem sequer exige que as cópias da cópia na escrita de cada bloco de dados sejam tiradas antes que o bloco seja modificado. Outros sistemas de arquivos fornecem snapshots, mas frequentemente com menos eficiência. Os snapshots do WAFL são representados na Figura 11.17.

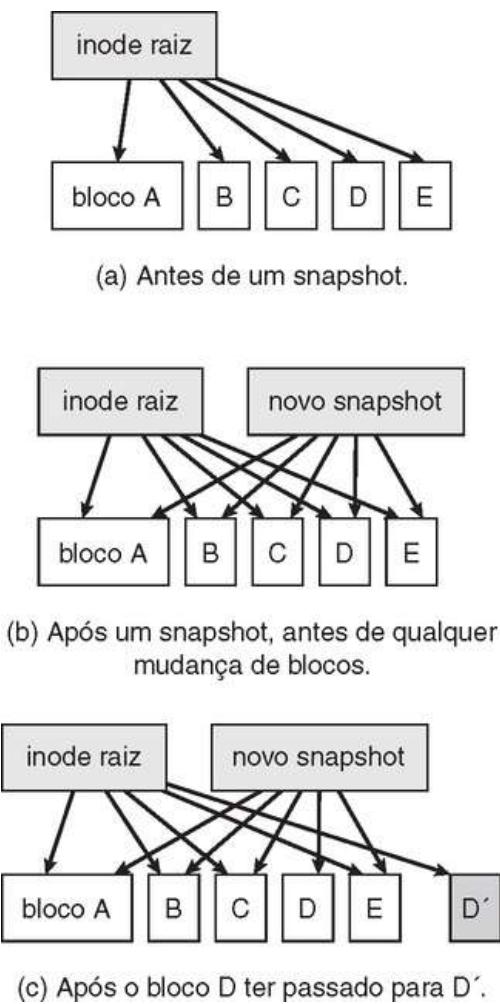


FIGURA 11.17 Snapshots no WAFL.

Versões mais novas do WAFL na verdade permitem snapshots de leitura e escrita, conhecidos como **clones**. Os clones também são eficientes, usando as mesmas técnicas dos snapshots. Neste caso, um snapshot somente de leitura captura o estado do sistema de arquivos, e um clone faz referência a esse snapshot somente de leitura. Quaisquer escritas no clone são armazenadas em novos blocos, e os ponteiros do clone são atualizados para que se refiram aos novos blocos. O snapshot original não é modificado, ainda dando uma visão para o sistema de arquivos como estava antes de o clone ser atualizado. Os clones também podem ser promovidos para substituir o sistema de arquivos original; isso envolve o descarte de todos os ponteiros antigos e quaisquer blocos antigos associados. Os clones são úteis para teste e upgrades, pois a versão original não é modificada e o clone é excluído quando o teste terminar ou se o upgrade falhar.

Outro recurso que vem naturalmente da implementação do sistema de arquivos WAFL é a **replicação**, a duplicação e o sincronismo de um conjunto de dados por uma rede para outro sistema. Primeiro, um snapshot de um sistema de arquivos WAFL é duplicado para outro sistema. Quando outro snapshot é tirado no sistema de origem, é relativamente fácil atualizar o sistema remoto enviando todos os blocos contidos no novo snapshot. Esses blocos são aqueles que foram alterados entre os momentos em que os dois snapshots foram tirados. O sistema remoto acrescenta esses blocos ao sistema de arquivos e atualiza seus ponteiros, e o novo sistema é então uma duplicata do sistema de origem no momento do segundo snapshot. A repetição desse processo

mantém o sistema remoto como uma cópia quase atualizada do primeiro sistema. Essa replicação é usada para recuperação de desastre. Se o primeiro sistema for destruído, a maior parte dos dados estará disponível para uso no sistema remoto.

Por fim, devemos observar que o sistema de arquivos ZFS da Sun admite snapshots, clones e replicação com eficiência semelhante.

11.10 Resumo

O sistema de arquivos reside no armazenamento secundário, projetado para manter grande quantidade de dados permanentemente. O meio de armazenamento secundário mais comum é o disco.

Os discos físicos podem ser segmentados em partições, para controlar o uso do meio e permitir vários sistemas de arquivos, possivelmente variáveis, por eixo de disco. Esses sistemas de arquivos são montados em uma arquitetura lógica de sistema de arquivos para torná-los disponíveis para uso. Os sistemas de arquivos normalmente são implementados em uma estrutura em camadas ou modular. Os níveis mais baixos lidam com as propriedades físicas dos dispositivos de armazenamento. Os níveis mais altos lidam com os nomes de arquivo simbólicos e as propriedades lógicas dos arquivos. Os níveis intermediários mapeiam os conceitos lógicos do arquivo em propriedades físicas do dispositivo.

Qualquer tipo de sistema de arquivos pode ter diferentes estruturas e algoritmos. Uma camada do VFS permite às camadas superiores tratarem de cada tipo de sistema de arquivos de maneira uniforme. Até mesmo os sistemas de arquivos remotos podem ser integrados à estrutura de diretório do sistema e agir por chamadas de sistema-padrão por meio da interface do VFS.

Os diversos arquivos podem receber espaço em disco de três maneiras: por meio de alocação contígua, interligada ou indexada. A alocação contígua pode sofrer de fragmentação externa. O acesso direto é bastante ineficaz com a alocação interligada. A alocação indexada pode exigir um overhead substancial para seu bloco de índice. Esses algoritmos podem ser otimizados de várias maneiras. O espaço contíguo pode ser ampliado por meio de extensões, para aumentar a flexibilidade e diminuir a fragmentação externa. A alocação indexada pode ser feita em clusters de vários blocos para aumentar o throughput e reduzir a quantidade de entradas de índice necessárias. A indexação em clusters grandes é semelhante à alocação contígua com extensões.

Os métodos de alocação de espaço livre também influenciam a eficiência de uso do espaço em disco, o desempenho do sistema de arquivos e a confiabilidade do armazenamento secundário. Os métodos usados incluem vetores de bits e listas interligadas. As otimizações incluem agrupamento, contadores e a FAT, que coloca a lista interligada em uma área contígua.

As rotinas de gerenciamento de diretório precisam considerar a eficiência, o desempenho e a confiabilidade. Uma tabela de hash é o método usado com mais frequência, por ser rápido e eficiente. Infelizmente, os danos a uma tabela ou uma falha no sistema podem resultar em inconsistência entre as informações de diretório e o conteúdo do disco. Um verificador de consistência pode ser utilizado para reparar os danos. As ferramentas de backup do sistema operacional permitem que os dados do disco sejam copiados para fita, permitindo ao usuário recuperar dados ou mesmo a perda de disco, devido à falha de hardware, bug do sistema operacional ou erro do usuário.

Os sistemas de arquivos de rede, como o NFS, utilizam a metodologia cliente-servidor para permitir que os usuários acessem arquivos e diretórios de máquinas remotas como se elas estivessem em sistemas de arquivos locais. As chamadas de sistema no cliente são traduzidas para protocolos de rede e retraduzidas para operações do sistema de arquivos no servidor. As redes e o acesso de múltiplos clientes criam desafios nas áreas de consistência de dados e desempenho.

Devido ao papel fundamental desempenhado pelos sistemas de arquivos na operação do sistema, seu desempenho e sua funcionalidade são cruciais. Técnicas como estruturas de log e caching ajudam a melhorar o desempenho, enquanto estruturas de log e RAID aumentam a confiabilidade. O sistema de arquivos WAFL é um exemplo da otimização do desempenho para combinar com uma carga de E/S específica.

Exercícios práticos

- 11.1. Considere um arquivo consistindo atualmente em 100 blocos. Suponha que o bloco de controle de arquivo (e o bloco de índice, no caso da alocação indexada) já esteja na memória. Calcule quantas operações de E/S de disco são necessárias para as estratégias de alocação contígua, interligada e indexada (único nível) se, para um bloco, as condições a seguir forem mantidas. No caso da alocação contígua, suponha que não haja espaço para crescimento no início, mas haja espaço para crescimento no final. Suponha também que a informação do bloco a ser acrescentado esteja armazenada na memória.
- a. O bloco é acrescentado no início.
 - b. O bloco é acrescentado no meio.
 - c. O bloco é acrescentado no final.
 - d. O bloco é removido do início.
 - e. O bloco é removido do meio.
 - f. O bloco é removido do final.
- 11.2. Que problemas poderiam ocorrer se um sistema permitisse que um sistema de arquivos fosse montado simultaneamente em mais de um local?
- 11.3. Por que o mapa de bits para a alocação de arquivos precisa ser mantido no armazenamento de massa, e não na memória principal?
- 11.4. Considere um sistema que oferece suporte para as estratégias de alocação contígua, interligada e indexada. Quais critérios deverão ser usados na decisão sobre qual estratégia é mais bem utilizada para determinado arquivo?
- 11.5. Um problema com a alocação contígua é que o usuário precisa pré-alocar espaço suficiente para cada arquivo. Se o arquivo crescer e se tornar maior que o espaço alocado para ele, ações especiais deverão ser tomadas. Uma solução para esse problema é definir uma estrutura de arquivos consistindo em uma área contígua inicial (de um tamanho especificado). Se essa área estiver cheia, o sistema operacional automaticamente define uma área de estouro que está ligada à área contígua inicial. Se a área de estouro estiver cheia, outra área de estouro é alocada. Compare essa implementação de um arquivo com as implementações contígua e interligada padrão.
- 11.6. Como os caches ajudam a melhorar o desempenho? Por que os sistemas não usam maiores caches ou caches maiores se eles são tão úteis?
- 11.7. Por que é vantajoso para o usuário que um sistema operacional aloque dinamicamente suas tabelas internas? Quais são as penalidades para o sistema operacional por fazer isso?
- 11.8. Explique como a camada VFS permite que um sistema operacional dê suporte a vários tipos de sistemas de arquivos com facilidade.

Exercícios

- 11.9. Considere um sistema de arquivos que usa um esquema de alocação contígua modificado, com suporte para extensões. Um arquivo é uma coleção de extensões, com cada extensão correspondendo a um conjunto de blocos contíguos. Um aspecto importante em tais sistemas é o grau de variabilidade no tamanho das extensões. Quais são as vantagens e as desvantagens dos esquemas a seguir?
- Todas as extensões têm o mesmo tamanho, e o tamanho é predeterminado.
 - As extensões podem ter qualquer tamanho e são alocadas dinamicamente.
 - As extensões podem ter alguns tamanhos fixos, e esses tamanhos são predeterminados.
- 11.10. Quais são as vantagens da variante de alocação interligada que usa uma FAT para encadear os blocos de um arquivo?
- 11.11. Considere um sistema onde o espaço livre é mantido em uma lista de espaço livre.
- Suponha que o ponteiro para a lista de espaço livre seja perdido. O sistema pode reconstruir a lista de espaço livre? Explique sua resposta.
 - Considere um sistema de arquivos semelhante ao que é usado pelo UNIX com a alocação indexada. Quantas operações de E/S de disco poderiam ser exigidas para ler o conteúdo de um pequeno arquivo local em `/a/b/c`? Suponha que nenhum dos blocos de disco esteja atualmente sendo mantido em cache.
 - Sugira um esquema para garantir que o ponteiro nunca seja perdido como resultado de falha na memória.
- 11.12. Alguns sistemas de arquivos permitem que o armazenamento em disco seja alocado em diferentes níveis de granularidade. Por exemplo, um sistema de arquivos poderia alocar 4 KB de espaço em disco como um único bloco de 4 KB ou como oito blocos de 512 bytes. Como poderíamos aproveitar essa flexibilidade para melhorar o desempenho? Que modificações teriam de ser feitas no esquema de gerenciamento de espaço livre a fim de dar suporte a esse recurso?
- 11.13. Discuta como as otimizações de desempenho para os sistemas de arquivos poderiam resultar em dificuldades na manutenção da consistência dos sistemas no caso de falhas do computador.
- 11.14. Considere um sistema de arquivos em um disco que tenha tamanhos de bloco lógico e físico de 512 bytes. Considere que as informações sobre cada arquivo já estejam na memória. Para cada uma das três estratégias de alocação (contígua, interligada e indexada), responda a estas perguntas:
- Como é realizado o mapeamento de endereço físico para lógico nesse sistema? (Para a alocação indexada, considere que um arquivo já tenha menos de 512 blocos de extensão.)
 - Se estivermos atualmente no bloco lógico 10 (o último bloco acessado foi o bloco 10) e quisermos acessar o bloco lógico 4, quantos blocos físicos precisam ser lidos do disco?
- 11.15. Considere um sistema de arquivos que utilize inodes para representar arquivos. Os blocos de disco possuem um tamanho de 8 KB, e um ponteiro para um bloco de disco requer 4 bytes. Esse sistema de arquivos possui 12 blocos de disco diretos, além de blocos de disco indiretos simples, duplo e triplo. Qual é o tamanho máximo de um arquivo que pode ser armazenado nesse sistema de arquivos?
- 11.16. A fragmentação em um dispositivo de armazenamento poderia ser eliminada pela nova compactação das informações. Os dispositivos de disco típicos não possuem registros de relocação ou de base (como os que são usados quando a memória deve ser compactada) e por isso como podemos relocar os arquivos? Indique três razões pelas quais a nova compactação e relocação de arquivos normalmente são evitadas.
- 11.17. Em que situações o uso da memória como um disco de RAM seria mais útil do que seu uso como um cache de disco?
- 11.18. Suponha que, no aumento em particular de um protocolo de acesso a arquivo remoto, cada cliente mantenha um cache de nomes que coloca em cache as traduções dos nomes de arquivo para descritores de arquivo correspondentes. Quais questões devemos levar em consideração na implementação do cache de nomes?
- 11.19. Explique por que o logging de metadados garante a recuperação de um sistema de arquivos após a falha de um sistema de arquivos.
- 11.20. Considere o seguinte esquema de backup:
- **Dia 1:** Copiar para um meio de backup todos os arquivos do disco.
 - **Dia 2:** Copiar para outro meio todos os arquivos alterados desde o dia 1.
 - **Dia 3:** Copiar para outro meio todos os arquivos alterados desde o dia 1.
- Isso é diferente da agenda dada na [Seção 11.7.4](#), pela qual todos os backups subsequentes copiam todos os arquivos modificados desde o primeiro backup completo. Quais são os benefícios desse sistema em relação ao da [Seção 11.7.4](#)? Quais são as desvantagens? As operações de restauração se tornam mais fáceis ou mais difíceis? Explique sua resposta.

Projetos de programação

Criando um sistema de arquivos

Este projeto consiste em criar e implementar um sistema de arquivos simples em cima de um disco simulado. A estrutura desse sistema de arquivos segue de perto os sistemas de arquivos baseados no UNIX, onde um superbloco é usado para descrever o estado do sistema de arquivos. Internamente, tal arquivo é armazenado no sistema usando uma estrutura de dados de inode. A estrutura do disco simulado aparece na [Figura 11.18](#).

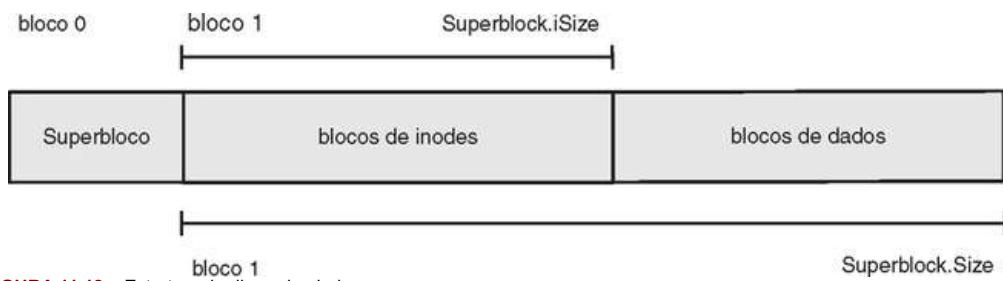


FIGURA 11.18 Estrutura do disco simulado.

Disco simulado

O disco simulado usa um arquivo chamado `DISK` para simular um disco com `NUM_BLOCKS` blocos de `BLOCK_SIZE` bytes por bloco. O disco simulado é representado com a classe Java `Disk.java` ([Figura 11.19](#)).

```
public class Disk {  
    public void read(int blocknum, byte[ ] buffer) { }  
    public void read(int blocknum, SuperBlock block) { }  
    public void read(int blocknum, InodeBlock block) { }  
    public void read(int blocknum, IndirectBlock block) { }  
    public void write(int blocknum, byte[ ] buffer) { }  
    public void write(int blocknum, SuperBlock block) { }  
    public void write(int blocknum, InodeBlock block) { }  
    public void write(int blocknum, IndirectBlock block) { }  
    public void stop( ) { }  
}
```

FIGURA 11.19 A classe Disk.

Primeiro, exploramos os três métodos a seguir nesta classe:

1. `void read(int blockNum, byte[] buffer)` - Lê do número de bloco especificado no disco para um buffer.
2. `void write(int blockNum, byte[] buffer)` - Escreve o conteúdo do buffer no número de bloco especificado no disco.
3. `int stop(boolean removeFile)` - Para o disco e informa quantas operações de leitura e escrita ocorreram. Se `removeFile` for `true`, esse método também excluirá o arquivo `DISK`.

Em cada um desses três métodos, `blockNum` precisa estar no intervalo `0..Disk.NUM_BLOCKS-1` e `buffer` precisa ser um array de bytes com exatamente `Disk.BLOCK_SIZE` bytes. Observe que os blocos precisam ser lidos e escritos em unidades de blocos inteiros, e não em partes de um bloco.

O construtor `Disk` procura um arquivo chamado `DISK` no diretório atual. Se não houver tal arquivo, o programa criará um e inicializará o primeiro bloco com valores 0. Todos os outros blocos precisam ser escritos pelo menos uma vez antes que o bloco possa ser lido.

As versões de overload de `read()` e `write()` são descritas mais adiante neste projeto.

Superbloco

O bloco 0 do disco é o **superbloco** que contém informações sobre o restante do disco. Você desejará manter uma cópia desse bloco na memória o tempo todo. Ele deve ser lido do arquivo `DISK` quando o

sistema de arquivos for iniciado e escrito de volta no arquivo `DISK` antes de encerrar. O superbloco mantém as seguintes variáveis:

1. `int size` - Número de blocos no sistema de arquivos.
2. `int iSize` - Número de blocos de índice.
3. `int freeList` - Primeiro bloco na lista de livres.

O tamanho do sistema de arquivos é registrado no superbloco para permitir que o sistema de arquivos use menos de um disco inteiro e dê suporte a vários tamanhos de disco. Em todas as estruturas de dados no disco, um **ponteiro** para um bloco de disco é um inteiro no intervalo `1..Disk.NUM_BLOCKS-1`. Como o bloco 0 (o superbloco) é tratado de forma exclusiva, você pode usar um número de bloco 0 para representar um ponteiro nulo. O superbloco é representado com o arquivo `SuperBlock.java`, que está disponível on-line (www.os-book.com).

Lista de espaço livre

O gerenciamento do espaço livre usará uma técnica de agrupamento semelhante à que descrevemos na [Seção 11.5.3](#). Especificamente, cada bloco da lista de blocos livres contém `Disk.POINTERS_PER_BLOCK` números de bloco, onde `Disk.POINTERS_PER_BLOCK = Disk.BLOCK_SIZE/4` (4 é o tamanho de um inteiro em bytes). O primeiro destes é o número de bloco do próximo bloco na lista de livres. As entradas restantes são números de bloco de blocos livres adicionais (cujo conteúdo é considerado insignificante). Enquanto o sistema está em execução, você desejará manter uma cópia do primeiro bloco da lista de livres na memória.

Estrutura de arquivos

Cada arquivo no sistema é descrito por um inode ([Figura 11.9](#)) e é definido pela seguinte classe:

```
class Inode {  
    // tamanho de um inode em bytes  
    public final static int SIZE = 64;  
    int flags;  
    int owner;  
    int fileSize;  
    public int pointer[ ] = new int[13];  
}
```

Se o campo `flags` for 0, ele será não usado. (Em um sistema de arquivos real, os bits desse inteiro distinguem ainda diferentes tipos de arquivo – diretórios, arquivos comuns, links simbólicos e coisas desse tipo – e indica permissões. Você não precisa implementar esses recursos. De modo semelhante, pode ignorar o campo `owner`.) O campo `fileSize` indica o tamanho atual do arquivo, em bytes.

O bloco 0 do disco é o superbloco, enquanto os blocos de 1 a `SuperBlock.iSize` são empacotados com blocos de inode. Cada bloco de inode contém `Disk.BLOCK_SIZE/Inode.SIZE` inodes. Os inodes são numerados consecutivamente a partir de 1 – e não 0! Logo, o bloco 1 do disco contém os inodes `1..Disk.BLOCK_SIZE/Inode.SIZE`, e assim por diante. Os arquivos são referenciados por esses números (chamados *inumbers*). Em um sistema de arquivos real, os arquivos de diretório são usados para traduzir nomes mnemônicos para inumbers, mas, para este projeto, usaremos inumbers diretamente. Representamos um bloco de inode com a seguinte classe:

```
class InodeBlock {  
    Inode node[ ] = new Inode[Disk.BLOCK_SIZE/Inode.SIZE];  
  
    public InodeBlock( ) {  
        for (int i = 0; i < Disk.BLOCK_SIZE/Inode.SIZE; i++)  
            node[i] = new Inode( );  
    }  
}
```

Os blocos restantes podem ser alocados como blocos diretos ou indiretos, ou colocados na lista de blocos livres. Eles são conhecidos coletivamente como **blocos de dados**. Os blocos de dados

contendo o conteúdo dos arquivos são chamados de **blocos diretos** (Figura 11.9). O array `pointer` na classe `Inode` aponta – direta ou indiretamente – para esses blocos. Os dez primeiros ponteiros apontam para blocos diretos. O décimo primeiro ponteiro – `pointer[10]` – aponta para um bloco indireto. Esse bloco indireto contém ponteiros para os próximos `Disk.BLOCK_SIZE/4` blocos diretos do arquivo. `pointer[11]` aponta para um bloco indireto duplo. Ele é preenchido com ponteiros para blocos indiretos, cada um contendo ponteiros para blocos de dados. De modo semelhante, o ponteiro final – `pointer[12]` – aponta para um bloco indireto triplo. É importante observar que o tamanho real do arquivo é determinado pelo campo `fileSize` do inode, e não pelos ponteiros.

Um ponteiro nulo (seja no inode ou em um dos blocos indiretos) pode indicar um buraco no arquivo. Por exemplo, se o campo `fileSize` indicar que deve haver cinco blocos, mas `pointer[2]==0`, então o terceiro bloco constitui um buraco. De modo semelhante, se o arquivo for grande o suficiente e `pointer[10]==0`, então os blocos de 11 a `POINTERS_PER_BLOCK + 10` são todos buracos. Uma tentativa de ler a partir de um buraco atua como se o arquivo estivesse preenchido com 0s; uma tentativa de escrever em um buraco faz o buraco ser preenchido. Os blocos são alocados conforme a necessidade e acrescentados ao arquivo. Os buracos são criados pela procura além do final do arquivo e depois pela escrita.

Outras operações com disco

As estruturas de dados `SuperBlock`, `InodeBlock` e `IndirectBlock` têm todos o mesmo tamanho, de modo que podem ser escritas ou lidas de qualquer bloco de disco. Por conveniência, fornecemos três versões de overload de `read()` e `write()` para a classe `Disk`, que permitem operações de leitura e escrita dos blocos de inode, blocos indiretos e superbloco.

Operações com o sistema de arquivos

Para este projeto, você precisa implementar os dez métodos na interface `FileSystem` ilustrados na Figura 11.20. Veja em seguida uma descrição de cada método:

```
public interface FileSystem {
    public int formatDisk(int size, int iSize);
    public int shutdown();
    public int create();
    public int open(int inumber);
    public int inumber(int fd);
    public int read(int fd, byte[ ] buffer);
    public int write(int fd, byte[ ] buffer);
    public int seek(int fd, int offset, int whence);
    public int close(int fd);
    public int delete(int inumber);
}
```

FIGURA 11.20 A interface `FileSystem`.

- `formatDisk()` inicializa o disco para o estado representando um sistema de arquivos vazio. Ele preenche o superbloco e vincula todos os blocos de dados na lista de blocos livres. O campo `size` representa o número de blocos de disco no sistema de arquivos, e `iSize` representa o número de blocos de inode.
- `shutdown()` fecha todos os arquivos abertos e encerra o disco simulado.
- `create()` cria um novo arquivo vazio e `open()` localiza um arquivo existente. Cada método retorna um inteiro no intervalo de 0 até 20, chamado **descritor de arquivo** (`fd` – de **file descriptor** – para abreviar). (O limite superior de 20 é razoável, pois geralmente um processo tem um número limitado de arquivos abertos de uma só vez.) O descritor de arquivo é um índice para um array chamado **tabela de descritor de arquivo**, representando os arquivos abertos. Cada entrada é associada a um arquivo aberto e também contém um **ponteiro de arquivo**, que é inicialmente definido como 0. O argumento de `open()` é o número de um arquivo existente. O método `inumber()` retorna o `inumber` do arquivo correspondente a um descritor de arquivo aberto.
- Os métodos `read()`, `write()`, `seek()` e `close()` se comportam de modo semelhante aos seus correspondentes no UNIX. O método `read()` lê até `buffer.length` bytes a partir do ponteiro de busca atual. O valor de retorno é o número de bytes lidos. Se houver menos do que `buffer.length` bytes entre o ponteiro de busca atual e o final do arquivo (conforme indicado pelo campo `size` na classe `Inode`), somente os bytes restantes serão lidos. Em particular, se o ponteiro de busca atual for maior ou igual ao tamanho do arquivo, `read()` retorna 0 e o `buffer` não é modificado. (O

ponteiro de busca atual não pode ser menor que 0.) O ponteiro de busca é incrementado pelo número de bytes lidos.

- O método `write()` transfere `buffer.length` bytes do `buffer` para o arquivo a partir do ponteiro de busca atual, avançando o ponteiro de busca por essa quantidade. Observe que permitimos que o ponteiro de busca seja maior que o tamanho do arquivo: nessa situação, buracos podem ser criados.
- O método `seek()` modifica o ponteiro de busca da seguinte forma:

```
public static final int SEEK_SET = 0;
public static final int SEEK_CUR = 1;
public static final int SEEK_END = 2;

. . .

switch (whence) {
    case SEEK_SET: seekPointer = offset; break;
    case SEEK_CUR: seekPointer += offset; break;
    case SEEK_END: seekPointer = file_size + offset; break;
}
```

No caso 0 (`SEEK_SET`), o deslocamento é relativo ao início do arquivo. No caso 1 (`SEEK_CUR`), o deslocamento é relativo ao ponteiro de busca atual. No caso 2 (`SEEK_END`), o deslocamento é relativo ao final do arquivo.

Para os casos 1 e 2, o valor do parâmetro `offset` pode ser positivo ou negativo; contudo, o ponteiro de busca resultante sempre precisa ser positivo ou 0. Se uma chamada a `seek()` resultar em um valor negativo para o ponteiro de busca, o ponteiro de busca fica inalterado, e a chamada retorna -1. Caso contrário, o valor retornado é o novo ponteiro de busca, representando a distância em bytes a partir do início do arquivo.

- O método `close()` escreve o inode no disco e libera a entrada da **tabela de arquivo**.
- O método `delete()` libera o inode e todos os blocos do arquivo. Uma tentativa de excluir o arquivo que está atualmente aberto resulta em erro.
- `shutdown()` fecha todos os arquivos abertos, esvazia todas as cópias de estruturas do disco na memória para o disco, chama o método `stop()` no disco e imprime informações de depuração e estatísticas.

Tabela de arquivo

A tabela de arquivo é uma estrutura de dados para registrar arquivos abertos. Para cada arquivo, você precisará de um ponteiro para uma cópia na memória de seu inode, seu número e seu ponteiro de busca atual. Contudo, é importante que você entenda totalmente como usá-lo a partir da interface `FileSystem`. Em particular, entenda os métodos para alocar e liberar slots nessa tabela, determinando se um descritor de arquivo é válido e acessando os dados associados a um descritor de arquivo.

A tabela de arquivo utiliza um array de bytes chamado `bitmap`, que indica se o índice especificado representa um arquivo aberto. `bitmap` é um array de 0s e 1s; 1 indica um arquivo aberto e 0 indica um slot aberto. A tabela de arquivo inclui um descritor de arquivo para cada arquivo. `FileDescriptor.java` (disponível on-line) contém dados sobre um arquivo, incluindo seu inode, `inumber` e ponteiro de busca atual. A tabela de arquivo mantém um array de objetos `FileDescriptor`. O índice no array `bitmap` corresponde ao objeto descritor de arquivo em `fdArray` com o mesmo índice (indicado pelo parâmetro `fd`). Cada objeto descritor de arquivo é uma instância da classe `FileDescriptor` que mantém informações sobre o arquivo especificado.

Dicas de implementação

Recomendamos desmembrar esse projeto em tarefas menores, de melhor manuseio. A seguir apresentamos uma estratégia que lista as tarefas individuais na ordem aproximada:

1. **Gerenciamento do espaço livre.** Escreva métodos para alocar e liberar blocos de disco.
Escreva também a parte de `formatDisk()` que monta a lista de livres em primeiro lugar.
2. **Acesso a bloco dentro de um arquivo.** Escreva um método que apanha um inode e um deslocamento de bloco dentro do arquivo e retorna um ponteiro para o número do bloco correspondente. O método deverá ter um argumento boolean `fillHole`, que especifica o que fazer se o bloco indicado não existir. Se o bloco não existir e `fillHole` for falso, o método deverá retornar 0; se `fillHole` for verdadeiro, o método deverá alocar um bloco, acrescentá-lo no arquivo e retornar seu número de bloco. A primeira versão é apropriada para `read()` e a

segunda é apropriada para `write()`. Você pode primeiro escrever e depois depurar esse método para pequenos arquivos, que podem ser armazenados inteiramente com blocos diretos. Depois, poderá modificá-lo para lidar também com grandes arquivos. Para arquivos grandes, se `fillHole` for verdadeiro, você pode ter que alocar um ou dois blocos indiretos e acrescentá-los ao arquivo.

3. **Acessando inodes.** Você precisará de métodos para ler um inode específico do disco (dado seu `inumber`) e escrever de volta um inode modificado. Lembre-se de que você só pode ler e escrever blocos inteiros, de modo que, para escrever um inode, terá de ler o bloco contendo o inode, modificar o inode e depois escrever o bloco de volta. Você também precisará de um método para alocar inodes.
4. **Lendo e escrevendo em intervalos arbitrários.** Neste ponto, a implementação de `read()` e `write()` deverá ser fácil. Uma requisição de leitura individual pode tocar em partes de vários blocos, de modo que você precisará de um loop que leia cada um dos blocos e copie a parte apropriada dele para a parte apropriada do argumento `buffer` da chamada de `read`. A implementação de `write()` é ligeiramente mais complicada; se um bloco for modificado apenas parcialmente, você terá de ler seu valor antigo, copiar os dados do buffer do cliente para a parte apropriada do bloco e depois escrever de volta.
5. **Diversos.** Preencha os métodos restantes da interface `FileSystem`. Talvez a única parte restante não trivial seja `delete()`, que precisa retornar todos os dados e blocos indiretos para a lista de livres e limpar o campo de flags do inode. Ele também precisa verificar se o arquivo não está atualmente aberto.

Teste

Você deverá testar completamente todos os dez métodos exigidos na sua implementação de `FileSystem`, incluindo criação, leitura, escrita, fechamento, reabertura e limpeza de todos os tipos de arquivos (pequenos, grandes, cheios de buracos e assim por diante). Você também deverá testar a verificação de erro no seu código.

Notas bibliográficas

O sistema FAT do MS-DOS foi explicado em [Norton e Wilton \[1988\]](#), e a descrição do OS/2 pode ser encontrada em [Iacobucci \[1988\]](#). Esses sistemas operacionais utilizam as CPUs Intel 8086 ([Intel \[1985b\]](#), [Intel \[1985a\]](#), [Intel \[1986\]](#) e [Intel \[1990\]](#)). Os métodos de alocação da IBM foram descritos em [Deitel \[1990\]](#). Os detalhes internos do sistema BSD UNIX foram abordados por completo em [McKusick e outros \[1996\]](#), e [McVoy e Kleiman \[1991\]](#) apresentaram otimizações para os métodos feitos no Solaris. O sistema de arquivos do Google é descrito em [Ghemawat e outros \[2003\]](#). FUSE pode ser encontrado em <http://fuse.sourceforge.net/>.

A alocação de arquivo em disco baseada no sistema buddy foi discutida por [Koch \[1987\]](#). Um esquema de organização de arquivos que garante a recuperação em um acesso foi discutido por [Larson e Kajla \[1984\]](#). Organizações de arquivo estruturadas em log para melhorar o desempenho e a coerência foram discutidas em [Rosenblum e Ousterhout \[1991\]](#), [Seltzer e outros \[1993\]](#) e [Seltzer e outros \[1995\]](#). Os algoritmos como as árvores balanceadas (e muito mais) são abordados por [Knuth \[1998\]](#) e [Cormen e outros \[2001\]](#). O código-fonte do ZFS para mapas de espaço pode ser encontrado em http://src.opensolaris.org/source/xref/onnv/onnv-gate/usr/src/uts/common/fs/zfs/space_map.c.

O caching de disco foi discutido por [McKeon \[1985\]](#) e [Smith \[1985\]](#). O caching no sistema operacional experimental Sprite foi descrito em [Nelson e outros \[1988\]](#). Discussões gerais referentes à tecnologia de armazenamento em massa foram oferecidas por [Chi \[1982\]](#) e [Hoagland \[1985\]](#). [Folk e Zoellick \[1987\]](#) cobriram toda a gama de estruturas de arquivo. [Silvers \[2000\]](#) discute a implementação do cache de página no sistema operacional NetBSD.

O Network File System (NFS) é discutido em [Sandberg e outros \[1985\]](#), [Sandberg \[1987\]](#), [Sun \[1990\]](#) e [Callaghan \[2000\]](#). NFS versão 4 é um padrão descrito em <http://www.ietf.org/rfc/rfc3530.txt>. As características das cargas de trabalho nos sistemas de arquivo distribuídos foram estudadas em [Baker e outros \[1991\]](#). [Ousterhout \[1991\]](#) discutiu o papel do estado distribuído nos sistemas de arquivos em rede. Os projetos estruturados em log para sistemas de arquivos em rede foram propostos em [Hartman e Ousterhout \[1995\]](#) e [Thekkath e outros \[1997\]](#). NFS e o UNIX File System (UFS) são descritos em [Vahalia \[1996\]](#) e [Mauro e McDougall \[2007\]](#). O sistema de arquivos do Windows NT, NTFS, foi explicado por [Solomon \[1998\]](#). O sistema de arquivos Ext2, usado no Linux, é descrito em [Bovet e Cesati \[2002\]](#), e o sistema de arquivos WAFL, em [Hitz e outros \[1995\]](#). A documentação do ZFS pode ser encontrada em <http://www.opensolaris.org/os/community/ZFS/docs>.

CAPÍTULO 12

Estrutura de armazenamento em massa

O sistema de arquivos pode ser visto logicamente como consistindo em três partes. No [Capítulo 10](#), vimos a interface com o usuário e o programador para o sistema de arquivos. No [Capítulo 11](#), descrevemos as estruturas de dados internas e os algoritmos usados pelo sistema operacional para implementar essa interface. Neste capítulo, discutimos o nível mais baixo do sistema de arquivos: as estruturas do armazenamento secundário e terciário. Primeiro, descrevemos a estrutura física dos discos magnéticos e fitas magnéticas, então descrevemos os algoritmos de escalonamento de disco, que definem a ordem da E/S em disco para melhorar o desempenho. Em seguida, discutimos a formatação e o gerenciamento dos blocos de boot, blocos danificados e swap space. Examinamos a estrutura do armazenamento secundário, explicando a confiabilidade do disco e a implementação em armazenamento estável. Concluímos com uma rápida descrição dos dispositivos de armazenamento terciário e os problemas que surgem quando um sistema operacional utiliza o armazenamento terciário.

OBJETIVOS DO CAPÍTULO

- Descrever a estrutura física dos dispositivos de armazenamento secundário e terciário e os efeitos resultantes sobre os usos dos dispositivos.
- Explicar as características de desempenho dos dispositivos de armazenamento em massa.
- Discutir os serviços do sistema operacional fornecidos para o armazenamento em massa, incluindo RAID e HSM.

12.1 Visão geral da estrutura de armazenamento em massa

Nesta seção, apresentamos uma visão geral da estrutura física dos dispositivos de armazenamento secundário e terciário.

12.1.1 Discos magnéticos

Os **discos magnéticos** são responsáveis pela maior parte do armazenamento secundário nos sistemas computadorizados modernos. Conceitualmente, os discos são simples (Figura 12.1). Cada **prato** do disco possui uma forma circular plana, como um CD. Os diâmetros comuns dos pratos variam de 1,8 a 5,25 polegadas. As duas superfícies de um prato são revestidas por um material magnético. Armazenamos informações gravando-as magneticamente sobre os pratos.

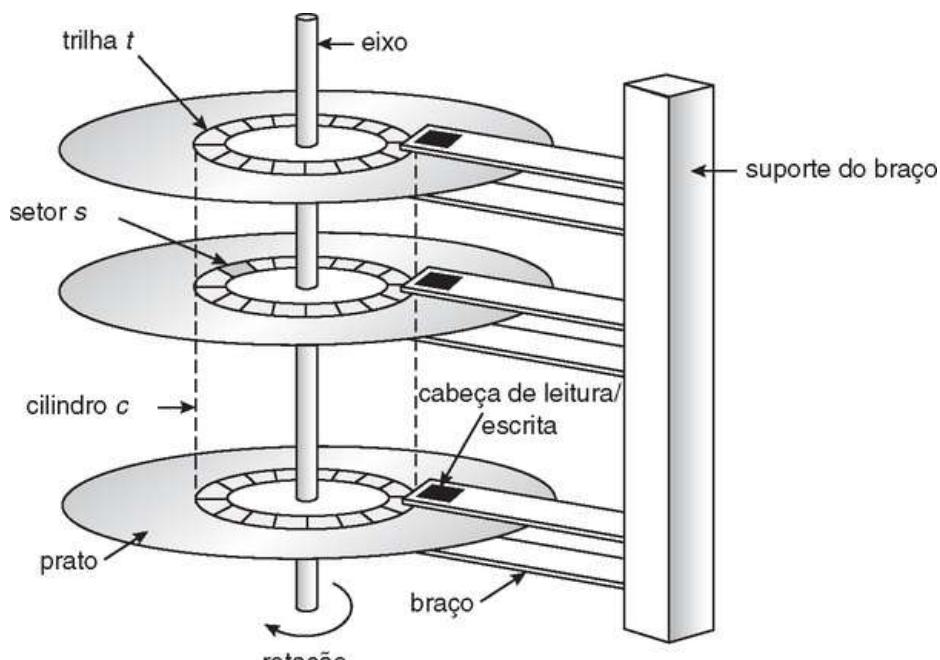


FIGURA 12.1 Mecanismo de disco com cabeça móvel.

Uma cabeça de leitura/escrita “voa” pouco acima de cada superfície de cada prato. As cabeças são presas a um **braço de disco**, que move todas as cabeças como uma só unidade. A superfície de um prato é dividida logicamente em **trilhas** circulares, que são subdivididas em **setores**. O conjunto de trilhas que estão em uma posição do braço forma um **cilindro**. Em uma unidade de disco pode haver milhares de cilindros concêntricos, e cada trilha pode conter centenas de setores. A capacidade de armazenamento das unidades de disco mais comuns é medida em gigabytes.

Quando o disco está em uso, o motor gira em alta velocidade. A maior parte das unidades gira de 60 a 200 vezes por segundo. A velocidade do disco possui duas medidas. A **taxa de transferência (transfer rate)** é a velocidade com que os dados fluem entre a unidade e o computador. O **tempo de posicionamento (positioning time)**, às vezes chamado de **tempo de acesso (access time)**, consiste em duas partes: o tempo para mover o braço do disco até o cilindro desejado, chamado **tempo de busca (seek time)**, e o tempo para que o setor desejado gire até a posição da cabeça de leitura/escrita, chamado **latência rotacional (rotational latency)**. Os discos mais comuns podem transferir megabytes de dados por segundo, e eles possuem tempos de busca e latências de rotação de vários milissegundos.

Como a cabeça do disco voa sobre uma almofada de ar extremamente fina (medida em micrônios), há o perigo de a cabeça poder fazer contato com a superfície do disco. Embora os pratos de disco sejam revestidos por uma fina camada protetora, às vezes a superfície magnética pode ser danificada. Esse acidente é denominado **colisão da cabeça (head crash)**. Uma colisão da cabeça normalmente não pode ser consertada. Todo o disco precisa ser substituído.

O disco pode ser **removível**, permitindo que diferentes discos sejam montados de acordo com a necessidade. Um disco magnético removível consiste em um prato mantido em um envoltório plástico para proteção contra danos enquanto não está na unidade de disco. **Disquetes** são discos magnéticos removíveis de baixo custo. A cabeça de uma unidade de disquete em geral entra em contato direto com a superfície do disco, de modo que a unidade é projetada para girar mais

lentamente do que uma unidade de disco rígido, a fim de reduzir o desgaste da superfície do disco. A capacidade de armazenamento de um disquete é de pouco mais que 1,44 MB. Existem discos removíveis que trabalham de forma muito semelhante a discos rígidos e possuem capacidades medidas em gigabytes.

Uma unidade de disco está conectada a um computador por um conjunto de fios, chamados barramento de E/S. Existem vários tipos de barramentos, incluindo barramentos **EIDE (Enhanced Integrated Drive Electronics)**, **ATA (Advanced Technology Attachment)**, **Serial ATA (SATA)**, **Universal Serial Bus (USB)**, **FC (Fiber Channel)** e **SCSI (Small Computer-System Interface)**. As transferências de dados em um barramento são executadas por processadores eletrônicos especiais, chamados **controladores**. O **controlador do host (host controller)** é o controlador do computador nesse barramento. Um **controlador de disco** está embutido em cada unidade de disco. Para realizar uma operação de E/S em disco, o computador coloca um comando no controlador do host, usando portas de E/S mapeadas na memória, conforme descrito na [Seção 9.7.3](#). O controlador do host, em seguida, envia o comando por mensagens ao controlador de disco, que opera o hardware da unidade de disco para executar o comando. Os controladores de disco possuem um cache embutido. A transferência de dados na unidade acontece entre o cache e a superfície do disco, e a transferência de dados para o host, em altas velocidades eletrônicas, ocorre entre o cache e o controlador do host.

TAXAS DE TRANSFERÊNCIA DE DISCO

Assim como muitos aspectos da computação, os números de desempenho publicados para os discos não são iguais aos números do desempenho real. As taxas de transferência declaradas sempre são menores que as **taxas de transferência efetivas**, por exemplo. A taxa de transferência pode ser a taxa em que os bits podem ser lidos da mídia magnética pela cabeça de leitura do disco, mas esta é diferente da taxa em que os blocos são entregues ao sistema operacional.

12.1.2 Fitas magnéticas

A **fita magnética** foi utilizada a princípio como um meio de armazenamento secundário. Embora sendo relativamente permanente e podendo armazenar grande quantidade de dados, seu tempo de acesso é lento em comparação com o da memória principal e o do disco magnético. Além disso, o acesso aleatório à fita magnética é cerca de mil vezes mais lento do que o acesso aleatório ao disco magnético; assim, as fitas não são muito úteis para armazenamento secundário. Elas são usadas principalmente para backup, para o armazenamento de informações usadas com pouca frequência e como um meio para transferir informações de um sistema para outro.

Uma fita é mantida em um carretel e avança e retrocede sob uma cabeça de leitura/escrita. A movimentação até a posição correta de uma fita pode levar minutos; porém, uma vez posicionada, pode gravar dados em velocidades comparáveis às das unidades de disco. As capacidades da fita variam muito, dependendo do tipo da unidade de fita. Tipicamente, as fitas armazenam de 20 GB a 200 GB. Algumas possuem compactação embutida, que pode mais do que dobrar o armazenamento efetivo. As fitas e seus drivers normalmente estão categorizados por largura, incluindo 4, 8 e 19 milímetros, e 1/4 e 1/2 polegada. O armazenamento em fita é descrito na [Seção 12.9](#).

FIREWIRE

FireWire refere-se a uma interface projetada para conectar dispositivos periféricos, como discos rígidos, unidades de DVD e as câmeras de vídeo digital a um sistema de computação. O FireWire foi desenvolvido inicialmente pela Apple Computer e tornou-se o padrão IEEE 1394 em 1995. O padrão FireWire original fornecia largura de banda de até 400 megabits por segundo. Recentemente, um novo padrão - FireWire 2 - surgiu e é identificado pelo padrão IEEE 1394b. O FireWire 2 oferece o dobro da taxa de dados do FireWire original - 800 megabits por segundo.

12.2 Estrutura do disco

As unidades de disco modernas são endereçadas como grandes arrays unidimensionais de **blocos lógicos**, sendo o bloco lógico a menor unidade de transferência. O tamanho de um bloco lógico normalmente é de 512 bytes, embora alguns discos possam ter a opção de um tamanho de bloco lógico diferente, como 1.024 bytes, durante a **formatação em baixo nível**. Essa opção é descrita na [Seção 12.5.1](#). O array unidimensional de blocos lógicos é mapeado sequencialmente nos setores do disco. O setor 0 é o primeiro setor da primeira trilha no cilindro mais externo. O mapeamento prossegue na sequência por essa trilha, depois pelo restante das trilhas nesse cilindro, e depois pelo restante dos cilindros, do mais externo ao mais interno.

Usando esse mapeamento, podemos – pelo menos, teoricamente – converter um número de bloco lógico em um endereço de disco no estilo antigo, que consiste em um número de cilindro, um número de trilha dentro desse cilindro e um número de setor dentro dessa trilha. Na prática, é difícil realizar essa tradução, por dois motivos. Primeiro, a maioria dos discos possui alguns setores defeituosos, mas o mapeamento esconde isso substituindo-os por setores de reserva em outra parte no disco. Segundo, o número de setores por trilha não é uma constante em algumas unidades.

Vamos dar uma olhada mais de perto no segundo motivo. Em meios que usam **velocidade linear constante (Constant Linear Velocity - CLV)**, a densidade dos bits por trilha é uniforme. Quanto mais afastada uma trilha estiver do centro do disco, maior é o seu tamanho, de modo que ela pode conter mais setores. Ao movermos das zonas externas para as zonas internas, a quantidade de setores por trilha diminui. As trilhas na zona mais externa normalmente mantêm 40% a mais de setores do que as trilhas na zona mais interna. A unidade aumenta sua velocidade de rotação enquanto a cabeça se move das trilhas mais externas para as mais internas, a fim de manter a mesma taxa de dados passando sob a cabeça. Esse método é usado nas unidades de CD-ROM e DVD-ROM. Como alternativa, a velocidade de rotação do disco pode permanecer constante, e a densidade dos bits diminui das trilhas mais internas para as trilhas mais externas, a fim de manter a taxa de dados constante. Esse método é usado nos discos rígidos e é conhecido como **velocidade angular constante (Constant Angular Velocity - CAV)**.

A quantidade de setores por trilha tem aumentado à medida que a tecnologia de disco melhora, e a zona mais externa de um disco normalmente possui várias centenas de setores por trilha. De modo semelhante, a quantidade de cilindros por disco tem aumentado; discos maiores possuem dezenas de milhares de cilindros.

12.3 Conexão de disco

Os computadores acessam o armazenamento em disco de duas maneiras. Uma delas é por meio de portas de E/S (ou **armazenamento conectado ao host**); isso é comum em sistemas pequenos. A outra maneira é por meio de um host remoto via sistema de arquivos distribuído; isso é denominado **armazenamento conectado por rede**.

12.3.1 Armazenamento conectado ao host

O armazenamento conectado ao host é o armazenamento acessado por meio das portas de E/S locais. Essas portas usam várias tecnologias. O PC desktop comum utiliza uma arquitetura de barramento de E/S denominada IDE ou ATA. Essa arquitetura admite um máximo de duas unidades por barramento de E/S. Um protocolo semelhante, mais novo, que possui cabeamento simplificado, é o SATA. Estações de trabalho e servidores de alto nível em geral usam arquiteturas de E/S mais sofisticadas, como SCSI e Fiber Channel (FC).

O SCSI é uma arquitetura de barramento. Seu meio físico costuma ser um cabo de fita com grande quantidade de condutores (normalmente, 50 ou 68). O protocolo SCSI admite um máximo de 16 dispositivos por barramento. Normalmente, o dispositivo inclui uma placa controladora no host (o **SCSI initiator**) e até 15 dispositivos de armazenamento (os **SCSI targets**). Um disco SCSI é um SCSI target comum, mas o protocolo provê a capacidade de endereçar até oito **unidades lógicas** em cada SCSI target. Um uso típico do endereço lógico da unidade é direcionar comandos para os componentes de um array RAID ou componentes de uma biblioteca de mídia removível (como um jukebox de CD enviando comandos para o mecanismo de troca de mídia ou para uma das unidades).

A FC é uma arquitetura serial de alta velocidade que pode operar sobre fibra óptica ou sobre um cabo de cobre com quatro condutores. Ela possui duas variantes. Uma é uma grande malha comutada com um espaço de endereços de 24 bits. Essa variante deverá ser dominante no futuro e é a base das **SANs (Storage-Area Networks)**, discutida na [Seção 12.3.3](#). Devido ao grande espaço de endereços e à natureza comutada da comunicação, vários hosts e dispositivos de armazenamento podem ser conectados à malha, permitindo grande flexibilidade na comunicação de E/S. Outra variante FC é um **loop arbitrado (FC-AL)**, que pode endereçar 126 dispositivos (unidades e controladores).

Uma grande variedade de dispositivos de armazenamento é adequada para uso como armazenamento conectado ao host. Entre eles estão unidades de disco rígido, arrays RAID e unidades de CD, DVD e fita. Os comandos de E/S que iniciam as transferências de dados para um dispositivo de armazenamento conectado ao host são leituras e escritas de blocos de dados lógicos, direcionados para unidades de armazenamento especificamente identificadas (como ID de barramento, ID SCSI e unidade lógica-alvo).

12.3.2 Armazenamento conectado à rede

Um dispositivo de armazenamento conectado à rede (**Network-Attached Storage - NAS**) é um sistema de armazenamento de uso especial, acessado remotamente por uma rede de dados ([Figura 12.2](#)). Os clientes acessam o armazenamento conectado à rede por meio de uma interface remote-procedure-call, como NFS para sistemas UNIX ou CIFS para máquinas Windows. As Remote Procedure Calls (RPCs) são transportadas via TCP ou UDP por uma rede IP – normalmente, a mesma rede local (**Local-Area Network - LAN**) que transporta todo o tráfego até os clientes. A unidade de armazenamento conectada à rede é implementada como um array RAID com software que implementa a interface remote procedure call. É mais fácil pensar no NAS como outro protocolo para acesso de armazenamento. Por exemplo, em vez de usar um driver de dispositivo SCSI e protocolos SCSI para acessar o armazenamento, um sistema usando NAS usaria RPC em cima de TCP/IP.

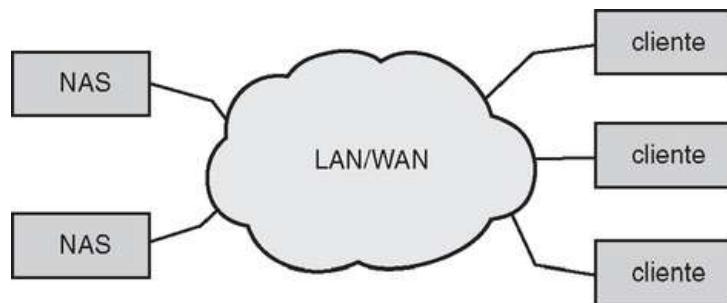


FIGURA 12.2 Armazenamento conectado à rede.

O armazenamento conectado à rede provê um modo conveniente para todos os computadores em uma LAN compartilharem um pool de armazenamento com a mesma facilidade de nomeação e acesso oferecida para o armazenamento conectado ao host local. Todavia, ele costuma ser menos eficiente e ter menor desempenho do que algumas opções de armazenamento conectadas diretamente.

O **iSCSI** é o protocolo de armazenamento conectado à rede mais recente. Basicamente, ele usa o protocolo de rede IP para transportar o protocolo SCSI. Assim, as redes, em vez dos cabos SCSI, podem ser usadas como interconexões entre os hosts e seu armazenamento. Como resultado, os hosts podem tratar seu armazenamento como se fosse conectado diretamente, mesmo que o armazenamento esteja distante do host.

12.3.3 SAN

Uma desvantagem dos sistemas de armazenamento conectados à rede é que as operações de E/S para armazenamento consomem largura de banda na rede de dados, aumentando, assim, a latência da comunicação na rede. Esse problema pode ser particularmente grave em grandes instalações cliente-servidor – a comunicação entre servidores e clientes compete pela largura de banda com a comunicação entre servidores e dispositivos de armazenamento.

Uma SAN (storage-area network) é uma rede privada (usando protocolos de armazenamento, em vez de protocolos de rede) entre os servidores e as unidades de armazenamento, como mostrado na **Figura 12.3**. O poder de uma SAN está em sua flexibilidade. Vários hosts e vários arrays de armazenamento podem estar conectados à mesma SAN, e o armazenamento pode ser alocado dinamicamente aos hosts. Um switch SAN permite ou proíbe o acesso entre os hosts e o armazenamento. Como exemplo, se um host estiver com pouco espaço em disco, a SAN pode ser configurada para alocar mais armazenamento a esse host. As SANs tornam possível que clusters de servidores compartilhem o mesmo armazenamento e que os arrays de armazenamento incluam múltiplas conexões de host diretas. Normalmente, elas possuem mais portas, e portas mais caras, do que os arrays de armazenamento.

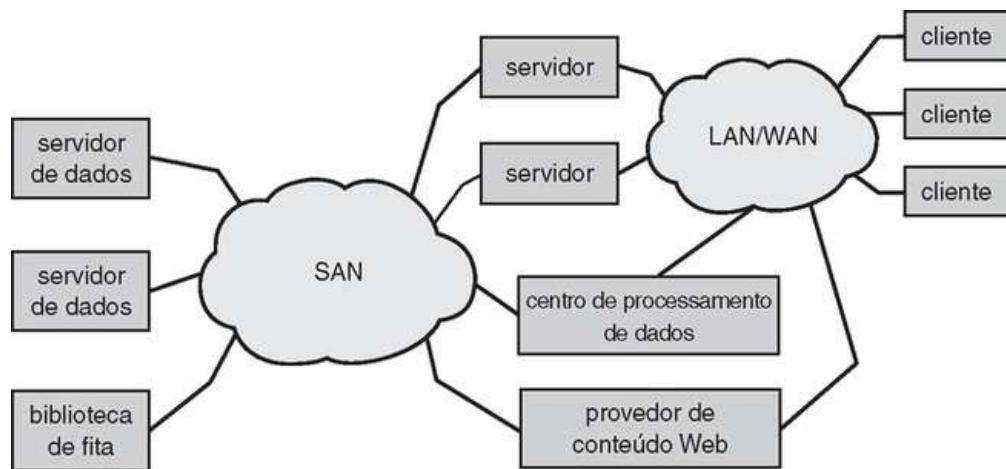


FIGURA 12.3 Storage-Area Network.

A FC é a interconexão SAN mais comum, embora a simplicidade do iSCSI esteja sendo cada vez mais utilizada. Uma alternativa emergente é uma arquitetura de barramento de uso especial, denominada InfiniBand, que provê suporte de hardware e software para redes de interconexão de alta velocidade para servidores e unidades de armazenamento.

12.4 Escalonamento de disco

Uma das responsabilidades do sistema operacional é usar o hardware de forma eficiente. Para as unidades de disco, atender a essa responsabilidade significa ter tempo de acesso rápido e largura de banda de disco. Como visto na [Seção 12.1.1](#), o tempo de acesso possui dois componentes principais: tempo de busca e latência de rotação. O **tempo de busca** é o tempo para o braço do disco mover as cabeças até o cilindro que contém o setor desejado, e a **latência de rotação** é o tempo adicional até o disco girar o setor desejado para a direção da cabeça do disco. A **largura de banda** do disco é o número total de bytes transferidos, dividido pelo tempo total entre a primeira requisição de serviço e o término da última transferência. Podemos melhorar tanto o tempo de acesso quanto a largura de banda controlando a ordem na qual as requisições de E/S de disco são atendidas.

Sempre que um processo precisa de E/S do ou para o disco, ele emite uma chamada de sistema para o sistema operacional. A requisição especifica várias informações:

- Se a operação é de entrada ou saída.
- O endereço de disco para a transferência.
- O endereço de memória para a transferência.
- A quantidade de bytes a serem transferidos.

Se a unidade de disco desejada e seu controlador estiverem disponíveis, a requisição pode ser atendida imediatamente. Se a unidade ou o controlador estiver ocupado, quaisquer requisições de atendimento novas serão colocadas na fila de requisições pendentes para essa unidade. Para um sistema de multiprogramação com muitos processos, a fila do disco normalmente pode ter centenas de requisições pendentes. Assim, quando uma requisição é concluída, o sistema operacional escolhe qual requisição pendente será atendida em seguida. Como o sistema operacional faz essa escolha? Qualquer um dentre vários algoritmos de escalonamento de disco pode ser usado, e discutimos sobre eles em seguida.

12.4.1 Escalonamento FCFS

A forma mais simples de escalonamento de disco é, naturalmente, o algoritmo primeiro a chegar, primeiro a ser atendido (First-Come, First-Served – FCFS). Esse algoritmo é intrinsecamente justo, mas em geral não provê o serviço mais rápido. Considere, por exemplo, uma fila de disco com requisições de E/S para os blocos nos cilindros

98, 183, 37, 122, 14, 124, 65, 67

nessa ordem. Se a cabeça de disco estiver inicialmente no cilindro 53, ela primeiro passará de 53 para 98, depois para 183, 37, 122, 14, 124, 65, e finalmente para 67, realizando um movimento de cabeça total de 640 cilindros. Esse escalonamento é representado na [Figura 12.4](#).

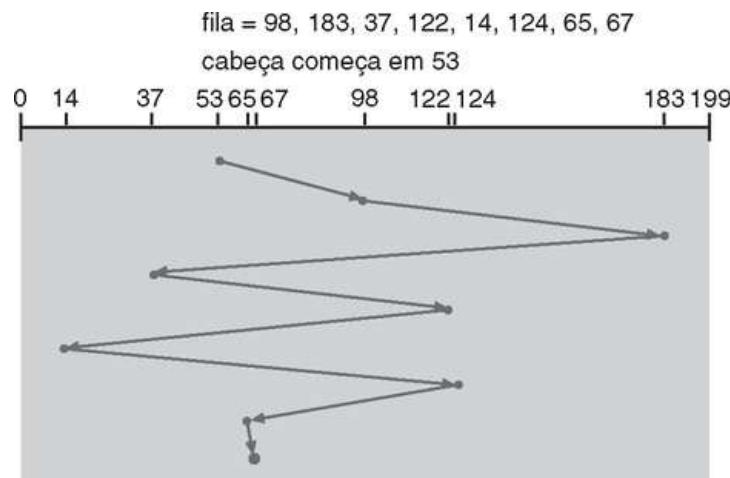


FIGURA 12.4 Escalonamento de disco FCFS.

A intensa oscilação de 122 a 14 e depois de volta para 124 ilustra o problema com esse escalonamento. Se as requisições para os cilindros 37 e 14 pudessem ser atendidas juntas, antes ou depois das requisições em 122 e 124, o movimento total da cabeça poderia ser diminuído substancialmente e, com isso, o desempenho poderia ser melhorado.

12.4.2 Escalonamento SSTF

Parece razoável atender a todas as requisições próximas à posição atual da cabeça, antes de mover

a cabeça para longe, a fim de atender a outras requisições. Essa suposição é a base para o algoritmo **menor tempo de busca primeiro** (**Shortest-Seek-Time-First** - **SSTF**). O algoritmo SSTF seleciona a requisição com o tempo de busca mínimo a partir da posição atual da cabeça. Como o tempo de busca aumenta com a quantidade de cilindros atravessados pela cabeça, SSTF escolhe a requisição pendente mais próxima da posição atual da cabeça.

Para nossa fila de requisição de exemplo, a requisição mais próxima à posição inicial da cabeça (53) está no cilindro 65. Quando estivermos no cilindro 65, a requisição mais próxima será no cilindro 67. A partir de lá, a requisição no cilindro 37 está mais próxima do que 98, de modo que 37 é atendido em seguida. Continuando, atendemos a requisição no cilindro 14, depois 98, 122, 124 e, finalmente, 183 ([Figura 12.5](#)). Esse método de escalonamento resulta em um movimento total da cabeça de apenas 236 cilindros – pouco mais de um terço da distância necessária para o escalonamento FCFS dessa fila de requisições. Claramente, esse algoritmo provê uma melhoria substancial no desempenho.

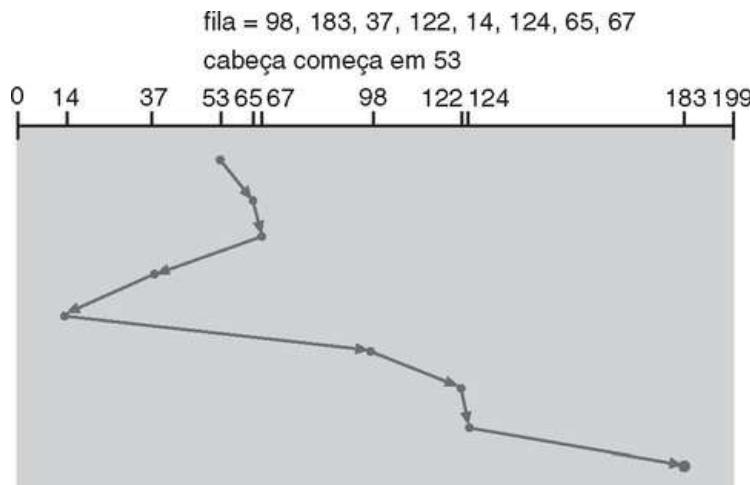


FIGURA 12.5 Escalonamento de disco SSTF.

O escalonamento SSTF é basicamente uma forma de escalonamento da primeira tarefa mais curta (Shortest-Job-First - SJF) e, como o escalonamento SJF, pode causar starvation de algumas requisições. Lembre-se de que as requisições podem chegar a qualquer momento. Suponha que tenhamos duas requisições na fila, para os cilindros 14 e 186, e enquanto atendemos à requisição de 14 chega uma nova requisição próxima de 14. Essa nova requisição será atendida em seguida, fazendo a requisição em 186 esperar. Enquanto essa requisição está sendo atendida, outra requisição perto de 14 poderia chegar. Teoricamente, um fluxo contínuo de requisições próximas umas das outras poderia chegar, fazendo a requisição do cilindro 186 esperar indefinidamente. Esse cenário se torna cada vez mais provável se a fila de requisições pendentes crescer muito.

Embora o algoritmo SSTF seja uma melhoria substancial em relação ao algoritmo FCFS, ele não é o ideal. No exemplo, podemos fazer algo melhor movendo a cabeça de 53 para 37, embora o último não seja o mais próximo, e depois para 14, antes de voltar para atender na ordem 65, 67, 98, 122, 124 e 183. Essa estratégia reduz o movimento total da cabeça para 208 cilindros.

12.4.3 Escalonamento SCAN

No **algoritmo SCAN**, o braço do disco começa em uma extremidade do disco e passa para a outra extremidade, atendendo às requisições à medida que alcança cada cilindro, até chegar à outra extremidade do disco. Na outra extremidade, a direção do movimento da cabeça é invertida, e o atendimento continua. A cabeça passa continuamente para frente e para trás no disco. O algoritmo SCAN às vezes é chamado de **algoritmo elevador**, pois o braço do disco se comporta como um elevador de um prédio, primeiro atendendo a todas as requisições de subida e depois retornando para atender às requisições no outro sentido.

Vamos voltar ao nosso exemplo para ilustrá-lo. Antes de aplicar o SCAN para escalonar as requisições dos cilindros 98, 183, 37, 122, 14, 124, 65 e 67, precisamos saber a direção do movimento da cabeça, além da posição atual da cabeça. Supondo que o braço do disco esteja se movendo para 0, e que a posição inicial da cabeça seja novamente 53, a cabeça atenderá a 37 e depois 14. No cilindro 0, o braço retornará e se moverá para a outra extremidade do disco, atendendo às requisições dos cilindros em 65, 67, 98, 122, 124 e 183 ([Figura 12.6](#)). Se a requisição chegar na fila logo na frente da cabeça, ela será atendida quase imediatamente; uma requisição chegando logo atrás da cabeça terá de esperar até o braço se mover até o final do disco, reverter a direção e retornar.

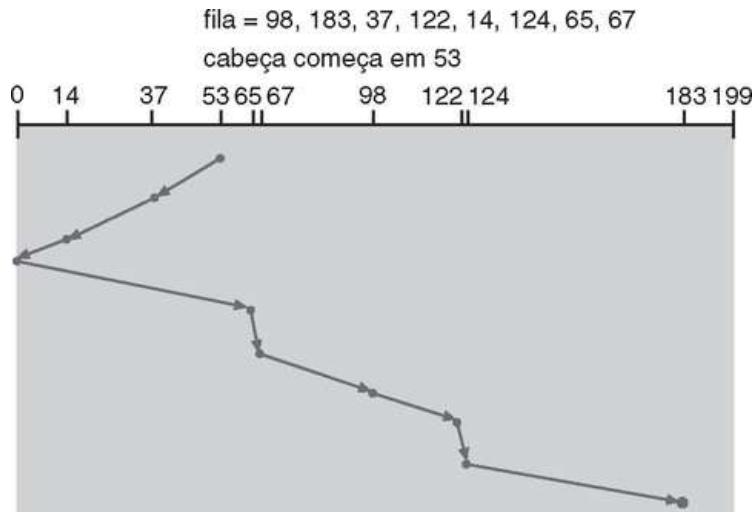


FIGURA 12.6 Escalonamento de disco SCAN.

Supondo uma distribuição uniforme de requisições para cilindros, considere a densidade das requisições quando a cabeça chega a uma extremidade e inverte a direção. Nesse ponto, poucas requisições estão na frente da cabeça, pois esses cilindros recentemente foram atendidos. A densidade maior das requisições está na outra extremidade do disco. Essas requisições também estão esperando há mais tempo; logo, por que não ir para lá primeiro? Essa é a ideia do próximo algoritmo.

12.4.4 Escalonamento C-SCAN

O **escalonamento SCAN circular (C-SCAN)** é uma variante do SCAN, projetada para fornecer um tempo de espera mais uniforme. Como o SCAN, o C-SCAN move a cabeça de uma extremidade do disco para a outra, atendendo às requisições durante o caminho. No entanto, quando a cabeça atinge a outra extremidade, ela imediatamente retorna ao início do disco, sem atender a quaisquer requisições na viagem de volta ([Figura 12.7](#)). O algoritmo de escalonamento C-SCAN basicamente trata os cilindros como uma lista circular que contorna do último cilindro para o primeiro.

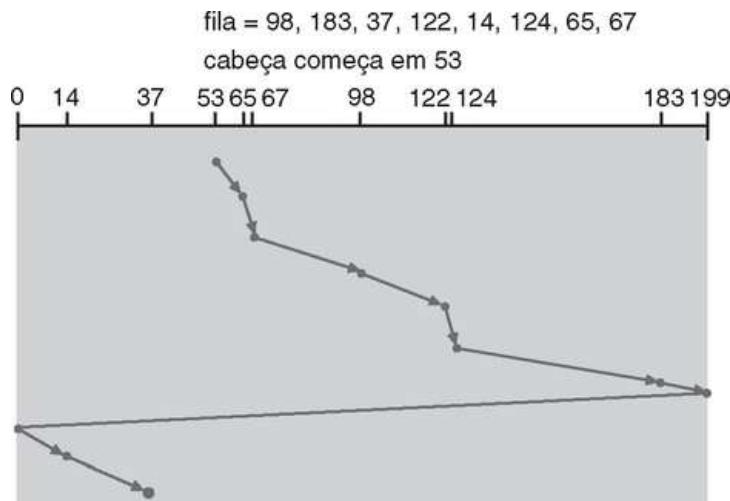


FIGURA 12.7 Escalonamento de disco C-SCAN.

12.4.5 Escalonamento LOOK

Conforme os descrevemos, tanto SCAN quanto C-SCAN movem o braço do disco por toda a extensão do disco. Na prática, nenhum desses algoritmos é implementado dessa maneira. Normalmente, o braço só vai até o ponto da última requisição em cada sentido. Depois, ele inverte o sentido, sem ir até a extremidade do disco. Essas versões de SCAN e C-SCAN são chamadas de **escalonamento LOOK** e **C-LOOK**, pois *procuram* (look for) uma requisição antes de continuar movendo em um sentido indicado ([Figura 12.8](#)).

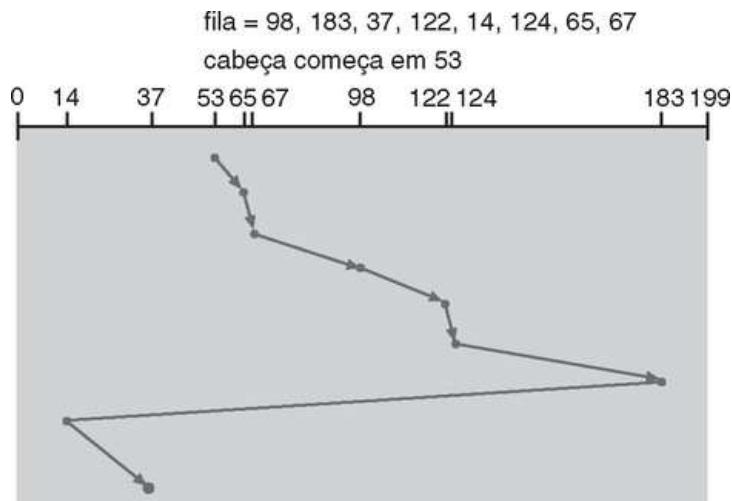


FIGURA 12.8 Escalonamento de disco C-LOOK.

12.4.6 Seleção de um algoritmo de escalonamento de disco

Com tantos algoritmos de escalonamento de disco, como escolhemos o melhor deles? O SSTF é comum e possui um apelo natural porque aumenta o desempenho em relação ao FCFS. SCAN e C-SCAN funcionam melhor para sistemas que impõem muita carga no disco, pois há menos chances de causar um problema de starvation. Para qualquer lista de requisições em particular, podemos definir a ordem ideal da recuperação, mas a computação necessária para encontrar o escalonamento ideal pode não justificar a economia em relação a SSTF ou SCAN. Com qualquer algoritmo de escalonamento, porém, o desempenho depende muito da quantidade e dos tipos de requisições. Por exemplo, suponha que a fila tenha apenas uma requisição pendente. Então, todos os algoritmos de escalonamento são forçados a se comportar da mesma maneira, pois têm apenas uma escolha para o local em que a cabeça do disco deverá se mover: todos se comportam como o escalonamento FCFS.

As requisições para o serviço de disco podem ser bastante influenciadas pelo método de alocação de arquivos. Um programa que lê um arquivo alocado de forma contígua gerará várias requisições próximas do disco, resultando em um movimento de cabeça limitado. Um arquivo interligado ou indexado, ao contrário, pode incluir blocos bastante espalhados no disco, resultando em maior movimento da cabeça.

O local dos diretórios e blocos de índice também é importante. Como cada arquivo precisa ser aberto para ser utilizado, e a abertura de um arquivo exige a busca na estrutura de diretórios, os diretórios serão acessados com frequência. Suponha uma entrada de diretório no primeiro cilindro e os dados de um arquivo no cilindro final. Nesse caso, a cabeça do disco precisa se mover por toda a extensão do disco. Agora, se a entrada de diretório está no cilindro do meio, a cabeça tem de mover somente metade da extensão. O uso de caches para diretórios e blocos de disco na memória principal também pode ajudar a reduzir o movimento do braço de disco, particularmente para requisições de leitura.

Devido a essas complexidades, o algoritmo de escalonamento de disco deve ser escrito como um módulo separado do sistema operacional, de modo a poder ser substituído por um algoritmo diferente, se necessário. Tanto SSTF quanto LOOK são opções razoáveis para o algoritmo-padrão.

Os algoritmos de escalonamento descritos aqui consideram apenas as distâncias de busca. Para os discos modernos, a latência de rotação pode ser quase tão grande quanto o tempo de busca médio. Entretanto, é difícil para o sistema operacional escalonar visando à melhor latência de rotação, pois os discos modernos não revelam o local físico dos blocos lógicos. Os fabricantes de disco têm aliviado esse problema implementando algoritmos de escalonamento de disco no hardware do controlador, embutido na unidade de disco. Se o sistema operacional enviar um lote de requisições ao controlador, ele poderá enfileirá-las e depois escaloná-las para melhorar o tempo de busca e a latência de rotação.

Se o desempenho da E/S fosse a única consideração, o sistema operacional estaria contente em passar a responsabilidade do escalonamento de disco para o hardware do disco. Todavia, na prática, o sistema operacional pode ter outras restrições sobre a ordem de atendimento das requisições. Por exemplo, a paginação por demanda pode ter prioridade em relação à E/S das aplicações, e as escritas são mais urgentes do que as leituras, se o cache estiver com poucas páginas livres. Além disso, pode ser desejável garantir a ordem de um conjunto de escritas de disco para tornar o sistema de arquivos robusto diante das falhas do sistema. Considere o que aconteceria se o sistema operacional alocasse uma página de disco a um arquivo e a aplicação escrevesse dados nessa página antes de o sistema operacional ter uma chance de enviar o inode modificado e a lista de espaços livres para o disco. Para acomodar tais requisitos, um sistema operacional pode escolher fazer seu

próprio escalonamento de disco e levar as requisições para o controlador de disco, uma a uma, para alguns tipos de E/S.

12.5 Gerenciamento de disco

O sistema operacional também é responsável por diversos outros aspectos do gerenciamento de disco. Aqui, discutimos a inicialização do disco, a partida do disco e a recuperação de blocos defeituosos.

12.5.1 Formatação de disco

Um disco magnético novo é como um quadro-negro vazio: tem apenas placas de um material de gravação magnético. Antes de um disco poder armazenar dados, ele precisa ser dividido em setores que o controlador de disco possa ler e escrever. Esse processo é denominado **formatação em baixo nível** ou **formatação física**. A formatação em baixo nível preenche o disco com uma estrutura de dados especial para cada setor. A estrutura dos dados para um setor consiste em um cabeçalho, uma área de dados (normalmente, 512 bytes de tamanho) e um término. O cabeçalho e o término contêm informações usadas pelo controlador de disco, como um número de setor e um **código para a correção de erros (Error Correcting Code - ECC)**. Quando o controlador escreve um setor de dados durante a E/S normal, o ECC é atualizado com um valor calculado a partir de todos os bytes na área de dados. Quando o setor é lido, o ECC é recalculado e comparado com o valor armazenado. Se os números armazenado e calculado forem diferentes, essa divergência indica que a área de dados do setor foi modificada e que o setor do disco pode estar defeituoso ([Seção 12.5.3](#)). O ECC é um código para a *correção* de erros porque contém informações suficientes para que, se apenas alguns bits de dados tiverem sido mudados, o controlador possa identificar quais bits foram mudados e calcular quais são seus valores corretos. Depois ele informa um **erro brando** recuperável. O controlador realiza o processamento do ECC automaticamente sempre que um setor é lido ou escrito.

A maioria dos discos rígidos é formatada em baixo nível na fábrica, como parte do processo de manufatura. Essa formatação permite ao fabricante testar o disco e inicializar o mapeamento dos números de bloco lógicos, detectando os setores livres com defeito no disco. Para muitos discos rígidos, quando o controlador de disco é instruído a formatar o disco em baixo nível, ele também pode ser informado sobre quantos bytes do espaço de dados devem ser deixados entre o cabeçalho e o término de todos os setores. Normalmente, é possível escolher entre alguns tamanhos, como 256, 512 e 1.024 bytes. A formatação de um disco com tamanho de setor grande significa que menos setores caberão em cada trilha, mas isso também significa que menos cabeçalhos e términos serão gravados em cada trilha, e mais espaço estará disponível para os dados do usuário. Alguns sistemas operacionais só podem lidar com um tamanho de setor de 512 bytes.

Para usar um disco para manter arquivos, o sistema operacional ainda precisa registrar suas próprias estruturas de dados no disco. Ele faz isso em duas etapas. A primeira etapa é **particionar** o disco em um ou mais grupos de cilindros. Por exemplo, uma partição pode manter uma cópia do código executável do sistema operacional, enquanto outra mantém os arquivos do usuário. A segunda etapa é a **formatação lógica** (ou criação de um sistema de arquivos). Nessa etapa, o sistema operacional armazena no disco as estruturas de dados iniciais do sistema de arquivos. Essas estruturas de dados podem incluir mapas de espaço livre e alocado (uma FAT ou inodes) e um diretório inicial vazio.

Para aumentar a eficiência, a maioria dos sistemas de arquivos agrupa blocos em pedaços maiores, constantemente chamados de **clusters**. A E/S em disco é feita por meio de blocos, mas a E/S do sistema de arquivos é feita por clusters, efetivamente garantindo que a E/S tenha mais características de acesso sequencial e menos de acesso aleatório.

Alguns sistemas operacionais dão a programas especiais a capacidade de usar uma partição do disco como uma grande sequência de blocos lógicos, sem quaisquer estruturas de dados do sistema de arquivos. Essa sequência às vezes é chamada de raw disk e a E/S realizada nela é denominada E/S bruta. Por exemplo, alguns sistemas de banco de dados preferem a E/S bruta por permitir que eles controlem o local exato no disco onde cada registro do banco de dados é armazenado. A E/S bruta evita todos os serviços do sistema de arquivos, como cache do buffer, lock de arquivo, busca prévia, alocação de espaço, nomes de arquivo e diretórios. Podemos tornar certas aplicações mais eficientes implementando seus próprios serviços de armazenamento de uso especial sobre uma partição bruta, mas a maioria das aplicações funciona melhor quando utiliza os serviços regulares do sistema de arquivos.

12.5.2 Bloco de boot

Para um computador dar boot - por exemplo, quando é ligado ou reinicializado - ele precisa ter um programa inicial para executar. Esse programa inicial de *boot* costuma ser simples. Ele inicializa todos os aspectos do sistema, dos registradores da CPU até os controladores de dispositivo e o conteúdo da memória principal, e depois inicia o sistema operacional. Para realizar seu trabalho, o programa de boot encontra o kernel do sistema operacional no disco, carrega o kernel na memória e

salta para um endereço inicial para começar a execução do sistema operacional.

Para a maioria dos computadores, o código de boot é armazenado na **memória somente de leitura (Read-Only Memory - ROM)**. Esse local é conveniente, pois a ROM não precisa de inicialização e está em um local fixo, que o processador pode começar a executar quando ligado ou reinicializado. E como a ROM é somente de leitura, ela não pode ser infectada por vírus de computador. O problema é que a mudança desse código de boot requer a mudança dos chips da ROM. Por esse motivo, os sistemas mais modernos armazenam um pequeno programa carregador de boot na ROM de boot, cuja única tarefa é trazer o programa de boot completo do disco. O programa de boot completo pode ser alterado com facilidade: uma nova versão pode ser copiada para o disco. O programa de boot completo é armazenado em uma partição chamada de "blocos de boot", em um local fixo no disco. Um disco que possui uma partição de boot é chamado **disco de boot** ou **disco do sistema**.

O código da ROM de boot instrui o controlador de disco a ler os blocos de boot para a memória (nenhum driver de dispositivo está carregado nesse ponto) e depois começa a executar esse código. O programa de boot completo é mais sofisticado do que o carregador de boot na ROM; ele é capaz de carregar o sistema operacional inteiro de um local não fixo no disco e iniciar a execução desse sistema operacional. Mesmo assim, o código de boot completo pode ser pequeno.

Vamos considerar, como exemplo, o processo de boot no Windows 2000. O sistema Windows 2000 coloca seu código de boot no primeiro setor do disco rígido (que ele chama de **registro mestre de boot** ou **MBR**). Além do mais, o Windows 2000 permite que um disco rígido seja dividido em uma ou mais partições; uma partição, identificada como **partição de boot**, contém o sistema operacional e drivers de dispositivos. O boot começa, em um sistema Windows 2000, executando o código que é residente na ROM do sistema. Esse código instrui o sistema a ler o código de boot do MBR. Além de conter código de boot, o MBR contém uma tabela listando as partições para o disco rígido e um flag indicando de qual partição o sistema deve ser inicializado. Isso é ilustrado na [Figura 12.9](#). Quando o sistema identifica a partição de boot, ele lê o primeiro setor dessa partição (que é chamado de **setor de boot**) e continua com o restante do processo de boot, que inclui a carga de vários subsistemas e serviços do sistema.

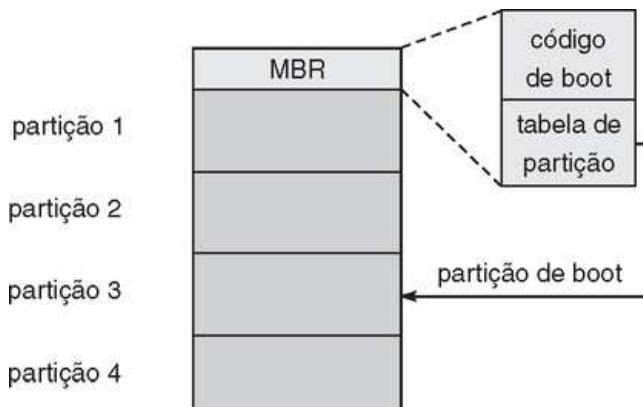


FIGURA 12.9 Booting do disco no Windows 2000.

12.5.3 Blocos defeituosos

Como os discos possuem partes móveis e tolerâncias pequenas (lembre-se de que a cabeça do disco voa logo acima da superfície do disco), eles são passíveis de falha. Às vezes, a falha é completa; nesse caso, o disco precisa ser substituído, com seu conteúdo restaurado de uma mídia de backup para o novo disco. No entanto, normalmente, um ou mais setores se tornam defeituosos. A maioria dos discos vem até mesmo com **blocos defeituosos** de fábrica. Dependendo do disco e do controlador em uso, esses blocos são tratados de várias maneiras.

Em discos simples, como alguns discos com controladores IDE, os blocos defeituosos são tratados manualmente. Por exemplo, o comando `format` do MS-DOS realiza uma formatação lógica e, como parte do processo, analisa o disco para encontrar blocos defeituosos. Se `format` encontrar um bloco defeituoso, ele escreve um valor especial na entrada correspondente na FAT, para indicar às rotinas de alocação que não usem esse bloco. Se os blocos apresentarem defeito durante a operação normal, um programa especial (como `chkdsk`) precisa ser executado manualmente para procurar os blocos defeituosos e separá-los. Os dados que residiam nos blocos defeituosos normalmente são perdidos.

Discos mais sofisticados, como os discos SCSI, usados em PCs de topo de linha e na maioria das estações de trabalho e servidores, são mais inteligentes em relação à recuperação de blocos defeituosos. O controlador mantém uma lista de blocos defeituosos no disco. A lista é inicializada

durante a formatação em baixo nível, na fábrica, e é atualizada durante o tempo de vida do disco. A formatação em baixo nível também reserva setores não visíveis ao sistema operacional. O controlador pode ser informado para substituir logicamente cada setor defeituoso por um dos setores de reserva. Esse esquema é conhecido como **reserva de setor (sector sparing ou sector forwarding)**.

Uma transação típica com setor defeituoso poderia ser a seguinte:

- O sistema operacional tenta ler o bloco lógico 87.
- O controlador calcula o ECC e descobre que o setor está defeituoso. Ele informa essa descoberta ao sistema operacional.
- Da próxima vez que o sistema for reinicializado, um comando especial é executado para dizer ao controlador SCSI para substituir o setor defeituoso por um reserva.
- Depois disso, sempre que o sistema requisitar o bloco lógico 87, a requisição é traduzida para o endereço do setor substituto pelo controlador.

Observe que esse redirecionamento pelo controlador poderia invalidar qualquer otimização pelo algoritmo de escalonamento de disco do sistema operacional! Por esse motivo, a maioria dos discos é formatada para fornecer alguns setores de reserva em cada cilindro, além de um cilindro de reserva. Quando um bloco defeituoso é remapeado, o controlador utiliza um setor de reserva do mesmo cilindro, se possível.

Como alternativa para a reserva de setor, alguns controladores podem ser instruídos a substituir um bloco defeituoso por **deslizamento de setor (sector slipping)**. Vejamos um exemplo. Suponha que o bloco lógico 17 apresente defeito, e o primeiro reserva disponível vem após o setor 202. Nessa situação, o deslizamento de setor remapeia todos os setores de 17 a 202, movendo-os para uma posição adiante; ou seja, o setor 202 é copiado para o reserva, o setor 201 para 202, e depois 200 para 201, e assim por diante, até o setor 18 ser copiado para o setor 19. O deslizamento de setores dessa maneira libera o espaço do setor 18, de modo que o setor 17 possa ser mapeado para ele.

A substituição de um bloco defeituoso em geral não é automática, pois os dados no bloco defeituoso normalmente são perdidos. Erros flexíveis poderiam disparar um processo em que uma cópia dos dados do bloco é feita e o bloco é reservado. Contudo, um **erro rígido** irrecuperável resulta em perda de dados. Qualquer arquivo que usasse esse bloco precisaria ser reparado (por exemplo, a partir da restauração de uma fita de backup), e isso exige intervenção manual.

12.6 Gerenciamento do swap space

O swapping foi apresentado primeiramente na [Seção 8.2](#), onde discutimos a movimentação de processos inteiros entre o disco e a memória principal. O swapping nesse ambiente ocorre quando a quantidade de memória física atinge um ponto criticamente baixo e os processos são movidos da memória para o swap space, para liberar a memória disponível. Na prática, poucos sistemas operacionais modernos implementam o swapping dessa forma. Em vez disso, os sistemas agora combinam o swapping com as técnicas de memória virtual ([Capítulo 9](#)) e páginas de swap, não necessariamente processos inteiros. De fato, alguns sistemas utilizam os termos *swapping* e *paging* para indicar a mesma coisa, refletindo a junção desses dois conceitos.

O **gerenciamento do swap space** é outra tarefa do sistema operacional em baixo nível. A memória virtual usa o espaço do disco como uma extensão da memória principal. Como o acesso ao disco é muito mais lento do que o acesso à memória, o uso do swap space diminui significativamente o desempenho do sistema. O objetivo principal para o projeto e a implementação do swap space é prover o melhor throughput para o sistema de memória virtual. Nesta seção, discutimos como o swap space é utilizado, onde ele está localizado no disco e como é gerenciado.

12.6.1 Uso do swap space

O swap space é usado de várias maneiras por sistemas operacionais, dependendo dos algoritmos de gerenciamento de memória implementados em uso. Por exemplo, os sistemas que implementam a swap podem usar o swap space para manter a imagem inteira do processo, incluindo os segmentos de código e de dados. Os sistemas de paginação podem armazenar páginas retiradas da memória principal. A quantidade do swap space necessária em um sistema, portanto, pode variar desde alguns megabytes de espaço em disco até gigabytes, dependendo da quantidade da memória física, da quantidade da memória virtual que fornece suporte e do modo como a memória virtual é utilizada.

Observe que é mais seguro superestimar do que subestimar o swap space, visto que, se um sistema estiver sem swap space, ele pode ser forçado a abortar os processos ou pode falhar. Superestimar desperdiça espaço em disco que poderia ser usado para os arquivos, mas não causa dano algum. Alguns sistemas recomendam a quantidade a ser reservada para o swap space. O Solaris, por exemplo, sugere a definição do swap space igual à quantidade pela qual a memória virtual excede a memória física paginável. No passado, o Linux sugeria a definição do swap space com o dobro da quantidade da memória física, mas a maioria dos sistemas Linux agora usa muito menos swap space. Na verdade, existe atualmente muito debate na comunidade Linux considerando se afinal deve haver algum swap space!

Alguns sistemas operacionais, como o UNIX, permitem o uso de vários swap spaces. Esses swap spaces são colocados em discos separados, de modo que a carga colocada sobre o sistema de E/S pela paginação e swap pode ser espalhada pelos dispositivos de E/S do sistema.

12.6.2 Locação do swap space

Um swap space pode residir em dois lugares. Ele pode estar incluído no sistema de arquivos normal ou estar em uma partição de disco separada. Se o swap space for um grande arquivo dentro do sistema de arquivos, as rotinas normais do sistema de arquivos poderão ser usadas para criá-lo, nomeá-lo e alocar seu espaço. Essa técnica, embora fácil de implementar, também é ineficaz. A navegação pela estrutura de diretório e pelas estruturas de dados para alocação de disco leva tempo e (possivelmente) acessos extras ao disco. A fragmentação externa pode aumentar bastante os tempos de troca, forçando múltiplas buscas durante a leitura ou a escrita da imagem de um processo. Podemos melhorar o desempenho colocando as informações de local do bloco em caching na memória física e usando ferramentas especiais para alocar fisicamente blocos contíguos para o arquivo de swap, mas o custo de atravessar as estruturas de dados do sistema de arquivos ainda permanece.

Como alternativa, o swap space pode ser criado em uma partição **bruta** separada, enquanto nenhum sistema de arquivos ou estrutura de diretório é colocado nesse espaço. Em vez disso, um gerenciador de armazenamento do swap space separado é utilizado para alocar e desalocar os blocos da partição bruta. Esse gerenciador utiliza algoritmos otimizados para velocidade, em vez de eficiência de armazenamento, porque o swap space é acessado com mais frequência do que os sistemas de arquivo (quando é usado). A fragmentação interna pode aumentar, mas isso é aceitável porque a vida dos dados no swap space é muito menor do que a dos arquivos no sistema de arquivos. O swap space é reinicializado no momento do boot, de modo que qualquer fragmentação tem vida curta. Essa técnica de partição bruta cria uma quantidade fixa de swap space durante o particionamento do disco. A inclusão de mais swap space exige o reparticionamento do disco (que envolve a movimentação de outras partições do sistema de arquivos ou a sua destruição e restauração a partir do backup) ou a inclusão de outro swap space em outro lugar.

Alguns sistemas operacionais são flexíveis e podem fazer a troca tanto em partições brutas quanto no espaço do sistema de arquivos. O Linux é um exemplo: a política e a implementação são separadas, permitindo ao administrador da máquina decidir que tipo será usado. A escolha é entre a conveniência da alocação e gerenciamento no sistema de arquivos e o desempenho do swap nas partições brutas.

12.6.3 Gerenciamento do swap space: um exemplo

Podemos ilustrar como o swap space é usado seguindo a evolução do swap e a paginação no UNIX. O kernel tradicional do UNIX começou com uma implementação do swap que copiava processos inteiros entre regiões de disco contíguas de disco e a memória. O UNIX depois evoluiu para uma combinação de swap e paginação, à medida que o hardware de paginação se tornava disponível.

No Solaris 1 (SunOS), os projetistas fizeram mudanças nos métodos-padrão do UNIX para melhorar a eficiência e refletir os desenvolvimentos tecnológicos. Quando um processo é executado, as páginas do segmento de texto contendo código são trazidas do sistema de arquivos, acessadas na memória principal e retiradas se forem selecionadas para paginação. É mais eficiente reler uma página do sistema de arquivos do que escrevê-la no swap space e depois relê-la de lá. O swap space é usado apenas como um local de armazenamento para páginas de memória **anônima**, que inclui a memória alocada para a pilha, heap e dados não inicializados de um processo.

ESTRUTURANDO O RAID

O armazenamento RAID pode ser estruturado de diversas maneiras. Por exemplo, um sistema pode ter discos conectados diretamente aos seus barramentos. Nesse caso, o sistema operacional ou software do sistema pode implementar a funcionalidade RAID. Como alternativa, um controlador de host inteligente pode controlar vários discos conectados e pode implementar RAID sobre esses discos no hardware. Finalmente, pode-se usar um **array de armazenamento**, ou **array RAID**. Um array RAID é uma unidade isolada, com seu próprio controlador, cache (normalmente) e discos. Ele é conectado ao host por meio de um ou mais controladores SCSI ATA padrão ou FC. Essa configuração comum permite que qualquer sistema operacional e software sem funcionalidade RAID tenha discos protegidos por RAID. Ela é usada até mesmo em sistemas que possuem camadas de software RAID, devido à sua simplicidade e flexibilidade.

Outras mudanças foram feitas nas últimas versões do Solaris. A maior delas é que o Solaris só aloca o swap space quando uma página é forçada a sair da memória física, em vez de quando a página da memória virtual é criada. Esse esquema provê melhor desempenho nos computadores modernos, que possuem mais memória física do que os sistemas mais antigos e costumam paginar menos.

O Linux é semelhante ao Solaris porque o swap space é usado apenas para memória anônima ou para regiões da memória compartilhadas por vários processos. O Linux permite uma ou mais áreas de swap a serem estabelecidas. Uma área de swap pode ser um arquivo de swap em um sistema de arquivos normal ou uma partição de espaço de swap bruta. Cada área de swap consiste em uma série de **slots de página** de 4 KB, que são usados para manter as páginas passadas. Associado a cada área de swap existe um **mapa de swap** - um array de contadores inteiros, cada um correspondendo a um slot de página na área de swap. Se o valor de um contador for 0, o slot de página correspondente estará disponível. Os valores maiores que 0 indicam que o slot de página é ocupado por uma página passada. O valor do contador indica o número de mapeamentos a terem as páginas passadas; por exemplo, um valor 3 indica que a página passada é mapeada para três processos diferentes (que podem ocorrer se a página passada estiver armazenando uma região da memória compartilhada por três processos). As estruturas de dados para a passagem nos sistemas Linux aparecem na [Figura 12.10](#).

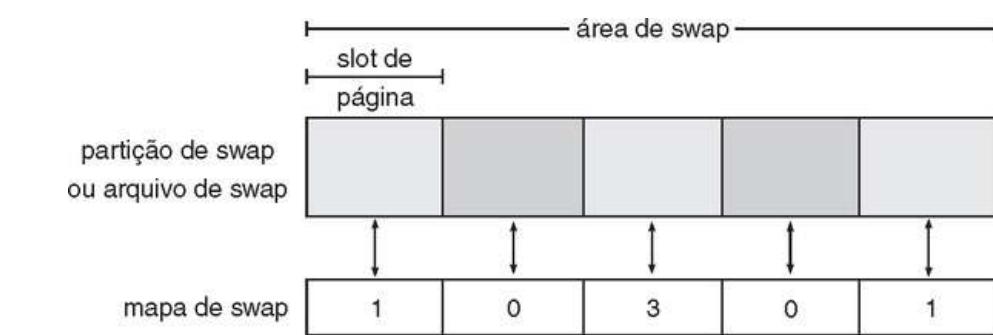


FIGURA 12.10 As estruturas de dados para o swapping nos sistemas Linux.

12.7 Estrutura RAID

As unidades de disco continuaram ficando menores e mais baratas, de modo que agora é economicamente viável conectar uma grande quantidade de discos a um sistema de computador. Ter uma grande quantidade de discos em um sistema gera oportunidades de melhorar a taxa em que os dados podem ser lidos ou escritos, se os discos forem operados em paralelo. Além do mais, essa configuração provê o potencial de melhorar a confiabilidade do armazenamento de dados, já que podemos armazenar informações redundantes em vários discos. Assim, a falha em um disco não ocasiona perda de dados. Diversas técnicas de organização de disco, chamadas coletivamente de **arrays redundantes de discos baratos (Redundant Arrays of Inexpensive Disks - RAIDs)**, normalmente são usadas para resolver questões de desempenho e confiabilidade.

No passado, RAIDs compostos de pequenos discos baratos eram vistos como uma alternativa econômica para discos grandes e caros; hoje, RAIDs são usados por sua maior confiabilidade e maior taxa de transferência de dados, e não por motivos econômicos. Daí, o *I* de RAID, que significava “inexpensive”, agora significa “independent”.

12.7.1 Melhoria de confiabilidade via redundância

Primeiro, vamos considerar a confiabilidade dos RAIDs. A chance de algum disco dentre um conjunto de N discos falhar é muito maior do que a chance de um único disco específico falhar. Suponha que o **tempo médio para a falha (mean time to failure - mttf)** de um único disco seja de 100.000 horas. Então, o tempo médio para a falha de algum disco em um array de 100 discos será $100.000/100 = 1.000$ horas ou 41,66 dias, o que não é muito tempo! Se armazenarmos apenas uma cópia dos dados, então cada falha no disco resultará na perda de uma quantidade significativa de dados – essa razão de perda de dados tão alta é inaceitável.

A solução para o problema de confiabilidade é introduzir a **redundância**; armazenamos informações extras que normalmente não são necessárias, mas que podem ser usadas no caso de falha de um disco, para reconstruir as informações perdidas. Assim, mesmo que um disco falhe, os dados não são perdidos.

A técnica mais simples (porém, mais cara) de introduzir a redundância é duplicar cada disco. Essa técnica é chamada **espelhamento (mirroring)**. Com o espelhamento, um disco lógico consiste em dois discos físicos, e cada escrita é executada nos dois discos. Se um dos discos falhar, os dados podem ser lidos do outro. Os dados só serão perdidos se o segundo disco falhar antes de o primeiro disco que falhou ser substituído.

O tempo médio para a falha de um volume espalhado – sendo *falha* a perda de dados – depende de dois fatores. Um é o tempo médio para a falha dos discos individuais; o outro é o **tempo médio para o reparo (mean time to repair - mttr)**, que é o tempo médio necessário para substituir um disco defeituoso e restaurar seus dados. Suponha que as falhas dos dois discos sejam **independentes**, ou seja, a falha de um disco não está ligada à falha do outro. Então, se o tempo médio para a falha de um único disco for 100.000 horas e o tempo médio para o reparo for de 10 horas, então o **tempo médio para a perda de dados (mean time to data loss - mttdl)** de um sistema de disco espelhado é $100.000^2/(2 * 10) = 500 * 10^6$ horas ou 57.000 anos!

Você precisa estar ciente de que a suposição de independência das falhas de disco não é válida. Faltas de energia e desastres naturais, como terremotos, incêndios e inundações, podem resultar em danos aos dois discos ao mesmo tempo. Além disso, defeitos de fabricação em lotes de discos podem causar falhas correlatas. À medida que os discos envelhecem, a probabilidade de falha aumenta, aumentando a chance de um segundo disco falhar enquanto o primeiro está sendo reparado. Contudo, apesar de todas essas considerações, os sistemas de disco espelhados proveem muito mais confiabilidade do que os sistemas de único disco.

A falta de energia é uma fonte de preocupação em particular, pois ocorrem com muito mais frequência do que os desastres naturais. Entretanto, mesmo com o espelhamento de discos, se houver escrita em andamento no mesmo bloco nos dois discos e faltar energia antes de os dois blocos estarem escritos, os dois blocos podem estar em um estado incoerente. Uma solução para esse problema é escrever uma cópia primeiro, depois a outra. Outra é acrescentar um cache de **RAM não volátil (NVRAM)** ao array RAID. Esse cache write-back é protegido contra perda de dados durante as faltas de energia, de modo que a escrita pode ser considerada completa nesse ponto, supondo que a NVRAM tenha algum tipo de proteção e correção de erro, como ECC ou espelhamento.

12.7.2 Melhoria de desempenho via paralelismo

Agora, vamos considerar como o acesso paralelo a vários discos melhora o desempenho. Com o espelhamento de disco, a taxa pela qual as requisições de disco podem ser tratadas é dobrada, pois as requisições de leitura podem ser enviadas a qualquer disco (desde que os dois discos em um par estejam funcionando, como quase sempre acontece). A taxa de transferência de cada leitura é igual

à de um sistema de único disco, mas a quantidade de leituras por unidade de tempo dobra.

Com vários discos, também podemos melhorar a taxa de transferência espalhando os dados por vários discos. Em sua forma mais simples, o **espalhamento de dados (data striping)** consiste em distribuir os bits de cada byte por vários discos; esse espalhamento é denominado **espalhamento em nível de bit**. Por exemplo, se tivermos um array de oito discos, escrevemos o bit i de cada byte no disco i . O array de oito discos pode ser tratado como um único disco, com setores oito vezes maiores que o tamanho normal e, mais importante, oito vezes maiores que a taxa de acesso. Nesse tipo de organização, cada disco participa de cada acesso (leitura ou escrita), de modo que a quantidade de acessos que podem ser processados por segundo é aproximadamente a mesma que em um único disco, mas cada acesso pode ler oito vezes mais dados no mesmo tempo gasto para um único disco.

O espalhamento em nível de bit pode ser generalizado para uma quantidade de discos que seja um múltiplo de 8 ou divisor de 8. Por exemplo, se usarmos um array de quatro discos, os bits i e $4 + i$ de cada byte vão para o disco i . Além do mais, o espalhamento não precisa estar no nível de bits de um byte: por exemplo, no **espalhamento em nível de bloco**, os blocos de um arquivo são espalhados por vários discos; com n discos, o bloco i de um arquivo vai para o disco $(i \bmod n) + 1$. Outros níveis de espalhamento, como bytes de um setor ou setores de um bloco, também são possíveis. O espalhamento em nível de bit é o mais comum.

No paralelismo de um sistema de disco, como atingido por meio do espalhamento, existem dois objetivos principais:

1. Aumentar o throughput de múltiplos acessos pequenos (ou seja, acessos de página) pelo balanceamento de carga.
2. Reduzir o tempo de resposta de acessos grandes.

12.7.3 Níveis de RAID

O espelhamento provê alta confiabilidade, mas é dispendioso. O espalhamento provê altas taxas de transferência de dados, mas não melhora a confiabilidade. Foram propostos diversos esquemas para fornecer redundância com menor custo, usando a ideia de espalhamento de disco combinada com bits de “paridade” (que descrevemos em seguida). Esses esquemas possuem diferentes opções de custo-desempenho e são classificados em níveis denominados **níveis de RAID**. Descrevemos os diversos níveis aqui; a [Figura 12.11](#) os mostra em forma de desenho (na figura, P indica bits de correção de erro e C indica uma segunda cópia dos dados). Em todos os casos representados na figura, o conteúdo de quatro discos de dados é armazenado e os discos extras são usados para armazenar informações redundantes para a recuperação de falhas.

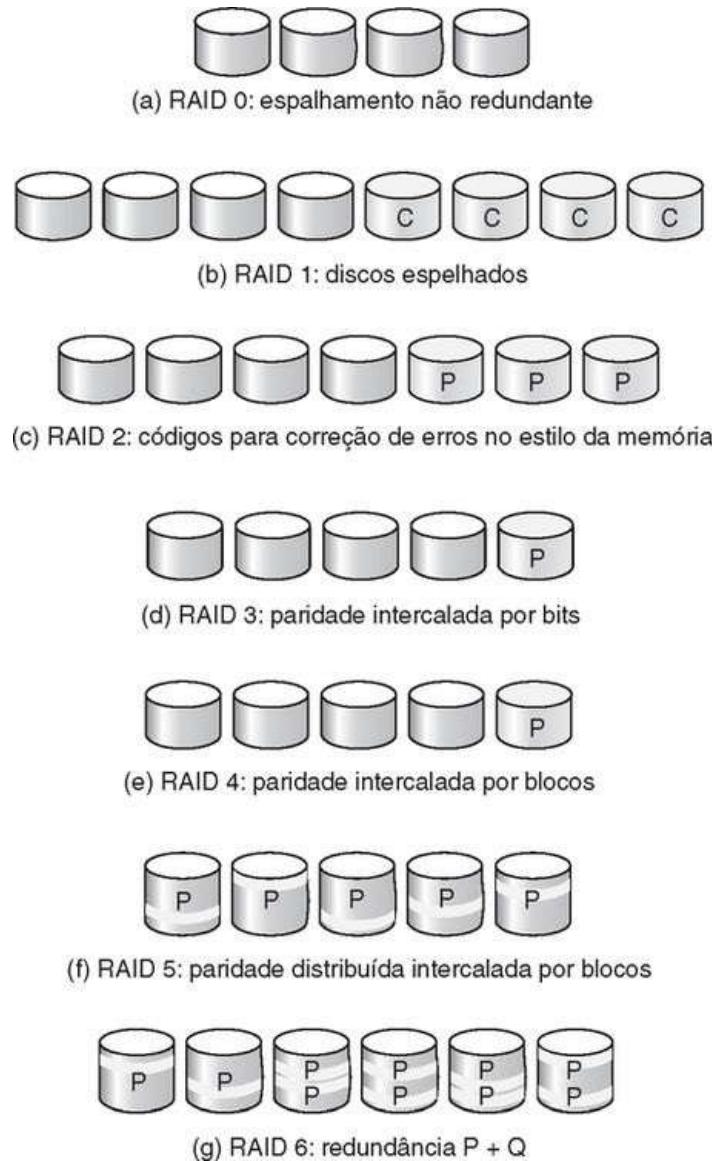


FIGURA 12.11 Níveis de RAID.

- **RAID nível 0.** RAID nível 0 refere-se aos arrays de disco com espalhamento no nível de blocos, mas sem redundância (como espelhamento ou bits de paridade), como mostrado na [Figura 12.11a](#).
- **RAID nível 1.** RAID nível 1 refere-se ao espelhamento de disco. A [Figura 12.11b](#) mostra uma organização espelhada.
- **RAID nível 2.** RAID nível 2 também é conhecido como **organização com código para a correção de erros (ECC) no estilo da memória**. Os sistemas de memória há muito tempo têm detectado determinados erros usando bits de paridade. Cada byte em um sistema de memória pode ter um bit de paridade associado a ele, que registra se a quantidade de bits no byte definidos como 1 é par (paridade = 0) ou ímpar (paridade = 1). Se um dos bits no byte for modificado (ou 1 tornando-se 0 ou 0 tornando-se 1), a paridade do byte muda e, assim, não combinará com a paridade armazenada. De modo semelhante, se o bit de paridade for modificado, ele não combinará com a paridade calculada. Assim, todos os erros em único bit serão detectados pelo sistema de memória. Os esquemas de correção de erro armazenam dois ou mais bits extras e podem reconstruir os dados se um único bit for danificado. A ideia do ECC pode ser usada diretamente nos arrays de disco por meio do espalhamento de bytes entre os discos. Por exemplo, o primeiro bit de cada byte poderia ser armazenado no disco 1, o segundo bit no disco 2, e assim por diante, até o oitavo bit ser armazenado no disco 8, e os bits de correção de erro serem armazenados em outros discos. Esse esquema é mostrado em forma representativa na [Figura 12.11c](#), onde os discos rotulados com P (paridade) armazenam os bits de correção de erro. Se um dos discos falhar, os bits restantes do byte e os bits de correção de erro associados poderão ser lidos de outros discos e usados para reconstruir os dados danificados. Observe que RAID nível 2 exige apenas o custo adicional de três discos para quatro discos de dados, ao contrário do RAID nível 1, que exige um custo adicional de quatro discos.

■ **RAID nível 3.** RAID nível 3, ou **organização com paridade intercalada por bits**, melhora o nível 2 considerando o fato de que, diferente dos sistemas de memória, os controladores de disco podem detectar se um setor foi lido corretamente, de modo que um único bit de paridade pode ser usado para correção de erro, bem como para detecção. A ideia é a seguinte: se um dos setores for danificado, saberemos qual é esse setor e poderemos descobrir se qualquer bit no setor é 1 ou 0 calculando a paridade dos bits correspondentes dos setores nos outros discos. Se a paridade dos bits restantes for igual à paridade armazenada, o bit que falta é 0; caso contrário, é 1. RAID nível 3 é tão bom quanto o nível 2, porém menos dispendioso na quantidade de discos extras exigida (ele tem o custo adicional de apenas um disco), de modo que o nível 2 não é usado na prática. Esse esquema é representado na [Figura 12.11d](#).

RAID nível 3 possui duas vantagens em relação ao nível 1. Primeiro, o armazenamento adicional é reduzido visto que um disco de paridade é necessário para vários discos normais, ao passo que um disco de espelho é necessário para cada disco no nível 1. Segundo, como as leituras e escritas de um byte estão espalhadas por vários discos com o espalhamento de dados em N discos, a taxa de transferência para leitura ou escrita de um único bloco é N vezes mais rápida que uma organização RAID nível 1. No lado negativo, RAID nível 3 admite uma quantidade menor de E/S por segundo, pois cada disco precisa participar de cada requisição de E/S.

Outro problema de desempenho com RAID 3 (assim como todos os níveis RAID baseados em paridade) é o custo da computação e escrita da paridade. Esse custo adicional resulta em escritas bem mais lentas, em comparação com os arrays RAID sem paridade. Para moderar essa penalidade no desempenho, muitos arrays de armazenamento RAID incluem um controlador de hardware com hardware de paridade dedicado. Esse controlador desloca o cálculo de paridade da CPU para o array. O array também possui um cache de RAM não volátil (Nonvolatile RAM – NVRAM), para armazenar os blocos enquanto a paridade é calculada e manter em buffer as escritas do controlador para os discos físicos. Essa combinação pode tornar o RAID com paridade quase tão rápido quanto o sem paridade. Na verdade, um array RAID com caching e paridade pode ser superior a um RAID sem caching e sem paridade.

■ **RAID nível 4.** O RAID nível 4, ou **organização com paridade intercalada por blocos**, utiliza o espalhamento no nível de bloco, como no RAID 0, e também mantém um bloco de paridade em um disco separado, para os blocos correspondentes dos N outros discos. Esse esquema aparece representado na [Figura 12.11e](#). Se um dos discos falhar, o bloco de paridade pode ser usado com os blocos correspondentes a partir dos outros discos para restaurar os blocos do disco que falhou.

Uma leitura de bloco acessa apenas um disco, permitindo que outras requisições sejam processadas pelos outros discos. Assim, a taxa de transferência de dados para cada acesso é mais lenta, mas vários acessos de leitura podem ser feitos em paralelo, levando a uma taxa de E/S geral maior. A taxa de transferência para grandes leituras é alta, pois todos os discos podem ser lidos em paralelo; grandes escritas também possuem grandes taxas de transferência, pois os dados e a paridade podem ser escritos em paralelo.

Pequenas escritas independentes não podem ser realizadas em paralelo. Uma escrita de dados do sistema operacional menor que um bloco exige que o bloco seja lido, modificado com os novos dados e escrito de volta. O bloco de paridade precisa ser atualizado também. Isso é conhecido como **leitura-modificação-escrita**. Assim, uma única escrita exige quatro acessos ao disco: dois para ler os dois blocos antigos e dois para escrever os dois novos blocos.

O WAFL ([Capítulo 1](#)) utiliza RAID nível 4 porque esse nível RAID permite que os discos sejam acrescentados a um conjunto RAID de modo transparente. Se os discos acrescentados forem inicializados com blocos contendo zeros, então o valor de paridade não muda, e o conjunto RAID ainda está correto.

■ **RAID nível 5.** O RAID nível 5, ou **paridade distribuída intercalada por blocos**, difere do nível 4 por espalhar os dados e a paridade entre todos os $N + 1$ discos, em vez de armazenar dados em N discos e paridade em um disco. Para cada bloco, um dos discos armazena a paridade e os outros armazenam os dados. Por exemplo, com um array de cinco discos, a paridade para o bloco n é armazenada no disco $(n \bmod 5) + 1$; os blocos n dos outros quatro discos armazenam os dados reais para esse bloco. Essa configuração é representada na [Figura 12.11f](#), onde os P_s são distribuídos por todos os discos. O bloco de paridade não pode armazenar a paridade para os blocos no mesmo disco, pois uma falha no disco resultaria em perda de dados e também paridade e, portanto, não seria recuperável. Espalhando a paridade por todos os discos no conjunto, o RAID 5 evita o potencial de uso demasiado de um único disco de paridade que pode ocorrer com RAID 4. O RAID 5 é o sistema RAID de paridade mais comum.

■ **RAID nível 6.** O RAID nível 6, também chamado **esquema de redundância P + Q**, é muito semelhante ao RAID nível 5, mas armazena informações redundantes extras para proteger contra múltiplas falhas no disco. Em vez de usar a paridade, são usados códigos para correção de erros, como os **códigos Reed-Solomon**. No esquema mostrado na [Figura 12.11g](#), 2 bits de dados redundantes são armazenados para cada 4 bits de dados - em vez de 1 bit de paridade no nível 5 - e o sistema pode tolerar duas falhas de disco.

■ **RAID nível 0 + 1 e 1 + 0.** O RAID nível 0 + 1 refere-se a uma combinação de níveis RAID 0 e 1.

O RAID 0 provê desempenho, enquanto o RAID 1 provê confiabilidade. Em geral, esse nível provê melhor desempenho do que o RAID 5. Ele é comum em ambientes em que o desempenho e a confiabilidade são importantes. Infelizmente, ele duplica a quantidade de discos necessária para o armazenamento, assim como o RAID 1 e, por isso, é mais dispendioso. No RAID 0 + 1, um conjunto de discos é espalhado, e depois tudo isso é espelhado para outro grupo equivalente.

Outra opção de RAID que está se tornando disponível comercialmente é RAID 1 + 0, em que os discos são espelhados em pares, e depois os pares de espelho resultantes são espalhados. Esse esquema possui algumas vantagens teóricas em relação ao RAID 0 + 1. Por exemplo, se um único disco falhar no RAID 0 + 1, o espalhamento inteiro ficará inacessível, deixando apenas o outro espalhamento disponível. Com uma falha no RAID 1 + 0, o único disco está indisponível, mas o disco que o espelha ainda está disponível, assim como todo o restante dos discos (Figura 12.12).

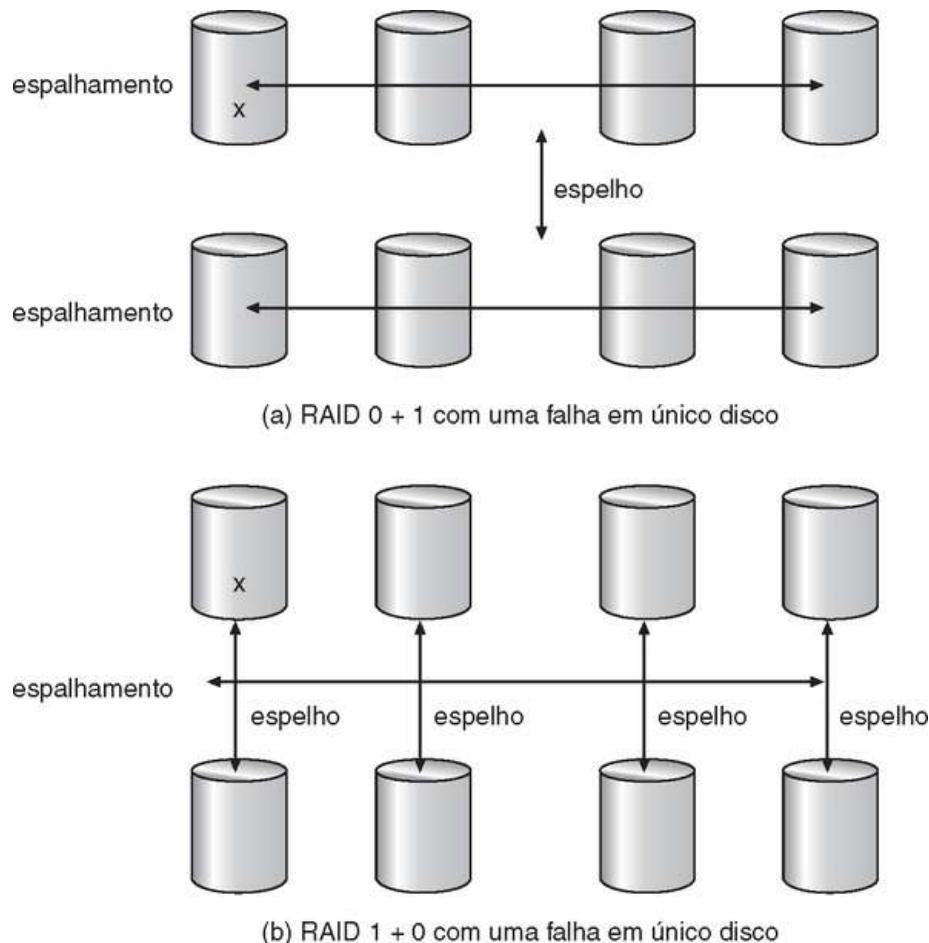


FIGURA 12.12 RAID 0 + 1 e 1 + 0.

Diversas variações foram propostas para os esquemas RAID básicos descritos aqui. Como resultado, poderá haver alguma confusão a respeito das definições exatas dos diferentes níveis de RAID.

A implementação de RAID é outra área de variação. Considere as seguintes camadas em que o RAID pode ser implementado.

- O software de gerenciamento de volume pode implementar o RAID dentro do kernel ou na camada de software de sistema. Nesse caso, o hardware de armazenamento pode fornecer um mínimo de recursos e ainda fazer parte de uma solução RAID completa. O RAID de paridade é muito lento quando implementado no software, de modo que normalmente RAID 0, 1 ou 0 + 1 é utilizado.
- O RAID pode ser implementado no hardware HBA (host bus-adapter). Somente os discos conectados diretamente ao HBA podem fazer parte de determinado conjunto RAID. Essa solução é econômica, mas não muito flexível.
- O RAID pode ser implementado no hardware do array de armazenamento. O array de armazenamento pode criar conjuntos RAID de diversos níveis e ainda dividir esses conjuntos em volumes menores, que são apresentados ao sistema operacional. O sistema operacional só precisa implementar o sistema de arquivos em cada um dos volumes. Os arrays podem ter múltiplas conexões disponíveis ou podem fazer parte de uma SAN, permitindo que múltiplos hosts tirem proveito dos recursos do array.

■ O RAID pode ser implementado na camada de interconexão da SAN por dispositivos de virtualização de disco. Nesse caso, um dispositivo fica entre os hosts e o armazenamento. Ele aceita comandos dos servidores e gerencia o acesso ao armazenamento. Ele poderia fornecer espelhamento, por exemplo, escrevendo cada bloco em dois dispositivos de armazenamento separados.

Outros recursos, como snapshots e replicação, também podem ser implementados em cada um desses níveis. A **replicação** envolve a duplicação automática de escritas entre sites separados para redundância e recuperação de desastre. A replicação pode ser síncrona ou assíncrona. Na replicação síncrona, cada bloco precisa ser escrito local e remotamente antes que a escrita seja considerada completa, enquanto na replicação assíncrona as escritas são agrupadas e escritas periodicamente. A replicação assíncrona pode resultar em perda de dados se o site principal falhar, mas é mais rápida e não possui limitações de distância.

A implementação desses recursos difere, dependendo da camada em que o RAID é implementado. Por exemplo, se o RAID for implementado no software, então cada host pode ter que executar e gerenciar sua própria replicação. Se a replicação for implementada no array de armazenamento ou na interconexão SAN, quaisquer que sejam o sistema operacional ou os recursos, os dados do host podem ser replicados.

O ARRAY DE ARMAZENAMENTO INSERV

A inovação, em um esforço para oferecer soluções melhores, mais rápidas e menos dispendiosas, constantemente ofusca as linhas que separavam tecnologias anteriores. Considere o array de armazenamento InServ da 3Par. Diferente da maioria dos outros arrays de armazenamento, o InServ não exige que um conjunto de discos seja configurado em um nível de RAID específico. Em vez disso, cada disco é dividido em “pedaços” de 256 MB. RAID é, então, aplicado em um nível de pedaço. Um disco, assim, pode participar de diversos níveis de RAID à medida que seus pedaços forem usados para diversos volumes.

O InServ também oferece snapshots semelhantes àqueles criados pelo sistema de arquivos WAFL. O formato dos snapshots InServ pode ser de leitura-escrita, bem como apenas somente de leitura, permitindo que vários hosts montem cópias de determinado sistema de arquivos sem precisar de suas próprias cópias do sistema de arquivos inteiro. Quaisquer mudanças que um host faça em sua própria cópia são copy-on-write e, assim, não são refletidas nas outras cópias.

Outro aspecto da maioria das implementações RAID é um disco ou discos hot spare. Um disco **hot spare (reserva a quente)** não é usado para dados, mas é configurado para ser usado como um substituto se algum outro disco falhar. Por exemplo, um disco hot spare pode ser usado para reconstruir um par do espelho se um dos discos no par falhar. Desse modo, o nível RAID pode ser restabelecido automaticamente, sem esperar a substituição do disco que falhou. A alocação de mais de um disco hot spare permite que mais de uma falha seja reparada sem intervenção humana.

12.7.4 Selecionando um nível de RAID

Dadas as muitas escolhas que eles têm, como os projetistas de sistemas escolhem um nível RAID? Uma consideração é o desempenho da reconstrução. Se um disco falhar, o tempo para recriar seus dados pode ser significativo. Esse pode ser um fator importante se uma fonte contínua de dados for exigida, como acontece nos sistemas de banco de dados de alto desempenho ou interativos. Além do mais, o desempenho da reconstrução influencia o tempo médio para a falha.

O desempenho da reconstrução varia com o nível de RAID utilizado. A reconstrução é mais fácil para RAID nível 1, pois os dados podem ser copiados de outro disco; para os outros níveis, precisamos acessar todos os outros discos no array para recriar dados em um disco que falhou. Os tempos de reconstrução podem ser horas para reconstrução no RAID 5 de grandes conjuntos de discos.

O RAID nível 0 é usado nas aplicações de alto desempenho onde a perda de dados não é crítica. O RAID nível 1 é popular para aplicações que exigem alta confiabilidade com recuperação rápida. RAID 0 + 1 e 1 + 0 são usados onde o desempenho e a confiabilidade são importantes, por exemplo, para pequenos bancos de dados. Devido ao grande espaço adicional necessário no RAID 1, o RAID nível 5 normalmente é recomendado para armazenar grande volume de dados. O nível 6 não é aceito atualmente por muitas implementações RAID, mas pode fornecer melhor confiabilidade do que o nível 5.

Os projetistas e administradores de armazenamento de sistemas RAID também precisam tomar várias outras decisões. Por exemplo, quantos discos devem estar em um set RAID? Quantos bits devem ser protegidos em cada bit de paridade? Se mais discos estiverem em um array, as taxas de transferência de dados serão mais altas, mas o sistema será mais dispendioso. Se mais bits estiverem protegidos por um bit de paridade, o espaço adicional devido aos bits de paridade será menor, mas a chance de um segundo disco falhar antes de o primeiro disco que falhou ser reparado será maior, e isso resultará em perda de dados.

Outra inovação é o **armazenamento utilitário**. Alguns sistemas de arquivos não se expandem nem encolhem. Nesses sistemas, o tamanho original é o único tamanho, e qualquer mudança exige cópia de dados. Um administrador pode configurar o InServ para oferecer a um host uma grande quantidade de armazenamento lógico que ocupa inicialmente apenas uma pequena quantidade de armazenamento físico. Quando o host começar a usar o armazenamento, os discos não usados serão alocados ao host, até o nível lógico original. O host, assim, pode acreditar que possui um espaço de armazenamento fixo, criar seus próprios sistemas de arquivos lá, e assim por diante. Os discos podem ser acrescentados ou removidos do sistema de arquivos pelo InServ sem que os sistemas de arquivos observem a mudança. Esse recurso pode reduzir o número de unidades necessárias pelos hosts ou, pelo menos, adiar a compra de discos até que eles sejam realmente necessários.

12.7.5 Extensões

Os conceitos de RAID têm sido generalizados para outros dispositivos de armazenamento, incluindo arrays de fitas, e até mesmo para o broadcast de dados por sistemas sem fio. Quando aplicadas aos arrays de fitas, as estruturas RAID são capazes de recuperar dados mesmo que uma das fitas em um array de fitas esteja danificada. Quando aplicado ao broadcast de dados, um bloco de dados é separado em unidades curtas e transmitido com uma unidade de paridade; se uma das unidades não for recebida por algum motivo, ela pode ser reconstruída a partir das outras unidades. Normalmente, os robôs de unidade de fita, contendo várias unidades de fita, espalharão dados por todas as unidades, para aumentar o throughput e diminuir o tempo de backup.

12.7.6 Problemas com RAID

Infelizmente, um RAID nem sempre garante que os dados estão disponíveis para o sistema operacional e seus usuários. Um ponteiro para um arquivo poderia estar errado, por exemplo, ou os ponteiros dentro da estrutura do arquivo poderiam estar errados. Escritas incompletas, se não recuperadas devidamente, podem resultar em dados corrompidos. Algum outro processo também poderia escrever acidentalmente sobre as estruturas de um sistema de arquivos. O RAID protege contra erros físicos da mídia, mas não outros erros de hardware ou software. Com um panorama tão grande de bugs de software e hardware, são grandes os perigos em potencial para os dados em um sistema.

O sistema de arquivos ZFS do Solaris utiliza uma técnica inovadora para solucionar esses problemas usando **checksums** (somas de verificação) - uma técnica que é usada para verificar a integridade dos dados. O sistema de arquivos ZFS mantém as checksums internas de todos os blocos, incluindo dados e metadados. Essas checksums não são mantidas com o bloco que está sendo verificado. Em vez disso, elas são armazenadas com o ponteiro para esse bloco ([Figura 12.13](#)). Considere um inode com ponteiros para seus dados. Dentro do inode está a checksum de cada bloco de dados. Se houver um problema com os dados, a checksum será incorreta e o sistema de arquivos saberá a respeito disso. Se os dados forem espelhados e houver um bloco com a checksum correta e um com uma checksum incorreta, o ZFS atualizará automaticamente o bloco ruim com o bom. De modo semelhante, a entrada de diretório que aponta para o inode tem uma checksum para o inode. Qualquer problema no inode é detectado quando o diretório é acessado. Essa checksum ocorre por todas as estruturas do ZFS, fornecendo um nível de consistência, detecção de erro e correção de erro muito mais alto do que existe nos conjuntos de discos RAID ou sistemas de arquivos-padrão. O overhead extra que é criado pelo cálculo da checksum e ciclos extras de leitura-modificação-escrita não é observável, pois o desempenho geral do ZFS é muito rápido.

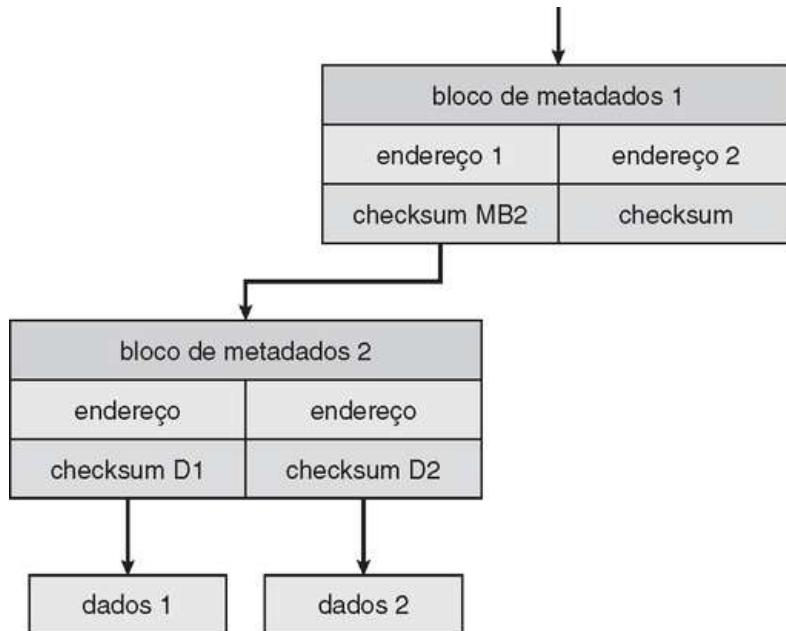
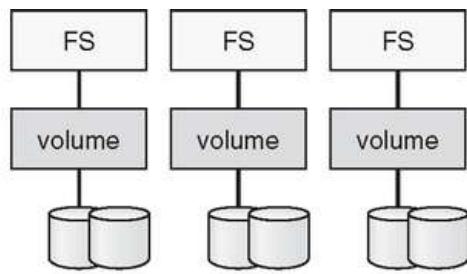


FIGURA 12.13 ZFS verifica a soma de todos os metadados e dados.

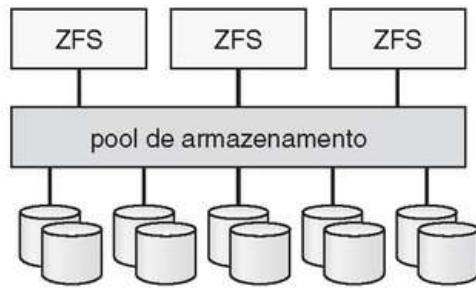
Outra questão com a maioria das implementações RAID é a falta de flexibilidade. Considere um array de armazenamento com 20 discos divididos em 4 conjuntos de 5 discos. Cada conjunto de 5 discos é um conjunto RAID nível 5. Como resultado, existem 4 volumes separados, cada um mantendo um sistema de arquivos. Mas e se um sistema de arquivos for muito grande para caber em um conjunto RAID nível 5 com 5 discos? E se outro sistema de arquivos precisar de muito pouco espaço? Se esses fatores forem conhecidos com antecedência, então os discos e volumes podem ser alocados devidamente. Porém, com muita frequência, o uso de disco e os requisitos variam com o tempo.

Mesmo que o array de armazenamento permitisse que o conjunto inteiro de vinte discos fosse criado como um grande conjunto RAID, outras questões poderiam surgir. Diversos volumes de vários tamanhos poderiam ser montados no conjunto. Porém, alguns gerenciadores de volumes não nos permitem alterar o tamanho do volume. Nesse caso, ficaríamos com o mesmo problema já descrito - divergência nos tamanhos do sistema de arquivos. Alguns gerenciadores de volumes permitem alterações de tamanho, mas alguns sistemas de arquivos não permitem o crescimento ou a redução do sistema de arquivos. Os volumes poderiam mudar de tamanho, mas os sistemas de arquivos precisariam ser recriados para tirar proveito dessas alterações.

ZFS combina gerenciamento do sistema de arquivos e gerenciamento de volume em uma unidade que oferece maior funcionalidade do que a separação tradicional dessas funções permite. Os discos, ou partições de discos, são agrupados por meio de conjuntos RAID em **pools** de armazenamento. Um pool pode conter um ou mais sistemas de arquivos ZFS. O espaço livre do pool inteiro está disponível a todos os sistemas de arquivos dentro desse pool. Como resultado, não existem limites artificiais sobre o uso de armazenamento e não é preciso relocar sistemas de arquivos entre volumes nem redimensionar volumes. O ZFS oferece cotas para limitar o tamanho de um sistema de arquivos e reservas para garantir que um sistema de arquivos possa crescer por uma quantidade especificada, mas essas variáveis podem ser alteradas pelo proprietário do sistema de arquivos a qualquer momento. A [Figura 12.14\(a\)](#) representa os volumes e sistemas de arquivos tradicionais, e a [Figura 12.14\(b\)](#) mostra o modelo ZFS.



(a) Volumes e sistemas de arquivos tradicionais.



(b) ZFS e armazenamento em pool.

FIGURA 12.14 (a) Volumes e sistemas de arquivos tradicionais e (b) um pool e sistemas de arquivos no ZFS.

12.8 Implementação do armazenamento estável

No [Capítulo 6](#), introduzimos o log de escrita antecipada, que exigia a disponibilidade do armazenamento estável. Por definição, as informações residindo no armazenamento estável *nunca* são perdidas. Para implementar esse armazenamento, precisamos replicar as informações necessárias em diversos dispositivos de armazenamento (normalmente, discos) com modos de falha independentes. Precisamos coordenar a escrita das atualizações de um modo que garanta que uma falha durante uma atualização não deixará todas as cópias em estado danificado e que, quando estivermos nos recuperando de uma falha, poderemos forçar todas as cópias para um valor coerente e correto, mesmo que ocorra outra falha durante a recuperação. No restante desta seção, discutiremos como atender as nossas necessidades.

Uma escrita em disco resulta em um destes três resultados:

1. **Término bem-sucedido:** Os dados foram escritos corretamente no disco.
2. **Falha parcial:** Uma falha ocorreu na mídia da transferência, de modo que somente alguns dos setores foram escritos com os novos dados, e o setor escrito durante a falha pode ter sido adulterado.
3. **Falha total:** A falha ocorreu antes de a escrita em disco ser iniciada, de modo que os valores de dados anteriores no disco permanecessem intactos.

Exigimos que, sempre que ocorrer uma falha durante a escrita de um bloco, o sistema a detecte e chame um procedimento de recuperação para restaurar o bloco a um estado coerente. Para isso, o sistema precisa manter dois blocos físicos para cada bloco lógico. Uma operação de saída é executada da seguinte maneira:

1. Escreva as informações no primeiro bloco físico.
2. Quando a primeira escrita terminar com sucesso, escreva a mesma informação no segundo bloco físico.
3. Declare a operação completa somente depois de a segunda escrita ser completada com sucesso.

Durante a recuperação de uma falha, cada par de blocos físicos é examinado. Se ambos forem iguais e não houver erro detectável, nenhuma outra ação será necessária. Se um bloco tiver um erro detectável, substituímos seu conteúdo pelo valor do outro bloco. Se o bloco não tiver erro detectável, mas os blocos diferirem em conteúdo, substituímos o conteúdo do primeiro bloco pelo valor do segundo. Esse procedimento de recuperação garante que uma escrita no armazenamento estável terá sucesso completo ou resultará em nenhuma mudança.

Podemos estender esse procedimento para permitir o uso de um número arbitrariamente grande de cópias de cada bloco de armazenamento estável. Embora ter um grande número de cópias reduza a probabilidade de falha, é bem razoável simular o armazenamento estável com apenas duas cópias. Os dados no armazenamento estável têm garantia de segurança, a menos que uma falha destrua todas as cópias.

Como a espera até que as escritas em disco se completem (E/S síncrona) é longa, muitos arrays de armazenamento acrescentam uma NVRAM como cache. Como a memória não é volátil (normalmente, ela tem alimentação por bateria como reserva para a alimentação da unidade), podemos confiar que ela armazenará os dados em rotina ao seu modo nos discos. Assim, ela é considerada parte do armazenamento estável. As escritas são muito mais rápidas do que em disco, de modo que o desempenho aumenta bastante.

12.9 Estrutura do armazenamento terciário

Você compraria um aparelho de CD ou DVD que tivesse dentro apenas um disco que você não pudesse retirar ou substituir? É claro que não. Você espera usar um aparelho de DVD ou CD com muitos discos relativamente baratos. Em um computador, o uso de muitos cartuchos baratos com uma unidade reduz o custo geral. Baixo custo é a característica definitiva do armazenamento terciário, que discutimos nesta seção.

12.9.1 Dispositivos de armazenamento terciário

Em razão do baixo custo, na prática, o armazenamento terciário é construído com **mídia removível**. Os exemplos mais comuns de mídia removível são disquetes, fitas e CDs e DVDs read-only, write-once e rewritable. Muitos outros tipos de dispositivos de armazenamento terciário também estão disponíveis, incluindo dispositivos removíveis que armazenam dados em memória flash e interagem com o sistema computadorizado por meio de uma interface USB.

12.9.1.1 Discos removíveis

Os discos removíveis são um tipo de armazenamento terciário. Os disquetes são um exemplo de discos magnéticos removíveis. Eles são compostos de um fino disco flexível, coberto com material magnético e envolvido por uma capa plástica protetora. Embora os disquetes comuns possam conter apenas cerca de 1 MB, uma tecnologia semelhante é usada para discos magnéticos removíveis que mantêm mais de 1 GB. Os discos magnéticos removíveis podem ser quase tão rápidos quanto os discos rígidos, embora a superfície de gravação ofereça maior risco de danos ocasionados por arranhões.

Um **disco óptico-magnético** é outro tipo de disco removível. Ele registra dados em uma placa rígida, coberta por material magnético, mas a tecnologia de gravação é bem diferente daquela usada para um disco magnético. A cabeça óptico-magnética voa muito mais longe da superfície do disco do que a cabeça do disco magnético, e o material magnético é revestido por uma grossa camada protetora de plástico ou vidro. Esse arranjo torna o disco muito mais resistente a colisões da cabeça.

A unidade do disco óptico-magnético possui uma bobina que produz um campo magnético; em temperatura ambiente, o campo é muito largo e muito fraco para magnetizar um bit no disco. Para escrever um bit, a cabeça do disco pisca um raio laser na superfície do disco. O laser visa a um pequeno ponto em que um bit deve ser escrito. O laser aquece esse ponto, tornando-o suscetível ao campo magnético. Assim, o campo magnético largo e fraco pode registrar um pequeno bit.

A cabeça óptico-magnética está muito longe do disco para ler os dados, detectando os pequenos campos magnéticos da mesma forma como a cabeça de um disco rígido faz. Em vez disso, a unidade lê um bit usando uma propriedade da luz laser, denominada **efeito de Kerr**. Quando um raio laser bate em um ponto magnético, a polarização do raio laser é girada em sentido horário ou anti-horário, dependendo da orientação do campo magnético. Essa rotação é o que a cabeça detecta para ler um bit.

Outra categoria de disco removível é o **disco óptico**. Esse disco não utiliza o magnetismo de forma alguma, mas usa um material especial que pode ser alterado pela luz do laser para ter pontos relativamente escuros ou claros. Um exemplo de tecnologia de disco óptico é o **disco de mudança de fase**, que é coberto com material que pode ser congelado para um estado cristalino ou amorfo. O estado cristalino é mais transparente e, por isso, um raio laser é mais claro quando passa pelo material e bate na camada reflexiva. A unidade de mudança de fase utiliza a luz do laser em três potências diferentes: baixa potência para ler dados, potência média para apagar o disco fundindo e congelando a mídia de gravação novamente para o estado cristalino, e uma potência alta para fundir a mídia para o estado amorfo, a fim de escrever no disco. Os exemplos mais comuns dessa tecnologia são o CD-RW e o DVD-RW re-graváveis.

Os tipos de discos descritos aqui podem ser usados indefinidamente. Eles são chamados **discos de leitura-escrita**. Ao contrário disso, os **WORM (Write-Once, Read-Many-Times)** podem ser escritos apenas uma vez. Um modo antigo de criar um disco WORM é manufaturar um fino filme de alumínio entre duas placas de vidro ou plástico. Para escrever um bit, a unidade usa um raio laser para queimar um pequeno furo por meio do alumínio. Essa queima não pode ser revertida. Embora seja possível destruir as informações em um disco WORM queimando furos em toda parte, é praticamente impossível alterar os dados no disco, pois os furos só podem ser acrescentados, e o código ECC associado a cada setor pode detectar esses acréscimos. Os discos WORM são considerados duráveis e confiáveis, pois a camada de metal está acomodada com segurança entre as placas protetoras de vidro ou plástico, e os campos magnéticos não podem danificar essa gravação. Uma tecnologia de única escrita mais recente registra em uma tinta à base de um polímero orgânico em vez de uma camada de alumínio: a tinta absorve o raio laser e forma marcas. Essa tecnologia é usada no CD-R e DVD-R regraváveis.

Discos somente de leitura, como CD-ROM e DVD, vêm de fábrica com os dados já gravados. Eles utilizam tecnologia semelhante à dos discos WORM (embora os buracos sejam prensados e não queimados) e são bastante duráveis.

A maioria dos discos removíveis é mais lenta do que seus correspondentes não removíveis. O processo de escrita é mais lento, assim como a rotação e, às vezes, o tempo de busca.

12.9.1.2 Fitas

A fita magnética é outro tipo de mídia removível. Na maioria das vezes, uma fita mantém mais dados do que um cartucho óptico ou de disco magnético. As unidades de fita e as unidades de disco possuem taxas de transferência similares. No entanto, o acesso aleatório à fita é muito mais lento do que uma busca no disco, pois exige uma operação de avanço ou retrocesso que leva dezenas de segundos ou até mesmo minutos.

Embora uma unidade de fita típica seja mais cara do que uma unidade de disco típica, o preço de um cartucho de fita é menor do que o preço da capacidade equivalente dos discos magnéticos. Assim, a fita é um meio econômico para fins que não exigem acesso aleatório rápido. As fitas normalmente são usadas para manter cópias de backup dos dados do disco. Elas também são usadas em grandes centros de supercomputadores, para manter volumes enormes de dados, usados na pesquisa científica e por grandes empresas comerciais.

Grandes instalações de fita utilizam trocadores de fita robóticos, que movem as fitas entre as unidades e locais de armazenamento em uma biblioteca de fita. Esses mecanismos dão ao computador acesso automatizado a uma grande quantidade de cartuchos de fita.

Uma biblioteca de fita robótica pode reduzir o custo geral do armazenamento de dados. Um arquivo residente em disco que não será necessário por um tempo pode ser **arquivado** em fita, onde o custo por gigabyte pode ser menor; se o arquivo for necessário no futuro, o computador pode colocá-lo de volta no armazenamento em disco para uso ativo. Uma biblioteca de fita robótica às vezes é denominada armazenamento **quase on-line**, pois está entre o alto desempenho dos discos magnéticos on-line e o baixo custo das fitas off-line, localizadas nas prateleiras de uma sala de armazenagem.

12.9.1.3 Tecnologia futura

No futuro, outras tecnologias de armazenamento podem se tornar importantes. Às vezes, tecnologias antigas são usadas de novas maneiras, à medida que a economia muda ou as tecnologias evoluem. Por exemplo, os discos em estado sólido, ou **SSDs**, estão tendo maior importância, tornando-se mais comuns. Em termos simples, um SSD é um disco que é usado como se fosse um disco rígido. Dependendo da tecnologia de memória utilizada, ele pode ser volátil ou não volátil. A tecnologia de memória também afeta o desempenho. Os SSDs não voláteis possuem as mesmas características dos discos rígidos tradicionais, mas podem ser mais confiáveis, pois não possuem partes móveis, e mais rápidos porque não possuem tempo de busca ou latência, existentes nos discos físicos. Além disso, eles usam menos energia. Porém, eles são mais caros por megabyte do que os discos rígidos tradicionais, possuem menos capacidade do que os maiores discos rígidos e podem ter tempos de vida mais curtos do que os discos rígidos, de modo que seus usos são limitados. Como um exemplo, os SSDs estão sendo usados como arrays de armazenamento para manter metadados que exigem alto desempenho, como um sistema de arquivos de registro em diário (*journaling*). SSDs também estão sendo acrescentados em notebooks, para torná-los menores, mais rápidos e mais econômicos em termos de energia.

Outra tecnologia de armazenamento promissora, o **armazenamento holográfico**, utiliza a luz do laser para registrar fotografias holográficas em mídia especial. Podemos pensar em um holograma como um array bidimensional de pixels. Cada pixel representa um bit: 0 para preto ou 1 para branco. Uma fotografia nítida pode manter milhões de bits de dados. E todos os pixels em um holograma são transferidos em um flash de luz a laser, de modo que a taxa de dados é muito alta. Com o desenvolvimento contínuo, o armazenamento holográfico pode se tornar comercialmente viável.

Outra tecnologia de armazenamento sob pesquisa ativa é baseada nos **sistemas mecânicos microeletrônicos (Microelectronic Mechanical Systems - MEMS)**. A ideia é aplicar as tecnologias de fabricação que produzem chips eletrônicos a fim de manufaturar pequenas máquinas de armazenamento de dados. Uma proposta requer a fabricação de um array de 10.000 pequenas cabeças de disco, com um centímetro quadrado de material de armazenamento magnético suspenso acima do array. Quando o material de armazenamento é movido acima das cabeças, cada cabeça acessa sua própria trilha linear de dados no material. O material de armazenamento pode ser deslocado ligeiramente de lado para permitir que todas as cabeças acessem sua própria trilha. Embora ainda não se saiba se essa tecnologia pode ter sucesso, ela poderá fornecer uma tecnologia de armazenamento de dados não volátil mais veloz que o disco magnético e mais barata do que a DRAM com semicondutores.

Não importa se a mídia de armazenamento é um disco magnético removível, um DVD ou uma fita magnética, o sistema operacional precisa fornecer várias capacidades para usar a mídia removível

para armazenamento de dados. Essas capacidades são discutidas na [Seção 12.9.2](#).

12.9.2 Suporte do sistema operacional

Duas tarefas importantes de um sistema operacional são gerenciar os dispositivos físicos e apresentar uma abstração de máquina virtual às aplicações. Neste capítulo, vimos que, para os discos rígidos, o sistema operacional provê duas abstrações. Uma é o dispositivo bruto, um array de blocos de dados. A outra é um sistema de arquivos. Para um sistema de arquivos em um disco magnético, o sistema operacional enfileira e escalona as requisições intercaladas vindas de várias aplicações. Agora, veremos como o sistema operacional realiza seu trabalho quando a mídia de armazenamento é removível.

12.9.2.1 Interface da aplicação

A maioria dos sistemas operacionais pode cuidar de discos removíveis quase exatamente como fazem com os discos fixos. Quando um cartucho vazio é inserido em uma unidade (ou montado), ele precisa ser formatado, e depois um sistema de arquivos vazio é gerado no disco. Esse sistema de arquivos é usado como um sistema de arquivos em um disco rígido.

As fitas são tratadas de formas diferentes. O sistema operacional normalmente apresenta uma fita como uma mídia de armazenamento bruta. Uma aplicação não abre um arquivo na fita; ela abre a unidade de fita inteira como um dispositivo bruto. Em geral, a unidade de fita depois será reservada para uso exclusivo dessa aplicação, até a aplicação encerrar ou fechar a unidade de fita. Essa exclusividade faz sentido, pois o acesso aleatório em uma fita pode levar dezenas de segundos ou até mesmo alguns minutos, de modo que intercalar acessos aleatórios às fitas de mais de uma aplicação provavelmente não funcionaria.

Quando a unidade de fita é apresentada como um dispositivo bruto, o sistema operacional não provê serviços do sistema de arquivos. A aplicação precisa decidir como usar o array de blocos. Por exemplo, um programa que realiza o backup de um disco rígido para fita poderia armazenar uma lista dos nomes de arquivos e tamanhos no início da fita e depois copiar os dados dos arquivos para a fita nessa ordem estabelecida.

É fácil ver os problemas que podem surgir com esse modo de usar a fita. Como cada aplicação cria suas próprias regras sobre como organizar uma fita, uma fita cheia de dados pode ser usada somente pelo programa que a criou. Por exemplo, mesmo sabendo que uma fita de backup contém uma lista de nomes de arquivo e tamanhos de arquivo seguidos pelos dados dos arquivos, ainda acharíamos difícil usá-la. Como exatamente são armazenados os nomes de arquivo? Os tamanhos dos arquivos estão em formato binário ou em ASCII? Os arquivos são escritos um por bloco ou todos eles estão concatenados em uma sequência de bits muito longa? Nem sequer sabemos o tamanho do bloco na fita, pois essa variável pode ser escolhida separadamente para cada bloco escrito.

Para uma unidade de disco, as operações básicas são `read()`, `write()` e `seek()`. As unidades de fita possuem um conjunto diferente de operações básicas. Em vez de `seek()`, uma unidade de fita usa a operação `locate()`. A operação `locate()` da fita é mais precisa do que a operação `seek()` do disco, pois posiciona a fita em um bloco lógico específico e não em uma trilha inteira. Localizar o bloco 0 é o mesmo que rebobinar a fita.

Para a maior parte dos tipos de unidades de fita, é possível localizar qualquer bloco que tenha sido escrito em uma fita. Todavia, em uma fita parcialmente preenchida, não é possível localizar no espaço vazio além da área escrita, pois a maioria das unidades de fita não gerencia seu espaço físico da mesma maneira que as unidades de disco. Para uma unidade de disco, os setores possuem um tamanho fixo, e o processo de formatação precisa ser usado para incluir setores vazios em suas posições finais antes de quaisquer dados poderem ser escritos. A maioria das unidades de fita possui um tamanho de bloco variável, e o tamanho de cada bloco é determinado no ato, quando esse bloco é escrito. Se for encontrada uma área da fita com defeito durante a escrita, essa área defeituosa será pulada e o bloco é escrito novamente. Essa operação explica por que não é possível localizar em qualquer espaço além da área escrita - as posições e números dos blocos lógicos ainda não foram determinados.

A maioria das unidades de fita possui uma operação `read_position()`, que retorna o número do bloco lógico onde se encontra atualmente a cabeça de leitura/escrita. Muitas unidades de fita admitem uma operação `space()` para o movimento relativo. Assim, por exemplo, a operação `space(-2)` moveria para trás dois blocos lógicos.

Para a maior parte dos tipos de unidades de fita, a escrita de um bloco tem o efeito colateral de apagar logicamente tudo além da posição da escrita. Na prática, esse efeito colateral significa que a maioria das unidades de fita é um dispositivo apenas de acréscimo, pois a atualização de um bloco no meio da fita também efetivamente apaga tudo além desse bloco. A unidade de fita implementa esse acréscimo colocando uma marca de fim de fita (End-Of-Tape - EOT) após um bloco escrito. A unidade se recusa a localizar além da marca de EOT, mas é possível localizar o EOT e depois começar a escrever. Isso apaga a marca de EOT antiga e coloca uma nova no final dos novos blocos.

A princípio, um sistema de arquivos pode ser implementado em uma fita. Mas muitas estruturas de dados e algoritmos do sistema de arquivos seriam diferentes daqueles usados para discos devido

à propriedade somente de acréscimo da fita.

12.9.2.2 Nomes de arquivos

Outra questão que o sistema operacional precisa tratar é como nomear arquivos na mídia removível. Para um disco fixo, os nomes não são difíceis. Em um PC, o nome do arquivo consiste em uma letra de unidade seguida por um nome de caminho. No UNIX, o nome do arquivo não contém uma letra de unidade, mas a tabela de montagem permite ao sistema operacional descobrir em que unidade o arquivo está localizado. Entretanto, se o disco é removível, conhecer uma unidade que continha o cartucho em algum momento no passado não significa saber como encontrar o arquivo. Se cada cartucho removível do mundo tivesse um número de série diferente, o nome de um arquivo em um dispositivo removível poderia ser iniciado por esse número de série, mas para garantir que dois números de série não sejam iguais seria preciso que cada um tivesse aproximadamente 12 dígitos de extensão. Quem poderia se lembrar dos nomes de seus arquivos se tivesse de decorar um número de série de 12 dígitos para cada um?

O problema se torna ainda mais difícil quando queremos escrever dados em um cartucho removível em um computador e depois usar o cartucho em outro computador. Se as duas máquinas são do mesmo tipo e têm o mesmo tipo de unidade removível, a única dificuldade é conhecer o conteúdo e o layout dos dados no cartucho. Contudo, se as máquinas ou as unidades forem diferentes, muitos problemas adicionais poderão surgir. Mesmo que as unidades sejam compatíveis, diferentes computadores podem armazenar bytes em diferentes ordens e podem usar diferentes codificações para números binários e até mesmo para letras (como ASCII em PCs *versus* EBCDIC em mainframes).

Os sistemas operacionais de hoje deixam o problema do espaço de nomes sem solução para a mídia removível e dependem das aplicações e dos usuários para descobrir como acessar e interpretar os dados. Felizmente, alguns tipos de mídia removível são tão bem padronizados que todos os computadores os utilizam da mesma maneira. Um exemplo é o CD. Os CDs de música utilizam um formato universal, entendido por qualquer unidade de CD. Os CDs de dados estão disponíveis apenas em alguns formatos diferentes, de modo que é comum que uma unidade de CD e o driver de dispositivo do sistema operacional sejam programados para lidar com todos os formatos comuns. Os formatos de DVD também são bem padronizados.

12.9.2.3 Gerenciamento de armazenamento hierárquico

Um **jukebox robótico** permite que o computador mude o cartucho removível em uma unidade de fita ou disco sem auxílio humano. Dois usos importantes para essa tecnologia são para backups e sistemas de armazenamento hierárquico. O uso de um jukebox para backups é simples: quando um cartucho está cheio, o computador instrui o jukebox a passar para o cartucho seguinte. Alguns jukeboxes mantêm dezenas de unidades e milhares de cartuchos, com braços robóticos controlando o movimento das fitas até as unidades.

Um sistema de armazenamento hierárquico estende a hierarquia de armazenamento além da memória principal e do armazenamento secundário (ou seja, disco magnético) para incorporar o armazenamento terciário. O armazenamento terciário é implementado como um jukebox de fitas ou discos removíveis. Esse nível de hierarquia de armazenamento é maior, mais barato e mais lento.

Embora o sistema de memória virtual possa ser estendido diretamente para o armazenamento terciário, essa extensão não costuma ser executada na prática. O motivo é que uma recuperação de um jukebox pode levar dezenas de segundos ou até mesmo minutos, e essa longa espera é intolerável para a paginação por demanda e para outras formas de uso da memória virtual.

O modo normal de incorporar o armazenamento terciário é estender o sistema de arquivos. Arquivos pequenos e usados com frequência permanecem no disco magnético, enquanto arquivos grandes e antigos, que não são usados ativamente, são arquivados no jukebox. Em alguns sistemas de arquivamento, a entrada do diretório para o arquivo continua existindo, mas o conteúdo do arquivo não ocupa mais espaço no armazenamento secundário. Se uma aplicação tenta abrir o arquivo, a chamada de sistema `open()` fica suspensa até o conteúdo do arquivo poder ser trazido do armazenamento terciário. Quando o conteúdo estiver disponível do disco magnético, a operação `open()` retornará o controle à aplicação, que prossegue para usar a cópia dos dados residentes no disco.

Hoje, o **gerenciamento do armazenamento hierárquico (Hierarchical Storage Management - HSM)** é encontrado em instalações que possuem enormes volumes de dados que são usados raramente, esporadicamente ou periodicamente. O trabalho atual em HSM inclui sua extensão para fornecer pleno **gerenciamento do ciclo de vida da informação (Information Life-cycle Management - ILM)**. Aqui, os dados passam do disco para a fita e de volta ao disco, conforme a necessidade, mas são excluídos segundo um horário ou de acordo com uma diretriz. Por exemplo, alguns sites salvam o e-mail por sete anos, mas querem ter certeza de que, ao final dos sete anos, ele estará destruído. Nesse ponto, os dados poderiam estar em disco, fita HSM e fita de backup. O ILM centraliza o conhecimento de onde os dados estão para que as diretrizes possam ser aplicadas em todos esses locais.

12.9.3 Aspectos de desempenho

Assim como qualquer componente do sistema operacional, os três aspectos mais importantes do desempenho do armazenamento terciário são velocidade, confiabilidade e custo.

12.9.3.1 Velocidade

A velocidade do armazenamento terciário possui dois aspectos: largura de banda e latência. Medimos a largura de banda em bytes por segundo. A **largura de banda sustentada** é a taxa de dados média durante uma transferência grande, ou seja, a quantidade de bytes dividida pelo tempo da transferência. A **largura de banda efetiva** calcula a média pelo tempo de E/S inteiro, incluindo o tempo para `seek()` ou `locate()` e qualquer tempo de troca de cartucho em um jukebox. Basicamente, a largura de banda sustentada é a taxa de dados quando o fluxo de dados está fluindo, e a largura de banda efetiva é a taxa de dados geral fornecida pela unidade. A *largura de banda de uma unidade* é entendida como indicando a largura de banda sustentada.

Para discos removíveis, a largura de banda varia de alguns megabytes por segundo, para a mais lenta, até 40 MB por segundo, para a mais rápida. As fitas possuem um intervalo similar de larguras de banda, desde alguns megabytes por segundo até mais de 30 MB por segundo.

O segundo aspecto da velocidade é a **latência de acesso**. Por essa medida de desempenho, os discos são muito mais rápidos do que as fitas. O armazenamento em disco é basicamente bidimensional - todos os bits estão em aberto. Um acesso em disco move o braço para o cilindro selecionado e espera pela latência de rotação, que pode levar menos de 5 milissegundos. Ao contrário, o armazenamento em fita é tridimensional. A qualquer momento, uma pequena parte da fita está acessível à cabeça, enquanto a maioria dos bits está enterrada centenas ou milhares de camadas de voltas de fita no carretel. Um acesso aleatório na fita exige desenrolar o carretel até o bloco selecionado alcançar a cabeça da fita, o que pode levar dezenas ou centenas de segundos. Assim, podemos dizer que o acesso aleatório dentro de um cartucho de fita é mais de mil vezes mais lento do que o acesso aleatório no disco.

Se um jukebox estiver envolvido, a latência de acesso pode ser significativamente mais alta. Para um disco removível ser mudado, a unidade precisa parar de girar, depois o braço robótico precisa trocar os cartuchos de disco e, então, a unidade precisa acelerar o novo cartucho. Essa operação pode levar vários segundos - cerca de cem vezes mais do que o tempo de acesso aleatório dentro de um disco. Assim, a troca de discos em um jukebox ocasiona uma penalidade de desempenho bastante alta.

Para fitas, o tempo do braço robótico é aproximadamente o mesmo para o disco. Contudo, para as fitas serem trocadas, a fita antiga precisa rebobinar antes de ser ejetada, e essa operação pode levar até 4 minutos. Depois que a nova fita estiver carregada na unidade, muitos segundos podem ser necessários para que a unidade seja calibrada até a fita e para preparar para E/S. Embora um jukebox de fita lento possa ter um tempo de troca de fita de 1 ou 2 minutos, esse tempo não é tão maior do que o tempo de acesso aleatório dentro de uma fita.

Assim, para generalizar, dizemos que o acesso aleatório em um jukebox de disco possui uma latência de dezenas de segundos, enquanto o acesso aleatório em um jukebox de fita possui uma latência de centenas de segundos; a troca de fitas é dispendiosa, mas a troca de discos, não. Precisamos ter cuidado para não generalizar demais: alguns jukeboxes de fita caros podem rebobinar, ejeter, carregar uma nova fita e avançar até um item de dados qualquer em menos de 30 segundos.

Se prestarmos atenção somente no desempenho das unidades em um jukebox, a largura de banda e a latência parecem razoáveis. No entanto, se em vez disso concentrarmos nossa atenção nos cartuchos encontraremos um terrível gargalo. Considere primeiro a largura de banda. A razão entre largura de banda e capacidade de armazenamento de uma biblioteca robótica é muito menos favorável do que a de um disco fixo. Para ler todos os dados armazenados em um disco rígido grande, poderíamos levar cerca de uma hora. Para ler todos os dados armazenados em uma grande biblioteca de fita, poderíamos levar anos. A situação com relação à latência de acesso é quase tão ruim quanto isso. Para ilustrar esse fato, se 100 requisições estiverem enfileiradas para uma unidade de disco, o tempo médio de espera será de aproximadamente 1 segundo. Se 100 requisições estiverem enfileiradas para uma biblioteca de fita, o tempo de espera médio poderá ser de mais de 1 hora. Há um baixo custo do armazenamento terciário devido ao compartilhamento de muitos cartuchos baratos de algumas unidades caras. Todavia, uma biblioteca removível é mais bem dedicada ao armazenamento de dados usados com pouca frequência, pois a biblioteca só pode satisfazer a uma quantidade bem pequena de requisições de E/S por hora.

12.9.3.2 Confiabilidade

Embora muitas vezes pensemos que *bom desempenho* significa *boa velocidade*, outro aspecto importante do desempenho é a *confiabilidade*. Se tentarmos ler alguns dados e não pudermos fazer isso por causa de uma falha na unidade ou na mídia, para todos os fins práticos, o tempo de acesso é infinitamente longo e a largura de banda é infinitamente pequena. Assim, é importante entender a confiabilidade da mídia removível.

Os discos magnéticos removíveis são um pouco menos confiáveis do que os discos rígidos fixos, pois eles provavelmente estão mais expostos a condições ambientais prejudiciais, como poeira, grandes mudanças de temperatura e umidade e forças mecânicas como choque e inclinação. Os discos ópticos são considerados bastante confiáveis, pois a camada que armazena os bits é protegida por uma camada de plástico ou vidro transparente. A confiabilidade da fita magnética varia muito, dependendo do tipo de unidade. Algumas unidades mais baratas desgastam as fitas após algumas dezenas de usos; outras unidades são delicadas o bastante para permitir milhões de reutilizações. Por comparação com a cabeça de um disco magnético, a cabeça em uma unidade de fita magnética é um ponto fraco. Uma cabeça de disco voa acima da mídia, mas uma cabeça de fita está em contato com a fita. A ação de esfregar sobre a fita pode desgastar a cabeça após alguns milhares ou algumas dezenas de milhares de horas.

Em resumo, podemos dizer que uma unidade de disco fixo provavelmente será mais confiável do que a unidade de disco removível ou unidade de fita, e um disco óptico será mais confiável do que um disco magnético ou fita. Entretanto, um disco magnético fixo possui um ponto fraco. Uma colisão da cabeça em um disco rígido em geral destrói os dados, enquanto uma falha de uma unidade de fita ou unidade de disco óptico deixa o cartucho de dados intacto.

12.9.3.3 Custo

O custo de armazenamento é outro fator importante. Aqui está um exemplo concreto de como a mídia removível pode reduzir o custo de armazenamento geral. Suponha que um disco rígido que mantém X GB tenha um preço de US\$200; desse valor, US\$190 são para a montagem, o motor e o controlador, e US\$10 são para as placas magnéticas. O custo do armazenamento para esse disco é de US\$200/ X por gigabyte. Agora, suponha que possamos manufaturar as placas em um cartucho removível. Para uma unidade e 10 cartuchos, o preço total é de US\$190 + US\$100 e a capacidade é 10X GB, de modo que o custo do armazenamento é de US\$29/ X por gigabyte. Mesmo que fosse um pouco mais caro criar um cartucho removível, o custo por gigabyte do armazenamento removível pode muito bem ser menor do que o custo por gigabyte de um disco rígido, pois o custo de um disco será equilibrado com o preço baixo de muitos cartuchos removíveis.

As Figuras 12.15, 12.16 e 12.17 mostram as tendências de custo por megabyte para a memória DRAM, discos rígidos magnéticos e unidades de fita. Os preços nos gráficos são os menores encontrados em várias revistas de informática e na World Wide Web no final de cada ano. Esses preços refletem o mercado de pequenos computadores do leitor dessas revistas, onde os preços são baixos por comparação com os mercados de computador de grande porte e minicomputador. No caso da fita, o preço é para uma unidade com uma fita. O custo geral do armazenamento de fita torna-se muito menor à medida que mais fitas são adquiridas para uso com a unidade, pois o preço de uma fita é uma pequena fração do preço da unidade. Contudo, em uma imensa biblioteca de fitas, contendo milhares de cartuchos, o custo do armazenamento é dominado pelo custo dos cartuchos de fita. Em 2004, o custo por GB dos cartuchos de fita estava em torno de US\$0,40.

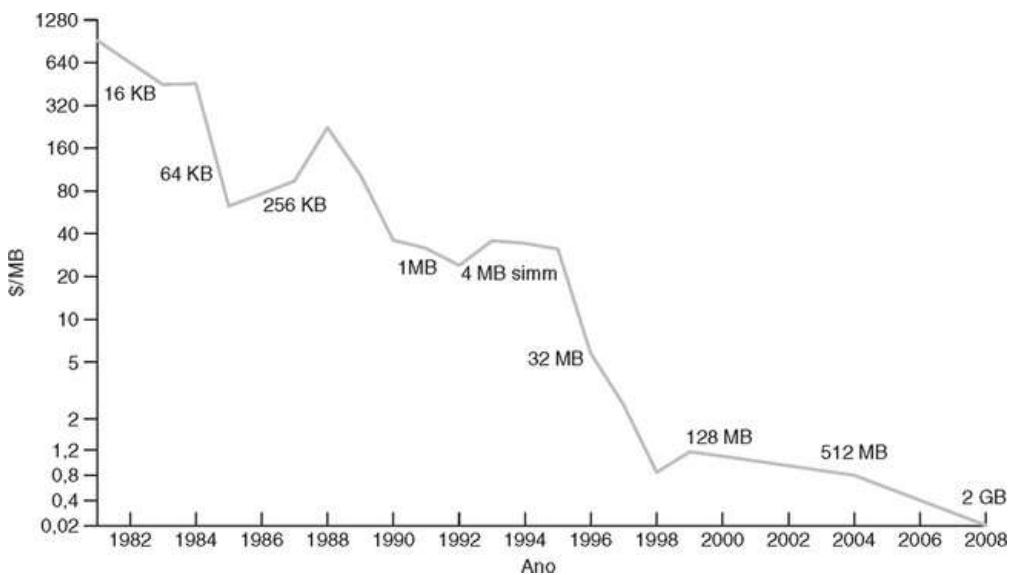


FIGURA 12.15 Preço por megabyte da DRAM, de 1981 a 2008.

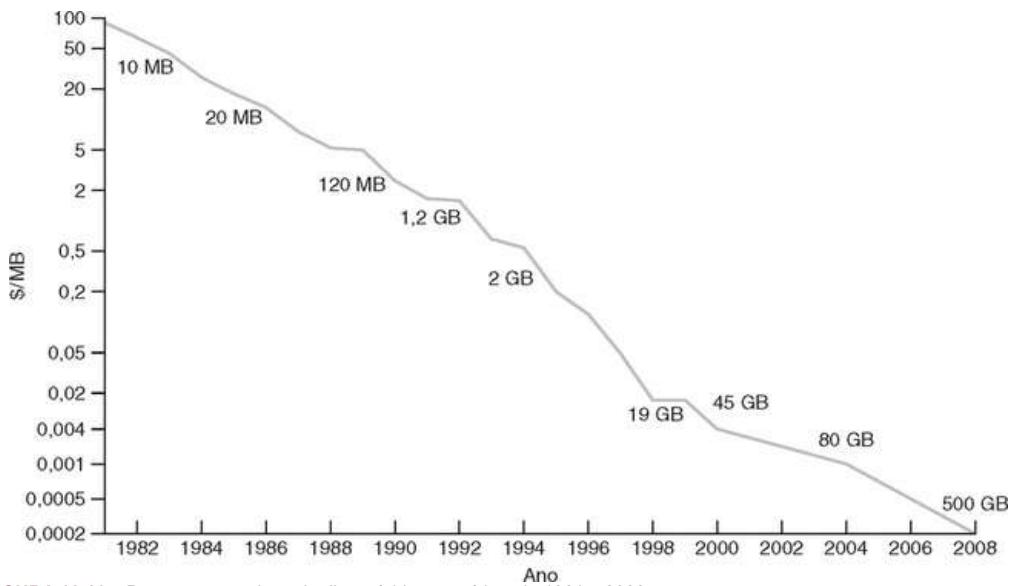


FIGURA 12.16 Preço por megabyte do disco rígido magnético, de 1981 a 2008.

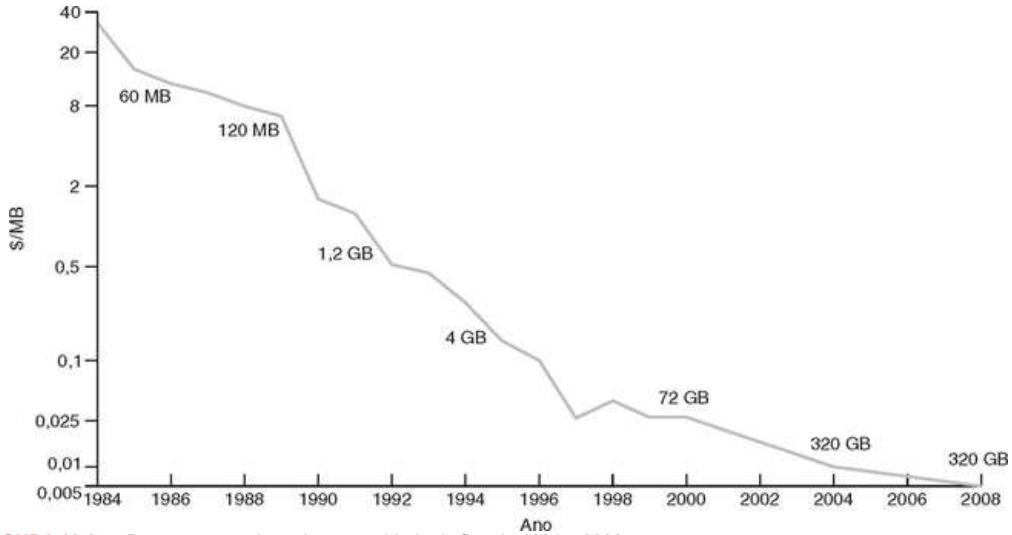


FIGURA 12.17 Preço por megabyte de uma unidade de fita, de 1984 a 2008.

Como podemos ver na [Figura 12.15](#), o custo da DRAM flutua bastante. No período de 1981 a 2004, podemos ver três quedas de preço (por volta de 1981, 1989 e 1996), quando a produção em excesso causou uma saturação no mercado. Também podemos ver dois períodos (por volta de 1987 e 1993), em que a escassez no mercado causou significativos aumentos de preço. No caso dos discos rígidos ([Figura 12.16](#)), as quedas de preço foram muito mais uniformes. Os preços de unidade de fita também caíram uniformemente até 1997 ([Figura 12.17](#)). Desde 1997, o preço por gigabyte das unidades de fita mais baratas deixou de ter sua queda drástica, embora a tecnologia de fita de nível médio (como DAT/DDS) tenha continuado a cair de preço e agora esteja se aproximando ao das unidades mais baratas. Os preços de unidade de fita não aparecem antes de 1984, pois, como mencionado, as revistas usadas na busca de preço visam ao mercado de computadores pequenos, e as unidades de fita não eram muito usadas com computadores pequenos antes de 1984.

Podemos ver por esses gráficos que o custo do armazenamento caiu bastante nos últimos vinte anos ou mais. Comparando esses gráficos, podemos ver que o preço do armazenamento em disco despencou em relação ao preço da DRAM e da fita.

O preço por megabyte de disco magnético melhorou em mais de quatro ordens de grandeza de 1981 a 2004, enquanto a melhoria correspondente para a memória principal foi de apenas três ordens de grandeza. A memória principal hoje é mais cara do que o armazenamento em disco por um fator de 100.

O preço por megabyte caiu muito mais rapidamente para unidades de disco do que para unidades de fita. Na verdade, o preço por megabyte de unidades de disco magnético está se aproximando ao de um cartucho de fita sem a unidade de fita. Como consequência, as bibliotecas de fitas de tamanhos pequeno e médio possuem um custo de armazenamento maior do que os sistemas de disco

com capacidade equivalente.

A queda drástica nos preços do disco tornou o armazenamento terciário em grande parte obsoleto: não temos mais tecnologia de armazenamento terciário, que é várias ordens de grandeza mais barata do que o disco magnético. Parece que o ressurgimento do armazenamento terciário terá de esperar uma descoberta tecnológica revolucionária. Enquanto isso, o armazenamento em fita será limitado principalmente a fins como backups de unidades de disco e arquivamento em enormes bibliotecas de fita, que ultrapassam muito a capacidade de armazenamento prática dos grandes conjuntos de disco.

12.10 Resumo

As unidades de disco são o principal dispositivo de E/S para armazenamento secundário na maioria dos computadores. A maioria dos dispositivos de armazenamento secundário é composta de discos magnéticos ou fitas magnéticas. Uma unidade de disco moderna é estruturada como um grande array unidimensional de blocos de disco lógicos. Geralmente, esses blocos lógicos possuem um tamanho de 512 bytes cada. Os discos podem ser conectados a um sistema computadorizado de duas maneiras: (1) usando as portas de E/S locais no computador host ou (2) por meio de uma conexão de rede.

As requisições para E/S de disco são geradas pelo sistema de arquivos e pelo sistema de memória virtual. Cada requisição especifica o endereço no disco a ser referenciado, na forma de um número de bloco lógico. Os algoritmos de escalonamento de disco podem melhorar a largura de banda efetiva, o tempo de resposta médio e a variância no tempo de resposta. Algoritmos como SSTF, SCAN, C-SCAN, LOOK e C-LOOK são projetados para fazer melhorias desse tipo por meio de estratégias para ordenação da fila do disco.

O desempenho pode ser prejudicado pela fragmentação externa. Alguns sistemas possuem utilitários que varrem o sistema de arquivos para identificar arquivos fragmentados; eles movimentam os blocos para diminuir a fragmentação. A desfragmentação de um sistema de arquivos pouco fragmentado pode melhorar muito o desempenho, mas o sistema pode ter um desempenho reduzido enquanto o processo está em andamento. Os sistemas de arquivos sofisticados, como o Fast File System do UNIX, incorporam muitas estratégias para controlar a fragmentação durante a alocação de espaço, de modo que a reorganização do disco não é necessária.

O sistema operacional gerencia os blocos do disco. Primeiro, um disco precisa ser formatado em baixo nível para criar os setores sobre o hardware bruto - discos novos normalmente já vêm pré-formatados. Depois, o disco é particionado e os sistemas de arquivos são criados, e os blocos de boot são alocados para armazenar o programa de boot do sistema. Finalmente, quando um bloco está adulterado, o sistema precisa ter um meio de separar esse bloco ou substituí-lo logicamente por um reserva.

Como um swap space eficiente é uma chave para o bom desempenho, os sistemas evitam o sistema de arquivos e usam o acesso bruto ao disco para a E/S de paginação. Alguns sistemas dedicam uma partição de disco raw para o swap space, e outros utilizam um arquivo dentro do sistema de arquivos em vez disso. Outros sistemas ainda permitem que o usuário ou o administrador do sistema tome a decisão, fornecendo as duas opções.

Devido à quantidade de armazenamento necessária em grandes sistemas, os discos constantemente são usados de formas redundantes por meio de algoritmos RAID. Esses algoritmos permitem que mais de um disco seja usado para determinada operação e permitem a operação continuada e até mesmo a recuperação automática no caso de uma falha no disco. Os algoritmos RAID são organizados em diferentes níveis, sendo que cada nível provê alguma combinação de confiabilidade e altas taxas de transferência.

O esquema de log por escrita antecipada exige a disponibilidade do armazenamento estável. Para implementar esse armazenamento, precisamos replicar as informações necessárias em diversos dispositivos de armazenamento não volátil (normalmente, discos) com modos de falha independentes. Também precisamos atualizar as informações de uma maneira controlada, para garantir que possamos recuperar os dados estáveis após qualquer falha durante a transferência ou recuperação de dados.

O armazenamento terciário é montado de unidades de disco e fita que utilizam mídia removível. Muitas tecnologias diferentes estão à disposição, incluindo fita magnética, discos magnéticos e óptico-magnéticos removíveis e discos ópticos.

Para os discos removíveis, o sistema operacional em geral provê os serviços completos de uma interface do sistema de arquivos, incluindo o gerenciamento de espaço e o escalonamento da fila de requisição. Para muitos sistemas operacionais, o nome de um arquivo em um cartucho removível é uma combinação de um nome de unidade e um nome de arquivo dentro dessa unidade. Essa convenção é mais simples, mas potencialmente mais confusa do que o uso de um nome que identifica um cartucho específico.

Para as fitas, o sistema operacional só provê uma interface bruta. Muitos sistemas operacionais não possuem suporte embutido para jukebox. O suporte para jukebox pode ser fornecido por um driver de dispositivo ou por uma aplicação privilegiada, projetada para backups ou para HSM.

Três aspectos importantes do desempenho são largura de banda, latência e confiabilidade. Muita largura de banda está disponível para discos e fitas, mas a latência do acesso aleatório para uma fita é muito mais lenta do que para um disco. A troca de cartuchos em um jukebox também é relativamente lenta. Como um jukebox possui uma razão baixa entre unidades e cartuchos, a leitura de uma grande fração dos dados em um jukebox pode levar muito tempo. A mídia óptica, que protege a camada sensível com um revestimento transparente, geralmente é mais robusta do que a mídia magnética, que mais provavelmente expõe o material magnético a uma maior possibilidade de danos físicos. Por fim, o custo do armazenamento diminuiu bastante nas duas últimas décadas,

principalmente para o armazenamento em disco.

Exercícios práticos

- 12.1. O escalonamento de disco, fora o escalonamento FCFS, é útil para um ambiente de único usuário? Explique sua resposta.
- 12.2. Explique por que o escalonamento SSTF tende a favorecer os cilindros do meio em detrimento dos cilindros mais internos e mais externos.
- 12.3. Por que a latência de rotação normalmente não é considerada no escalonamento de disco? Como você modificaria SSTF, SCAN e C-SCAN para incluir a otimização da latência?
- 12.4. Como o uso de um disco de RAM afetaria sua seleção de um algoritmo de escalonamento de disco? Que fatores você teria que considerar? As mesmas considerações também se aplicam ao escalonamento de disco rígido, dado que o sistema de arquivos armazena os blocos usados recentemente em um cache de buffer na memória principal?
- 12.5. Por que é importante balancear a E/S do sistema de arquivos entre os discos e controladores em um sistema de um ambiente multitarefa?
- 12.6. Quais são as escolhas envolvidas na releitura de páginas de código do sistema de arquivos *versus* usar o espaço de swap para armazená-las?
- 12.7. Existe alguma maneira de implementar o armazenamento verdadeiramente estável? Explique sua resposta.
- 12.8. O termo "Fast Wide SCSI-II" indica um barramento SCSI que opera a uma taxa de dados de 20 megabytes por segundo quando move um pacote de bytes entre um host e um dispositivo. Suponha que uma unidade de disco Fast Wide SCSI-II gire a 7.200 RPM, tenha um tamanho de setor de 512 bytes e mantenha 160 setores por trilha.
- a. Estime a taxa de transferência sustentada dessa unidade em megabytes por segundo.
 - b. Suponha que a unidade tenha 7.000 cilindros, 20 trilhas por cilindro, um tempo de troca de cabeça (de um prato para outro) de 0,5 milissegundo e um tempo de busca no cilindro adjacente de 2 milissegundos. Use essa informação adicional para dar uma estimativa precisa da taxa de transferência sustentada para uma transferência em grande quantidade.
 - c. Suponha que o tempo de busca médio para a unidade seja de 8 milissegundos. Estime as operações de E/S por segundo e a taxa de transferência efetiva para uma carga de trabalho de acesso aleatório que leia setores individuais espalhados pelo disco.
 - d. Calcule as operações de E/S de acesso aleatório por segundo e a taxa de transferência para tamanhos de E/S de 4 kilobytes, 8 kilobytes e 64 kilobytes.
 - e. Se várias requisições estiverem na fila, um algoritmo de escalonamento como SCAN deverá ser capaz de reduzir a distância média de busca. Suponha que uma carga de trabalho de acesso aleatório esteja lendo páginas de 8 kilobytes, o tamanho médio da fila seja 10 e o algoritmo de escalonamento reduza o tempo médio de busca para 3 milissegundos. Agora, calcule as operações de E/S por segundo e a taxa de transferência efetiva da unidade.
- 12.9. Mais de uma unidade de disco pode estar conectada a um barramento SCSI. Em particular, um barramento Fast Wide SCSI-II (ver [Exercício 12.8](#)) pode estar conectado a, no máximo, 15 unidades de disco. Lembre-se de que esse barramento tem uma largura de banda de 20 megabytes por segundo. A qualquer momento, apenas um pacote pode ser transferido no barramento entre o cache interno de algum disco e o host. Porém, um disco pode estar movendo seu braço enquanto algum outro disco está transferindo um pacote no barramento. Além disso, um disco pode estar transferindo dados entre seus pratos magnéticos e seu cache interno enquanto algum outro disco está transferindo um pacote no barramento. Considerando as taxas de transferência que você calculou para as diversas cargas de trabalho no [Exercício 12.8](#), discuta quantos discos podem ser usados, de modo eficaz, por um barramento Fast Wide SCSI-II.
- 12.10. O remapeamento de blocos defeituosos por reserva de setor ou por deslizamento de setor pode influenciar o desempenho. Suponha que a unidade no [Exercício 12.8](#) tenha um total de 100 setores defeituosos em locais aleatórios e que cada setor defeituoso seja mapeado a um spare localizado em uma trilha diferente dentro do mesmo cilindro. Estime o número de operações de E/S por segundo e a taxa de transferência efetiva para uma carga de trabalho de acesso aleatório consistindo em leituras de 8 kilobytes, considerando um tamanho de fila de 1 (ou seja, a escolha do algoritmo de escalonamento não é um fator). Qual é o efeito de um setor defeituoso sobre o desempenho?
- 12.11. Em um jukebox de disco, qual seria o efeito de ter mais arquivos abertos do que o número de unidades no jukebox?
- 12.12. Se os discos rígidos magnéticos passarem a ter o mesmo custo por gigabyte das fitas, as fitas se tornarão obsoletas ou elas ainda serão necessárias? Explique sua resposta.
- 12.13. Às vezes se diz que a fita é um meio de acesso sequencial, enquanto um disco magnético é um meio de acesso aleatório. Na verdade, a adequação de um dispositivo de armazenamento para o acesso aleatório depende do tamanho da transferência. O termo *taxa de transferência streaming* indica a taxa para a transferência de dados que está em andamento, excluindo o efeito da latência de acesso. Ao contrário, a *taxa de transferência efetiva* é a razão do total de bytes pelo total de segundos, incluindo o tempo de overhead, como a latência de acesso. Suponha que, em um computador, o cache de nível 2 tenha uma latência de acesso de 8 nanossegundos e uma taxa de transferência streaming de 800 megabytes por segundo, a memória principal tenha uma latência de acesso de 60 nanossegundos e uma taxa de transferência streaming de 80 megabytes por segundo, o disco magnético tenha uma latência de acesso de 15 milissegundos e uma taxa de transferência streaming de 5 megabytes por segundo, e uma unidade de fita tenha uma latência de acesso de 60 segundos e uma taxa de transferência streaming de 2 megabytes por segundo.
- a. O acesso aleatório faz a taxa de transferência efetiva de um dispositivo diminuir, pois nenhum dado é transferido durante o tempo de acesso. Para o disco descrito, qual é a taxa de transferência efetiva se um acesso médio for acompanhado de uma taxa de

- transferência de (1) 512 bytes; (2) 8 kilobytes; (3) 1 megabyte; e (4) 16 megabytes?
- b. A utilização de um dispositivo é a razão entre a taxa de transferência efetiva e a taxa de transferência streaming. Calcule a utilização da unidade de disco para cada um dos quatro tamanhos de transferência dados no item a.
 - c. Suponha que uma utilização de 25% (ou maior) seja considerada aceitável. Usando os valores de desempenho dados, calcule o menor tamanho de transferência para o disco que ocasiona uma utilização aceitável.
 - d. Complete a seguinte sentença: Um disco é um dispositivo de acesso aleatório para transferências maiores do que _____ bytes e é um dispositivo de acesso sequencial para transferências menores.
 - e. Calcule os tamanhos de transferência mínimos que ocasionam a utilização aceitável para cache, memória e fita.
 - f. Quando uma fita é um dispositivo de acesso aleatório e quando ela é um dispositivo de acesso sequencial?

12.14. Suponha que concordemos que 1 kilobyte sejam 1.024 bytes, 1 megabyte sejam 1.024² bytes e 1 gigabyte sejam 1.024³ bytes. Essa progressão continua por terabytes, petabytes e exabytes (1.024⁶). Diversos projetos científicos propostos planejam registrar e armazenar alguns exabytes de dados durante a próxima década. Para responder às perguntas a seguir, você precisará fazer algumas suposições razoáveis; escreva as suposições que você precisa fazer.

- a. Quantos discos seriam necessários para manter 4 exabytes de dados?
- b. Quantas fitas magnéticas seriam necessárias para manter 4 exabytes de dados?
- c. Quantas fitas ópticas seriam necessárias para manter 4 exabytes de dados (ver [Exercício 12.34](#))?
- d. Quantos cartuchos de armazenamento holográfico seriam necessários para manter 4 exabytes de dados (ver [Exercício 12.33](#))?
- e. Quantos centímetros cúbicos de espaço de armazenamento seriam exigidos por cada uma dessas opções?

Exercícios

12.15. Nenhuma das disciplinas de escalonamento de disco, exceto FCFS, é verdadeiramente *justa* (pode ocorrer starvation).

- Explique por que essa afirmação é verdadeira.
- Descreva um meio de modificar algoritmos como SCAN para garantir a imparcialidade.
- Explique por que a imparcialidade é um objetivo importante em um sistema de tempo compartilhado.
- Indique três ou mais exemplos de circunstâncias em que é importante que o sistema operacional seja *injusto* no atendimento das requisições de E/S.

12.16. Suponha que determinada unidade de disco tenha 5.000 cilindros, numerados de 0 a 4999.

A unidade atualmente está atendendo a uma requisição no cilindro 143, e a requisição anterior foi no cilindro 125. A fila de requisições pendentes, na ordem FIFO, é
86, 1470, 913, 1774, 948, 1509, 1022, 1750, 130

Começando na posição atual da cabeça, qual é a distância total (em cilindros) que o braço do disco se move para satisfazer a todas as requisições pendentes para cada um dos seguintes algoritmos de escalonamento de disco?

- FCFS
- SSTF
- SCAN
- LOOK
- C-SCAN
- C-LOOK

12.17. A física elementar afirma que, quando um objeto está sujeito a uma aceleração constante a , a relação entre a distância d e o tempo t é dada por

$$d = \frac{1}{2}at^2.$$

Suponha que, durante uma busca, o disco no [Exercício 12.16](#) acelere o braço do disco em uma razão constante para a primeira metade da busca e depois desacelere o braço do disco na mesma razão para a segunda metade da busca. Suponha que o disco possa realizar uma busca para um cilindro adjacente em 1 milissegundo, e uma busca completa por todos os 5.000 cilindros em 18 milissegundos.

- A distância de uma busca é o número de cilindros que a cabeça move. Explique por que o tempo de busca é proporcional à raiz quadrada da distância da busca.
- Escreva uma equação para o tempo de busca como uma função da distância da busca.
Essa equação deverá estar na forma

$$t = c + y\sqrt{2}$$

onde t é o tempo em milissegundos e L é a distância de busca em cilindros.

- Calcule o tempo de busca total para cada um dos escalonamentos do [Exercício 12.16](#). Determine qual escalonamento é o mais rápido (tem o menor tempo de busca total).
- O *percentual de aumento de velocidade* é o tempo economizado dividido pelo tempo original. Qual é o percentual de aumento de velocidade do escalonamento mais rápido em relação ao FCFS?

12.18. Suponha que o disco no [Exercício 12.17](#) gire a 7.200 RPM.

- Qual é a latência de rotação média dessa unidade de disco?
- Que distância de busca pode ser coberta no tempo encontrado para o item a?

12.19. A busca com aceleração do Exemplo 12.17 é típica para unidades de disco rígido. Ao contrário, disquetes (e muitos discos rígidos fabricados antes de meados da década de 1980) normalmente realizam buscas a uma velocidade fixa. Suponha que o disco no [Exercício 12.17](#) tenha uma busca com velocidade constante, em vez de uma busca com aceleração constante, de modo que o tempo de busca tem a forma

$$t = x + yL$$

onde t é o tempo em milissegundos e L é a distância de busca. Suponha que o tempo para a busca em um cilindro adjacente seja de 1 milissegundo, como antes, e que o tempo para a busca em cada cilindro adicional seja de 0,5 milissegundo.

- a. Escreva uma equação para esse tempo de busca em função da distância de busca.
 - b. Usando a função do tempo de busca, calcule o tempo de busca total para cada um dos escalonamentos do [Exercício 12.16](#). Sua resposta é a mesma que aquela para o [Exercício 12.17\(c\)](#)?
 - c. Qual é o percentual de aumento de velocidade do escalonamento mais rápido em relação ao FCFS neste caso?
- 12.20. Compare o desempenho do escalonamento C-SCAN e SCAN, assumindo uma distribuição uniforme de requisições. Considere o tempo de resposta médio (o tempo entre a chegada de uma requisição e o término do atendimento dessa requisição), a variação no tempo de resposta e a largura de banda efetiva. Como o desempenho depende dos tamanhos relativos do tempo de busca e latência de rotação?
- 12.21. As requisições não costumam ser distribuídas uniformemente. Por exemplo, podemos esperar que um cilindro contendo a FAT ou os inodes do sistema de arquivos seja acessado com mais frequência do que um cilindro que contém apenas arquivos. Suponha que você saiba que 50% das requisições sejam para um número pequeno e fixo de cilindros.
- a. Algum dos algoritmos de escalonamento discutidos neste capítulo seria particularmente bom para este caso? Explique sua resposta.
 - b. Proponha um algoritmo de escalonamento de disco que ofereça desempenho ainda melhor, tirando proveito desse “ponto quente” no disco.
 - c. Os sistemas de arquivos normalmente encontram blocos de dados por meio de uma tabela de indireção, como FAT no DOS e inodes no UNIX. Descreva uma ou mais maneiras de tirar proveito dessa indireção para melhorar o desempenho do disco.
- 12.22. Uma organização RAID nível 1 poderia alcançar melhor desempenho para requisições de leitura do que uma organização RAID nível 0 (com espalhamento de dados não redundante)? Se puder, como?
- 12.23. Considere uma organização RAID nível 5 compreendendo cinco discos, com a paridade para conjuntos de quatro blocos em quatro discos armazenada no quinto disco. Quantos blocos são acessados para realizar o seguinte:
- a. Uma escrita de um bloco de dados
 - b. Uma escrita de sete blocos de dados consecutivos
- 12.24. Compare o throughput alcançado por uma organização RAID nível 5 com aquele alcançado por uma organização RAID nível 1 para o seguinte:
- a. Operações de leitura em blocos isolados
 - b. Operações de leitura em múltiplos blocos contíguos
- 12.25. Compare o desempenho das operações de escrita alcançado por uma organização RAID nível 5 com aquele alcançado por uma organização RAID nível 1.
- 12.26. Suponha que você tenha uma configuração misturada compreendendo discos organizados como RAID nível 1 e como RAID nível 5. Suponha que o sistema tenha flexibilidade para decidir qual organização de disco usar para armazenar determinado arquivo. Que arquivos deverão ser armazenados nos discos RAID nível 1 e quais nos discos RAID nível 5 a fim de otimizar o desempenho?
- 12.27. A confiabilidade de uma unidade de disco rígido normalmente é descrita em termos de uma quantidade chamada *tempo médio entre falhas* (*mean time between failures - MTBF*). Embora essa quantidade seja chamada de “tempo”, o MTBF na realidade é medido em horas de unidade por falha.
- a. Se um sistema contém 1.000 unidades de disco, cada uma das quais com um MTBF de 750.000 horas, qual dos seguintes descreve a frequência com que uma falha de unidade ocorrerá nesse conjunto de discos: uma vez por milênio, uma vez por século, uma vez por década, uma vez por ano, uma vez por mês, uma vez por semana, uma vez por dia, uma vez por hora, uma vez por minuto ou uma vez por segundo?
 - b. As estatísticas de mortalidade indicam que, na média, um morador dos Estados Unidos tem chance de cerca de 1:1.000 de morrer entre 20 e 21 anos. Deduza as horas de MTBF para 20 anos de idade. Converta esse valor de horas para anos. O que esse MTBF lhe diz a respeito do tempo de vida esperado de alguém com 20 anos?
 - c. O fabricante garante um MTBF de 1 milhão de horas para determinado modelo de unidade de disco. O que você pode concluir sobre o número de anos pelos quais uma dessas unidades está sob garantia?
- 12.28. Discuta as vantagens e desvantagens relativas da reserva de setor e do deslizamento de setor.
- 12.29. Discuta os motivos pelos quais o sistema operacional poderia exigir informações precisas sobre como os blocos são armazenados em um disco. Como o sistema operacional poderia

melhorar o desempenho do sistema de arquivos com esse conhecimento?

12.30. O sistema operacional em geral trata os discos removíveis como sistemas de arquivos compartilhados, mas atribui uma unidade de fita apenas a uma aplicação de cada vez. Indique três motivos que poderiam explicar essa diferença no tratamento de discos e fitas. Descreva os recursos adicionais que um sistema operacional precisaria para dar suporte ao acesso compartilhado do sistema de arquivos a um jukebox de fita. As aplicações compartilhando o jukebox de fita precisam de quaisquer propriedades especiais ou elas poderiam usar os arquivos como se estivessem residentes em disco? Explique sua resposta.

12.31. Quais seriam os efeitos sobre o custo e o desempenho se o armazenamento em fita tivesse a mesma densidade de área do armazenamento em disco? (*Densidade de área* é o número de gigabits por polegada quadrada.)

12.32. Você pode usar estimativas simples para comparar o custo e o desempenho de um sistema de armazenamento de terabytes feito inteiramente de discos com um que incorpora o armazenamento terciário. Suponha que os discos magnéticos mantenham 10 GB cada um, custem US\$1.000, transfiram 5 MB por segundo e tenham uma latência de acesso média de 15 milissegundos. Suponha também que uma biblioteca de fita custe US\$10 por gigabyte, transfira 10 MB por segundo e tenha uma latência de acesso média de 20 segundos. Calcule o custo total, a taxa de dados máxima total e o tempo de espera médio para um sistema de disco puro. Se você fizer quaisquer suposições sobre a carga de trabalho, descreva-as e justifique-as. Agora, suponha que 5% dos dados sejam usados frequentemente, de modo que tenham de residir no disco, mas os outros 95% sejam arquivados na biblioteca de fita. Suponha, ainda, que o sistema de disco trate de 95% das requisições e que a biblioteca trate dos outros 5%. Quais são o custo total, a taxa de dados total máxima e o tempo de espera médio para esse sistema de armazenamento hierárquico?

12.33. Imagine que uma unidade de armazenamento holográfico tenha sido inventada. Suponha que a unidade holográfica custe US\$10.000 e tenha um tempo de acesso médio de 40 milissegundos. Suponha que ela utilize um cartucho de US\$100 do tamanho de um CD. Esse cartucho mantém 40.000 imagens e cada imagem é uma figura em preto e branco quadrada com resolução de 6.000×6.000 pixels (cada pixel armazena 1 bit). Suponha que a unidade possa ler ou escrever uma figura em 1 milissegundo. Responda às perguntas a seguir.

- Cite alguns bons usos para esse dispositivo.
- Como esse dispositivo afetaria o desempenho de E/S de um sistema de computação?
- Que outros tipos de dispositivos de armazenamento, se houver algum, se tornariam obsoletos como resultado da invenção desse dispositivo?

12.34. Suponha que um cartucho de disco óptico de 5,25 polegadas e um lado tenha uma densidade de superfície de 1 gigabit por polegada quadrada. Suponha que uma fita magnética tenha uma densidade de superfície de 20 megabits por polegada quadrada e tenha 1/2 polegada de largura por 600 metros de extensão. Calcule uma estimativa das capacidades de armazenamento desses dois tipos de cartucho. Suponha que exista uma fita óptica que tenha o mesmo tamanho físico da fita, mas a mesma densidade de armazenamento do disco óptico. Que volume de dados poderia ser mantido pela fita óptica? Qual seria um preço de mercado razoável para a fita óptica se a fita magnética custa US\$25?

12.35. Discuta como um sistema operacional poderia manter uma lista de espaço livre para um sistema de arquivos residente em fita. Suponha que a tecnologia de fita seja apenas de acréscimo e que ela utilize a marca de EOT e os comandos `locate`, `space` e `read position` conforme descrevemos na [Seção 12.9.2.1](#).

Problemas de programação

12.36. Escreva um programa em Java que simule os algoritmos de escalonamento de disco discutidos na [Seção 12.4](#). Particularmente, crie classes separadas que implementem os seguintes algoritmos de escalonamento:

- a. FCFS
- b. SSTF
- c. SCAN
- d. C-SCAN
- e. LOOK

Cada algoritmo implementará a seguinte interface:

```
public interface DiskScheduler
{
    // atende as requisições
    // retorna a quantidade de movimento da
    // cabeça para o algoritmo em particular
    public int serviceRequests( );
}
```

O método `serviceRequests()` retornará a quantidade de movimento de cabeça exigida pelo algoritmo de escalonamento de disco.

Strings de referência consistindo na requisição de cilindros de disco serão fornecidos pela classe `Generator` que está disponível on-line (www.os-book.com). A classe `Generator` produz requisições aleatórias para os cilindros numerados entre 0 e 99. A API para a classe `Generator` se parece com:

```
// produz uma lista de tamanho default de
// requisições de cilindro
public Generator( )
// produz uma lista de requisições de cilindro
// de tamanho count
public Generator(int count)
// retorna a lista de requisições de cilindro
public int[ ] get cylinders( )
```

Cada algoritmo implementando a interface `DiskScheduler` precisa fornecer um construtor que recebe (1) um array inteiro de requisições de cilindro e (2) a posição do cilindro inicial da cabeça de disco. Supondo que a classe `FCFS` implemente `DiskScheduler` de acordo com a política FCFS, um exemplo ilustrando seu uso aparece a seguir:

```
Generator ref = new Generator(1000);

int[ ] referenceString = ref.get cylinders( );
DiskScheduler fcfs = new FCFS(referenceString, 13);
System.out.println("FCFS = " + fcfs.
serviceRequests( ));
```

Este exemplo constrói 1.000 requisições de cilindro aleatórias e inicia o algoritmo FCFS no cilindro 13.

Quando você tiver acabado, compare as quantidades de movimento de cabeça exigidas pelos

diversos algoritmos de escalonamento de disco.

Notas bibliográficas

Discussões de arrays redundantes de discos independentes (RAID) são apresentadas por [Patterson e outros \[1988\]](#) e no estudo detalhado de [Chen e outros \[1994\]](#). As arquiteturas de sistema de disco para computação de alto desempenho são discutidas por [Katz e outros \[1989\]](#). As melhorias nos sistemas RAID são descritas em [Wilkes e outros \[1996\]](#) e [Yu e outros \[2000\]](#). [Teorey e Pinkerton \[1972\]](#) apresentam uma análise comparativa antiga dos algoritmos de escalonamento de disco. Eles usam simulações que modelam um disco para o qual o tempo de busca é linear no número de cilindros atravessados. Para esse disco, LOOK é uma boa escolha para tamanhos de fila abaixo de 140 e C-LOOK é bom para tamanhos de fila acima de 100. [King \[1990\]](#) descreve maneiras de melhorar o tempo de busca movendo o braço do disco quando ele está ocioso por outro lado. [Seltzer e outros \[1990\]](#) e [Jackson e Wilkes \[1991\]](#) descrevem algoritmos de escalonamento de disco que consideram a latência de rotação além do tempo de busca. As otimizações de escalonamento que exploram os tempos ociosos do disco são discutidas em [Lumb e outros \[2000\]](#). [Worthington e outros \[1994\]](#) discutem o desempenho do disco e mostram o impacto insignificante no desempenho do gerenciamento de defeitos. O posicionamento de dados quentes para melhorar os tempos de busca foi considerada por [Ruemmler e Wilkes \[1991\]](#) e [Akyurek e Salem \[1993\]](#). [Ruemmler e Wilkes \[1994\]](#) descrevem um modelo de desempenho preciso para uma unidade de disco moderna. [Worthington e outros \[1995\]](#) dizem como determinar as propriedades de disco de baixo nível, como estrutura de zona, e esse trabalho é levado adiante por [Schindler e Gregory \[1999\]](#). Questões de gerenciamento de energia nos discos são discutidas em [Douglis e outros \[1994\]](#), [Douglis e outros \[1995\]](#), [Greenawalt \[1994\]](#) e [Golding e outros \[1995\]](#).

O tamanho da E/S e a aleatoriedade da carga de trabalho têm uma influência considerável no desempenho do disco. [Ousterhout e outros \[1985\]](#) e [Ruemmler e Wilkes \[1993\]](#) informam diversas características de carga de trabalho interessantes, incluindo que a maioria dos arquivos é pequena, a maioria dos arquivos recém-criados é excluída logo depois, a maioria dos arquivos abertos para leitura é lida sequencialmente em sua totalidade e a maioria das buscas é curta. [McKusick e outros \[1984\]](#) descrevem o Berkeley Fast File System (FFS), que usa muitas técnicas sofisticadas para obter um bom desempenho para uma grande variedade de cargas de trabalho. [McVoy e Kleiman \[1991\]](#) discutem outras melhorias no FFS básico. [Quinlan \[1991\]](#) descreve como implementar um sistema de arquivos no armazenamento WORM com um cache em disco magnético; [Richards \[1990\]](#) discute uma técnica de sistema de arquivos para o armazenamento terciário. [Maher e outros \[1994\]](#) proveem uma visão geral da integração dos sistemas de arquivos distribuídos e armazenamento terciário.

O conceito de hierarquia de armazenamento foi estudado por mais de um quarto de século. Por exemplo, um trabalho de 1970, de [Mattson e outros \[1970\]](#), descreve uma técnica matemática para prever o desempenho de uma hierarquia de armazenamento. [Alt \[1993\]](#) descreve a acomodação do armazenamento removível em um sistema operacional comercial, e [Miller e Katz \[1993\]](#) descrevem as características do acesso ao armazenamento terciário em um ambiente de supercomputação. [Benjamin \[1990\]](#) provê uma visão geral dos requisitos de armazenamento de massa para o projeto EOSDIS na NASA. O gerenciamento e o uso de discos conectados à rede e discos programáveis são discutidos em [Gibson e outros \[1997b\]](#), [Gibson e outros \[1997a\]](#), [Riedel e outros \[1998\]](#) e [Lee e Thekkath \[1996\]](#).

A tecnologia de armazenamento holográfico é o assunto de um artigo de [Psaltis e Mok \[1995\]](#); uma coleção de trabalhos sobre esse tópico datada de 1963 foi montada por [Sincerbox \[1994\]](#). [Asthana e Finkelstein \[1995\]](#) descrevem várias tecnologias de armazenamento emergentes, incluindo o armazenamento holográfico, fita óptica e interceptação de elétrons. [Toigo \[2000\]](#) provê uma descrição profunda sobre a tecnologia de disco moderna e várias tecnologias de armazenamento futuras em potencial.

CAPÍTULO 13

Sistemas de E/S

As duas tarefas principais de um computador são E/S e processamento. Em muitos casos, a tarefa principal é a E/S, e o processamento é meramente casual. Por exemplo, quando visitamos uma página Web ou editamos um arquivo, nosso interesse imediato é ler ou entrar com algumas informações, e não calcular uma resposta.

O papel do sistema operacional na E/S do computador é gerenciar e controlar as operações de E/S e dispositivos de E/S. Embora os tópicos relacionados apareçam em outros capítulos, aqui reunimos as partes para pintar uma imagem completa da E/S. Primeiro, descrevemos os fundamentos do hardware de E/S porque a natureza da interface de hardware impõe requisitos sobre as facilidades internas do sistema operacional. Em seguida, discutimos os serviços de E/S fornecidos pelo sistema operacional e a incorporação desses serviços na interface de E/S da aplicação. Em seguida, explicamos como o sistema operacional preenche a lacuna entre a interface de hardware e a interface da aplicação. Também discutimos o mecanismo STREAMS no UNIX System V, que permite a uma aplicação montar, dinamicamente, pipelines de driver. Finalmente, discutimos os aspectos de desempenho da E/S e os princípios de projeto do sistema operacional que melhoraram o desempenho da E/S.

OBJETIVOS DO CAPÍTULO

- Explorar a estrutura do subsistema de E/S de um sistema operacional.
- Discutir os princípios e as complexidades do hardware de E/S.
- Explicar os aspectos de desempenho do hardware e software de E/S.

13.1 Visão geral

O controle dos dispositivos conectados ao computador é uma preocupação importante dos projetistas de sistema operacional. Como os dispositivos de E/S variam muito em sua função e velocidade (considere um mouse, um disco rígido e um jukebox de CD-ROM), métodos variados são necessários para controlá-los. Esses métodos formam o *subsistema de E/S* do kernel, que isola o restante do kernel da complexidade de gerenciamento dos dispositivos de E/S.

A tecnologia do dispositivo de E/S exibe duas tendências em conflito. Por um lado, vemos uma padronização crescente das interfaces de software e hardware. Essa tendência nos ajuda a incorporar gerações de dispositivo melhoradas aos computadores e sistemas operacionais existentes. Por outro lado, vemos uma variedade cada vez mais ampla de dispositivos de E/S. Alguns dispositivos novos são diferentes dos dispositivos anteriores, portanto é um grande desafio incorporá-los aos nossos computadores e sistemas operacionais. Esse desafio é atendido por uma combinação de técnicas de hardware e software. Os elementos de hardware básicos da E/S, como portas, barramentos e controladores de dispositivos, acomodam uma grande variedade de dispositivos de E/S. Para englobar os detalhes e singularidades dos diferentes dispositivos, o kernel de um sistema operacional é estruturado para usar módulos de driver de dispositivo. Os **drivers de dispositivo** apresentam uma interface uniforme de acesso ao dispositivo para o subsistema de E/S, assim como as chamadas de sistema proveem uma interface-padrão entre a aplicação e o sistema operacional.

13.2 Hardware de E/S

Os computadores operam com muitos tipos de dispositivos. A maioria se encaixa nas categorias gerais de dispositivos de armazenamento (discos, fitas), dispositivos de transmissão (placas de rede, modems) e dispositivos de interface humana (monitor, teclado, mouse). Outros dispositivos são mais especializados, como aqueles envolvidos na direção de um avião de combate militar ou de uma nave espacial. Nessas aeronaves, um ser humano gera entrada para o computador de bordo por meio de um joystick e pedais, e o computador envia os comandos de saída que fazem os motores movimentarem lemes, flapes e aceleradores. Apesar da incrível variedade de dispositivos de E/S, só precisamos de alguns poucos conceitos para entender como os dispositivos são conectados e como o software pode controlar o hardware.

Um dispositivo se comunica com um computador enviando sinais por um cabo ou ainda pelo ar. O dispositivo se comunica com a máquina por meio de um ponto de conexão (ou **porta**), por exemplo, uma porta serial. Se os dispositivos utilizam um conjunto de fios comum, a conexão é chamada de **barramento** (bus). Um **barramento** é um conjunto de fios e um protocolo rigidamente definido que especifica um conjunto de mensagens que podem ser enviadas nos fios. Em termos de eletrônica, as mensagens são transportadas por padrões de voltagens elétricas aplicados aos fios com tempos definidos. Quando o dispositivo A possui um cabo que se conecta ao dispositivo B, o dispositivo B possui um cabo que se conecta ao dispositivo C e o dispositivo C se conecta a uma porta no computador, esse arranjo é chamado de **daisy chain**. Uma daisy chain normalmente funciona como um barramento.

Os barramentos são muito utilizados na arquitetura do computador, e variam em seus métodos de sinalização, velocidade, throughput e métodos de conexão. A [Figura 13.1](#) mostra uma estrutura de barramento típica do PC. Essa figura mostra um **barramento PCI** (o barramento de sistema comum do PC) que conecta o subsistema de memória do processador aos dispositivos rápidos, e um **barramento de expansão**, que conecta dispositivos relativamente lentos, como o teclado e as portas serial e USB. Na parte superior direita da figura, quatro discos são conectados em um barramento SCSI ligado a um controlador SCSI. Outros barramentos comuns utilizados para interconectar as principais partes de um computador são **PCI-X**, com throughput de até 4,3 GB; **PCI Express** (PCIe), com throughput de até 16 GB; e **HyperTransport**, com throughput de até 20 GB.

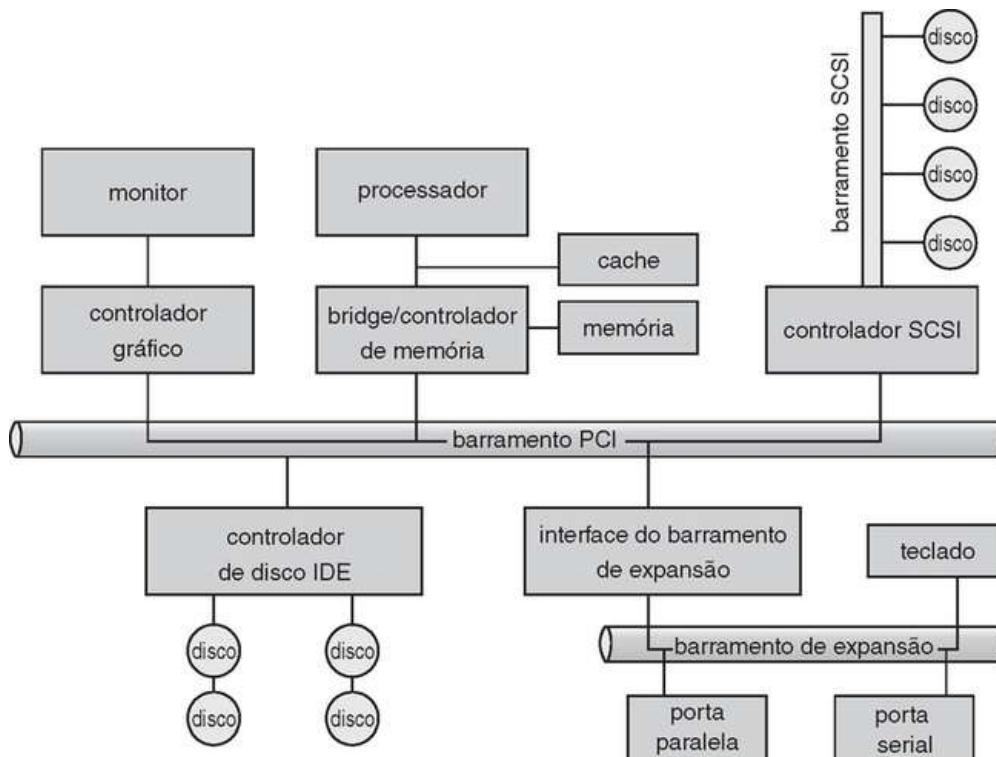


FIGURA 13.1 Uma estrutura típica de barramento do PC.

Um **controlador** é uma coleção de circuitos eletrônicos que podem operar uma porta, um barramento ou um dispositivo. Um controlador de porta serial é um controlador de dispositivo simples. Ele é um único chip (ou parte de um chip) no computador que controla os sinais nos fios de uma porta serial. Por outro lado, um controlador de barramento SCSI não é simples. Como o

protocolo SCSI é complexo, o controlador de barramento SCSI é implementado como uma placa de circuito separada (ou um **adaptador de host**), conectada ao computador. Ele contém processador, microcódigo e alguma memória local, para permitir processar as mensagens do protocolo SCSI. Alguns dispositivos possuem seus próprios controladores embutidos. Se você examinar uma unidade de disco, verá uma placa de circuito impresso presa a um lado. A placa é o controlador de disco. Ele implementa o lado do disco do protocolo para algum tipo de conexão, SCSI ou ATA, por exemplo. Ele possui microcódigo e um processador para realizar muitas tarefas, como mapeamento de setores defeituosos, pré-busca, uso de buffers e caches.

Como o processador pode dar comandos e dados a um controlador para realizar uma transferência de E/S? A resposta curta é que o controlador possui um ou mais registros para sinais de dados e controle. O processador se comunica com o controlador lendo e escrevendo padrões de bits nesses registradores. Uma maneira de realizar essa comunicação é por meio de instruções especiais de E/S, que especificam a transferência de um byte ou palavra para um endereço de porta de E/S. A instrução de E/S ativa as linhas do barramento para selecionar o dispositivo apropriado e mover os bits para dentro ou fora de um registrador do dispositivo. Como alternativa, o controlador do dispositivo pode aceitar E/S **mapeada em memória**. Nesse caso, os registradores de controle de dispositivo são mapeados no espaço de endereços do processador. A CPU executa requisições de E/S usando as instruções de transferência de dados para ler e escrever nos registradores de controle de dispositivo.

Alguns sistemas utilizam as duas técnicas. Por exemplo, os PCs utilizam instruções de E/S para controlar alguns dispositivos e E/S mapeada na memória para controlar outros. A [Figura 13.2](#) mostra os endereços de porta de E/S comuns do PC. O controlador gráfico possui portas de E/S para as operações básicas de controle, mas o controlador possui uma grande região mapeada na memória para manter o conteúdo da tela. O processo envia a saída para a tela escrevendo dados na região mapeada na memória. O controlador gera a imagem da tela com base no conteúdo dessa memória. Essa técnica é simples de usar. Além do mais, a escrita de milhões de bytes na memória gráfica é mais rápida do que emitir milhões de instruções de E/S. Entretanto, a facilidade de escrita em um controlador de E/S mapeada na memória é prejudicada por uma desvantagem. Como um tipo comum de falha de software é uma escrita por meio de um ponteiro incorreto para uma região da memória não desejada, o registrador de um dispositivo mapeado na memória é vulnerável à modificação acidental. Logicamente, a memória protegida ajuda a reduzir esse risco.

intervalo de endereços de E/S (hexadecimal)	dispositivo
000-00F	controlador de DMA
020-021	controlador de interrupção
040-043	temporizador
200-20F	controlador de jogos
2F8-2FF	porta serial (secundária)
320-32F	controlador de disco rígido
378-37F	porta paralela
3D0-3DF	controlador gráfico
3F0-3F7	controlador de unidade de disquete
3F8-3FF	porta serial (principal)

FIGURA 13.2 Locações de porta de E/S de dispositivo nos PCs (parcial).

Uma porta de E/S consiste em quatro registradores, chamados registradores de (1) entrada de dados; (2) saída de dados; (3) status; e (4) controle.

- O **registrador de entrada de dados** é lido pelo host para obter entrada.
- O **registrador de saída de dados** é escrito pelo host para enviar saída.
- O **registrador de status** contém bits que podem ser lidos pelo host. Esses bits indicam estados, por exemplo, se o comando atual foi completado, se um byte está disponível para ser lido do registrador de entrada de dados e se houve um erro no dispositivo.
- O **registrador de controle** pode ser escrito pelo host para iniciar um comando ou alterar o modo de um dispositivo. Por exemplo, um certo bit no registrador de controle de uma porta serial escolhe entre a comunicação full-duplex e half-duplex, outro ativa a verificação de paridade, um terceiro bit define o tamanho da palavra como 7 ou 8 bits, e outros bits selecionam uma das velocidades admitidas pela porta serial.

Os registradores de dados normalmente possuem de 1 a 4 bytes. Alguns controladores possuem chips FIFO, que podem manter vários bytes de dados de entrada ou saída para expandir a capacidade do controlador além do tamanho do registrador de dados. Um chip FIFO pode manter um pequeno burst de dados até o dispositivo ou host ser capaz de receber esses dados.

13.2.1 Polling

O protocolo completo para interação entre o host e um controlador pode ser intrincado, mas a noção básica do *handshaking* é simples. Explicamos o handshaking por meio de um exemplo. Suponha que 2 bits sejam usados para coordenar o relacionamento produtor-consumidor entre o controlador e o host. O controlador indica seu estado por meio do bit *ocupado* no registrador de *status*. (Lembre-se de que *marcar* um bit significa escrever 1 no bit, e *apagar* o bit significa escrever 0 nele.) O controlador marca o bit *ocupado* quando ele está trabalhando e apaga o bit *ocupado* quando está pronto para aceitar o próximo comando. O host sinaliza seus desejos por meio do bit *comando pronto* no registrador de *comando*. O host marca o bit *comando pronto* quando um comando está disponível para o controlador executar. Para este exemplo, o host escreve a saída por meio de uma porta, coordenando com o controlador por meio do handshaking, da seguinte forma:

1. O host repetidamente lê o bit *ocupado* até ele ser desligado.
2. O host liga o bit *escrever* no registrador de *comando* e escreve um byte no registrador de *saída de dados*.
3. O host liga o bit *comando pronto*.
4. Quando o controlador observa que o bit *comando pronto* está ligado, ele liga o bit *ocupado*.
5. O controlador lê o registrador de comando e vê o comando *write*. Ele lê o registrador de *saída de dados* para apanhar o byte e realiza a E/S para o dispositivo.
6. O controlador desliga o bit *comando pronto*, desliga o bit de *erro* no registrador de *status*, para indicar que a E/S de dispositivo teve sucesso e desliga o bit *ocupado*, para indicar que terminou. Esse loop é repetido para cada byte.

Na etapa 1, o host está **busy-waiting** ou **polling**: ele está em um loop, lendo o registrador de *status* o tempo todo, até o bit *ocupado* ser desligado. Se o controlador e o dispositivo forem rápidos, esse método é razoável. No entanto, se a espera for longa, o host deverá passar para outra tarefa. Como o host sabe quando o controlador ficou ocioso? Para alguns dispositivos, o host precisa atender ao dispositivo rapidamente, ou então dados serão perdidos. Por exemplo, quando os dados estão fluindo em uma porta serial ou de um teclado, o pequeno buffer no controlador estoura e os dados são perdidos se o host esperar muito tempo antes de voltar a ler os bytes.

Em muitas arquiteturas, três ciclos de instrução de CPU são suficientes para consultar um dispositivo: *ler* um registrador do dispositivo, realizar o *AND lógico* para extrair um bit de status e *desviar* se não for zero. Nitidamente, a operação básica de pooling é eficiente. Contudo, o pooling se torna ineficiente quando é tentado de forma repetida, enquanto raramente encontra um dispositivo pronto para atender, embora outro processamento útil da CPU permaneça sem ser feito. Nesses casos, pode ser mais eficiente arrumar as coisas para o controlador de hardware notificar a CPU quando o dispositivo estiver pronto para ser atendido, em vez de exigir que a CPU consulte sequencialmente o término da E/S. O mecanismo de hardware que permite a um dispositivo notificar a CPU é chamado de **interrupção**.

13.2.2 Interrupções

O mecanismo básico de interrupção funciona da seguinte maneira. O hardware da CPU possui um fio chamado **linha de requisição de interrupção (interrupt request)**, que a CPU percebe depois de executar cada instrução. Quando a CPU detecta que um controlador enviou um sinal na linha de requisição de interrupção, ela realiza um salvamento de estado e desvia para a **rotina do tratador de interrupção**, em um endereço fixo na memória. O tratador de interrupção determina a causa da interrupção, efetua a restauração do estado e executa uma instrução de retorno da interrupção para a CPU voltar ao estado de execução anterior à interrupção. Dizemos que o controlador de dispositivo *levanta* uma interrupção enviando um sinal na linha de requisição de interrupção, a CPU *captura* a interrupção e *despacha* para o tratador de interrupção, e o tratador *limpa* a interrupção atendendo ao dispositivo. A [Figura 13.3](#) resume o ciclo de E/S controlado por interrupção.

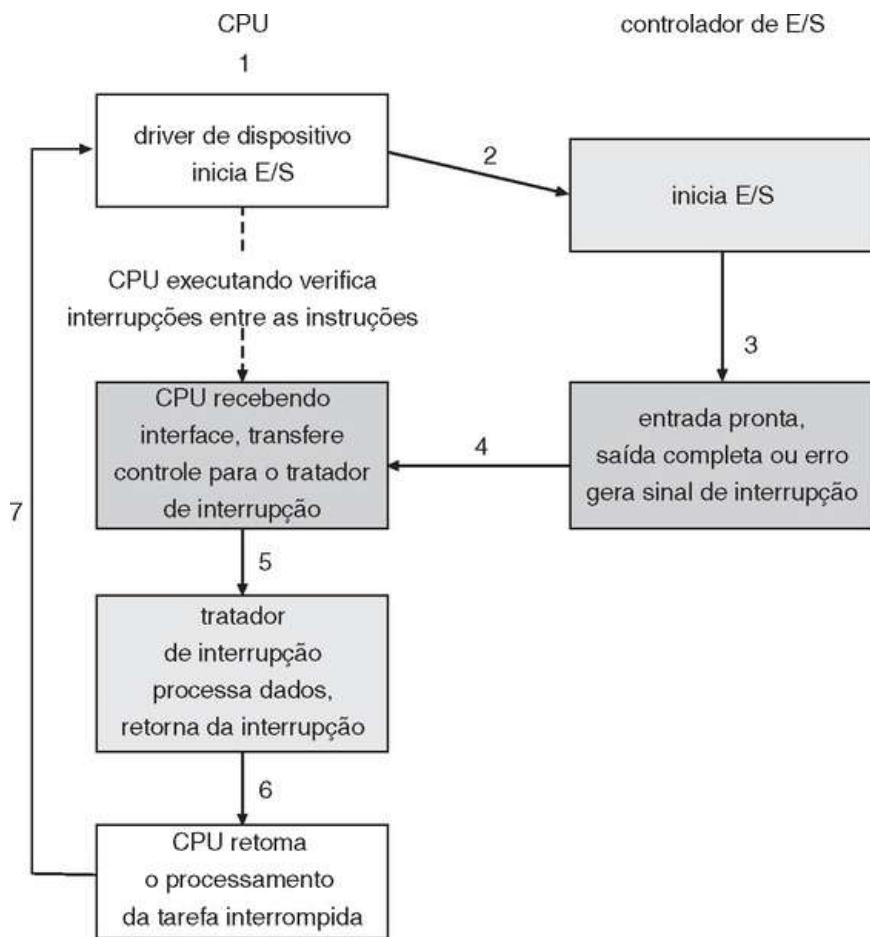


FIGURA 13.3 Ciclo de E/S controlado por interrupção.

Esse mecanismo básico de interrupção permite que a CPU responda a um evento assíncrono, como um controlador de dispositivo ficando pronto para o atendimento. Em um sistema operacional moderno, precisamos de recursos de tratamento de interrupção mais sofisticados.

1. Precisamos da capacidade de adiar o tratamento da interrupção durante um processamento crítico.
2. Precisamos de um modo eficiente de despachar para o tratador de interrupção apropriado para um dispositivo, sem primeiro efetuar o polling de todos os dispositivos para verificar qual levantou a interrupção.
3. Precisamos de interrupções multinível, para o sistema operacional poder distinguir entre interrupções de alta e baixa prioridade e poder responder com o grau de urgência apropriado.

No hardware de computador moderno, esses três recursos são fornecidos pela CPU e pelo **hardware do controlador de interrupção**.

A maioria das CPUs possui duas linhas de requisição de interrupção. Uma é a **interrupção não mascarável (nonmaskable interrupt)**, reservada para eventos irrecuperáveis como erros de memória. A segunda linha de interrupção é **mascarável**: ela pode ser desativada pela CPU antes da execução de sequências de instrução críticas, que não poderão ser interrompidas. A interrupção mascarável é usada pelos controladores de dispositivo para requisitar atendimento.

O mecanismo de interrupção aceita um **endereço** – um número que seleciona uma rotina de tratamento de interrupção em um pequeno conjunto. Na maior parte das arquiteturas, esse endereço é um deslocamento em uma tabela chamada **vetor de interrupções**. Esse vetor contém os endereços de memória de tratadores de interrupção especializados. A finalidade de um mecanismo de interrupção vetorializada é reduzir a necessidade de um único tratador de interrupção pesquisar todas as fontes de interrupção possíveis para determinar qual precisa ser atendida. Entretanto, na prática, os computadores têm mais dispositivos (e, portanto, tratadores de interrupção) do que elementos de endereço no vetor de interrupções. Um modo comum de resolver esse problema é usar a técnica de **encadeamento de interrupções**, em que cada elemento no vetor de interrupções aponta para o início de uma lista de tratadores de interrupção. Quando uma interrupção é levantada, os tratadores na lista correspondente são chamados um por um, até ser encontrado um que possa atender a requisição. Essa estrutura é um meio-termo entre o custo adicional de uma imensa tabela de interrupções e a ineficiência de um despacho para um único tratador de interrupção.

A [Figura 13.4](#) mostra o esquema do vetor de interrupções do processador Pentium da Intel. Os

eventos 0 a 31, que são não mascaráveis, são usados para sinalizar diversas condições de erro. Os eventos 32 a 255, que são mascaráveis, são usados para diversos propósitos como interrupções geradas pelo dispositivo.

número do vetor	descrição
0	erro de divisão
1	exceção de depuração (execução passo a passo)
2	interrupção não mascarável
3	breakpoint
4	estouro detectado pela INTO
5	limite de intervalo excedido
6	código de operação inválido
7	dispositivo indisponível
8	dupla falha
9	overrun de segmento do coprocessador (reservado)
10	segmento de estado de tarefa (TSS) inválido
11	segmento ausente
12	falha no segmento de pilha
13	proteção geral
14	falha de página (page fault)
15	(reservado pela Intel, não deve ser usado)
16	erro de ponto flutuante
17	verificação de alinhamento
18	verificação de máquina
19-31	(reservado pela Intel, não deve ser usado)
32-255	interrupções mascaráveis

FIGURA 13.4 Tabela de vetor de eventos do processador Pentium da Intel.

O mecanismo de interrupção também implementa um sistema de **níveis de prioridade de interrupção**. Esse mecanismo permite que a CPU adie o tratamento de interrupções de baixa prioridade sem abandonar todas as interrupções e torna possível uma interrupção de alta prioridade tomar o lugar da execução de uma interrupção de baixa prioridade.

Um sistema operacional moderno interage com o mecanismo de interrupção de várias maneiras. No momento do boot, o sistema operacional consulta sequencialmente os barramentos do hardware para determinar quais dispositivos estão presentes e instala os tratadores de interrupção correspondentes no vetor de interrupções. Durante a E/S, os diversos controladores de dispositivos levantam interrupções quando estão prontos para serem atendidos. Essas interrupções significam que a saída foi concluída ou que os dados de entrada estão disponíveis ou que foi detectada uma falha. O mecanismo de interrupção também é usado para tratar de uma grande variedade de **exceções**, como divisão por zero, acesso a um endereço de memória protegido ou inexistente, ou tentativa de executar uma instrução privilegiada a partir do modo do usuário. Os eventos que disparam interrupções possuem uma propriedade comum: eles são ocorrências que induzem a CPU a executar uma rotina urgente, com conteúdo próprio.

Um sistema operacional tem outros usos interessantes para um mecanismo de hardware e software eficiente que salva uma pequena quantidade de estado do processador e depois chama uma rotina privilegiada no kernel. Por exemplo, muitos sistemas operacionais utilizam o mecanismo de interrupção para paginação da memória virtual. Uma falha de página (page fault) é uma exceção que levanta uma interrupção. A interrupção suspende o processo atual e desvia para o tratador de falha de página no kernel do sistema. Esse tratador salva o estado do processo, move o processo para a fila de espera, realiza o gerenciamento de cache de página, escalona uma operação de E/S para buscar a página, escalona outro processo para retomar a execução e depois retorna da interrupção.

Outro exemplo é encontrado na implementação de chamadas de sistema. Em geral, um programa usa chamadas de biblioteca para enviar chamadas de sistema. As rotinas de biblioteca verificam os argumentos dados pela aplicação, montam uma estrutura de dados para levar os argumentos ao kernel e depois executam uma instrução especial, chamada **interrupção de software** (ou **trap**).

Essa instrução possui um operando que identifica o serviço do kernel desejado. Quando a chamada de sistema executa a instrução de interceptação, o hardware de interrupção salva o estado do código do usuário, passa para o modo supervisor e despacha para a rotina do kernel que implementa o serviço requisitado. A interceptação recebe uma prioridade de interrupção relativamente baixa em comparação com aquelas atribuídas às interrupções de dispositivo - executar uma chamada de sistema em favor de uma aplicação é menos urgente do que atender a um controlador de dispositivos antes de sua fila FIFO estourar e perder dados.

As interrupções também podem ser usadas para gerenciar o fluxo de controle dentro do kernel. Por exemplo, considere o processamento exigido para completar uma leitura no disco. Uma etapa é copiar os dados do espaço do kernel para o buffer do usuário. Essa cópia é demorada, mas não urgente - não deve impedir outro tratamento de interrupção de alta prioridade. Outra etapa é iniciar a próxima E/S pendente para essa unidade de disco. Essa etapa possui prioridade mais alta: se os discos tiverem de ser usados de forma eficiente, precisamos iniciar a próxima E/S assim que a anterior for concluída. Consequentemente, um *par* de tratadores de interrupção implementa o código do kernel que completa a leitura do disco. O tratador de alta prioridade registra o status da E/S, limpa a interrupção do dispositivo, inicia a próxima E/S pendente e levanta uma interrupção de baixa prioridade para completar o trabalho. Mais tarde, quando a CPU não estiver ocupada com o trabalho de alta prioridade, a interrupção de baixa prioridade será despachada. O tratador correspondente completa a E/S no nível do usuário copiando dados dos buffers do kernel para o espaço da aplicação e depois chamando o escalonador para colocar a aplicação na fila de prontos (ready queue).

Uma arquitetura de kernel com threads é bastante adequada para implementar múltiplas prioridades de interrupção e impor a precedência do tratamento de interrupção em relação ao processamento de segundo plano no kernel e às rotinas da aplicação. Ilustramos esse ponto com o kernel do sistema Solaris. No Solaris, os tratadores de interrupção são executados por threads do kernel. Um intervalo de altas prioridades é reservado para essas threads. Essas prioridades dão aos tratadores de interrupção precedência em relação ao código da aplicação e manutenção do kernel e implementam os relacionamentos de prioridade entre os tratadores de interrupção. As prioridades fazem o escalonador de threads do Solaris se apropriar de tratadores de interrupção de baixa prioridade em favor daqueles de prioridade mais alta, e a implementação em threads permite ao hardware multiprocessado executar vários tratadores de interrupção concorrentemente.

Resumindo, as interrupções são utilizadas nos sistemas operacionais modernos para tratar dos eventos assíncronos e desviar para rotinas, em modo supervisor, no kernel. Para permitir que o trabalho mais urgente seja feito primeiro, os computadores modernos utilizam um sistema de prioridades de interrupção. Controladores de dispositivos, falhas do hardware e chamadas de sistema levantam interrupções para disparar rotinas do kernel. Como as interrupções são muito usadas para o processamento sensível ao tempo, é necessário o tratamento eficiente das interrupções para o bom desempenho no sistema.

13.2.3 Acesso direto à memória

Para um dispositivo que realiza grandes transferências, como uma unidade de disco, parece ser um desperdício usar um processador caro, de uso geral, para observar bits de status e inserir dados em um registrador de controlador 1 byte de cada vez - processo denominado **E/S programada (Programmed I/O - PIO)**. Muitos computadores evitam sobrecarregar a CPU principal com PIO passando parte desse trabalho para um processador de uso específico, chamado **controlador de acesso direto à memória (Direct Memory Access - DMA)**. Para iniciar uma transferência de DMA, o host escreve um bloco de comando de DMA na memória. Esse bloco contém um ponteiro para a origem de uma transferência, um ponteiro para o destino da transferência e um contador do número de bytes a serem transferidos. A CPU escreve o endereço desse bloco de comando no controlador de DMA, depois prossegue com outro trabalho. O controlador de DMA prossegue operando diretamente sobre o barramento da memória, colocando endereços no barramento para realizar as transferências sem a ajuda da CPU principal. Um controlador de DMA simples é um componente-padrão nos PCs, mas as **placas de controle do barramento (bus-mastering) de E/S** nos PCs normalmente contêm seu próprio hardware de DMA de alta velocidade.

O handshaking entre o controlador de DMA e o controlador de dispositivo é realizado por meio de um par de fios chamado requisição (request) de DMA e confirmação (acknowledge) de DMA. O controlador de dispositivo coloca um sinal no fio de requisição de DMA quando uma palavra de dados está disponível para transferência. O sinal faz o controlador de DMA se apoderar do barramento de memória, colocar o endereço desejado nos fios de endereço de memória e colocar um sinal no fio de confirmação de DMA. Quando o controlador de dispositivo recebe o sinal de confirmação de DMA, ele transfere a palavra de dados para a memória e remove o sinal de requisição de DMA.

Quando a transferência inteira terminar, o controlador de DMA interrompe a CPU. Esse processo é representado na [Figura 13.5](#). Quando o controlador de DMA se apodera do barramento de memória, a CPU está momentaneamente impedida de acessar a memória principal, embora ainda possa

acessar itens de dados e seu cache principal e secundário. Embora esse **roubo de ciclo** possa atrasar a computação da CPU, a passagem do trabalho de transferência de dados para um controlador de DMA em geral melhora o desempenho total do sistema. Algumas arquiteturas utilizam endereços físicos de memória para o DMA, mas outras realizam o **acesso direto à memória virtual (Direct Virtual Memory Access - DVMA)**, usando endereços virtuais que passam pelo processo de tradução para endereços físicos. O DVMA pode realizar transferência entre dois dispositivos mapeados na memória sem a intervenção da CPU ou do uso da memória principal.

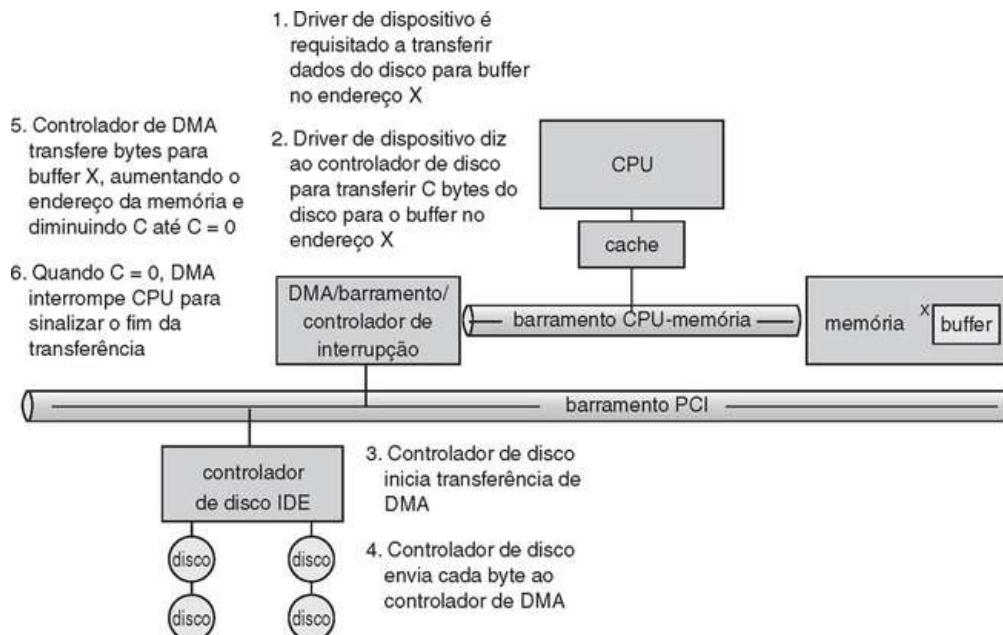


FIGURA 13.5 Etapas em uma transferência de DMA.

Em kernels no modo protegido, o sistema operacional em geral impede que os processos emitam comandos do dispositivo diretamente. Essa disciplina protege os dados contra violações de controle de acesso e também protege o sistema contra o uso indevido dos controladores de dispositivos, o que poderia causar uma falha no sistema. Em vez disso, o sistema operacional exporta funções que um processo suficientemente privilegiado pode usar para acessar operações em baixo nível no hardware subjacente. Em kernels sem proteção de memória, os processos podem acessar diretamente os controladores de dispositivos. Esse acesso direto pode ser usado para obter alto desempenho, pois pode evitar comunicação do kernel, trocas de contexto e camadas de software do kernel. Infelizmente, isso interfere com a segurança e a estabilidade do sistema. A tendência nos sistemas operacionais de uso geral é proteger a memória e dispositivos, para o sistema poder tentar se proteger contra aplicações maliciosas ou com erros.

13.2.4 Resumo de hardware da E/S

Embora os aspectos de hardware da E/S sejam complexos quando considerados no nível de detalhe dos projetistas do hardware eletrônico, os conceitos que acabamos de descrever são suficientes para se entender muitos aspectos de E/S dos sistemas operacionais. Vamos rever os principais conceitos:

- Barramento.
- Controlador.
- Porta de E/S e seus registradores.
- Relacionamento de handshaking entre o host e um controlador de dispositivo.
- Execução desse handshaking em um loop de polling ou por meio de interrupções.
- Entrega desse trabalho para um controlador de DMA para grandes transferências.

Demos um exemplo básico do handshaking que ocorre entre um controlador de dispositivo e o host nesta seção. Na realidade, a grande variedade de dispositivos existentes impõe um problema para aqueles que implementam o sistema operacional. Cada tipo de dispositivo possui seu próprio conjunto de capacidades, definições de bit de controle e protocolo para interagir com o host – e todos eles são diferentes. Como o sistema operacional pode ser projetado de modo que possamos conectar novos dispositivos ao computador sem reescrever o sistema operacional? Além disso, quando os dispositivos variam tanto assim, como o sistema operacional pode prover uma interface de E/S conveniente e uniforme para as aplicações? Abordamos essas questões a seguir.

13.3 Interface de E/S da aplicação

Nesta seção, vamos discutir sobre as técnicas de estruturação e as interfaces para o sistema operacional que permitem aos dispositivos de E/S serem tratados de um modo-padrão, uniforme. Explicamos, por exemplo, como uma aplicação pode abrir um arquivo em um disco sem saber que tipo de disco é, e como novos discos e outros dispositivos podem ser acrescentados a um computador sem incomodar o sistema operacional.

Como outros problemas complexos de engenharia de software, a técnica usada aqui envolve abstração, encapsulamento e camadas de software. Especificamente, podemos abstrair as diferenças detalhadas nos dispositivos de E/S identificando alguns tipos gerais. Cada tipo geral é acessado por meio de um conjunto padronizado de funções - uma **interface**. As diferenças são englobadas em módulos do kernel denominados drivers de dispositivo, que internamente são ajustados a dispositivos específicos mas que exportam uma das interfaces-padrão. A [Figura 13.6](#) ilustra como as partes relacionadas com a E/S do kernel são estruturadas nas camadas de software.

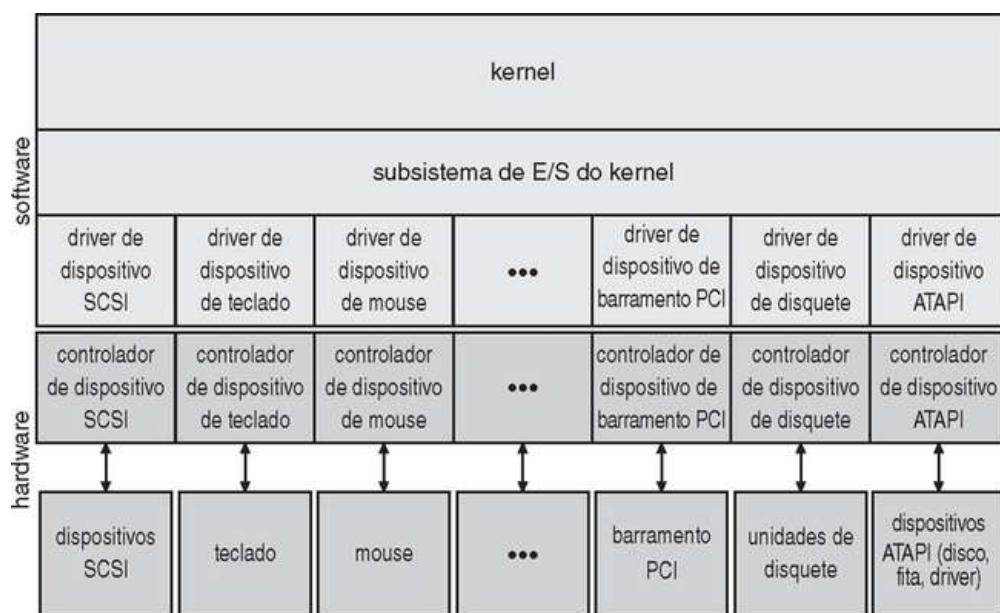


FIGURA 13.6 Uma estrutura de E/S do kernel.

A finalidade da camada de driver de dispositivo é ocultar as diferenças entre os controladores de dispositivos do subsistema de E/S do kernel, assim como as chamadas de sistema de E/S englobam o comportamento dos dispositivos em algumas classes genéricas que ocultam as diferenças de hardware das aplicações. Tornar o subsistema de E/S independente do hardware simplifica a tarefa do desenvolvedor de sistema operacional. Isso também beneficia os fabricantes de hardware. Eles projetam novos dispositivos para serem compatíveis com uma interface de controlador de host existente (como SCSI-2) ou então escrevem drivers de dispositivo para realizar a interface do novo hardware com sistemas operacionais populares. Assim, podemos conectar novos periféricos a um computador sem esperar o fornecedor do sistema operacional desenvolver o código de suporte.

Infelizmente para os fabricantes de hardware de dispositivo, cada tipo de sistema operacional possui seus próprios padrões para a interface com o driver de dispositivo. Determinado dispositivo pode vir com vários controladores de dispositivo - por exemplo, drivers para MS-DOS, Windows 95/98, Windows NT/2000 e Solaris. Os dispositivos variam em muitas dimensões, conforme ilustramos na [Figura 13.7](#).

aspecto	variação	exemplo
modo de transferência de dados	caracteres blocos	terminal disco
método de acesso	sequencial aleatório	modem CD-ROM
agenda de transferência	síncrono assíncrono	fita teclado
compartilhamento	dedicado compartilhável	fita teclado
velocidade do dispositivo	latência tempo de busca taxa de transferência atraso entre operações	
direção da E/S	somente de leitura somente de escrita leitura/escrita	CD-ROM controlador gráfico disco

FIGURA 13.7 Características dos dispositivos de E/S.

- **Fluxo de caracteres ou bloco.** Um dispositivo de fluxo de caracteres transfere bytes um por um, enquanto um dispositivo de bloco transfere um bloco de bytes como uma unidade.
- **Acesso sequencial ou aleatório.** Um dispositivo sequencial transfere dados em uma ordem fixa, determinada pelo dispositivo, enquanto o usuário de um dispositivo de acesso aleatório pode instruir o dispositivo a buscar em qualquer um dos locais de armazenamento de dados disponíveis.
- **Síncrono ou assíncrono.** Um dispositivo síncrono realiza transferências de dados com tempos de resposta previsíveis. Um dispositivo assíncrono exibe tempos de resposta irregulares ou imprevisíveis.
- **Compartilhável ou dedicado.** Um dispositivo compartilhável pode ser usado simultaneamente por vários processos ou threads; um dispositivo dedicado não pode.
- **Velocidade de operação.** As velocidades do dispositivo variam desde alguns bytes por segundo até alguns gigabytes por segundo.
- **Leitura/escrita, somente leitura ou somente escrita.** Alguns dispositivos realizam entrada e saída, mas outros aceitam apenas uma direção para a transferência de dados.

Para fins de acesso da aplicação, muitas dessas diferenças são escondidas pelo sistema operacional, e os dispositivos são agrupados em alguns tipos convencionais. Descobriu-se que os estilos resultantes de acesso ao dispositivo são úteis e bastante aplicáveis. Embora as chamadas de sistema exatas possam diferir entre os sistemas operacionais, as categorias de dispositivos são razoavelmente padronizadas. As principais convenções de acesso são E/S em bloco, E/S de fluxo de caracteres, acesso a arquivo mapeado na memória e sockets de rede. Os sistemas operacionais também proveem chamadas de sistema especiais para acessar alguns dispositivos adicionais, como um relógio para a hora do dia e um temporizador. Alguns sistemas operacionais proveem um conjunto de chamadas de sistema para dispositivos de apresentação gráfica, vídeo e áudio.

A maioria dos sistemas operacionais também possui um **escape**, ou **porta dos fundos (back door)**, que passa comandos de forma transparente, de uma aplicação para um driver de dispositivo. No UNIX, essa chamada de sistema é `ioctl()` (para controle de E/S). A chamada de sistema `ioctl()` permite a uma aplicação acessar qualquer funcionalidade que possa ser implementada por qualquer driver de dispositivo, sem a necessidade de inventar uma nova chamada de sistema. A chamada de sistema `ioctl()` possui três argumentos. O primeiro é o descritor de arquivo que conecta a aplicação ao driver, referindo-se a um dispositivo de hardware gerenciado por esse driver. O segundo é um inteiro que seleciona um dos comandos implementados no driver. O terceiro é um ponteiro para uma estrutura de dados qualquer na memória, permitindo assim que a aplicação e o driver comuniquem qualquer informação de controle necessária ou dados.

13.3.1 Dispositivos de bloco e caractere

A **interface de dispositivo de bloco** captura todos os aspectos necessários para acessar unidades de disco e outros dispositivos orientados a bloco. Esperamos que o dispositivo entenda comandos como `read()` e `write()` e, se for um dispositivo de acesso aleatório, ele terá um comando `seek()` para especificar qual bloco será transferido em seguida. As aplicações acessam tal dispositivo por meio de uma interface do sistema de arquivos. Podemos ver que `read()`, `write()` e `seek()` capturam os comportamentos essenciais dos dispositivos de armazenamento em bloco, de modo que as aplicações não precisam conhecer as diferenças em baixo nível entre esses dispositivos.

O próprio sistema operacional, e também as aplicações especiais, como sistemas de gerenciamento de banco de dados, pode preferir acessar um dispositivo de bloco como uma simples sequência linear de blocos. Esse modo de acesso às vezes é chamado de **E/S bruta**. Se a própria aplicação realizar seu trabalho com buffer, o uso de um sistema de arquivos causará um uso extra,

desnecessário. De modo semelhante, se uma aplicação fornece seu próprio lock de blocos ou regiões de arquivo, então quaisquer serviços de lock do sistema operacional serão no mínimo redundantes e contraditórios no pior dos casos. Para evitar esses conflitos, o acesso bruto aos dispositivos passa o controle do dispositivo diretamente para a aplicação, deixando que o sistema operacional fique de lado. Infelizmente, nenhum serviço do sistema operacional é realizado sobre esse dispositivo. Um meio-termo que está se tornando comum é que o sistema operacional permita um modo de operação sobre um arquivo que desative o uso de buffers e locks. No mundo do UNIX, isso é chamado de **E/S direta**.

O acesso a arquivo mapeado na memória pode ser disposto em uma camada acima dos drivers de dispositivo de bloco. Em vez de prover operações de leitura e escrita, uma interface mapeada na memória provê acesso ao armazenamento em disco por meio de uma sequência de bytes na memória principal. A chamada de sistema que mapeia um arquivo na memória retorna o endereço da memória virtual contendo uma cópia do arquivo. As transferências de dados reais são realizadas apenas quando necessárias para atender o acesso ao código residente na memória. Como as transferências são tratadas pelo mesmo mecanismo usado para o acesso à memória virtual por demanda de página, a E/S mapeada na memória é eficiente. O mapeamento na memória também é conveniente para programadores - o acesso a um arquivo mapeado na memória é tão simples quanto ler e escrever na memória. Os sistemas operacionais que proveem memória virtual utilizam a interface de mapeamento para serviços do kernel. Por exemplo, para executar um programa, o sistema operacional mapeia o executável na memória e depois transfere o controle para o endereço de entrada do executável. A interface de mapeamento também é utilizada para o acesso do kernel ao swap space no disco.

Um teclado é um exemplo de um dispositivo que é acessado por meio da **interface de fluxo de caracteres**. As chamadas básicas do sistema nessa interface permitem que uma aplicação apanhe [get()] ou envie [put()] um caractere. Em cima dessa interface, bibliotecas podem ser montadas para prover acesso a uma linha de cada vez, com serviços de buffer e edição (por exemplo, quando um usuário pressiona a tecla Backspace, o caractere anterior é removido do fluxo de entrada). Esse estilo de acesso é conveniente para dispositivos de entrada como teclado, mouse e modem, que produzem dados para entrada "espontaneamente", ou seja, em momentos que podem não ser previstos pela aplicação. Esse estilo de acesso também é bom para dispositivos de saída, como impressoras ou placas de áudio, que se encaixam no conceito de fluxo linear de bytes.

13.3.2 Dispositivos de rede

Como as características de desempenho e endereçamento da E/S de rede são muito diferentes daquelas da E/S de disco, a maioria dos sistemas operacionais provê uma interface de E/S de rede, diferente da interface read()-write()-seek() usada para os discos. Uma interface disponível em muitos sistemas operacionais, incluindo UNIX e Windows NT, é a interface de **socket** de rede.

Pense em uma tomada de parede para a eletricidade: qualquer aparelho elétrico pode ser conectado. Por analogia, as chamadas de sistema na interface de socket permitem a uma aplicação criar um socket para conectar um socket local a um endereço remoto (que conecta a um socket criado por outra aplicação), para escutar qualquer aplicação remota, para conectar ao socket local e para enviar e receber pacotes pela conexão. Para dar suporte à implementação dos servidores, a interface de socket também provê uma função chamada select(), que controla um conjunto de sockets. Uma chamada a select() retorna informações sobre quais sockets possuem um pacote esperando para ser recebido e quais sockets têm espaço para aceitar um pacote a ser enviado. O uso de select() elimina a consulta e a espera ocupada que, de outra forma, seriam necessárias para a E/S de rede. Essas funções englobam os comportamentos essenciais das redes, facilitando bastante a criação de aplicações distribuídas que possam usar qualquer hardware e pilha de protocolos da rede subjacente.

Muitas outras técnicas para a comunicação entre processos e a comunicação por rede foram implementadas. Por exemplo, o Windows NT provê uma interface para a placa de interface de rede e uma segunda interface para os protocolos de rede. No UNIX, que possui um longo histórico como campo de prova para a tecnologia de rede, encontramos pipes half-duplex, FIFOs full-duplex, STREAMS full-duplex, filas de mensagens e sockets.

13.3.3 Relógios e temporizadores

A maioria dos computadores possui relógios e temporizadores de hardware que proveem três funções básicas:

- Dar a hora atual.
- Dar o tempo transcorrido.
- Definir um temporizador para disparar a operação X no momento T.

Essas funções são muito usadas pelo sistema operacional e também por aplicações sensíveis ao tempo. Infelizmente, as chamadas de sistema que implementam essas funções não são padronizadas entre os sistemas operacionais.

O hardware para medir o tempo transcorrido e disparar operações é chamado de **temporizador de intervalo programável (programmable interval time)**. Ele pode ser definido para esperar por determinado tempo e depois gerar uma interrupção. Ele pode ser definido para realizar essa operação uma vez ou para repetir o processo, gerando interrupções periódicas. O escalonador usa esse mecanismo para gerar uma interrupção que se apropriará de um processo no final de sua fatia de tempo. O subsistema de E/S de disco o utiliza para invocar o esvaziamento de buffers de cache sujos periodicamente para o disco, e o subsistema de rede o utiliza para cancelar operações que estão prosseguindo muito lentamente, devido ao congestionamento ou falhas da rede. O sistema operacional também pode prover uma interface para os processos do usuário usarem os temporizadores. O sistema operacional pode admitir mais requisições de temporizador do que a quantidade de canais de hardware de temporizador, simulando relógios virtuais. Para fazer isso, o kernel (ou o driver de dispositivo de temporizador) mantém uma lista de interrupções desejadas por suas próprias rotinas e por requisições do usuário, classificada na ordem de hora mais cedo em primeiro lugar. Ele define o temporizador para a hora mais cedo. Quando o temporizador interrompe, o kernel sinaliza ao requisitante e recarrega o temporizador com a próxima hora na sequência.

Em muitos computadores, a taxa de interrupção gerada pelo relógio de hardware está entre 18 e 60 tique-tiques por segundo. Essa resolução é muito bruta, pois um computador moderno pode executar centenas de milhões de instruções por segundo. A precisão dos gatilhos é limitada pela resolução bruta do temporizador, juntamente com o custo adicional de manter relógios virtuais. Além disso, se os tique-tiques do temporizador forem usados para manter a hora no relógio do sistema, ele poderá ficar defasado. Na maioria dos computadores, o relógio do hardware é construído de um contador de alta frequência. Em alguns computadores, o valor desse contador pode ser lido por um registrador de dispositivo, de modo que o contador pode ser considerado um relógio de alta resolução. Embora esse relógio não gere interrupções, ele provê medições precisas de intervalos de tempo.

13.3.4 E/S bloqueante e não bloqueante

Outro aspecto da interface de chamada de sistema está relacionado com a escolha entre E/S bloqueante e E/S não bloqueante. Quando uma aplicação emite uma chamada de sistema **bloqueante**, a execução da aplicação é suspensa. A aplicação é movida da fila de execução do sistema operacional para uma fila de espera (wait queue). Depois de a chamada de sistema ser terminada, a aplicação é movida de volta para a fila de pronto (ready queue), onde fica elegível para retomar a execução. Ao retomar a execução, ela receberá os valores retornados pela chamada de sistema. As ações físicas realizadas pelos dispositivos de E/S costumam ser assíncronas - elas exigem um tempo variável ou imprevisível. Apesar disso, a maioria dos sistemas operacionais utiliza as chamadas de sistema bloqueantes para a interface da aplicação, pois o código de aplicação bloqueante é mais fácil de entender do que o código de aplicação não bloqueante.

Alguns processos no nível de usuário precisam de E/S **não bloqueante**. Um exemplo é uma interface de usuário que recebe entrada de teclado e mouse enquanto processa e exibe dados na tela. Outro exemplo é uma aplicação de vídeo que lê quadros de um arquivo no disco enquanto, simultaneamente, descompacta e exibe a saída no monitor.

Um escritor de aplicação pode sobrepor a execução com a E/S escrevendo uma aplicação multithreads. Algumas threads podem realizar chamadas de sistema bloqueante, enquanto outras continuam executando. Os desenvolvedores do Solaris usaram essa técnica para implementar uma biblioteca de E/S assíncrona em nível de usuário, liberando dessa tarefa o escritor da aplicação. Alguns sistemas operacionais proveem chamadas de sistema para E/S não bloqueante. Uma chamada não bloqueante não interrompe a execução da aplicação por um tempo estendido. Em vez disso, ela retorna rapidamente, com um valor de retorno que indica quantos bytes foram transferidos.

Uma alternativa a uma chamada de sistema não bloqueante é uma chamada de sistema assíncrona. Esse tipo de chamada retorna imediatamente, sem esperar o término da E/S. A aplicação continua a executar seu código. O término da E/S em algum momento futuro é comunicado à aplicação, seja pela configuração de alguma variável no espaço de endereços da aplicação, seja pelo disparo de um sinal ou interrupção de software, ou por uma rotina de call-back executada fora do fluxo de controle linear da aplicação. A diferença entre as chamadas de sistema não bloqueantes e assíncronas é que um `read()` não bloqueante retorna imediatamente com quaisquer dados disponíveis - o número total de bytes requisitados, menos ou nenhum. Uma chamada `read()` assíncrona requisita uma transferência realizada por completo, mas que terminará em algum momento futuro. Esses dois métodos de E/S são mostrados na [Figura 13.8](#).

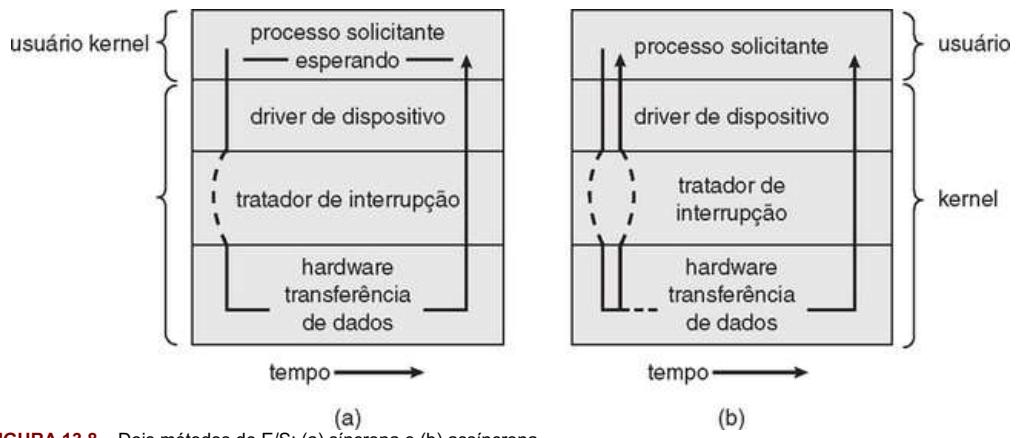


FIGURA 13.8 Dois métodos de E/S: (a) síncrona e (b) assíncrona.

Um bom exemplo de comportamento não bloqueante é a chamada de sistema `select()` para sockets de rede. Essa chamada de sistema apanha um argumento que especifica um tempo de espera máximo. Definindo-a como 0, uma aplicação pode consultar a atividade da rede não bloqueante. Contudo, o uso de `select()` introduz um custo adicional extra, pois a chamada `select()` só verifica se a E/S é possível. Para uma transferência de dados, `select()` precisa ser acompanhado de algum tipo de comando `read()` ou `write()`. Uma variação dessa técnica, encontrada no Mach, é uma chamada de leitura múltipla bloqueante. Ela especifica as leituras desejadas para diversos dispositivos em uma chamada de sistema e retorna assim que qualquer um deles terminar.

13.4 Subsistema de E/S do kernel

Os kernels proveem muitos serviços relacionados com a E/S. Vários serviços - escalonamento, uso de buffers e caches, spooling, reserva de dispositivo e tratamento de erros - são fornecidos pelo subsistema de E/S do kernel e montados na infraestrutura do hardware e do driver de dispositivo. O subsistema de E/S também é responsável por proteger-se de processos errantes e usuários maliciosos.

13.4.1 Escalonamento de E/S

Escalonar um conjunto de requisições de E/S significa determinar uma boa ordem em que serão executadas. A ordem em que as aplicações emitem chamadas de sistema raramente é a melhor opção. O escalonamento pode melhorar o desempenho geral do sistema, pode compartilhar o acesso ao dispositivo de forma mais justa entre os processos e pode reduzir o tempo de espera médio para o término da E/S. Aqui está um exemplo simples para ilustrar. Suponha que um braço do disco esteja perto do início de um disco e que três aplicações emitem chamadas de leitura bloqueante. A aplicação 1 requisita um bloco perto do final do disco, a aplicação 2 requisita um perto do início e a aplicação 3 requisita um no meio do disco. O sistema operacional pode reduzir a distância que o braço do disco atravessa atendendo às aplicações na ordem 2, 3, 1. A arrumação da ordem de atendimento dessa maneira é a essência do escalonamento de E/S.

Os desenvolvedores de sistema operacional implementam o escalonamento mantendo uma fila de requisições para cada dispositivo. Quando uma aplicação emite uma chamada de sistema de E/S bloqueante, a requisição é colocada na fila para esse dispositivo. O escalonador de E/S arruma a ordem da fila para melhorar a eficiência geral do sistema e o tempo de resposta médio experimentado pelas aplicações. O sistema operacional também pode tentar ser justo, de modo que nenhuma aplicação receba um serviço especialmente fraco, ou então poderá dar atendimento prioritário para requisições sensíveis a atraso. Por exemplo, requisições do subsistema de memória virtual podem ter prioridade em relação a requisições da aplicação. Vários algoritmos de escalonamento para E/S de disco estão detalhados na [Seção 12.4](#).

Quando um kernel admite E/S assíncrona, ele precisa ser capaz de registrar muitas requisições de E/S ao mesmo tempo. Para essa finalidade, o sistema operacional poderia conectar a fila de entrada a uma **tabela de status de dispositivo**. O kernel gerencia essa tabela, que contém uma entrada para cada dispositivo de E/S, como mostra a [Figura 13.9](#). Cada entrada da tabela indica o tipo do dispositivo, endereço e estado (não funcionando, ocioso ou ocupado). Se o dispositivo estiver ocupado com uma requisição, o tipo da requisição e outros parâmetros serão armazenados na entrada da tabela para esse dispositivo.

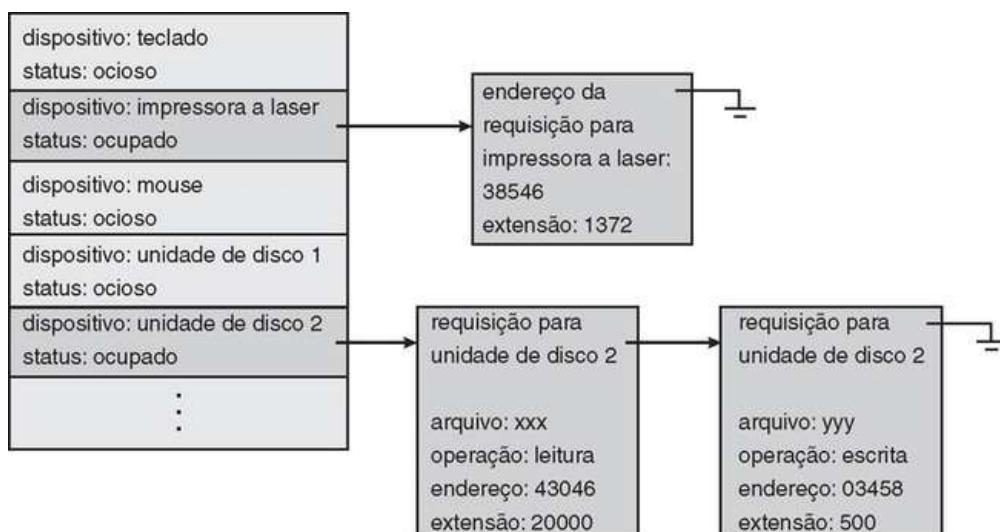


FIGURA 13.9 Tabela de status de dispositivo.

Um modo como o subsistema de E/S melhora a eficiência do computador é escalonando as operações de E/S. Outro modo é usando o espaço de armazenamento na memória principal ou no disco, por meio de técnicas que utilizam buffers, caches e spooling.

13.4.2 Buffers

Um **buffer** é uma área da memória que armazena dados que estão sendo transferidos entre dois

dispositivos ou entre um dispositivo e uma aplicação. Os buffers são utilizados por três motivos. Um motivo é para lidar com uma divergência de velocidade entre o produtor e o consumidor de um fluxo de dados. Suponha, por exemplo, que um arquivo esteja sendo recebido via modem para armazenamento no disco rígido. O modem é cerca de mil vezes mais lento do que o disco rígido. Assim, um buffer é criado na memória principal para acumular os bytes recebidos do modem. Quando um buffer de dados inteiro tiver chegado, o buffer pode ser escrito no disco em uma única operação. Como a escrita no disco não é instantânea e o modem ainda precisa de um lugar para armazenar dados adicionais que chegam, são usados dois buffers. Depois de o modem preencher o primeiro buffer, a escrita em disco será requisitada. O modem, então, começa a preencher o segundo buffer, enquanto o primeiro é escrito no disco. Quando o modem tiver preenchido o segundo buffer, a escrita em disco do primeiro já deverá estar concluída, de modo que o modem pode passar para o primeiro buffer enquanto o disco escreve o segundo. Esse **duplo buffer** desacopla o produtor dos dados do seu consumidor, aliviando assim os requisitos de temporização entre eles. A necessidade desse desacoplamento é mostrada na [Figura 13.10](#), que lista as enormes diferenças nas velocidades de dispositivo do hardware típico do computador.

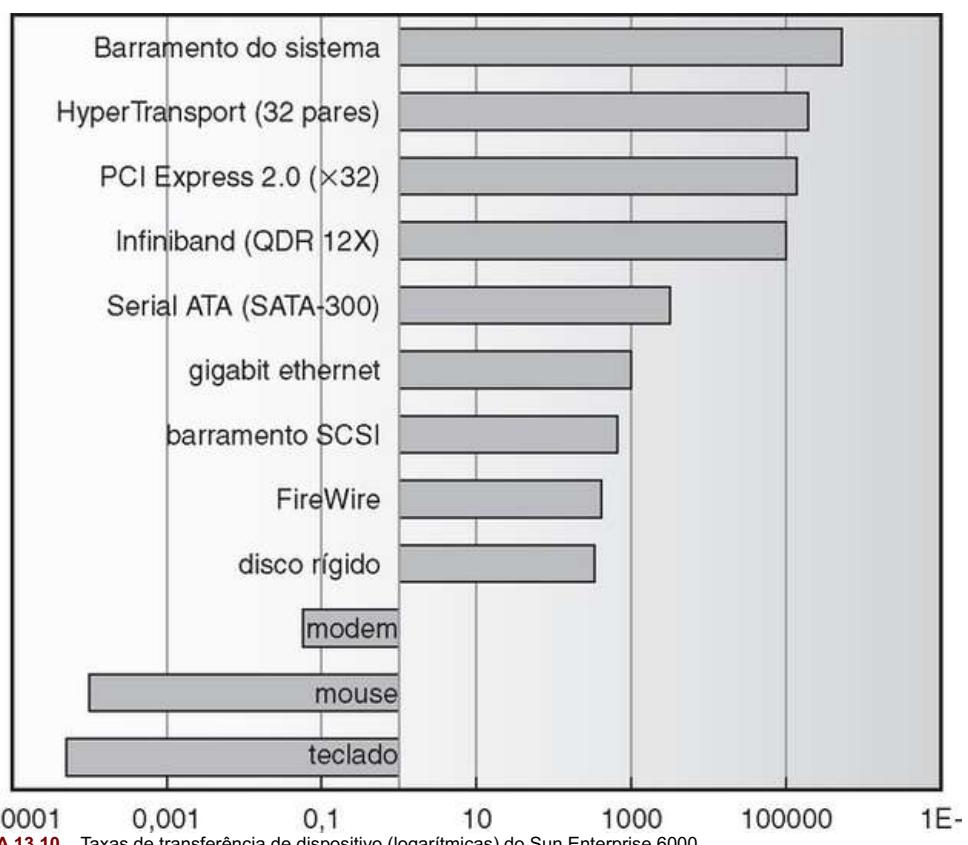


FIGURA 13.10 Taxas de transferência de dispositivo (logarítmicas) do Sun Enterprise 6000.

Um segundo uso dos buffers é oferecer adaptações para dispositivos que possuem tamanhos de transferência de dados diferentes. Essas disparidades são comuns especialmente em redes de computadores, nas quais os buffers são bastante usados para fragmentação e remontagem de mensagens. No lado do envio, uma mensagem grande é fragmentada em pequenos pacotes de rede. Os pacotes são enviados pela rede, e o lado receptor os coloca em um buffer de remontagem para formar uma imagem dos dados originais.

Um terceiro uso dos buffers é admitir a semântica de cópia para a E/S da aplicação. Um exemplo esclarecerá o significado da “semântica de cópia”. Suponha que uma aplicação tenha um buffer de dados que deseja escrever no disco. Ela chama a chamada de sistema `write()`, fornecendo um ponteiro para o buffer e um inteiro especificando a quantidade de bytes a escrever. Depois de a chamada de sistema retornar, o que acontece se a aplicação mudar o conteúdo do buffer? Com a **semântica de cópia**, a versão dos dados escrita em disco com certeza é a versão no momento da chamada de sistema por parte da aplicação, independentemente de quaisquer mudanças subsequentes no buffer da aplicação. Um modo simples pelo qual o sistema operacional pode garantir a semântica de cópia é fazer a chamada de sistema `write()` copiar os dados da aplicação para um buffer do kernel antes de retornar o controle à aplicação. A escrita em disco é realizada a partir do buffer do kernel, de modo que as mudanças subsequentes no buffer da aplicação não têm efeito. A cópia dos dados entre os buffers do kernel e o espaço de dados da aplicação é comum nos

sistemas operacionais, apesar do custo adicional ocasionado por essa operação, devido à semântica limpa. O mesmo efeito pode ser obtido de forma mais eficiente pelo uso inteligente do mapeamento de memória virtual e proteção de página por cópia na escrita.

13.4.3 Caches

Um **cache** é uma região da memória rápida que mantém cópias dos dados. O acesso à cópia em cache é mais eficiente do que o acesso aos dados originais. Por exemplo, as instruções do processo em execução são armazenadas em disco, colocadas em cache na memória física e copiadas novamente para os caches secundário e principal da CPU. A diferença entre um buffer e um cache é que um buffer pode manter a única cópia existente de um item de dados, enquanto um cache, por definição, mantém uma cópia, em armazenamento mais rápido, de um item que reside em outro lugar.

As funções que utilizam buffers e caches são distintas, mas, às vezes, uma região da memória pode ser usada para as duas finalidades. Por exemplo, para preservar a semântica de cópia e permitir o escalonamento eficiente da E/S em disco, o sistema operacional utiliza buffers na memória principal para manter os dados do disco. Esses buffers também são usados como um cache, para melhorar a eficiência da E/S para arquivos compartilhados pelas aplicações ou que estão sendo escritos e relidos rapidamente. Quando o kernel recebe uma requisição de E/S de arquivo, ele primeiro acessa o cache do buffer para ver se essa região do arquivo já está disponível na memória principal. Se estiver, uma E/S no disco físico pode ser evitada ou adiada. Além disso, as escritas em disco são acumuladas no cache do buffer por vários segundos, de modo que grandes transferências são reunidas para permitir escalonamentos de escrita eficientes. Essa estratégia de adiar as escritas para prover eficiência na E/S é discutida, no contexto do acesso a arquivo remoto, na [Seção 17.3](#).

13.4.4 Spooling e reserva de dispositivo

Um **spool** é um buffer que mantém a saída para um dispositivo, como uma impressora, que não pode aceitar fluxos de dados intercalados. Embora uma impressora só possa atender a uma única tarefa de cada vez, várias aplicações podem querer imprimir sua saída ao mesmo tempo, sem que ela fique misturada. O sistema operacional resolve esse problema interceptando toda a saída para a impressora. A saída de cada aplicação é mantida em um arquivo de disco separado. Quando uma aplicação termina de imprimir, o sistema de spooling envia o arquivo de spool correspondente para saída na impressora. O sistema de spooling copia os arquivos de spool enfileirados para a impressora, um de cada vez. Em alguns sistemas operacionais, o spooling é controlado por um processo daemon do sistema. Em outros sistemas operacionais, ele é manipulado por uma thread no kernel. De qualquer forma, o sistema operacional provê uma interface de controle que permite que usuários e administradores do sistema apresentem a fila, removam tarefas indesejadas antes de elas serem impressas, suspendam a impressão enquanto a impressora está trabalhando, e assim por diante.

Alguns dispositivos, como unidades de fita e impressoras, não podem multiplexar as requisições de E/S de forma útil para várias aplicações simultâneas. O spooling é a única maneira pela qual os sistemas operacionais podem coordenar a saída concorrente. Outra forma de lidar com o acesso concorrente ao dispositivo é prover facilidades explícitas para coordenação. Alguns sistemas operacionais (incluindo VMS) proveem suporte ao acesso exclusivo ao dispositivo, permitindo que um processo aloque um dispositivo ocioso e desaloque esse dispositivo quando não for mais necessário. Outros sistemas operacionais impõem um limite de um descritor de arquivo aberto para tal dispositivo. Muitos sistemas operacionais proveem funções que permitem aos processos coordenarem o acesso exclusivo entre eles. Por exemplo, o Windows NT provê chamadas de sistema para esperar até um objeto do dispositivo se tornar disponível. Ele também possui um parâmetro para a chamada de sistema `open()`, que declara os tipos de acesso a serem permitidos para outras threads concorrentes. Nesses sistemas, fica a critério das aplicações evitar o deadlock.

13.4.5 Tratamento de erros

Um sistema operacional que usa memória protegida pode se proteger contra muitos tipos de erros de hardware e da aplicação, de modo que uma falha completa do sistema não é o resultado comum de cada pequeno problema mecânico. Os dispositivos e as transferências de E/S podem falhar de muitas maneiras, seja por motivos passageiros, como quando uma rede fica sobrecarregada, seja por motivos “permanentes”, como quando um controlador de disco apresenta defeito. Os sistemas operacionais normalmente podem compensar as falhas passageiras de modo eficaz. Por exemplo, uma falha em um `read()` no disco resulta em uma nova tentativa de `read()`, e um erro em um `send()` na rede resulta em um `resend()`, se o protocolo especificar isso. Infelizmente, se um componente importante experimentar uma falha permanente, é provável que o sistema operacional não possa se recuperar.

Geralmente, uma chamada de sistema para E/S retornará 1 bit de informação sobre o status da chamada, sinalizando o sucesso ou a falha. No sistema operacional UNIX, uma variável inteira adicional, chamada `errno`, é usada para retornar um código de erro - um dentre aproximadamente 100 valores -, indicando a natureza geral da falha (por exemplo, argumento fora do intervalo, ponteiro incorreto ou arquivo não aberto). Por outro lado, alguns tipos de hardware podem prover informações de erro bastante detalhadas, embora muitos sistemas operacionais atuais não sejam projetados para transmitir essas informações a uma aplicação. Por exemplo, uma falha de um dispositivo SCSI é informada pelo protocolo SCSI em termos de uma **chave de detecção (sense key)** que identifica a natureza geral da falha, como um erro de hardware ou uma requisição ilegal; um **código de detecção adicional (additional sense code)**, que indica a categoria da falha, como um parâmetro de comando incorreto ou uma falha no autoteste; e um **qualificador de código de detecção adicional (additional sense code qualifier)**, que provê ainda mais detalhes, como qual parâmetro de comando está errado ou qual subsistema de hardware falhou no seu autoteste. Além disso, muitos dispositivos SCSI mantêm páginas internas de informações de log de erro, que podem ser requisitadas pelo host, mas isso raramente acontece.

13.4.6 Proteção da E/S

Os erros estão intimamente relacionados com a questão da proteção. Um programa do usuário pode atrapalhar, por acidente ou de propósito, a operação normal do sistema ao tentar emitir instruções de E/S ilegais. Podemos usar diversos mecanismos para garantir que esses problemas não ocorram no sistema.

Para evitar que os usuários realizem E/S ilegal, definimos todas as instruções de E/S como instruções privilegiadas. Assim, os usuários não podem emitir instruções de E/S diretamente; eles precisam fazer isso com o sistema operacional. Para realizar a E/S, um programa do usuário executa uma chamada do sistema para requisitar que o sistema operacional realize a E/S em seu favor ([Figura 13.11](#)). O sistema operacional, executando no modo monitor, verifica e certifica-se de que a requisição é válida e, se for, realiza a operação de E/S requisitada. O sistema operacional, em seguida, retorna ao usuário.

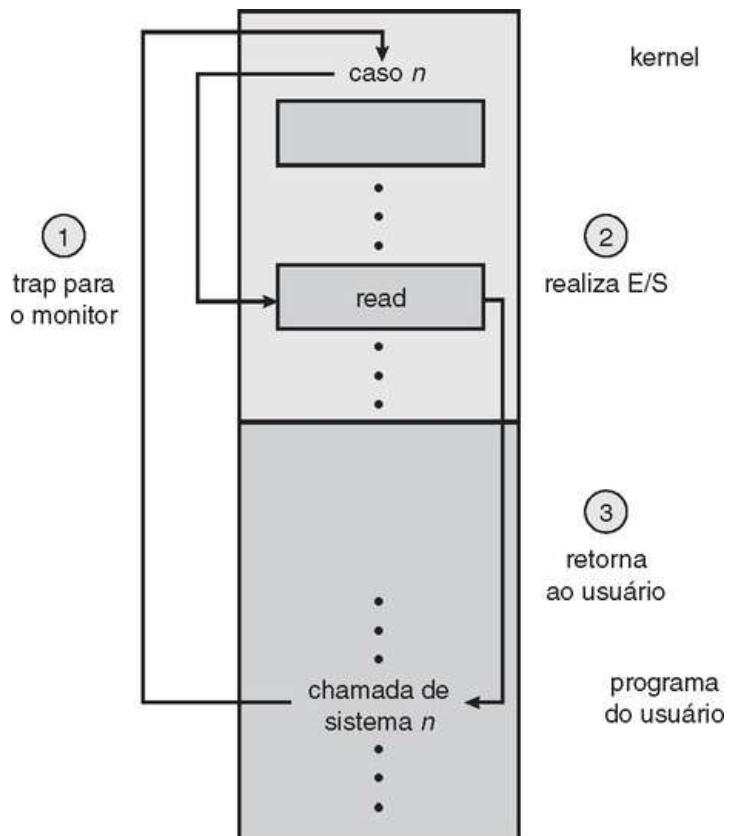


FIGURA 13.11 Uso de uma chamada de sistema para realizar E/S.

Além disso, quaisquer locais de memória mapeada ou de porta de E/S precisam ser protegidos contra acesso do usuário pelo sistema de proteção de memória. Observe que um kernel não pode negar todo acesso do usuário. A maioria dos jogos gráficos e software de edição e reprodução de vídeo precisa de acesso direto à memória do controlador gráfico mapeado na memória para agilizar

o desempenho dos gráficos, por exemplo. O kernel poderia, nesse caso, fornecer um mecanismo de lock para permitir que uma seção da memória gráfica (representando uma janela na tela) fosse alocada a um processo de cada vez.

13.4.7 Estruturas de dados do kernel

O kernel precisa manter informações de estado sobre o uso dos componentes de E/S. Ele faz isso por meio de diversas estruturas de dados no kernel, como a estrutura da tabela de arquivos abertos, da [Seção 11.1](#). O kernel usa muitas estruturas semelhantes para acompanhar as conexões de rede, comunicações de dispositivo de caractere e outras atividades de E/S.

O UNIX provê acesso do sistema de arquivos a uma série de entidades, como arquivos do usuário, dispositivos brutos e os espaços de endereços dos processos. Embora cada uma dessas entidades admita uma operação `read()`, a semântica é diferente. Por exemplo, para ler um arquivo do usuário, o kernel precisa sondar o cache do buffer antes de decidir se realizará uma E/S em disco. Para ler um disco bruto, o kernel precisa garantir que o tamanho da requisição tem um múltiplo do tamanho de setor do disco e está alinhado em um limite de setor. Para ler uma imagem do processo, basta copiar dados da memória. O UNIX engloba essas diferenças dentro de uma estrutura uniforme usando uma técnica orientada a objeto. O registro de arquivos abertos, mostrado na [Figura 13.12](#), contém uma tabela de despacho que mantém ponteiros para as rotinas apropriadas, dependendo do tipo de arquivo.

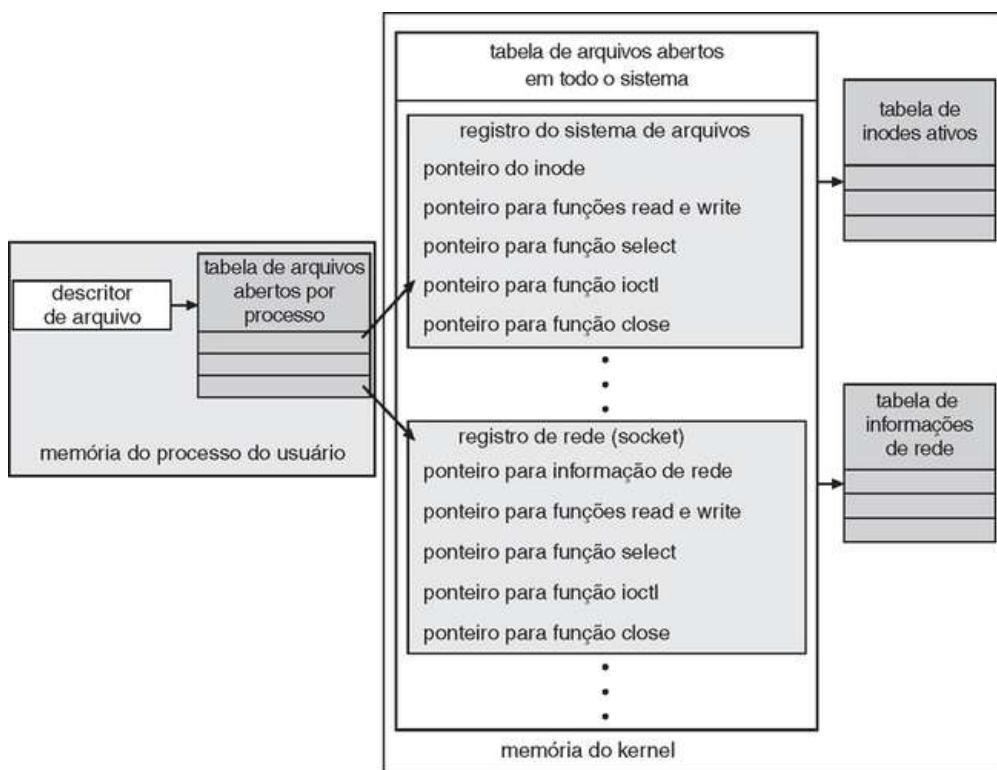


FIGURA 13.12 Estrutura do kernel de E/S do UNIX.

Alguns sistemas operacionais utilizam métodos orientados a objeto de forma ainda mais extensa. Por exemplo, o Windows NT utiliza uma implementação de troca de mensagens para a E/S. Uma requisição de E/S é convertida em uma mensagem enviada por meio do kernel para o gerenciador de E/S e depois para o driver de dispositivo, cada um podendo alterar o conteúdo da mensagem. Para a saída, a mensagem contém os dados a serem escritos. Para a entrada, a mensagem contém um buffer para receber os dados. A técnica de troca de mensagens pode aumentar o custo adicional, por comparação com as técnicas de procedimento que utilizam estruturas de dados compartilhadas, mas simplifica a estrutura e o projeto do sistema de E/S, aumentando a flexibilidade.

13.4.8 Resumo do subsistema de E/S no kernel

Resumindo, o subsistema de E/S coordena uma extensa coleção de serviços disponíveis às aplicações e a outras partes do kernel. O subsistema de E/S supervisiona esses procedimentos:

- Gerenciamento do espaço de nomes para arquivos e dispositivos.
- Controle de acesso a arquivos e dispositivos.

- Controle de operação [por exemplo, um modem não pode usar `seek()`].
 - Alocação de espaço no sistema de arquivos.
 - Alocação de dispositivo.
 - Buffers, caches e spooling.
 - Escalonamento de E/S.
 - Monitoramento de status do dispositivo, tratamento de erro e recuperação de falhas.
 - Configuração e inicialização de driver de dispositivo.
- Os níveis mais altos do subsistema de E/S acessam dispositivos por meio da interface uniforme fornecida pelos drivers de dispositivo.

13.5 Transformando requisições de E/S em operações de hardware

Já descrevemos o handshaking entre um driver de dispositivo e um controlador de dispositivo, mas não explicamos como o sistema operacional conecta uma requisição da aplicação a um conjunto de fios da rede ou a um setor de disco específico. Vamos considerar, por exemplo, a leitura de um arquivo do disco. A aplicação refere-se aos dados por um nome de arquivo. Dentro de um disco, o sistema de arquivos é mapeado do nome de arquivo, por meio dos diretórios do sistema de arquivos, para obter a alocação de espaço do arquivo. Por exemplo, no MS-DOS, o nome é associado a um número que indica uma entrada na tabela de alocação de arquivos, e essa entrada da tabela informa quais blocos de disco estão alocados ao arquivo. No UNIX, o nome está associado a um número de inode, e o inode correspondente contém a informação de alocação de espaço. Mas como é feita a conexão entre o nome do arquivo e o controlador de disco (o endereço da porta de hardware ou os registradores do controlador mapeados na memória)?

Um método é aquele usado pelo MS-DOS, um sistema operacional relativamente simples. A primeira parte de um nome de arquivo no MS-DOS, antes do sinal de dois-pontos, é uma letra que identifica o dispositivo de hardware específico. Por exemplo, *c:* é a primeira parte de cada nome de arquivo no disco rígido principal. O fato de *c:* representar o disco rígido principal está embutido no sistema operacional; *c:* é mapeado para um endereço de porta específico por meio de uma tabela de dispositivos. Devido ao separador dois-pontos, o espaço do nome de dispositivo é separado do espaço de nomes do sistema de arquivos. Essa separação facilita para o sistema operacional a associação de funcionalidade extra a cada dispositivo. Por exemplo, é fácil invocar o spooling sobre quaisquer arquivos escritos para a impressora.

Se, em vez disso, o espaço de nomes do dispositivo estiver incorporado no espaço de nomes normal do sistema de arquivos, como acontece no UNIX, os serviços normais de nomes do sistema de arquivos são fornecidos automaticamente. Se o sistema de arquivos prover controle de proprietário e acesso a todos os nomes de arquivo, então os dispositivos possuem controle de proprietários e acesso. Como os arquivos são armazenados em dispositivos, essa interface provê acesso ao sistema de E/S em dois níveis. Os nomes podem ser usados para acessar os próprios dispositivos ou para acessar os arquivos armazenados nos dispositivos.

O UNIX representa os nomes de dispositivo no espaço de nomes normal do sistema de arquivos. Ao contrário de um nome de arquivo do MS-DOS, que tem o separador de dois-pontos, um nome de caminho é o nome de um dispositivo. O UNIX tem uma **tabela de montagem** que associa prefixos de nomes de caminho a nomes de dispositivo específicos. Para resolver um nome de caminho, o UNIX pesquisa o nome na tabela de montagem para encontrar o maior prefixo que combina; a entrada correspondente na tabela de montagem provê o nome de dispositivo. Esse nome de dispositivo também tem a forma de um nome no espaço de nomes do sistema de arquivos. Quando o UNIX pesquisa esse nome nas estruturas de diretório do sistema de arquivos, ele não encontra um número de inode, mas um número de dispositivo *<principal, secundário>*. O número de dispositivo principal identifica um driver de dispositivo que deve ser chamado para tratar da E/S nesse dispositivo. O número de dispositivo secundário é passado ao driver de dispositivo para indexar em uma tabela de dispositivo. A entrada da tabela de dispositivo correspondente dá o endereço de porta ou o endereço mapeado na memória do controlador de dispositivo.

Os sistemas operacionais modernos conseguem uma flexibilidade significativa a partir dos vários estágios das tabelas de pesquisa no caminho entre uma requisição e um controlador de dispositivo físico. Os mecanismos que passam requisições entre as aplicações e os drivers são genéricos. Assim, podemos introduzir novos dispositivos e drivers em um computador sem ter de recompilar o kernel. Na verdade, alguns sistemas operacionais têm a capacidade de carregar drivers de dispositivo por demanda. No momento do boot, o sistema primeiro sonda os barramentos do hardware para determinar quais dispositivos estão presentes, então carrega os drivers necessários, imediatamente ou quando forem requisitados pela primeira vez por uma requisição de E/S.

Em seguida, descrevemos o ciclo de vida típico de uma requisição de leitura bloqueante, conforme representada na [Figura 13.13](#). A figura sugere que uma operação de E/S exige muitas etapas que, juntas, consomem uma quantidade enorme de ciclos de CPU.

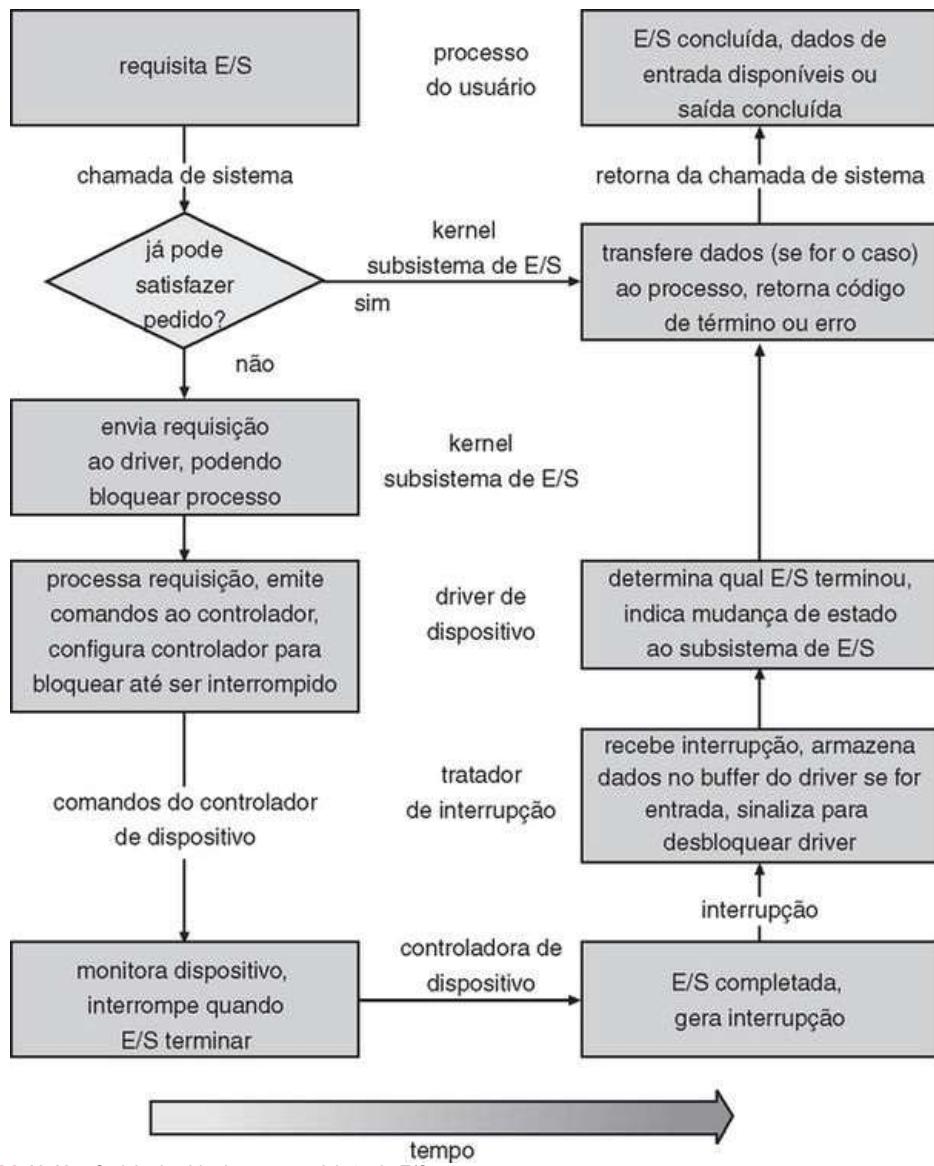


FIGURA 13.13 O ciclo de vida de uma requisição de E/S.

1. Um processo emite uma chamada de sistema `read()` bloqueante para o descritor de um arquivo aberto (`open`) anteriormente.
2. O código da chamada de sistema no kernel verifica se os parâmetros estão corretos. No caso de entrada, se os dados já estiverem disponíveis no cache do buffer, eles serão retornados ao processo e a requisição de E/S é completada.
3. Caso contrário, uma E/S física precisa ser realizada. O processo é removido da fila de execução e colocado na fila de espera para o dispositivo, onde a requisição de E/S é escalonada. Mais cedo ou mais tarde, o subsistema de E/S envia a requisição para o driver de dispositivo. Dependendo do sistema operacional, a requisição é enviada por uma chamada de sub-rotina ou por uma mensagem no kernel.
4. O driver de dispositivo aloca espaço de buffer no kernel para receber os dados e escalona a E/S. A qualquer momento, o driver envia comandos para o controlador de dispositivo escrevendo nos registradores de controle do dispositivo.
5. O controlador do dispositivo opera sobre o hardware do dispositivo para realizar a transferência de dados.
6. O driver pode consultar status e dados ou então pode ter configurado uma transferência de DMA para a memória do kernel. Consideramos que a transferência é gerenciada por um controlador de DMA, que gera uma interrupção quando a transferência termina.
7. O tratador de interrupção correto recebe a interrupção por meio da tabela do vetor de interrupção, armazena quaisquer dados necessários, sinaliza o driver de dispositivo e retorna da interrupção.
8. O driver de dispositivo recebe o sinal, determina qual requisição de E/S foi completada, determina o status da requisição e sinaliza ao subsistema de E/S do kernel que a requisição foi concluída.

9. O kernel transfere dados ou retorna códigos ao espaço de endereço do processo requisitante, movendo o processo da fila de espera para a fila de prontos.
10. A mudança do processo para a fila de prontos desbloqueia o processo. Quando o escalonador atribui o processo à CPU, o processo retoma a execução no final da chamada de sistema.

13.6 STREAMS

O UNIX System V possui um mecanismo interessante, chamado **STREAMS**, que permite que uma aplicação monte canalizações de código de driver dinamicamente. Um fluxo (stream) é uma conexão full-duplex entre um driver de dispositivo e um processo no nível de usuário. Ele consiste em uma **cabeça de fluxo (stream head)**, que realiza a interface com o processo do usuário, uma **extremidade de driver**, que controla o dispositivo, e zero ou mais **módulos de fluxo** entre a cabeça de fluxo e a extremidade de driver. Cada um desses componentes contém um par de filas - uma fila de leitura e uma fila de escrita. A troca de mensagens é usada para transferir dados entre as filas. A estrutura STREAMS aparece na [Figura 13.14](#).

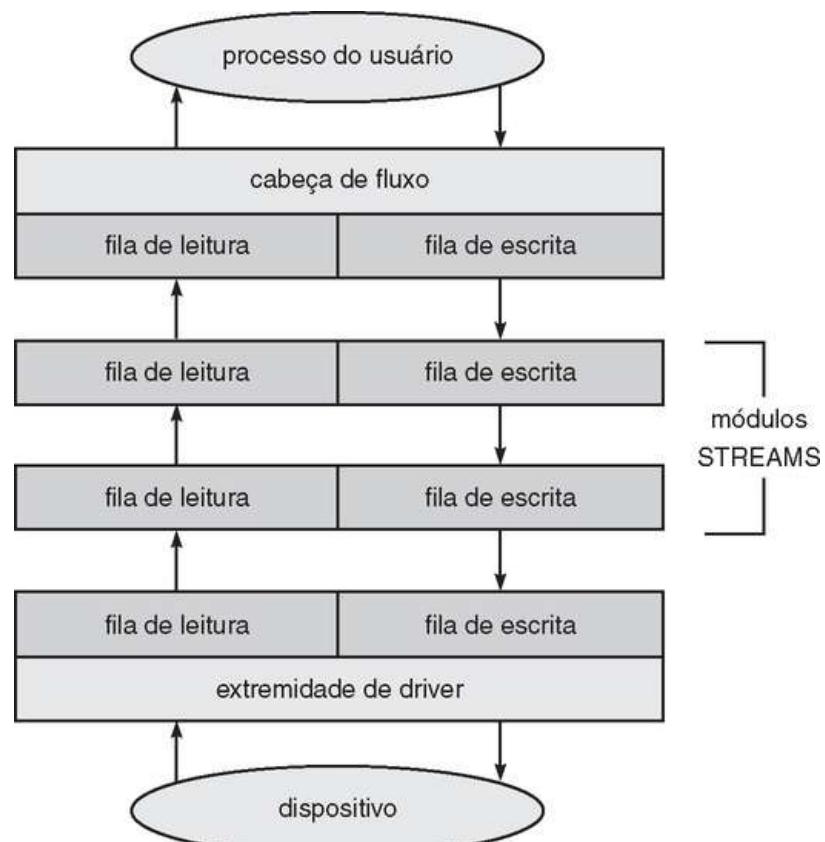


FIGURA 13.14 A estrutura STREAMS.

Os módulos proveem a funcionalidade do processamento STREAMS e são *empurrados* para um fluxo usando a chamada de sistema `ioctl()`. Por exemplo, um processo pode abrir um dispositivo de porta serial por meio de um fluxo e pode empurrar um módulo para tratar da edição da entrada. Como as mensagens são trocadas entre as filas em módulos adjacentes, uma fila em um módulo pode estourar uma fila adjacente. Para evitar que isso aconteça, uma fila pode admitir controle de fluxo. Sem controle de fluxo, uma fila aceita todas as mensagens e as envia imediatamente para a fila no módulo adjacente, sem colocá-las em buffer. A fila que tem suporte para controle de fluxo coloca as mensagens em buffer e não aceita mensagens sem haver espaço suficiente no buffer; esse processo envolve a troca de mensagens de controle entre as filas nos módulos adjacentes.

Um processo usuário escreve dados em um dispositivo usando as chamadas de sistema `write()` ou `putmsg()`. A chamada de sistema `write()` escreve dados brutos no fluxo, enquanto `putmsg()` permite que o processo do usuário especifique uma mensagem. Independentemente da chamada de sistema usada pelo processo do usuário, a cabeça do fluxo copia os dados para uma mensagem e a entrega à fila para o próximo módulo em sequência. Essa cópia de mensagens continua até a mensagem ser copiada para a extremidade do driver e daí ao dispositivo. Da mesma forma, o processo do usuário lê dados da cabeça de fluxo usando as chamadas de sistema `read()` ou `getmsg()`. Se `read()` for usada, a cabeça de fluxo apanha uma mensagem de sua fila adjacente e retorna dados comuns (um fluxo de bytes não estruturado) ao processo. Se `getmsg()` for usada, uma mensagem é retornada ao processo.

A E/S do STREAMS é assíncrona (ou não bloqueante), com a exceção de quando o processo do usuário se comunica com a cabeça de fluxo. Ao escrever no fluxo, o processo do usuário bloqueará, supondo que a próxima fila usa controle de fluxo, até haver espaço para copiar a mensagem. De

modo semelhante, o processo do usuário bloqueará quando estiver lendo do fluxo até os dados estarem disponíveis.

Como dissemos, a extremidade de driver - como a cabeça de fluxo e os módulos - possui uma fila de leitura e escrita. Todavia, a extremidade de driver precisa responder a interrupções, como aquela disparada quando um quadro está pronto para ser lido de uma rede. Ao contrário da cabeça de fluxo, que pode bloquear se não puder copiar uma mensagem para a fila seguinte em linha, a extremidade de driver precisa tratar de todos os dados que chegam. Os drivers também precisam admitir controle de fluxo. Entretanto, se o buffer de um dispositivo estiver cheio, um dispositivo costuma descartar as mensagens que chegam. Considere uma placa de rede cujo buffer de entrada esteja cheio. A placa de rede deve descartar quaisquer outras mensagens até haver espaço no buffer para armazenar as mensagens que chegam.

O benefício do uso do STREAMS é que ele provê uma estrutura para uma técnica modular e incremental de escrever drivers de dispositivos e protocolos de rede. Os módulos podem ser usados por diferentes STREAMS e, em virtude disso, por diferentes dispositivos. Por exemplo, um módulo de rede pode ser usado por uma placa de rede Ethernet e por uma placa de rede sem fio 802.11. Além do mais, em vez de tratar a E/S de dispositivo de caractere como um fluxo de bytes não estruturado, o STREAMS permite o suporte para limites de mensagem e informações de controle na comunicação entre os módulos. O suporte para o STREAMS está espalhado entre a maioria das variantes do UNIX, e esse é o método preferido para escrever protocolos e drivers de dispositivos. Por exemplo, System V UNIX e Solaris implementam o mecanismo de socket por meio do STREAMS.

13.7 Desempenho

A E/S é um fator importante no desempenho do sistema. Ela impõe demandas pesadas sobre a CPU para executar o código do driver de dispositivo e escalar processos de modo justo e eficiente enquanto bloqueiam e desbloqueiam. As trocas de contexto resultantes pressionam a CPU e seus caches de hardware. A E/S também expõe quaisquer ineficiências nos mecanismos do tratamento de interrupção no kernel. Além disso, a E/S carrega o barramento da memória durante a cópia de dados entre os controladores e a memória física e novamente durante as cópias entre os buffers do kernel e o espaço de dados da aplicação. O tratamento controlado de todas essas demandas é uma das principais preocupações de uma arquitetura.

Embora os computadores modernos possam lidar com muitos milhares de interrupções por segundo, o tratamento de interrupção é uma tarefa relativamente dispendiosa: cada interrupção faz o sistema realizar uma mudança de estado, executar o tratador de interrupção e depois restaurar o estado. A E/S programada pode ser mais eficiente do que a E/S controlada por interrupção, se o número de ciclos gastos na espera ocupada não for excessivo. Um término de E/S normalmente desbloqueia um processo, levando ao custo adicional total de uma troca de contexto.

O tráfego da rede também pode causar uma alta taxa de troca de contexto. Considere, por exemplo, um login remoto de uma máquina para outra. Cada caractere digitado na máquina local precisa ser transportado para a máquina remota. Na máquina local, o caractere é digitado, uma interrupção do teclado é gerada e o caractere é passado pelo tratador de interrupção até o driver de dispositivo, para o kernel e depois para o processo do usuário. O processo do usuário emite uma chamada de sistema para E/S de rede, a fim de enviar o caractere para a máquina remota. O caractere, em seguida, flui para o kernel local, passa pelas camadas de rede que montam um pacote de rede, e depois para o driver de dispositivo de rede. O driver de dispositivo de rede transfere o pacote para o controlador de rede, que envia o caractere e gera uma interrupção. A interrupção é passada de volta para o kernel para terminar a chamada de sistema de E/S de rede.

Agora, o hardware de rede do sistema remoto recebe o pacote e uma interrupção é gerada. O caractere é desempacotado dos protocolos de rede e entregue ao daemon de rede apropriado. O daemon de rede identifica qual sessão de login remoto está envolvida e passa o pacote para o subdaemon apropriado para essa sessão. No decorrer desse fluxo existem trocas de contexto e trocas de estado ([Figura 13.15](#)). Normalmente, o receptor ecoa o caractere de volta para o emissor; essa técnica duplica o trabalho.

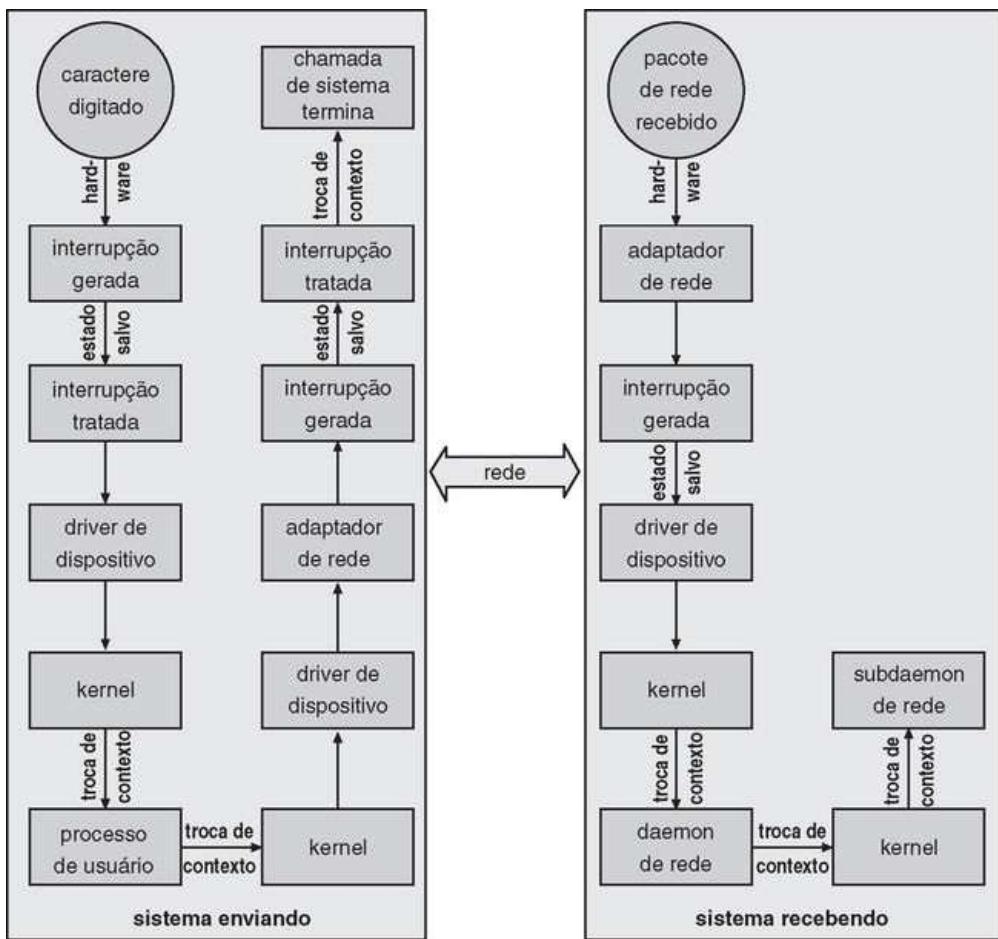


FIGURA 13.15 Comunicações entre computadores.

Os desenvolvedores do Solaris reimplementaram o daemon **telnet** usando threads no kernel para eliminar as trocas de contexto envolvidas na movimentação de cada caractere entre os daemons e o kernel. A Sun estima que essa melhoria aumentou a quantidade máxima de logins de rede de cem para mil em um servidor grande.

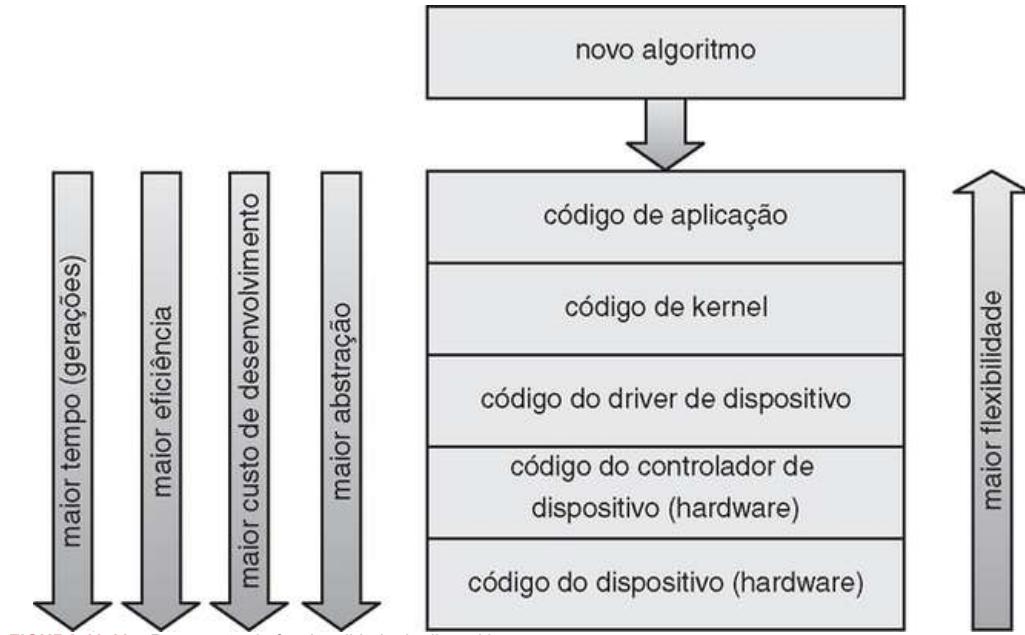


FIGURA 13.16 Progressão da funcionalidade do dispositivo.

Outros sistemas utilizam **processadores front-end** separados para E/S de terminal, a fim de reduzir a carga da interrupção na CPU principal. Por exemplo, um **concentrador de terminais**

pode multiplexar o tráfego de centenas de terminais remotos em uma única porta de um computador grande. Um **canal de E/S** é uma CPU dedicada, de uso especial, encontrada em computadores de grande porte (mainframes) e em outros sistemas de alto nível. A tarefa de um canal é desafogar o trabalho de E/S da CPU principal. A ideia é que os canais mantenham os dados fluindo tranquilamente, enquanto a CPU principal permanece livre para processar os dados. Assim como os controladores de dispositivos e os controladores de DMA, encontrados em computadores menores, um canal pode processar programas mais genéricos e sofisticados, de modo que os canais podem ser ajustados para cargas de trabalho específicas.

Podemos empregar vários princípios para melhorar a eficiência da E/S:

- Reduzir a quantidade de trocas de contexto.
- Reduzir o número de vezes que os dados precisam ser copiados na memória enquanto passam entre dispositivo e aplicação.
- Reduzir a frequência das interrupções usando transferências grandes, controladores inteligentes e polling - se a espera ocupada puder ser reduzida.
- Aumentar a concorrência, usando controladores que conhecem DMA, ou canais, para tirar da CPU o trabalho de simples cópia de dados.
- Mover o processamento de primitivas para o hardware, permitindo sua operação nos controladores de dispositivos simultânea com a operação da CPU e do barramento.
- Balancear o desempenho da CPU, subsistema de memória, barramento e E/S, pois uma sobrecarga em qualquer um causará ociosidade nos outros.

Os dispositivos de E/S variam muito em complexidade. Por exemplo, um mouse é simples. Os movimentos do mouse e os cliques dos botões são convertidos em valores numéricos passados do hardware, atravessando o driver de dispositivo do mouse, até a aplicação. Por outro lado, a funcionalidade oferecida pelo driver de dispositivo de disco do NT é complexa. Ele não apenas gerencia os discos individuais, mas também implementa arrays RAID ([Seção 12.7](#)). Para isso, ele converte a requisição de leitura ou escrita de uma aplicação em um conjunto coordenado de operações de E/S em disco. Além do mais, implementa sofisticados algoritmos de tratamento de erros e recuperação de dados, realizando muitas ações para otimizar o desempenho do disco.

Onde a funcionalidade da E/S precisa ser implementada: no hardware do dispositivo, no driver de dispositivo ou no software da aplicação? Às vezes, observamos a progressão representada na [Figura 13.16](#).

- Inicialmente, implementamos algoritmos de E/S experimentais no nível da aplicação, pois o código da aplicação é flexível, e os bugs da aplicação provavelmente não causarão falhas no sistema. Além do mais, desenvolvendo código no nível da aplicação, evitamos a necessidade de reiniciar ou recarregar drivers de dispositivos após cada mudança no código. Entretanto, uma implementação no nível da aplicação pode ser ineficaz, devido ao custo adicional das trocas de contexto e porque a aplicação não pode tirar proveito das estruturas de dados internas do kernel e da funcionalidade do kernel (como o uso eficiente de mensagens, threads e locks no kernel).
- Quando um algoritmo no nível da aplicação tiver demonstrado seu valor, podemos reimplementá-lo no kernel. Isso pode melhorar o desempenho, mas o esforço de desenvolvimento é mais desafiador, pois o kernel de um sistema operacional é um sistema de software grande e complexo. Além do mais, uma implementação no kernel precisa ser completamente depurada para evitar adulteração de dados e falhas no sistema.
- O desempenho mais alto pode ser obtido por uma implementação especializada no hardware, seja no dispositivo ou no seu controlador. As desvantagens de uma implementação no hardware incluem a dificuldade e o gasto para fazer melhorias ou consertar bugs, o maior tempo de desenvolvimento (meses em vez de dias) e a menor flexibilidade. Por exemplo, um controlador RAID no hardware pode não prover meio algum para o kernel influenciar a ordem ou o local das leituras e escritas de blocos individuais, mesmo que o kernel tenha informações especiais sobre a carga de trabalho que permitiriam que o kernel melhorasse o desempenho da E/S.

13.8 Resumo

Os elementos de hardware básicos envolvidos na E/S são barramentos, controladores de dispositivos e os próprios dispositivos. O trabalho de mover dados entre os dispositivos e a memória principal é realizado pela CPU como E/S programada ou é deixado para um controlador de DMA. O módulo do kernel que controla um dispositivo é um driver de dispositivo. A interface de chamada de sistema fornecida para as aplicações é projetada para lidar com várias categorias básicas de hardware, incluindo dispositivos de bloco, dispositivos de caractere, arquivos mapeados na memória, sockets de rede e temporizadores de intervalo programados. As chamadas de sistema normalmente bloqueiam o processo que as emite, mas as chamadas não bloqueantes e assíncronas são usadas pelo próprio kernel e também por aplicações que não podem dormir enquanto esperam que uma operação de E/S termine.

O subsistema de E/S do kernel provê diversos serviços. Entre eles estão o escalonamento de E/S, buffers, caches, spooling, reserva de dispositivo e tratamento de erro. Outro serviço é a tradução de nomes, para fazer a conexão entre dispositivos de hardware e os nomes de arquivo simbólicos usados pelas aplicações. Isso envolve vários níveis de mapeamento que traduzem um nome em sequência de caracteres para drivers de dispositivo específico e endereços de dispositivo e depois para endereços físicos das portas de E/S ou controladores de barramento. Esse mapeamento pode ocorrer dentro do espaço de nomes do sistema de arquivos, como no UNIX, ou então em um espaço separado para nomes de dispositivos, como no MS-DOS.

O STREAMS é uma implementação e metodologia que provê uma estrutura para um enfoque modular e incremental para a escrita de drivers de dispositivos e protocolos de rede. Por meio de fluxos, os drivers podem ser empilhados, com os dados passando por eles sequencial e bidirecionalmente, para o processamento.

As chamadas de sistema de E/S são dispendiosas em termos de consumo de CPU, devido às muitas camadas de software entre um dispositivo físico e a aplicação. Essas camadas implicam custos adicionais de várias origens: a troca de contexto para atravessar a fronteira de proteção do kernel, o tratamento de sinal e interrupção, a fim de atender os dispositivos de E/S, e a carga na CPU e no sistema de memória, a fim de copiar dados entre os buffers do kernel e o espaço da aplicação.

Exercícios práticos

- 13.1. Cite três vantagens de colocar funcionalidade em um controlador de dispositivo, em vez de no kernel. Cite três desvantagens.
- 13.2. O exemplo de handshaking na [Seção 13.2](#) usou dois bits: um bit *ocupado* e um bit *comando pronto*. É possível implementar esse handshaking com apenas um bit? Se for, descreva o protocolo. Se não for, explique por que um bit não é suficiente.
- 13.3. Por que um sistema poderia usar a E/S controlada por interrupção para gerenciar uma única porta serial e o polling da E/S para gerenciar um processador front-end, como um concentrador de terminais?
- 13.4. O polling para um término de E/S pode desperdiçar um grande número de ciclos de CPU se o processador repetir um loop de espera ocupada muitas vezes antes que a E/S termine. Mas se o dispositivo de E/S estiver pronto para atendimento, o polling poderá ser muito mais eficiente do que o caching e o despacho de uma interrupção. Descreva uma estratégia híbrida que combine realizar polling, interrupções e dormir para o atendimento do dispositivo de E/S. Para cada uma dessas três estratégias (polling puro, interrupções puras, híbrida), descreva um ambiente de computação no qual essa estratégia é mais eficiente do que as outras duas.
- 13.5. Como o DMA aumenta a concorrência do sistema? De que forma ele complica o projeto do hardware?
- 13.6. Por que é importante expandir o barramento do sistema e as velocidades de dispositivo à medida que a velocidade da CPU aumenta?
- 13.7. Faça a distinção entre um driver STREAMS e um módulo STREAMS.

Exercícios

- 13.8. Quando múltiplas interrupções de diferentes dispositivos aparecem quase ao mesmo tempo, um esquema de prioridade poderia ser usado para determinar a ordem em que as interrupções serão atendidas. Discuta quais questões precisam ser consideradas na atribuição de prioridades a diferentes interrupções.
- 13.9. Quais são as vantagens e desvantagens do suporte à E/S mapeada na memória para os registradores de controle de dispositivo?
- 13.10. Considere os seguintes cenários de E/S em um PC monousuário.
- Mouse usado com interface gráfica do usuário
 - Unidade de fita em um sistema operacional multitarefa (considere que nenhuma pré-alocação de dispositivo está disponível)
 - Unidade de disco contendo arquivos do usuário
 - Placa gráfica com conexão direta do barramento, acessível por meio da E/S mapeada na memória
- Para cada um desses cenários de E/S, você projetaria o sistema operacional para usar buffers, spooling, caches ou uma combinação deles? Você usaria E/S por polling ou E/S controlada por interrupção? Dê os motivos para as suas escolhas.
- 13.11. Na maioria dos sistemas multiprogramados, os programas do usuário acessam a memória por meio de endereços virtuais, enquanto o sistema operacional utiliza endereços físicos brutos para acessar a memória. Quais são as implicações desse projeto para o início das operações de E/S pelo programa do usuário e sua execução pelo sistema operacional?
- 13.12. Quais são os diversos tipos de overheads de desempenho associados ao atendimento de uma interrupção?
- 13.13. Descreva três circunstâncias sob as quais a E/S bloqueante seria utilizada. Descreva três circunstâncias sob as quais a E/S não bloqueante seria usada. Por que não apenas implementar a E/S não bloqueante e ter processos em espera ocupada (busy wait) até que seus dispositivos estejam prontos?
- 13.14. Normalmente, ao término de uma E/S de dispositivo, uma única interrupção é gerada e tratada de modo apropriado pelo processador host. Em certos ambientes, porém, o código que deve ser executado no término da E/S pode ser desmembrado em partes separadas. A primeira executa imediatamente após o término da E/S e escalona uma segunda interrupção para a parte do código restante, a ser executada em outro momento. Qual é a finalidade de usar essa estratégia no projeto de tratadores de interrupção?
- 13.15. Alguns controladores de DMA admitem o acesso à memória virtual, onde os destinos das operações de E/S são especificados como endereços virtuais e uma tradução de endereço virtual para físico é realizada durante o DMA. Como esse projeto complica o projeto do controlador de DMA? Quais são as vantagens de prover tal funcionalidade?
- 13.16. O UNIX coordena as atividades dos componentes de E/S do kernel manipulando as estruturas de dados compartilhadas no kernel, enquanto o Windows NT utiliza a passagem de mensagens orientada a objeto entre os componentes de E/S do kernel. Discuta três prós e três contras de cada técnica.
- 13.17. Escreva (em pseudocódigo) uma implementação dos relógios virtuais, incluindo o enfileiramento e o gerenciamento das requisições de temporizador para o kernel e as aplicações. Considere que o hardware provê três canais de temporizador.
- 13.18. Discuta as vantagens e as desvantagens de garantir a transferência confiável de dados entre os módulos na abstração STREAMS.

Notas bibliográficas

[Vahalia \[1996\]](#) provê uma boa introdução da E/S e redes no UNIX. [Leffler e outros \[1989\]](#) detalham as estruturas de E/S e os métodos empregados no BSD UNIX. [Milenkovic \[1987\]](#) discute a complexidade dos métodos de E/S e sua implementação. O uso e a programação dos diversos protocolos de comunicação entre processos e protocolos de rede no UNIX são explorados em [Stevens \[1992\]](#). [Brain \[1996\]](#) documenta a interface de aplicação do Windows NT. A implementação de E/S no MINIX OS é descrita em [Tanenbaum e Woodhull \[1997\]](#). [Custer \[1994\]](#) inclui informações detalhadas sobre a implementação da E/S por troca de mensagens no NT.

Para obter os detalhes do tratamento de E/S e funcionalidade do mapeamento de memória no nível de hardware, os manuais de referência de processador ([Motorola \[1993\]](#) e [Intel \[1993\]](#)) estão entre as melhores fontes. Hennessy e Patterson [2002] descrevem os sistemas multiprocessados e as questões de coerência do cache. [Tanenbaum \[1990\]](#) descreve o projeto de E/S do hardware em baixo nível, e [Sargent e Shoemaker \[1995\]](#) proveem um guia do programador para o hardware e software do PC em baixo nível. O mapa de endereços de E/S de dispositivo no IBM PC é apresentado em [IBM \[1983\]](#). A edição de março de 1994 da *IEEE Computer* é dedicada a hardware e software de E/S. [Rago \[1993\]](#) provê uma boa discussão sobre STREAMS.

PARTE V

PROTEÇÃO E SEGURANÇA

ESBOÇO

[Capítulo 18: Introdução a Proteção e segurança](#)

[Capítulo 19: Proteção](#)

[Capítulo 20: Segurança](#)

Introdução a Proteção e segurança

Os mecanismos de proteção proveem acesso controlado a um sistema, limitando os tipos de acesso a arquivos permitidos aos usuários. Além disso, a proteção precisa garantir que somente os processos com autorização apropriada do sistema operacional possam operar sobre segmentos de memória, CPU e outros recursos.

A proteção é provida por um mecanismo que controla o acesso dos programas, processos ou usuários aos recursos definidos por um sistema computadorizado. Esse mecanismo precisa prover um meio para a especificação dos controles a serem impostos, juntamente com um meio de execução.

A segurança garante a autenticação dos usuários do sistema para proteger a integridade das informações armazenadas no sistema (tanto dados quanto código), além dos recursos físicos do sistema computadorizado. O sistema de segurança impede o acesso não autorizado a um sistema, destruição maliciosa ou alteração de dados, além da introdução accidental da inconsistência.

CAPÍTULO 14

Proteção

Os processos em um sistema operacional precisam ser protegidos contra as atividades uns dos outros. Para oferecer essa proteção, podem usar diversos mecanismos para garantir que somente os processos que obtiveram a devida autorização do sistema operacional possam operar em arquivos, segmentos de memória, CPU e outros recursos de um sistema.

Proteção refere-se a um mecanismo para controlar o acesso de programas, processos ou usuários aos recursos definidos por um sistema computadorizado. Esse mecanismo precisa prover um meio para especificação dos controles a serem impostos, junto com algum meio de execução. Fazemos uma distinção entre proteção e segurança, que é uma medida de confiança de que a integridade de um sistema e seus dados serão preservados. A garantia de segurança é um tópico muito mais amplo do que a proteção, e trataremos dela no [Capítulo 15](#).

OBJETIVOS DO CAPÍTULO

- Discutir os objetivos e princípios de proteção em um sistema computadorizado moderno.
- Explicar como os domínios de proteção, combinados com uma matriz de acesso, são usados para especificar os recursos que um processo pode acessar.
- Examinar sistemas de proteção baseados em capacidade e linguagem.

14.1 Objetivos da proteção

À medida que os sistemas computadorizados se tornaram mais sofisticados e difusos em suas aplicações, a necessidade de proteger sua integridade também cresceu. A proteção foi concebida originalmente como um auxiliar para os sistemas operacionais multiprogramados, de modo que usuários não confiáveis pudessem seguramente compartilhar um espaço de nomes lógicos comum, como um diretório de arquivos, ou compartilhar um espaço de nomes físicos comum, como a memória. Os conceitos modernos de proteção evoluíram para aumentar a confiabilidade de qualquer sistema complexo que utilize recursos compartilhados.

Precisamos prover proteção por vários motivos. O mais óbvio é a necessidade de impedir a violação maldosa e intencional de uma restrição de acesso por um usuário. No entanto, de importância mais geral é a necessidade de garantir que cada componente de programa ativo em um sistema só utilize recursos do sistema de maneiras consistentes com as políticas determinadas. Esse requisito é fundamental para um sistema confiável.

A proteção pode melhorar a confiabilidade, detectando erros latentes nas interfaces entre os subsistemas componentes. A detecção antecipada de erros da interface pode impedir a contaminação de um subsistema saudável por um subsistema defeituoso. Além disso, um recurso não protegido não pode se proteger contra uso (ou mau uso) por um usuário não autorizado ou incompetente. Um sistema orientado à proteção provê meios de distinguir entre o uso autorizado e o não autorizado.

O papel da proteção em um sistema computadorizado é prover um mecanismo para a execução das políticas que governam o uso do recurso. Essas políticas podem ser estabelecidas de diversas maneiras. Algumas são fixas no projeto do sistema, outras são formuladas pelo gerenciamento de um sistema. Outras ainda são definidas pelos usuários individuais para proteger seus próprios arquivos e programas. Um sistema de proteção precisa ter a flexibilidade de executar uma série de políticas.

As políticas para uso de recursos podem variar por aplicação e podem mudar com o tempo. Por esses motivos, a proteção não é mais a preocupação exclusiva do projetista de um sistema operacional. O programador de aplicação também precisa usar mecanismos de proteção para proteger os recursos criados e admitidos por um subsistema de aplicação contra o uso indevido. Neste capítulo, descrevemos os mecanismos de proteção que o sistema operacional precisa prover para que um projetista de aplicação possa usá-los no projeto do seu próprio software de proteção.

Observe que *mecanismos* é diferente de *políticas*. Os mecanismos determinam *como* algo será feito; as políticas decidem *o que* será feito. A separação entre política e mecanismo é importante por flexibilidade. As políticas provavelmente mudam de um lugar para outro ou de um momento para outro. No pior dos casos, cada mudança na política exigiria uma mudança no mecanismo subjacente. O uso de mecanismos gerais possibilita que evitemos essa situação.

14.2 Princípios de proteção

Frequentemente, um princípio orientador pode ser usado por todo um projeto, como o projeto de um sistema operacional. Seguir esse princípio simplifica as decisões de projeto e mantém o sistema consistente e fácil de entender. Um princípio orientador fundamental para a proteção, testado pelo tempo, é o **princípio do menor privilégio**. Ele determina que programas, usuários e até mesmo sistemas recebam apenas privilégios suficientes para realizar suas tarefas.

Considere a analogia de um guarda de segurança com uma chave de acesso. Se essa chave permitir que o guarda entre apenas nas áreas públicas que ele vigia, então o mau uso da chave resultará em um dano mínimo. Contudo, se a chave permitir o acesso a todas as áreas, então o dano decorrente de perda, roubo, mau uso, cópia ou outro comprometimento será muito maior.

Um sistema operacional seguindo o princípio do menor privilégio implementa seus recursos, programas, chamadas de sistema e estruturas de dados de modo que a falha ou o comprometimento de um componente realize o mínimo de dano e permita que o mínimo de dano seja feito. O estouro de um buffer em um processo daemon do sistema poderá causar a falha do daemon, por exemplo, mas não deverá permitir a execução de código da pilha de processos, o que permitiria que um usuário remoto ganhasse o máximo de privilégios e acesso ao sistema inteiro (como acontece com frequência hoje em dia).

Tal sistema operacional também fornece chamadas de sistema e serviços que permitem que as aplicações sejam escritas com controles de acesso minuciosos. Ele fornece mecanismos para permitir privilégios quando forem necessários e desativá-los quando não forem necessários. Também é benéfica a criação de trilhas de auditoria para todo o acesso de função privilegiado. A trilha de auditoria permite que o programador, administrador de sistemas ou agente legal rastreie todas as atividades de proteção e segurança no sistema.

O gerenciamento de usuários com o princípio do menor privilégio acarreta a criação de uma conta separada para cada usuário, apenas com os privilégios que o usuário precisa. Um operador que precisa montar fitas e arquivos de backup no sistema tem acesso apenas aos comandos e arquivos necessários para realizar a tarefa. Alguns sistemas implementam o controle de acesso baseado em posição (Role-Based Access Control - RBAC) para fornecer essa funcionalidade.

Os computadores implementados em uma instalação de computação sob o princípio do menor privilégio podem ser limitados a executar serviços específicos, acessar hosts remotos específicos por meio de serviços específicos e realizar isso durante horários específicos. Normalmente, essas restrições são implementadas pela ativação ou desativação de cada serviço e pelo uso das listas de controle de acesso, conforme descrevemos nas [Seções 10.6.2 e 14.6](#).

O princípio do menor privilégio pode ajudar a produzir um ambiente de computação mais seguro. Infelizmente, isso não acontece com frequência. Por exemplo, o Windows 2000 possui um complexo esquema de proteção em seu núcleo, e mesmo assim tem muitas falhas na segurança. Por comparação, o Solaris é considerado relativamente seguro, embora seja uma variante do UNIX, que historicamente foi projetado com pouca proteção em mente. Um motivo para a diferença pode ser que o Windows 2000 tenha mais linhas de código e mais serviços que o Solaris, e por isso tem mais a proteger. Outro motivo poderia ser que o esquema de proteção no Windows 2000 é incompleto ou protege os aspectos errados do sistema operacional, deixando outras áreas vulneráveis.

14.3 Domínio de proteção

Um computador é uma coleção de processos e objetos. Com *objetos* queremos dizer **objetos de hardware** (como CPU, segmentos de memória, impressoras, discos e unidades de fita) e **objetos de software** (como arquivos, programas e semáforos). Cada objeto possui um nome exclusivo, que o diferencia de todos os outros objetos no sistema, e cada um só pode ser acessado por meio de operações bem definidas e significativas. Os objetos são essencialmente tipos de dados abstratos.

As operações possíveis podem depender do objeto. Por exemplo, em uma CPU, só podemos executar. Os segmentos de memória podem ser lidos e escritos, enquanto um CD-ROM ou DVD-ROM só pode ser lido. Unidades de fita podem ser lidas, escritas e rebobinadas. Os arquivos de dados podem ser criados, abertos, lidos, escritos, fechados e excluídos; os arquivos de programa podem ser lidos, escritos, executados e excluídos.

Um processo só deverá ter permissão para acessar os recursos para os quais tem autorização. Além do mais, a qualquer momento, um processo só deve ser capaz de acessar aqueles recursos exigidos para concluir sua tarefa. Esse segundo requisito, denominado princípio *precisa saber* (*need-to-know*), é útil na limitação da quantidade de danos que um processo defeituoso pode causar no sistema. Por exemplo, quando o processo p chama o procedimento $A()$, o procedimento precisa ter permissão para acessar apenas suas próprias variáveis e os parâmetros formais passados para ele; ele não deverá ser capaz de acessar todas as variáveis do processo p . De modo semelhante, considere o caso em que o processo p chama um compilador para compilar um arquivo em particular. O compilador não deverá ser capaz de acessar arquivos arbitrariamente, mas deve ter acesso apenas a um subconjunto de arquivos bem definido (como arquivo-fonte, arquivo de listagem, e assim por diante) relacionado com o arquivo a ser compilado. Por sua vez, o compilador pode ter arquivos privados, usados para fins de contabilidade ou otimização, que o processo p não deverá ser capaz de acessar. O princípio *precisa saber* é semelhante ao princípio do menor privilégio, discutido na [Seção 14.2](#), no sentido de que os objetivos da proteção são minimizar os riscos de possíveis violações de segurança.

14.3.1 Estrutura do domínio

Para facilitar o esquema que acabamos de descrever, um processo opera dentro de um **domínio de proteção**, que especifica os recursos que o processo pode acessar. Cada domínio define um conjunto de objetos e os tipos de operações que podem ser invocadas sobre cada objeto. A capacidade de executar uma operação sobre um objeto é um **direito de acesso**. Um domínio é uma coleção de direitos de acesso, cada um sendo um par ordenado $\langle\text{nome do objeto}, \{\text{operações}\}\rangle$. Por exemplo, se o domínio D tiver o direito de acesso $\langle\text{arquivo } F, \{\text{read, write}\}\rangle$, então um processo executando no domínio D pode ler e escrever o arquivo F ; porém, ele não pode realizar qualquer outra operação sobre esse objeto.

Os domínios não precisam ser disjuntos; eles podem compartilhar direitos de acesso. Por exemplo, na [Figura 14.1](#), temos três domínios: D_1 , D_2 e D_3 . O direito de acesso $\langle O_4, \{\text{print}\}\rangle$ é compartilhado por D_2 e D_3 , indicando que um processo executando em um desses dois domínios pode imprimir o objeto O_4 . Observe que um processo precisa estar executando no domínio D_1 para ler e escrever o objeto O_1 , enquanto apenas os processos no domínio D_3 podem executar o objeto O_2 .

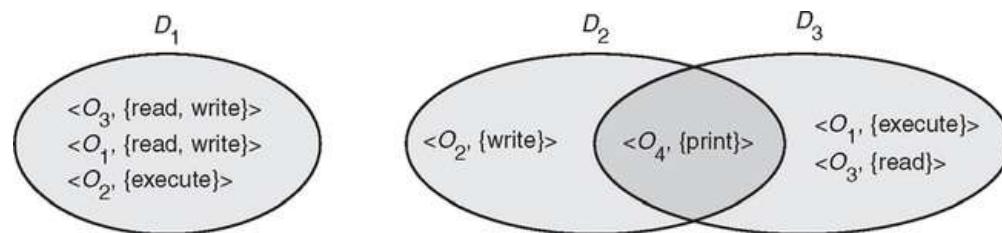


FIGURA 14.1 Sistema com três domínios de proteção.

A associação entre um processo e um domínio pode ser **estática**, se o conjunto de recursos disponíveis a um processo for fixo por toda a vida desse processo, ou **dinâmica**. Como se poderia esperar, o estabelecimento de domínios dinâmicos de proteção é mais complicado do que o estabelecimento de domínios estáticos de proteção.

Se a associação entre processos e domínios for fixa e quisermos aderir ao princípio “precisa saber”, então um mecanismo precisa estar disponível para mudar o conteúdo de um domínio. A razão origina-se do fato de um processo poder executar em duas fases diferentes e poder, por exemplo, precisar de acesso de leitura em uma fase e acesso de escrita em outra. Se um domínio for estático, ele terá de definir o domínio para incluir acesso de leitura e escrita. Contudo, esse arranjo

provê mais direitos do que são necessários em cada uma das duas fases, pois temos acesso de leitura na fase em que só precisamos de acesso de escrita, e vice-versa. Assim, o princípio “precisa saber” é violado. Temos de permitir que o conteúdo de um domínio seja modificado, para que o domínio sempre reflita o mínimo de direitos de acesso necessário.

Se a associação for dinâmica, haverá um mecanismo à disposição para permitir a **troca de domínio**, possibilitando que o processo troque de um domínio para outro. Também podemos querer permitir que o conteúdo de um domínio seja trocado. Se não pudermos mudar o conteúdo de um domínio, poderemos prover o mesmo efeito criando um novo domínio com o conteúdo alterado e passando para esse novo domínio quando quisermos mudar o conteúdo do domínio.

Um domínio pode ser percebido de diversas maneiras:

- Cada *usuário* pode ser um domínio. Nesse caso, o conjunto de objetos que podem ser acessados depende da identidade do usuário. A troca de domínio ocorre quando o usuário é mudado – em geral, quando um usuário se desconecta e outro usuário se conecta ao sistema.
- Cada *processo* pode ser um domínio. Nesse caso, o conjunto de objetos que podem ser acessados depende da identidade do processo. A troca de domínio ocorre quando um processo envia uma mensagem para outro processo e depois aguarda uma resposta.
- Cada *procedimento* pode ser um domínio. Nesse caso, um conjunto de objetos que podem ser acessados corresponde às variáveis locais definidas dentro do procedimento. A troca de domínio ocorre quando é feita uma chamada de procedimento.

Discutiremos a troca de domínio com mais detalhes na [Seção 14.4](#).

Considere o modelo-padrão do modo dual (modo monitor-usuário) de execução do sistema operacional. Quando um processo é executado no modo monitor, ele pode executar instruções privilegiadas e, assim, obter controle completo do sistema computadorizado. Ao contrário, quando o processo executa no modo usuário, ele só pode invocar instruções não privilegiadas. Consequentemente, ele só pode executar dentro do seu espaço de memória predefinido. Esses dois modos protegem o sistema operacional (executando no domínio do monitor) dos processos do usuário (executando no domínio do usuário). Em um sistema operacional multiprogramado, dois domínios de proteção são insuficientes, pois os usuários também querem estar protegidos um do outro. Portanto, um esquema mais elaborado é necessário. Ilustramos esse esquema examinando dois sistemas operacionais marcantes – UNIX e MULTICS –, para ver como eles implementam esses conceitos.

14.3.2 Um exemplo: UNIX

No sistema operacional UNIX, um domínio está associado ao usuário. A troca do domínio corresponde à mudança temporária da identificação do usuário. Essa mudança é realizada por meio do sistema de arquivos da seguinte maneira. Uma identificação de proprietário e um bit de domínio (conhecido como bit *setuid*) estão associados a cada arquivo. Quando o bit setuid está *ligado* e um usuário executa esse arquivo, o userID é definido como sendo do proprietário do arquivo; porém, quando o bit está *desligado*, o userID não muda. Por exemplo, quando um usuário A (isto é, com *userID = A*) começa a executar um arquivo localizado em B, cujo bit de domínio associado está *desligado*, o *userID* do processo é definido como A. Quando o bit setuid está *ligado*, o *userID* é definido como aquele do proprietário do arquivo: B. Quando o processo termina, essa mudança de *userID* temporária termina.

Outros métodos são usados para mudar domínios nos sistemas operacionais em que os IDs do usuário são usados para definição de domínio, pois quase todos os sistemas precisam prover tal mecanismo. Esse mecanismo é usado quando uma facilidade privilegiada de outra maneira precisa se tornar disponível à população de usuários em geral. Por exemplo, pode ser desejável permitir que os usuários accessem uma rede sem permitir que escrevam seus próprios programas de rede. Nesse caso, em um sistema UNIX, o bit setuid em um programa de rede seria definido, causando a mudança do ID do usuário quando o programa fosse executado. O ID do usuário mudaria para o de um usuário com privilégio de acesso à rede (como *root*, o ID do usuário mais poderoso). Um problema com esse método é que, se um usuário conseguir criar um arquivo com ID do usuário *root* e com seu bit setuid *ativo*, esse usuário pode se tornar *root* e fazer qualquer coisa no sistema.

Uma alternativa a esse método, usada em outros sistemas operacionais, é colocar os programas privilegiados em um diretório especial. O sistema operacional seria projetado para mudar o ID do usuário de qualquer programa executado desse diretório, seja para o equivalente de *root* ou para o ID do usuário do proprietário do diretório. Isso elimina um problema de segurança com os programas de setuid, em que os crackers os criam e escondem para uso futuro (usando nomes obscuros de arquivo ou diretório). Contudo, esse método é menos flexível do que aquele usado no UNIX.

Ainda mais restritivos, portanto mais protetores, são os sistemas que não permitem uma mudança de ID do usuário. Nesses casos, técnicas especiais precisam ser usadas para permitir que os usuários accessem facilidades privilegiadas. Por exemplo, um **processo daemon** pode ser iniciado no momento do boot e executado como um ID do usuário especial. Os usuários, então, executam um programa separado, que envia requisições para esse processo sempre que precisam utilizar essa

facilidade. Esse método é usado pelo sistema operacional TOPS-20.

Em qualquer um desses sistemas, é preciso tomar muito cuidado na escrita de programas privilegiados. Qualquer descuido pode resultar em uma total falta de proteção no sistema. Em geral, esses programas são os primeiros a ser atacados por pessoas que tentam se infiltrar em um sistema; infelizmente, os invasores costumam ter sucesso. Por exemplo, a segurança tem sido invadida em muitos sistemas UNIX por causa do recurso setuid. Discutiremos sobre a segurança no [Capítulo 15](#).

14.3.3 Um exemplo: MULTICS

No sistema MULTICS, os domínios de proteção são organizados hierarquicamente em uma estrutura de anel. Cada anel corresponde a um único domínio ([Figura 14.2](#)). Os anéis são numerados de 0 a 7. Sejam D_i e D_j dois anéis de domínio quaisquer. Se $j < i$, então D_i é um subconjunto de D_j , ou seja, um processo executando no domínio D_j possui mais privilégios do que um processo executando no domínio D_i . Um processo executando no domínio D_0 tem mais privilégios. Se houver somente dois anéis, esse esquema é equivalente ao modo de execução monitor-usuário, no qual o modo monitor corresponde a D_0 e o modo usuário corresponde a D_1 .

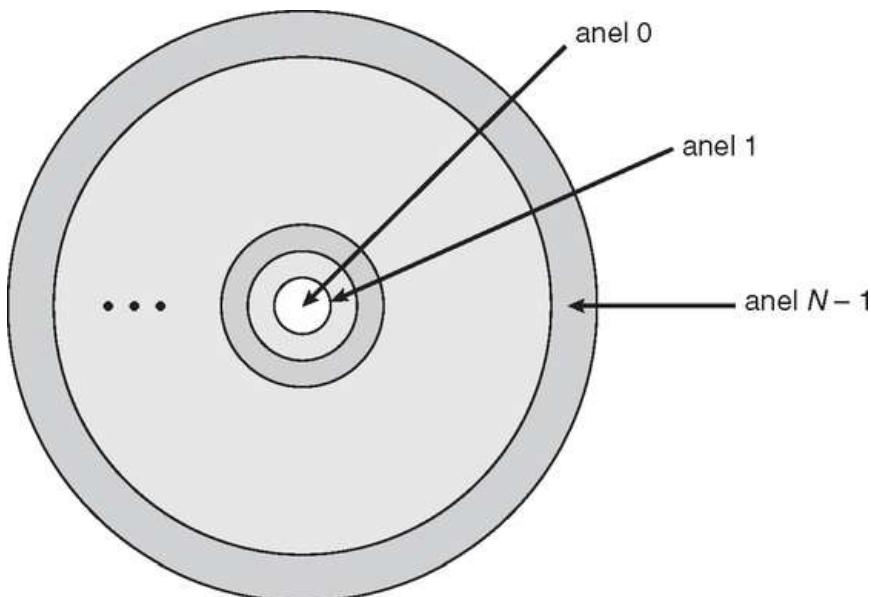


FIGURA 14.2 Estrutura de anel do MULTICS.

O MULTICS possui um espaço de endereços de segmento; cada segmento é um arquivo e cada segmento está associado a um dos anéis. Uma descrição de segmento inclui uma entrada que identifica o número do anel. Além disso, ele inclui três bits de acesso para controlar leitura, escrita e execução. A associação entre segmentos e anéis é uma decisão política, com a qual não nos preocupamos aqui.

Um contador do *número do anel atual* está associado a cada processo, identificando o anel em que o processo está sendo executado. Quando um processo está executando no anel i , ele não pode acessar um segmento associado ao anel j ($j < i$). Ele pode acessar um segmento associado ao anel k ($k \geq i$). Contudo, o tipo de acesso é restrinido de acordo com os bits de acesso associados a esse segmento.

A troca de domínio no MULTICS ocorre quando um processo atravessa de um anel para outro, chamando um procedimento em um anel diferente. Obviamente, essa troca precisa ser feita de maneira controlada; caso contrário, um processo poderia começar executando no anel 0, e não haveria qualquer proteção. Para permitir a troca de domínio controlada, modificamos o campo de anel do descriptor de segmento para incluir o seguinte:

- **Par de acesso.** Um par de inteiros, $b1$ e $b2$, de modo que $b1 \leq b2$.
- **Límite.** Um inteiro $b3$, de modo que $b3 > b2$.
- **Lista de portas.** Identifica os pontos de entradas (ou **portas**) em que os segmentos podem ser chamados.

Se um processo executando no anel i chamar um procedimento (ou segmento) com um par de acesso ($b1, b2$), então a chamada é permitida se $b1 \leq i \leq b2$, e o número de anel atual do processo continua sendo i . Caso contrário, ocorre uma interceptação para o sistema operacional, e a situação é tratada da seguinte forma:

- Se $i < b1$, então a chamada tem permissão para ocorrer, pois temos uma transferência para um anel (ou domínio) com menos privilégios. Todavia, se são passados parâmetros que se referem a

segmentos em um anel inferior (ou seja, segmentos não acessíveis ao procedimento que chamou), então esses segmentos precisam ser copiados para uma área que pode ser acessada pelo procedimento chamado.

- Se $i > b_2$, então a chamada tem permissão somente se b_3 for maior ou igual a i , e a chamada foi direcionada para um dos pontos de entrada designados na lista de portas. Esse esquema permite que os processos com direitos de acesso limitados chamem procedimentos em anéis inferiores, que possuem mais direitos de acesso, mas somente de uma maneira bastante controlada.

A principal desvantagem da estrutura de anel (ou hierárquica) é que ela não nos permite impor o princípio “precisa saber”. Em particular, se um objeto tiver de ser acessível no domínio D_j , mas não acessível no domínio D_i , então precisamos ter $j < i$. Esse requisito significa que cada segmento acessível em D_i também precisa ser acessível em D_j .

O sistema de proteção do MULTICS é mais complexo e menos eficaz do que aqueles usados nos sistemas operacionais atuais. Se a proteção interfere com a facilidade de uso do sistema ou diminui significativamente o desempenho do sistema, então seu uso precisa ser pesado com cuidado em relação à finalidade do sistema. Por exemplo, você desejaría ter um sistema de proteção complexo em um computador usado por uma universidade para processar as notas dos alunos e também usado pelos alunos para o trabalho de aula. Um sistema de proteção semelhante não seria adequado para um computador que está sendo usado para triturar números, em que o desempenho é de importância primordial. Preferiríamos separar o mecanismo da política de proteção, permitindo que o mesmo sistema tenha proteção complexa ou simples, dependendo das necessidades dos usuários. Para separar o mecanismo da política, precisamos de modelos de proteção mais genéricos.

14.4 Matriz de acesso

Nosso modelo de proteção pode ser visto de forma abstrata como uma matriz, chamada **matriz de acesso**. As linhas da matriz de acesso representam domínios, e as colunas representam objetos. Cada entrada na matriz consiste em um conjunto de direitos de acesso. Como a coluna define objetos explicitamente, podemos omitir o nome do objeto do direito de acesso. A entrada $\text{access}(i,j)$ define um conjunto de operações que um processo, executando no domínio D_i , pode invocar sobre o objeto O_j .

Para ilustrar esses conceitos, consideramos a matriz de acesso mostrada na [Figura 14.3](#). Existem quatro domínios e quatro objetos, três arquivos (F_1, F_2, F_3) e uma impressora a laser. Um processo executando no domínio D_1 pode ler os arquivos F_1 e F_3 . Um processo executando no domínio D_4 tem os mesmos privilégios que um executando no domínio D_1 , mas, além disso, ele também pode escrever nos arquivos F_1 e F_3 . Observe que a impressora a laser só pode ser acessada por um processo executando no domínio D_2 .

domínio \ objeto	F_1	F_2	F_3	impressora
D_1	read		read	
D_2				print
D_3		read	execute	
D_4	read write		read write	

FIGURA 14.3 Matriz de acesso.

O esquema de matriz de acesso nos provê o mecanismo para especificar uma série de políticas. O mecanismo consiste em implementar a matriz de acesso e garantir que as propriedades semânticas aqui esboçadas sejam mantidas. Mais especificamente, temos de garantir que um processo executando no domínio D_i possa acessar apenas os objetos especificados na linha i , e então somente conforme permitido pelas entradas na matriz de acesso.

A matriz de acesso pode implementar decisões de política com relação à proteção. As decisões de política envolvem quais direitos devem estar incluídos na entrada número (i,j) . Também temos de decidir sobre o domínio em que cada processo é executado. A última política é decidida pelo sistema operacional.

Os usuários decidem sobre o conteúdo das entradas de matriz de acesso. Quando um usuário cria um novo objeto O_j , a coluna O_j é acrescentada à matriz de acesso com as entradas de inicialização apropriadas, conforme ditado pelo criador. O usuário pode decidir entrar com alguns direitos em algumas entradas na coluna j e outros direitos em outras entradas, conforme a necessidade.

A matriz de acesso provê um mecanismo apropriado para definir e implementar o controle estrito para a associação estática e dinâmica entre os processos e os domínios. Quando passamos um processo de um domínio para outro, estamos executando uma operação (switch) sobre esse objeto (o domínio). Podemos controlar a troca de domínio incluindo domínios entre os objetos da matriz de acesso. De modo semelhante, quando mudamos o conteúdo da matriz de acesso, estamos realizando uma operação sobre um objeto: a matriz de acesso. Mais uma vez, podemos controlar essas mudanças incluindo a própria matriz de acesso como um objeto. Na realidade, como cada entrada na matriz de acesso pode ser modificada individualmente, temos de considerar cada entrada na matriz de acesso como um objeto a ser protegido. Agora, precisamos considerar apenas as operações possíveis sobre esses novos objetos (domínios e a matriz de acesso) e decidir como queremos que os processos sejam capazes de executar essas operações.

Os processos devem ser capazes de trocar de um domínio para outro. A troca do domínio D_i para o domínio D_j é permitida se e somente se o direito de acesso $\text{switch} \in \text{access}(i,j)$. Assim, na [Figura 14.4](#), um processo executando no domínio D_2 pode passar para o domínio D_3 ou para o domínio D_4 . Um processo no domínio D_4 pode passar para D_1 , e um no domínio D_1 pode passar para D_2 .

domínio \ objeto	F_1	F_2	F_3	impres-sora laser	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch
D_3		read	execute					
D_4	read write		read write		switch			

FIGURA 14.4 Matriz de acesso da Figura 14.3 com domínios como objetos.

A permissão controlada para mudar o conteúdo das entradas da matriz de acesso exige três operações adicionais: *copy*, *owner* e *control*. Examinamos essas operações a seguir.

A capacidade de copiar um direito de acesso de um domínio (ou linha) da matriz de acesso para outro é indicada por um asterisco (*) anexado ao direito de acesso. O direito de *cópia* (*copy*) permite a cópia do direito de acesso somente dentro da coluna (ou seja, para o objeto) para a qual o direito é definido. Por exemplo, na Figura 14.5(a), um processo executando no domínio D_2 pode copiar a operação *read* para qualquer entrada associada ao arquivo F_2 . Em virtude disso, a matriz de acesso da Figura 14.5(a) pode ser modificada para a matriz de acesso mostrada na Figura 14.5(b).

domínio \ objeto	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute		

(a)

domínio \ objeto	F_1	F_2	F_3
D_1	execute		write*
D_2	execute	read*	execute
D_3	execute	read	

(b)

FIGURA 14.5 Matriz de acesso com direitos de cópia.

Esse esquema possui duas variantes:

1. Um direito é copiado de $access(i,j)$ para $access(k,j)$; depois, ele é removido de $access(i,j)$; essa ação é uma *transferência* de um direito, em vez de uma cópia.
2. A propagação do direito de *cópia* pode ser limitada, ou seja, quando o direito R^* é copiado de $access(i,j)$ para $access(k,j)$, somente o direito R (não R^*) é criado. Um processo executando no domínio D_k não pode copiar o direito R adiante.

Um sistema só pode selecionar um desses três direitos de *cópia* ou então pode prover todos os três identificando-os como direitos separados: *cópia*, *transferência* e *cópia limitada*.

Também precisamos de um mecanismo para permitir o acréscimo de novos direitos e remoção de alguns direitos. O direito de *proprietário* (*owner*) controla essas operações. Se $access(i,j)$ incluir o direito de *proprietário*, então um processo executando no domínio D_i pode acrescentar ou remover

qualquer direito na entrada da coluna j . Por exemplo, na [Figura 14.6\(a\)](#), o domínio D_1 é o proprietário de F_1 e, por isso, pode acrescentar e excluir qualquer direito válido na coluna F_1 . De modo semelhante, o domínio D_2 é o proprietário de F_2 e F_3 e, por isso, pode acrescentar e remover qualquer direito válido dentro dessas duas colunas. Assim, a matriz de acesso da [Figura 14.6\(a\)](#) pode ser modificada para a matriz de acesso mostrada na [Figura 14.6\(b\)](#).

domínio \ objeto	F_1	F_2	F_3
D_1	owner execute		write
D_2		read* owner	read* owner write
D_3	execute		

(a)

domínio \ objeto	F_1	F_2	F_3
D_1	owner execute		write
D_2		owner read* write*	read* owner write
D_3		write	write

(b)

FIGURA 14.6 Matriz de acesso com direitos de proprietário.

Os direitos de *cópia* e *proprietário* permitem que um processo mude as entradas em uma coluna. Um mecanismo também é necessário para mudar as entradas em uma linha. O direito de *controle* se aplica apenas a objetos de domínio. Se *access* (i,j) incluir o direito de *controle* (*control*), então um processo executando no domínio D_i pode remover qualquer direito de acesso da linha j . Por exemplo, suponha que na [Figura 14.4](#) incluamos o direito de *controle* em *access* (D_2, D_4). Então, um processo executando no domínio D_2 poderia modificar o domínio D_4 , como mostra a [Figura 14.7](#).

domínio \ objeto	F_1	F_2	F_3	impres- sora laser	D_1	D_2	D_3	D_4
D_1	read		read			switch		
D_2				print			switch	switch control
D_3		read	execute					
D_4	write		write		switch			

FIGURA 14.7 Matriz de acesso modificada da [Figura 14.4](#).

Os direitos de *cópia* e *proprietário* nos proveem um mecanismo para limitar a propagação dos direitos de acesso. Entretanto, eles não nos dão as ferramentas apropriadas para impedir a propagação (ou divulgação) de informações. O problema de garantir que nenhuma informação

mantida inicialmente em um objeto pode migrar fora do seu ambiente de execução é denominado **problema de confinamento**. Esse problema, em geral, não tem solução (ver Notas Bibliográficas ao final do capítulo).

Essas operações sobre os domínios e a matriz de acesso não são importantes por si sós, mas ilustram a capacidade do modelo de matriz de acesso de permitir a implementação e o controle de requisitos de proteção dinâmicos. Novos objetos e novos domínios podem ser criados dinamicamente e incluídos no modelo da matriz de acesso. Contudo, mostramos apenas que o mecanismo básico está aqui; os projetistas e usuários do sistema precisam tomar as decisões políticas com relação a quais domínios devem ter acesso a quais objetos e de que maneiras.

14.5 Implementação da matriz de acesso

Como a matriz de acesso pode ser implementada de forma eficiente? Em geral, a matriz será esparsa, ou seja, a maioria das entradas estará vazia. Embora as técnicas de estrutura de dados estejam disponíveis para representar matrizes esparsas, elas não são particularmente úteis para essa aplicação, devido ao modo como a facilidade de proteção é utilizada. Aqui, primeiro descrevemos vários métodos de implementação da matriz de acesso e depois comparamos os métodos.

14.5.1 Tabela global

A implementação mais simples da matriz de acesso é uma tabela global consistindo em um conjunto de triplas ordenadas $\langle\text{domínio}, \text{objeto}, \text{conjunto de direitos}\rangle$. Sempre que uma operação M é executada sobre um objeto O_j dentro do domínio D_i , a tabela global é pesquisada em busca de uma tripla $\langle D_i, O_j, R_k \rangle$, com $M \in R_k$. Se essa tripla for encontrada, a operação tem permissão para continuar; caso contrário uma condição de exceção (ou erro) é levantada.

Essa implementação sofre de várias desvantagens. A tabela costuma ser grande e, assim, não pode ser mantida na memória principal, de modo que uma E/S adicional é necessária. As técnicas de memória virtual são utilizadas para gerenciar essa tabela. Além disso, é difícil tirar proveito de agrupamentos especiais de objetos ou domínios. Por exemplo, se todos puderem ler determinado objeto, esse objeto precisará ter uma entrada separada em cada domínio.

14.5.2 Listas de acesso para objetos

Cada coluna na matriz de acesso pode ser implementada como uma lista de acesso para um objeto, conforme descrevemos na [Seção 10.6.2](#). Obviamente, as entradas vazias podem ser descartadas. A lista resultante para cada objeto consiste em pares ordenados de $\langle\text{domínio}, \text{conjunto de direitos}\rangle$, que definem todos os domínios com um conjunto não vazio de direitos de acesso para esse objeto.

Essa técnica pode ser estendida facilmente para definir uma lista e um conjunto-*padrão* de direitos de acesso. Quando uma operação M sobre um objeto O_j for tentada no domínio D_i , pesquisamos a lista de acesso para o objeto O_j , procurando uma entrada $\langle D_i, R_k \rangle$ com $M \in R_k$. Se a entrada for localizada, permitimos a operação; caso contrário, verificamos o conjunto-*padrão*. Se M estiver no conjunto-*padrão*, então permitimos o acesso. Caso contrário, o acesso é negado e ocorre uma condição de exceção. Por questão de eficiência, podemos verificar o conjunto-*padrão* primeiro e depois pesquisar a lista de acesso.

14.5.3 Listas de capacidade para domínios

Em vez de associar as colunas da matriz de acesso aos objetos como listas de acesso, podemos associar cada linha ao seu domínio. Uma **lista de capacidade** para um domínio é uma lista de objetos junto com as operações permitidas sobre esses objetos. Um objeto é representado por seu nome ou endereço físico, chamado **capacidade**. Para executar a operação M sobre o objeto O_j , o processo executa a operação M , especificando a capacidade (ou ponteiro) para o objeto O_j como parâmetro. A **posse** simples da capacidade significa que o acesso é permitido.

A lista de capacidade está associada a um domínio, mas nunca está acessível diretamente a um processo executando nesse domínio. Em vez disso, a lista de capacidade por si só é um objeto protegido, mantido pelo sistema operacional e acessado pelo usuário apenas indiretamente. A proteção baseada em capacidade conta com o fato de que as capacidades nunca têm permissão para migrar para qualquer espaço de endereço acessível diretamente por um processo do usuário (onde poderiam ser modificados). Se todas as capacidades estiverem seguras, o objeto que protegem também estará seguro contra acesso não autorizado.

As capacidades foram propostas originalmente como uma espécie de ponteiro de segurança, para atender a necessidade de proteção de recursos prevista quando os computadores multiprogramados atingiram a maioridade. A ideia de um ponteiro inherentemente protegido (do ponto de vista de um usuário do sistema) provê o alicerce para a proteção que pode ser estendida para o nível de aplicações.

Para prover a proteção inherente, temos de distinguir as capacidades dos outros tipos de objetos e que precisam ser interpretadas por uma máquina abstrata em que os programas de mais alto nível são executados. As capacidades são distinguidas de outros dados de duas maneiras:

- Cada objeto possui uma **chave (tag)** para indicar seu tipo como uma capacidade ou como dado acessível. As próprias chaves não podem ser diretamente acessíveis por um programa de aplicação. O suporte para hardware ou firmware pode ser usado para executar essa restrição. Embora somente um bit seja necessário para distinguir entre as capacidades e outros objetos, mais bits são utilizados. Essa extensão permite que todos os objetos sejam reunidos com seus

próprios tipos pelo hardware. Assim, o hardware pode distinguir inteiros, números de ponto flutuante, ponteiros, booleanos, caracteres, instruções, capacidades e valores não inicializados por suas chaves.

- Como alternativa, o espaço de endereços associado a um programa pode ser dividido em duas partes. Uma parte é acessível ao programa e contém os dados e instruções normais do programa. A outra parte, contendo a lista de compatibilidade, é acessível apenas pelo sistema operacional. Um espaço de memória segmentado ([Seção 8.6](#)) é útil para dar suporte a essa técnica.

Vários sistemas de proteção baseados em capacidade foram desenvolvidos; vamos descrevê-los rapidamente na [Seção 14.8](#). O sistema operacional Mach também usa uma versão da proteção baseada em capacidade.

14.5.4 Um mecanismo lock-key

O **esquema lock-key** é um meio-termo entre listas de acesso e listas de capacidade. Cada objeto possui uma lista de padrões de bit exclusivos, chamados **locks** (**fechaduras**). De modo semelhante, cada domínio possui uma lista de padrões de bit exclusivos, chamados **keys** (**chaves**). Um processo executando em um domínio pode acessar um objeto somente se esse domínio possui uma chave que combina com um dos locks do objeto.

Como acontece com as listas de capacidade, a lista de chaves para um domínio precisa ser administrada pelo sistema operacional em favor do domínio. Os usuários não têm permissão para examinar ou modificar a lista de chaves (ou locks) diretamente.

14.5.5 Comparação

Como você poderia esperar, a escolha de uma técnica para implementar uma matriz de acesso envolve várias escolhas. O uso de uma tabela global é simples; porém, a tabela pode ser muito grande e normalmente não pode tirar proveito de agrupamentos especiais de objetos ou domínios. As listas de acesso correspondem diretamente às necessidades dos usuários. Quando um usuário cria um objeto, ele pode especificar quais domínios podem acessar o objeto, bem como as operações permitidas. No entanto, como as informações de direitos de acesso para determinado domínio não estão no local, é difícil determinar o conjunto de direitos de acesso para cada domínio. Além disso, cada acesso ao objeto precisa ser verificado, exigindo uma pesquisa da lista de acesso. Em um grande sistema com longas listas de acesso, essa pesquisa pode ser demorada.

As listas de capacidade não correspondem diretamente às necessidades dos usuários; contudo, elas são úteis para localizar informações para determinado processo. O processo tentando acessar precisa apresentar uma capacidade para esse acesso. Depois, o sistema de proteção só precisa verificar se a capacidade é válida. Contudo, a revogação de capacidades pode ser ineficaz ([Seção 14.7](#)).

Como mencionado, o mecanismo lock-key é um meio-termo entre as listas de acesso e de capacidade. O mecanismo pode ser eficaz e flexível, dependendo do tamanho das chaves. As chaves podem ser passadas livremente de um domínio para outro. Além disso, os privilégios de acesso podem ser revogados pela simples técnica de mudar alguns dos locks associados ao objeto ([Seção 14.7](#)).

A maioria dos sistemas utiliza uma combinação de listas de acesso e capacidades. Quando um processo tenta acessar um objeto, a lista de acesso é pesquisada. Se o acesso for negado, ocorrerá uma condição de exceção. Caso contrário, uma capacidade será criada e anexada ao processo. Outras referências utilizam a capacidade para demonstrar que o acesso está permitido. Após o último acesso, a capacidade é destruída. Essa estratégia é usada no sistema MULTICS e no sistema CAL.

Como exemplo de como essa estratégia funciona, considere um sistema de arquivos em que cada arquivo possui uma lista de acesso associada. Quando um processo abre um arquivo, a estrutura de diretório é pesquisada para localizar o arquivo, a permissão de acesso é verificada e os buffers são alocados. Toda essa informação é registrada em uma nova entrada em uma tabela de arquivo associada ao processo. A operação retorna um índice para essa tabela para o arquivo recém-aberto. Todas as operações sobre o arquivo são feitas pela especificação do índice para a tabela de arquivos. A entrada na tabela de arquivos, então, aponta para o arquivo e seus buffers. Quando o arquivo é fechado, a entrada da tabela de arquivos é excluída. Como a tabela de arquivos é mantida pelo sistema operacional, o usuário não pode danificá-la acidentalmente. Assim, o usuário pode acessar apenas os arquivos abertos. Como o acesso é verificado quando o arquivo é aberto, a proteção é garantida. Essa estratégia é usada no sistema UNIX.

O direito para acessar ainda *precisa* ser verificado em cada acesso, e a entrada da tabela de arquivos só possui uma capacidade para as operações permitidas. Se um arquivo for aberto para leitura, então a capacidade para o acesso de leitura é colocada na entrada da tabela de arquivos. Se for feita uma tentativa de escrever no arquivo, o sistema determina essa tentativa de violar a proteção comparando a operação requisitada com a capacidade indicada na entrada da tabela de arquivos.

14.6 Controle de acesso

Na [Seção 10.6.2](#), descrevemos como os controles de acesso podem ser usados nos arquivos dentro de um sistema de arquivos. Cada arquivo e diretório recebe um owner, um grupo ou, possivelmente, uma lista de usuários, e para cada uma dessas entidades a informação de controle de acesso é atribuída. Uma função semelhante pode ser acrescentada a outros aspectos de um sistema computadorizado. Um bom exemplo disso é encontrado no Solaris 10.

O Solaris 10 avança a proteção disponível nos sistemas operacionais da Sun Microsystems acrescentando explicitamente o princípio do menor privilégio por meio do **controle de acesso baseado em posição (RBAC)**. Essa facilidade gira em torno dos privilégios. Um privilégio é o direito de executar uma chamada de sistema ou usar uma opção dentro dessa chamada de sistema (como abrir um arquivo com acesso de escrita). Os privilégios podem ser atribuídos a processos, limitando-os ao acesso exato de que precisam para realizar seu trabalho. Os privilégios e os programas também podem ser atribuídos a **posições (roles)**. Os usuários recebem papéis ou podem ganhar papéis com base nas senhas para os papéis. Desse modo, um usuário pode apanhar uma posição que ative um privilégio, permitindo-o executar um programa para realizar uma tarefa específica, conforme representado na [Figura 14.8](#). Essa implementação de privilégios diminui o risco à segurança associado aos superusuários e programas setuid.

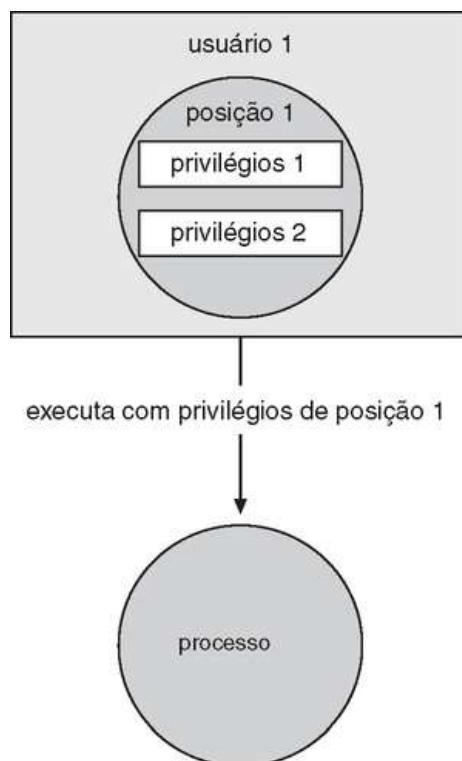


FIGURA 14.8 Controle de acesso baseado em posição no Solaris 10.

Observe que essa facilidade é semelhante à matriz de acesso descrita na [Seção 14.4](#). Esse relacionamento será explorado ainda mais nos exercícios ao final do capítulo.

14.7 Revogação de direitos de acesso

Em um sistema de proteção dinâmico, às vezes podemos ter de revogar direitos de acesso para os objetos compartilhados por diferentes usuários. Diversas questões sobre revogação poderão surgir:

■ **Imediata versus adiada.** A revogação ocorre imediatamente ou ela é adiada? Se a revogação for adiada, podemos descobrir quando ela ocorrerá?

■ **Seletiva versus geral.** Quando um direito de acesso a um objeto é revogado, ele afeta *todos* os usuários que possuem um direito de acesso a esse objeto ou podemos especificar um grupo seletivo de usuários cujos direitos de acesso devem ser revogados?

■ **Parcial versus total.** Um subconjunto dos direitos associados a um objeto pode ser revogado ou temos de revogar todos os direitos de acesso para esse objeto?

■ **Temporária versus permanente.** O acesso pode ser revogado permanentemente (ou seja, o direito de acesso revogado nunca mais estará disponível) ou o acesso pode ser revogado e mais tarde ser obtido novamente?

Com um esquema de lista de acesso, a revogação é fácil. A lista de acesso é pesquisada para os direitos de acesso a serem revogados, e eles são excluídos da lista. A revogação é imediata e pode ser geral ou seletiva, total ou parcial e permanente ou temporária.

Entretanto, as capacidades apresentam um problema de revogação muito mais difícil, como já dissemos. Como as capacidades são distribuídas por todo o sistema, temos de localizá-las antes de poder revogá-las. Os esquemas que implementam a revogação para as capacidades incluem os seguintes:

■ **Reaquisição.** Periodicamente, as capacidades são excluídas de cada domínio. Se um processo quiser usar uma capacidade, ele poderá descobrir que a capacidade foi excluída. O processo pode, então, tentar readquirir a capacidade. Se o acesso tiver sido revogado, o processo não poderá readquirir a capacidade.

■ **Ponteiros de apoio.** Uma lista de ponteiros é mantida com cada objeto, apontando para todas as capacidades associadas a esse objeto. Quando a revogação é requisitada, podemos seguir esses ponteiros, alterando as capacidades conforme a necessidade. Esse esquema foi adotado no sistema MULTICS. Ele é bastante genérico, mas sua implementação é dispendiosa.

■ **Indireção.** As capacidades apontam indireta e não diretamente para os objetos. Cada capacidade aponta para uma entrada exclusiva em uma tabela global, que por sua vez aponta para o objeto. Implementamos a revogação pesquisando a tabela global para a entrada desejada e excluindo-a. Quando um acesso é requisitado, descobre-se que a capacidade aponta para uma entrada ilegal na tabela. As entradas da tabela podem ser reutilizadas para outras capacidades sem dificuldade, pois tanto a capacidade quanto a entrada da tabela contêm o nome exclusivo do objeto. O objeto para uma capacidade e sua entrada na tabela precisam combinar. Esse esquema foi adotado no sistema CAL. Ele não permite a revogação seletiva.

■ **Chaves.** Uma chave é um padrão de bits exclusivo que pode ser associado a uma capacidade. Essa chave é definida quando a capacidade é criada e não pode ser modificada nem inspecionada pelo processo que possui essa capacidade. Uma **chave mestra** associada a cada objeto pode ser definida ou substituída com a operação set-key. Quando uma capacidade é criada, o valor atual da chave mestra é associado à capacidade. Quando a capacidade é exercida, sua chave é comparada com a chave mestra. Se as chaves combinarem, a operação tem permissão para continuar; caso contrário, uma condição de exceção é levantada. A revogação substitui a chave mestra por meio de um novo valor pela operação set-key, invalidando todas as capacidades anteriores para esse objeto.

Esse esquema não permite a revogação seletiva, pois somente uma chave-mestra está associada a cada objeto. Se associarmos uma lista de chaves a cada objeto, então a revogação seletiva pode ser implementada. Por fim, podemos agrupar todas as chaves em uma tabela global de chaves. Uma capacidade só é válida se a chave combinar com alguma chave na tabela global. Implementamos a revogação removendo da tabela a chave que combina. Com esse esquema, uma chave pode ser associada a vários objetos, e várias chaves podem ser associadas a cada objeto, provendo o máximo de flexibilidade.

Em esquemas baseados em chave, as operações para definir, inserir nas listas e excluir chaves das listas não deverão estar disponíveis a todos os usuários. Em particular, seria razoável permitir que apenas o proprietário de um objeto defina as chaves para esse objeto. Essa opção, porém, é uma decisão política, que o sistema de proteção pode implementar, mas não deve definir.

14.8 Sistemas baseados em capacidade

Nesta seção, analisamos dois sistemas de proteção baseados em capacidade. Esses sistemas variam em sua complexidade e no tipo de políticas que podem ser implementadas sobre eles. Nenhum deles é muito utilizado, mas são interessantes para provar os fundamentos das teorias de proteção.

14.8.1 Um exemplo: Hydra

O Hydra é um sistema de proteção baseado em capacidade, que provê flexibilidade considerável. O sistema implementa um conjunto fixo de direitos de acesso, incluindo formas básicas de acesso, como direito a leitura, escrita ou execução de um segmento da memória. Além disso, um usuário (do sistema de proteção) pode declarar outros direitos. A interpretação dos direitos definidos pelo usuário é realizada unicamente pelo programa do usuário, mas o sistema provê proteção de acesso para o uso desses direitos e também para o uso de direitos definidos pelo sistema. Essas facilidades constituem um desenvolvimento significativo na tecnologia de proteção.

As operações sobre os objetos são definidas por procedimento. Os procedimentos que implementam tais operações são por si só uma forma de objeto, e eles são acessados indiretamente pelas capacidades. Os nomes dos procedimentos definidos pelo usuário precisam ser identificados ao sistema de proteção se tiver de lidar com objetos do tipo definido pelo usuário. Quando a definição de um objeto se torna conhecida ao Hydra, os nomes das operações sobre o tipo se tornam **direitos auxiliares**. Os direitos auxiliares podem ser descritos em uma capacidade para uma instância do tipo. Para um processo realizar uma operação sobre um objeto tipificado, a capacidade que mantém para esse objeto precisa conter o nome da operação sendo invocada entre seus direitos auxiliares. Essa restrição permite que a discriminação dos direitos de acesso seja feita com base em cada instância e em cada processo.

O Hydra também provê **ampliação de direitos**. Esse esquema permite que um procedimento seja certificado como *confiável* para atuar sobre um parâmetro formal de um tipo especificado, em favor de qualquer processo que mantenha um direito de executar o procedimento. Os direitos mantidos por um procedimento confiável são independentes e podem exceder os direitos mantidos pelo processo que chama. Contudo, tal procedimento não pode ser considerado universalmente confiável (o procedimento não tem permissão para atuar sobre outros tipos, por exemplo) e a confiabilidade não pode ser estendida a quaisquer outros procedimentos ou segmentos de programa que possam ser executados por um processo.

A ampliação permite o acesso de procedimentos de implementação às variáveis de representação de um tipo de dados abstrato. Se um processo mantém uma capacidade para um objeto tipificado *A*, por exemplo, essa capacidade pode incluir um direito auxiliar para invocar alguma operação *P*, mas não incluiria qualquer um dos chamados direitos do kernel, como leitura, escrita ou execução, sobre o segmento que representa *A*. Essa capacidade provê a um processo um meio de acesso indireto (por meio da operação *P*) à representação de *A*, mas somente para fins específicos.

Entretanto, quando um processo invoca a operação *P* sobre um objeto *A*, a capacidade para acessar *A* pode ser amplificada enquanto o controle passa para o corpo do código de *P*. Essa ampliação pode ser necessária para permitir que *P* tenha o direito de acessar o segmento de armazenamento que representa *A*, de modo a implementar a operação que *P* define sobre o tipo de dados abstrato. O corpo do código de *P* pode ter permissão para ler ou escrever diretamente no segmento de *A*, embora o processo que chama não possa. Ao retornar de *P*, a capacidade para *A* é restaurada ao seu estado original, não ampliado. Esse caso é típico quando os direitos mantidos por um processo para o acesso a um segmento protegido devem mudar dinamicamente, dependendo da tarefa a ser realizada. O ajuste dinâmico de direitos é realizado para garantir a coerência de uma abstração definida pelo programador. A ampliação de direitos pode ser indicada explicitamente na declaração de um tipo abstrato para o sistema operacional Hydra.

Quando um usuário passa um objeto como argumento para um procedimento, podemos ter de garantir que o procedimento não poderá modificar o objeto. Podemos implementar essa restrição prontamente passando um direito de acesso que não tenha o direito de modificação (escrita). Todavia, se a ampliação puder ocorrer, o direito de modificar pode ser redeclarado. Assim, o requisito de proteção do usuário pode ser contornado. De modo geral, um usuário pode confiar que um procedimento realizará sua tarefa corretamente. Entretanto, essa suposição nem sempre é correta, devido aos erros de hardware ou software. O Hydra soluciona esse problema restringindo as ampliações.

O mecanismo de chamada de procedimento do Hydra foi projetado como uma solução direta para o problema dos subsistemas mutuamente suspeitos. Esse problema é definido da seguinte maneira. Suponha haver um programa que possa ser invocado como um serviço por muitos usuários diferentes (por exemplo, uma rotina de classificação, um compilador, um jogo). Quando os usuários invocam esse programa de serviço, eles correm o risco de o programa apresentar um problema e danificar os dados ou reter algum direito de acesso aos dados, que será utilizado (sem autorização) mais tarde. De modo semelhante, o programa de serviço pode ter alguns arquivos privados (para

fins de contabilidade, por exemplo) que não devem ser acessados diretamente pelo programa do usuário que o chama. O Hydra provê mecanismos para lidar diretamente com esse problema.

Um subsistema do Hydra é montado em cima do seu kernel de proteção e pode exigir proteção dos seus próprios componentes. Um subsistema interage com o kernel por meio de chamadas a um conjunto de primitivas definidas pelo kernel, que define os direitos de acesso aos recursos definidos pelo subsistema. O projetista do subsistema pode definir as políticas para uso desses recursos pelos processos do usuário, mas são impostas pelo uso da proteção de acesso-padrão proporcionada pelo sistema de capacidade.

Os programadores podem fazer uso direto do sistema de proteção, depois de se acostumarem com seus recursos no manual de referência apropriado. O Hydra provê uma grande biblioteca de procedimentos definidos pelo sistema, que podem ser chamados pelos programas do usuário. Os programadores podem incorporar explicitamente chamadas a esses procedimentos do sistema no código de seus programas ou usaria um programa tradutor que estivesse ligado ao Hydra.

14.8.2 Um exemplo: sistema Cambridge CAP

Uma técnica diferente para a proteção baseada em capacidade foi tomada no projeto do sistema Cambridge CAP. O sistema de capacidade do CAP é mais simples e superficialmente menos poderoso do que o do Hydra. No entanto, um exame mais detalhado mostra que ele também pode ser usado para prover proteção segura de objetos definidos pelo usuário. O CAP possui dois tipos de capacidades. O tipo comum é denominado **capacidade de dados**. Ele pode ser usado para prover acesso aos objetos, mas os únicos direitos fornecidos são os direitos-padrão de leitura, escrita ou execução dos segmentos de armazenamento individuais associados ao objeto. As capacidades de dados são interpretadas pelo microcódigo na máquina CAP.

O segundo tipo de capacidade é a chamada **capacidade de software**, que é protegida, mas não interpretada, pelo microcódigo do CAP. Ela é interpretada por um procedimento *protegido* (ou seja, privilegiado), que pode ser escrito por um programador de aplicação como parte de um subsistema. Um tipo específico de ampliação de direitos está associado a um procedimento protegido. Ao executar o corpo do código de tal procedimento, um processo adquire temporariamente os direitos de ler ou escrever ele mesmo o conteúdo de uma capacidade de software. Esse tipo específico de ampliação de direitos corresponde a uma implementação das primitivas *seal* e *unseal* sobre as capacidades. Naturalmente, esse privilégio ainda está sujeito à verificação de tipo, para garantir que somente as capacidades de software para um tipo abstrato especificado são passadas a qualquer procedimento desse tipo. A confiança universal não é colocada em qualquer código além do microcódigo da máquina CAP. (Consulte as Notas Bibliográficas.)

A interpretação de uma capacidade de software fica com o subsistema, por meio dos procedimentos protegidos que contém. Esse esquema permite que uma série de políticas de proteção sejam implementadas. Embora um programador possa definir seus próprios procedimentos protegidos (qualquer um dos quais podendo estar incorreto), a segurança do sistema geral não pode ser comprometida. O sistema de proteção básico não permitirá que um procedimento protegido não verificado, definido pelo usuário, tenha acesso a quaisquer segmentos de armazenamento (ou capacidades) que não pertençam ao ambiente de proteção em que reside. A consequência mais séria de um procedimento protegido inseguro é uma quebra de proteção do subsistema pelo qual esse procedimento tem responsabilidade.

Os projetistas do sistema CAP observaram que o uso das capacidades de software proporcionava economias consideráveis na formulação e na implementação de políticas de proteção correspondentes aos requisitos dos recursos abstratos. Todavia, um projetista de subsistemas que deseja utilizar essa capacidade não pode estudar um manual de referência, como acontece com o Hydra. Em vez disso, ele precisa aprender os princípios e as técnicas de proteção, pois o sistema não lhe provê uma biblioteca de procedimentos.

14.9 Proteção baseada na linguagem

Até o ponto em que a proteção é fornecida nos sistemas computadorizados existentes, normalmente ela é alcançada por meio de um kernel do sistema operacional, que atua como um agente de segurança para inspecionar e validar cada tentativa de acessar um recurso protegido. Como a validação de acesso abrangente pode ser uma fonte considerável de trabalho adicional, precisamos dar suporte de hardware ou reduzir o custo de cada validação, ou então temos de aceitar que o projetista do sistema possa comprometer os objetivos da proteção. Será muito difícil satisfazer a todos esses objetivos se a flexibilidade para implementar as políticas de proteção for restrita pelos mecanismos de suporte fornecidos ou se os ambientes de proteção se tornarem maiores do que o necessário para assegurar maior eficiência operacional.

À medida que os sistemas operacionais se tornam mais complexos – particularmente à medida que tentam prover interfaces de usuário de nível mais alto –, os objetivos da proteção têm se tornado muito mais refinados. Os projetistas de sistemas de proteção têm se baseado bastante nas ideias originadas nas linguagens de proteção e, especialmente, nos conceitos de tipos de dados e objetos abstratos. Os sistemas de proteção agora se preocupam não apenas com a identidade de um recurso ao qual o acesso é tentado, mas também com a natureza funcional desse acesso. Nos sistemas de proteção mais novos, a preocupação com a função a ser invocada se estende além de um conjunto de funções definidas pelo sistema, como os métodos-padrão de acesso a arquivo, para incluir também as funções que podem ser definidas pelo usuário.

As políticas para uso de recursos também podem variar, dependendo da aplicação, e podem estar sujeitas a mudanças com o tempo. Por esses motivos, a proteção não pode mais ser considerada uma preocupação apenas do projetista de um sistema operacional. Ela também deve estar disponível como uma ferramenta para uso pelo projetista da aplicação, para que os recursos de um subsistema de aplicações possam ser protegidos contra violação ou influência de um erro.

14.9.1 Imposição baseada em compilador

Neste ponto, as linguagens de programação entram em cena. Especificar o controle desejado de acesso a um recurso compartilhado em um sistema significa fazer uma afirmação declarativa sobre o recurso. Esse tipo de afirmação pode ser integrado a uma linguagem por uma extensão de sua facilidade de uso de tipos. Quando a proteção é declarada junto com o tipo dos dados, o projetista de cada subsistema pode especificar seus requisitos para proteção, bem como sua necessidade de uso de outros recursos em um sistema. Essa especificação deve ser dada diretamente enquanto um programa é codificado e na linguagem em que o próprio programa é expresso. Essa técnica possui diversas vantagens significativas:

1. As necessidades de proteção são declaradas, em vez de programadas como uma sequência de chamadas de procedimentos de um sistema operacional.
2. Os requisitos de proteção podem ser declarados independentemente das facilidades fornecidas por um sistema operacional em particular.
3. Os meios de imposição não precisam ser fornecidos pelo projetista de um subsistema.
4. Uma notação declarativa é natural porque os privilégios de acesso estão bastante relacionados com o conceito linguístico do tipo de dados.

Diversas técnicas podem ser providas pela implementação de uma linguagem de programação para impor a proteção, mas qualquer uma delas precisa depender de algum grau de suporte de uma máquina subjacente e de seu sistema operacional. Por exemplo, suponha que uma linguagem seja usada para gerar código para execução no sistema Cambridge CAP. Nesse sistema, toda referência de armazenamento feita sobre o hardware ocorre indiretamente por meio de uma capacidade. Essa restrição impede que qualquer processo acesse um recurso fora de seu ambiente de proteção a qualquer momento. Entretanto, um programa pode impor restrições quaisquer sobre o modo como um recurso pode ser usado durante a execução de determinado segmento de código por algum processo. Podemos implementar essas restrições mais prontamente usando as capacidades de software fornecidas pelo CAP. A implementação de uma linguagem poderia prover procedimentos protegidos padronizados para interpretar as capacidades de software, que perceberiam as políticas de proteção que poderiam estar especificadas na linguagem. Esse esquema coloca a especificação da política à disposição dos programadores, enquanto os libera da implementação de sua imposição.

Mesmo que um sistema não ofereça um kernel de proteção tão poderoso quanto aqueles do Hydra ou do CAP, ainda existem mecanismos à disposição para a implementação de especificações de proteção em uma linguagem de programação. A distinção principal é que a *segurança* dessa proteção não será tão grande quanto a admitida pelo kernel de proteção, já que o mecanismo precisa contar com mais suposições sobre o estado operacional do sistema. Um compilador pode separar as referências para as quais pode certificar que nenhuma violação de proteção poderia ocorrer daquelas para as quais uma violação poderia ser possível, e elas podem ser tratadas de formas diferentes. A segurança proporcionada por essa forma de proteção conta com a suposição de que o código gerado pelo compilador não será modificado antes ou durante sua execução.

Quais, então, são os méritos relativos da imposição baseada unicamente em um kernel, ao contrário da imposição fornecida por um compilador?

■ **Segurança.** A imposição por um kernel provê um grau de segurança do próprio sistema de proteção mais alto do que a geração de código de verificação de proteção por um compilador. Em um esquema com o suporte do compilador, a segurança conta com a exatidão do tradutor, com algum mecanismo básico de gerência de armazenamento que proteja os segmentos dos quais o código compilado é executado, e, por último, com a segurança dos arquivos dos quais um programa é carregado. Algumas dessas considerações também se aplicam ao kernel de proteção como suporte do software, mas não tão intensamente, pois o kernel pode residir em segmentos fixos do armazenamento físico e pode ser carregado apenas de um arquivo designado. Com um sistema com capacidade reunida, em que todo o cálculo de endereço é realizado pelo hardware ou por um microprograma fixo, é possível haver uma segurança ainda maior. A proteção com suporte do hardware também é relativamente imune às violações de proteção que poderiam ocorrer como resultado de defeitos no hardware ou no software do sistema.

■ **Flexibilidade.** Existem limites para a flexibilidade de um kernel de proteção na implementação de uma política definida pelo usuário, embora ele possa conter facilidades adequadas para o sistema fornecer a imposição de suas próprias políticas. Com uma linguagem de programação, a política de proteção pode ser declarada e a imposição fornecida conforme a necessidade, por meio de uma implementação. Se uma linguagem não provê flexibilidade suficiente, ela pode ser estendida ou substituída, com menos perturbação de um sistema em serviço do que seria causada pela modificação do kernel de um sistema operacional.

■ **Eficiência.** A maior eficiência é obtida quando a imposição da proteção tem o suporte direto do hardware (ou microcódigo). Até o ponto em que o suporte do software é exigido, a imposição baseada na linguagem tem a vantagem de a imposição de acesso estática poder ser verificada off-line, no momento da compilação. Além disso, como um compilador inteligente pode ajustar o mecanismo de imposição a fim de atender a necessidade especificada, o custo adicional fixo das chamadas do kernel pode ser evitado.

Resumindo, a especificação da proteção em uma linguagem de programação permite a descrição de alto nível das políticas para a alocação e uso dos recursos. Uma implementação de linguagem pode prover software para imposição da proteção quando a verificação automática, com suporte do hardware, não estiver disponível. Além disso, ela pode interpretar especificações de proteção para gerar chamadas sobre qualquer sistema de proteção fornecido pelo hardware e pelo sistema operacional.

Um modo de tornar a proteção disponível ao programa de aplicação é por meio do uso de uma capacidade de software que poderia ser usada como um objeto de computação. Inerente a esse conceito está a ideia de que determinados componentes do programa poderiam ter o privilégio de criar ou examinar essas capacidades de software. Um programa de criação de capacidade seria capaz de executar uma operação primitiva que selaria uma estrutura de dados, tornando seu conteúdo inacessível a quaisquer componentes do programa que não mantivessem os privilégios seal ou unseal. Eles poderiam copiar a estrutura de dados ou passar seu endereço a outros componentes do programa, mas não poderiam obter acesso ao seu conteúdo. O motivo para introduzir essas capacidades de software é trazer um mecanismo de proteção para a linguagem de programação. O único problema com o conceito, conforme proposto, é que o uso das operações seal e unseal utiliza uma técnica de procedimento para a especificação da proteção. Uma notação de não procedimento, ou declarativa, parece ser o modo preferido para tornar a proteção disponível ao programador da aplicação.

É necessário um mecanismo de controle de acesso seguro e dinâmico para distribuição de capacidades aos recursos do sistema entre os processos do usuário. Para contribuir com a confiabilidade geral de um sistema, o mecanismo de controle de acesso deve ser seguro de usar. Para ser útil na prática, ele também deve ser razoavelmente eficaz. Esse requisito ocasionou o desenvolvimento de uma série de construções de linguagem que permitem ao programador declarar várias restrições sobre o uso de um recurso gerenciado específico. (Ver as Notas Bibliográficas.) Essas construções proveem mecanismos para três funções:

1. Distribuição de capacidades de forma segura e eficiente entre os processos do cliente. Em particular, os mecanismos garantem que um processo do usuário usará o recurso gerenciado apenas se recebeu uma capacidade para esse recurso.
2. Especificação do tipo de operações que determinado processo pode invocar em um recurso alocado (por exemplo, um leitor de um arquivo deverá ter permissão apenas para ler o arquivo, enquanto um escritor deverá ser capaz de ler e escrever). Não deve ser preciso conceder o mesmo conjunto de direitos a cada processo do usuário e deve ser impossível para um processo aumentar seu conjunto de direitos de acesso, exceto com a autorização do mecanismo de controle de acesso.
3. Especificação da ordem em que determinado processo pode invocar as diversas operações de um recurso (por exemplo, um arquivo precisa ser aberto antes de poder ser lido). Deve ser possível dar a dois processos restrições diferentes sobre a ordem em que podem invocar as operações do recurso alocado.

A incorporação dos conceitos de proteção nas linguagens de programação, como uma ferramenta prática para o projeto do sistema, está em sua infância. A proteção se tornará uma questão de maior preocupação para os projetistas de novos sistemas, com arquiteturas distribuídas e requisitos cada vez mais rigorosos sobre a segurança dos dados. Aí, então, a importância de notações de linguagem adequadas, para expressar os requisitos de proteção, será mais amplamente reconhecida.

14.9.2 Proteção em Java

Como a Java foi projetada para executar em um ambiente distribuído, a máquina virtual Java (JVM) apresenta vários mecanismos de proteção embutidos. Programas Java são compostos de **classes**, cada uma sendo uma coleção de campos de dados e funções (chamadas **métodos**) que operam sobre esses campos. A JVM carrega uma classe em resposta a uma requisição para criar instâncias (ou objetos) dessa classe. Um dos recursos mais novos e úteis da Java é seu suporte para carregar dinamicamente classes não confiáveis por uma rede e executar classes mutuamente não confiáveis dentro da mesma JVM.

Devido a essas capacidades da Java, a proteção é uma preocupação fundamental. As classes executando na mesma JVM podem ter diferentes origens e podem não ser igualmente confiáveis. Como resultado, impor a proteção na granularidade do processo JVM é insuficiente. De forma intuitiva, se uma requisição para abrir um arquivo tiver de ser permitida, isso dependerá de qual classe requisitou a abertura. O sistema operacional não tem esse conhecimento.

Assim, tais decisões de proteção são tratadas dentro da JVM. Quando a JVM carrega uma classe, ela atribui a classe a um domínio de proteção que dá as permissões dessa classe. O domínio de proteção em que a classe é atribuída depende do URL do qual a classe foi carregada e de quaisquer assinaturas digitais no arquivo de classe. (As assinaturas digitais são abordadas na [Seção 15.4.1.3](#)) Um arquivo de política configurável determina as permissões concedidas ao domínio (e suas classes). Por exemplo, as classes carregadas de um servidor confiável poderiam ser colocadas em um domínio de proteção que lhes permita acessar arquivos no diretório home do usuário, enquanto as classes carregadas de um servidor não confiável poderiam não receber qualquer permissão de arquivo.

Pode ser complicado para a JVM determinar que classe é responsável por uma requisição para acessar um recurso protegido. Os acessos são realizados de forma indireta, por meio de bibliotecas do sistema ou de outras classes. Por exemplo, considere uma classe sem permissão para abrir conexões de rede. Ela poderia chamar uma biblioteca do sistema para requisitar o carregamento do conteúdo de um URL. A JVM precisa decidir se abrirá ou não uma conexão de rede para essa requisição. Mas qual classe deverá ser usada para determinar se a conexão deve ser permitida: a aplicação ou a biblioteca do sistema?

A filosofia adotada na Java é exigir que a classe de biblioteca permita explicitamente uma conexão de rede. Em geral, para acessar um recurso protegido, algum método na sequência de chamada que resultou na requisição precisa declarar explicitamente o privilégio de acessar o recurso. Assim, esse método *toma responsabilidade* pela requisição; presume-se que ele também realizará quaisquer verificações necessárias para garantir a segurança da requisição. Nem todo método tem permissão para declarar um privilégio; ele só pode fazer isso se sua classe estiver em um domínio de proteção com permissão para exercer o privilégio.

Essa técnica de implementação é denominada **inspeção da pilha**. Cada thread na JVM possui uma pilha associada, com suas chamadas de método em andamento. Quando o código que o chamou potencialmente não for confiável, um método executará uma requisição de acesso dentro de um bloco `doPrivileged` para realizar o acesso ao recurso protegido direta ou indiretamente. `doPrivileged()` é um método estático na classe `AccessController`, que recebe uma classe com um método `run()` para invocar. Quando o bloco `doPrivileged` é iniciado, o quadro da pilha para esse método é anotado, para indicar esse fato. Depois, o conteúdo do bloco é executado. Quando, mais tarde, for requisitado um acesso a um recurso protegido, seja por esse método ou um método que ele chama, uma chamada a `checkPermissions()` é utilizada para invocar a inspeção da pilha, a fim de determinar se a requisição deve ser permitida. A inspeção examina os quadros da pilha na pilha da thread que chama, começando com o quadro acrescentado mais recentemente e seguindo em direção ao mais antigo. Se for encontrado inicialmente um quadro da pilha com a anotação `doPrivileged()`, então `checkPermissions()` retorna imediatamente, permitindo o acesso. Se for encontrado um quadro da pilha para o qual o acesso não está permitido, com base no domínio de proteção da classe do método, então `checkPermissions()` lança uma `AccessControlException`. Se a inspeção da pilha esgotar toda a pilha sem encontrar qualquer um desses tipos de quadro, então a permissão do acesso dependerá da implementação (por exemplo, algumas implementações da JVM podem permitir o acesso, outras implementações podem revogá-lo).

A inspeção da pilha é ilustrada na [Figura 14.9](#). Aqui, o método `gui()` de uma classe no domínio de proteção *applet não confiável* realiza duas operações, primeiro um `get()` e depois `open()`. A primeira é uma chamada do método `get()` de uma classe no domínio de proteção *carregador de URL*, que tem permissão para abrir [`open()`] sessões para os sites no domínio `lucent.com`, em particular um servidor proxy `proxy.lacent.com` para apanhar URLs. Por esse motivo, a chamada `get()`

do applet não confiável terá sucesso: a chamada `checkPermissions()` na biblioteca *networking* encontra o quadro da pilha do método `get()`, que realizou seu `open()` em um bloco `doPrivileged`. Todavia, a chamada `open()` do applet não confiável resultará em uma exceção, pois a chamada `checkPermissions()` não encontra uma anotação `doPrivileged` antes de encontrar um quadro da pilha do método `gui()`.

domínio de proteção: applet não confiável	carregador de URL	rede
permissão de socket: none	*.lucent.com:80, connect	any
classe: gui: ... get(url); open(addr); ...	get(URL u): ... doPrivileged { open('proxy.lucent.com:80'); } <request u from proxy> ...	open(Addr a): ... checkPermission(a, connect); connect(a);

FIGURA 14.9 Inspeção da pilha.

Naturalmente, para que a inspeção da pilha funcione, um programa não pode ser capaz de modificar as anotações em seu próprio quadro da pilha, nem fazer outras manipulações de inspeção da pilha. Essa é uma das diferenças mais importantes entre a Java e muitas outras linguagens (incluindo C++). Um programa Java não pode acessar a memória diretamente. Ele pode manipular apenas um objeto para o qual tem uma referência. As referências não podem ser forjadas, e as manipulações são feitas apenas por meio de interfaces bem definidas. A compatibilidade é imposta por meio de uma sofisticada coleção de verificações em tempo de carga e de execução. Como resultado, um objeto não pode manipular sua pilha em tempo de execução, pois não pode obter uma referência à pilha ou a outros componentes do sistema de proteção.

De um modo mais genérico, as verificações em tempo de carga e em tempo de execução da Java impõem **segurança de tipo** das classes Java. A segurança de tipo garante que as classes não possam tratar inteiros como ponteiros, escrever além do final de um array ou acessar a memória de outras maneiras quaisquer. Em vez disso, um programa só pode acessar um objeto por meio dos métodos definidos nesse objeto pela sua classe. Esse é o fundamento da proteção em Java, pois permite que uma classe **encapsule** e proteja seus dados e métodos contra outras classes carregadas na mesma JVM. Por exemplo, uma variável pode ser definida como privada, para somente a classe que a contém poder acessá-la, ou protegida, para que só possa ser acessada pela classe que a contém, por subclasses dessa classe ou por classes no mesmo pacote. A segurança de tipo garante que essas restrições poderão ser impostas.

14.10 Resumo

Os sistemas computadorizados contêm muitos objetos e eles precisam ser protegidos contra má utilização. Os objetos podem ser de hardware (como memória, tempo de CPU ou dispositivos de E/S) ou software (como arquivos, programas e tipos de dados abstratos). Um direito de acesso é a permissão para realizar uma operação sobre um objeto. Um domínio é um conjunto de direitos de acesso. Os processos são executados em domínios e podem usar qualquer um dos direitos de acesso no domínio para acessar e manipular objetos. Durante seu tempo de vida, um processo pode ser vinculado a um domínio de proteção ou pode ter permissão para passar de um domínio para outro.

A matriz de acesso é um modelo geral de proteção que provê um mecanismo para proteção sem impor uma política de proteção em particular no sistema ou em seus usuários. A separação de política e mecanismo é uma propriedade de projeto importante.

A matriz de acesso é esparsa. Ela é implementada como listas de acesso associadas a cada objeto ou como listas de capacidades associadas a cada domínio. Podemos incluir a projeção dinâmica no modelo de matriz de acesso considerando domínios e a própria matriz de acesso como objetos. A revogação dos direitos de acesso em um modelo de proteção dinâmico é mais fácil de implementar com um esquema de lista de acesso do que com uma lista de capacidades.

Os sistemas reais são muito mais limitados que o modelo geral e costumam prover proteção apenas para arquivos. O UNIX é representativo, provendo proteção de leitura, escrita e execução para o proprietário, grupo e público em geral para cada arquivo. O MULTICS usa uma estrutura de anel além do acesso ao arquivo. O Hydra, o sistema Cambridge CAP e o Mach são sistemas de capacidade que estendem a proteção aos objetos de software definidos pelo usuário. O Solaris 10 implementa o princípio do menor privilégio por meio do controle de acesso baseado em posição, uma forma de matriz de acesso.

A proteção baseada na linguagem provê uma arbitragem mais minuciosa das requisições e privilégios do que o sistema operacional é capaz de fornecer. Por exemplo, uma única JVM Java pode executar várias threads, cada uma com uma classe de proteção diferente. Ela impõe as requisições de recursos por meio de uma sofisticada inspeção da pilha e por meio da segurança de tipo da linguagem.

Exercícios práticos

- 14.1. Quais são as principais diferenças entre listas de capacidade e listas de acesso?
- 14.2. Um arquivo MCP do Burroughs B7000/B6000 pode ser marcado como dados confidenciais. Quando um arquivo desse tipo é excluído, sua área de armazenamento é sobreescrita com bits aleatórios. Para qual finalidade esse esquema seria útil?
- 14.3. Em um sistema de proteção com estrutura de anel, o nível 0 tem o maior acesso aos objetos, e o nível n (onde $n > 0$) tem menos direitos de acesso. Os direitos de acesso de um programa em determinado nível na estrutura de anel são considerados um conjunto de capacidades. Qual é a relação entre as capacidades de um domínio no nível j e um domínio no nível i a um objeto (para $j > i$)?
- 14.4. O sistema RC 4000, entre outros, definiu uma árvore de processos tal que todos os descendentes de um processo podem receber recursos (objetos) e direitos de acesso somente por seus ancestrais. Assim, um descendente nunca pode ter a capacidade de fazer algo que seus ancestrais não possam fazer. A raiz da árvore é o sistema operacional, que tem a capacidade de fazer qualquer coisa. Suponha que o conjunto de direitos de acesso seja representado por uma matriz de acesso, A . $A(x,y)$ define os direitos de acesso do processo x ao objeto y . Se x for um descendente de z , qual é a relação entre $A(x,y)$ e $A(z,y)$ para um objeto y qualquer?
- 14.5. Que problemas de proteção podem surgir se uma pilha compartilhada for usada para a passagem de parâmetros?
- 14.6. Considere um ambiente de computação onde um número exclusivo é associado a cada processo e cada objeto no sistema. Suponha que um processo com número n só possa acessar um objeto com número m se $n > m$. Que tipo de estrutura de proteção temos?
- 14.7. Considere um ambiente de computação onde um processo tenha recebido o privilégio de acessar um objeto apenas n vezes. Sugira um esquema para implementar essa política.
- 14.8. Se todos os direitos de acesso a um objeto forem removidos, o objeto não pode mais ser acessado. Nesse ponto, o objeto também deverá ser removido, e o espaço que ele ocupa deverá ser retornado ao sistema. Sugira uma implementação eficiente desse esquema.
- 14.9. Por que é difícil proteger um sistema em que os usuários têm permissão para realizar sua própria E/S?
- 14.10. As listas de capacidade normalmente são mantidas dentro de um espaço de endereços do usuário. Como o sistema garante que o usuário não possa modificar o conteúdo da lista?

Exercícios

- 14.11. Considere o esquema de proteção de anel no MULTICS. Se tivéssemos que implementar as chamadas de sistema de um sistema operacional típico e armazená-las em um segmento associado ao anel 0, quais deveriam ser os valores armazenados no campo de anel do descritor de segmento? O que acontecerá durante uma chamada de sistema quando um processo executando em um anel com número alto invocar um procedimento no anel 0?
- 14.12. A matriz de controle de acesso poderia ser usada para determinar se um processo pode passar, digamos, do domínio A para o domínio B e gozar dos privilégios de acesso do domínio B. Essa técnica é equivalente a incluir os privilégios de acesso do domínio B naqueles do domínio A?
- 14.13. Considere um sistema computadorizado em que “jogos de computador” podem ser jogados por alunos apenas entre as 22 horas e as 6 horas, pelos membros da faculdade entre 17 horas e 8 horas, e pelo pessoal do centro de computação em todos os horários. Sugira um esquema para implementar essa política de forma eficiente.
- 14.14. Que recursos do hardware são necessários em um sistema computadorizado para a manipulação de capacidade eficiente? Podem ser usados para a proteção da memória?
- 14.15. Discuta os pontos fortes e fracos da implementação de uma matriz de acesso usando listas associadas a objetos.
- 14.16. Discuta os pontos fortes e fracos da implementação de uma matriz de acesso usando capacidades que são associadas aos domínios.
- 14.17. Explique por que um sistema baseado em capacidade, como o Hydra, fornece maior flexibilidade do que o esquema de proteção de anel na imposição das políticas de proteção.
- 14.18. Discuta a necessidade de amplificação de direitos no Hydra. Como essa prática se compara com as chamadas entre anéis em um esquema de proteção em estrutura de anel?
- 14.19. Qual é o princípio “precisa saber”? Por que é importante que um sistema de proteção esteja em conformidade com esse princípio?
- 14.20. Discuta quais dos seguintes sistemas permitem que os projetistas de módulo imponham o princípio “precisa saber”.
- Esquema de proteção em estrutura de anel do MULTICS.
 - Capacidades do Hydra.
 - Esquema de inspeção de pilha da JVM.
- 14.21. Descreva como o modelo de proteção Java seria sacrificado se um programa Java tivesse permissão para alterar diretamente as anotações do seu frame de pilha.
- 14.22. Qual é a semelhança entre a facilidade de matriz de acesso e a facilidade de controle de acesso baseada em posição? Qual é a diferença?
- 14.23. Como o princípio do menor privilégio ajuda na criação dos sistemas de proteção?
- 14.24. Como os sistemas que implementam o princípio do menor privilégio ainda têm falhas de proteção que levam a violações de segurança?

Notas bibliográficas

O modelo de proteção de matriz de acesso entre domínios e objetos foi desenvolvido por [Lampson \[1969\]](#) e [Lampson \[1971\]](#). [Popek \[1974\]](#) e [Saltzer e Schroeder \[1975\]](#) proveram excelentes análises sobre o assunto de proteção. [Harrison e outros \[1976\]](#) usaram uma versão formal desse modelo para permitir que provassem matematicamente as propriedades de um sistema de proteção.

O conceito de capacidade evoluiu das *palavras-código* de Iliffe e Jodeit, que foram implementadas no computador da Rice University ([Iliffe e Jodeit \[1962\]](#)). O termo *capacidade* foi introduzido por [Dennis e Horn \[1966\]](#).

O sistema Hydra foi descrito por [Wulf e outros \[1981\]](#). O sistema CAP foi descrito por [Needham e Walker \[1977\]](#). [Organick \[1972\]](#) discutiu o sistema de proteção com estrutura de anel do MULTICS.

A revogação foi discutida por [Redell e Fabry \[1974\]](#), [Cohen e Jefferson \[1975\]](#) e [Ekanadham e Bernstein \[1979\]](#). O princípio da separação entre política e mecanismo foi defendido pelo projetista do Hydra ([Levin e outros \[1975\]](#)). O problema de confinamento foi discutido pela primeira vez por [Lampson \[1973\]](#) e examinado mais a fundo por [Lipner \[1975\]](#).

O uso de linguagens de nível mais alto para especificar o controle de acesso foi sugerido inicialmente por [Morris \[1973\]](#), que propôs o uso das operações *seal* e *unseal* discutidas na [Seção 14.9](#). [Kieburz e Silberschatz \[1978\]](#), [Kieburz e Silberschatz \[1983\]](#) e [McGraw e Andrews \[1979\]](#) propuseram diversas construções da linguagem para lidar com os esquemas dinâmicos gerais do gerenciamento de recursos. [Jones e Liskov \[1978\]](#) consideraram como um esquema de controle de acesso estático pode ser incorporado em uma linguagem de programação que aceite tipos de dados abstratos. O uso do suporte mínimo do sistema operacional para impor a proteção foi defendido pelo Exokernel Project ([Ganger e outros \[2002\]](#), [Kaashoek e outros \[1997\]](#)). A extensibilidade do código do sistema por mecanismos de proteção baseados em linguagem foi discutida em [Bershad e outros \[1995\]](#). Outras técnicas para impor a proteção incluem sandboxing ([Goldberg e outros \[1996\]](#)) e isolamento de falha no software ([Wahbe e outros \[1993b\]](#)). As questões de redução do overhead associado aos custos de proteção e permissão do acesso em nível de usuário aos dispositivos em rede foram discutidas em [McCanne e Jacobson \[1993\]](#) e [Basu e outros \[1995\]](#).

Uma análise mais detalhada da inspeção da pilha, incluindo comparações com outras técnicas de segurança Java, poderá ser encontrada em [Wallach e outros \[1997\]](#) e [Gong e outros \[1997\]](#).

CAPÍTULO 15

Segurança

Proteção, conforme discutimos no [Capítulo 14](#), é estritamente um problema *interno*. Como provemos acesso controlado aos programas e dados armazenados em um sistema computadorizado? **Segurança**, por outro lado, exige não apenas um sistema de proteção adequado, mas também uma consideração do ambiente *externo* dentro do qual o sistema opera. A proteção do sistema é ineficaz se a autenticação do usuário está comprometida ou o programa é executado por um usuário não autorizado.

Os recursos do computador precisam ser protegidos contra acesso não autorizado, destruição ou alteração maliciosa, e introdução accidental de inconsistência. Esses recursos incluem informações armazenadas no sistema (tanto dados quanto código), além da CPU, memória, discos, fitas e rede, que são o computador. Neste capítulo, começamos examinando maneiras como os recursos podem ser mal utilizados accidental ou propositadamente. Depois, exploramos um habilitador de segurança fundamental - a criptografia. Finalmente, examinamos os mecanismos para proteger contra ou detectar ataques.

OBJETIVOS DO CAPÍTULO

- Discutir as ameaças à segurança e ataques.
- Explicar os fundamentos da criptografia, autenticação e hashing.
- Examinar os usos da criptografia na computação.
- Descrever as diversas contramedidas para os ataques à segurança.

15.1 O problema da segurança

Em muitas aplicações, garantir a segurança do sistema computadorizado é um esforço bastante considerável. Grandes sistemas comerciais, contendo dados de folha de pagamento ou outros, são alvos convidativos para os ladrões. Os sistemas que contêm dados pertencentes a operações corporativas podem ser de interesse para concorrentes inescrupulosos. Além do mais, a perda de tais dados, seja por acidente ou por fraude, pode prejudicar seriamente a capacidade de funcionamento da empresa.

No [Capítulo 14](#), discutimos os mecanismos que o sistema operacional pode prover (com ajuda apropriada do hardware) que permitem aos usuários protegerem seus recursos (normalmente, programas e dados). Esses mecanismos funcionam bem somente enquanto os usuários estão em conformidade com o uso pretendido e o acesso a esses recursos. Dizemos que um sistema é **seguro** se seus recursos forem usados e acessados conforme o pretendido, sob todas as circunstâncias. Infelizmente, a segurança total não poderá ser alcançada. Apesar disso, precisamos ter mecanismos para tornar as violações de segurança uma ocorrência rara, e não uma norma.

Violações de segurança (ou mau uso) do sistema podem ser categorizadas como intencionais (maliciosas) ou acidentais. É mais fácil proteger contra mau uso accidental do que contra mau uso malicioso. Em sua maior parte, os mecanismos de proteção são o núcleo da proteção contra acidentes. A lista a seguir inclui várias formas de violações de segurança acidentais ou maliciosas. Devemos notar que, em nossa discussão de segurança, usamos os termos *intruso* e *cracker* para aqueles que tentam quebrar a segurança. Além disso, uma **ameaça** é o potencial para uma violação de segurança, como a descoberta de uma vulnerabilidade, enquanto um **ataque** é a tentativa de quebrar a segurança.

- **Quebra de confidencialidade.** Esse tipo de violação envolve a leitura não autorizada de dados (ou roubo de informação). Normalmente, uma quebra de confidencialidade é o objetivo de um intruso. Capturar dados secretos de um sistema ou de um stream de dados, como informações de cartão de crédito ou informações de identidade para roubo de identidade, pode resultar diretamente em dinheiro para o intruso.
- **Quebra de integridade.** Essa violação envolve modificação não autorizada dos dados. Esses ataques podem, por exemplo, resultar em passagem de responsabilidade a uma parte inocente ou modificação do código-fonte de uma aplicação comercial importante.
- **Quebra de disponibilidade.** Essa violação envolve destruição não autorizada de dados. Alguns crackers preferem mexer e ganhar status ou conseguir direitos do que ganhar financeiramente. A modificação de site é um exemplo comum desse tipo de quebra de segurança.
- **Roubo de serviço.** Essa violação envolve o uso não autorizado dos recursos. Por exemplo, um intruso (ou programa de intrusão) pode instalar um daemon em um sistema, que atua como servidor de arquivos.
- **Negação de serviço.** Essa violação envolve impedir o uso legítimo do sistema. Os ataques de **negação de serviço**, ou **DOS** (Denial-Of-Service), às vezes são acidentais. O worm original da Internet se transformou em um ataque DOS quando um bug falhou ao atrasar sua rápida divulgação. Discutimos os ataques de DOS na [Seção 15.3.3](#).

Os atacantes utilizam diversos métodos-padrão em suas tentativas de quebrar a segurança. O mais comum é o **mascaramento**, em que um participante de uma comunicação finge ser outro alguém (outro host ou outra pessoa). Com o mascaramento, os atacantes quebram a **autenticação**, a exatidão da identificação; eles podem então obter acesso que normalmente não seria permitido ou escalar seus privilégios – obter privilégios aos quais normalmente não teriam concessão. Outro ataque comum é reproduzir uma troca de dados capturada. Um **ataque de reprodução** consiste em repetição maliciosa ou fraudulenta de uma transmissão de dados válida. Às vezes, a reprodução compreende um ataque inteiro – por exemplo, em uma repetição de uma requisição para transferir dinheiro. Porém, frequentemente, isso é feito junto com a **modificação de mensagem**, novamente para escalar privilégios. Considere o dano que poderia ser feito se uma requisição de autenticação tivesse a informação de um usuário legítimo substituída pela informação de um usuário não autorizado. Outro tipo de ataque é o **ataque do homem no meio**, em que um atacante entra no fluxo de dados de uma comunicação fingindo ser o emissor para o receptor, e vice-versa. Em uma comunicação em rede, um ataque do homem no meio pode ser precedido por um **sequestro de sessão**, em que uma sessão de comunicação ativa é interceptada. Vários métodos de ataque são representados na [Figura 15.1](#).

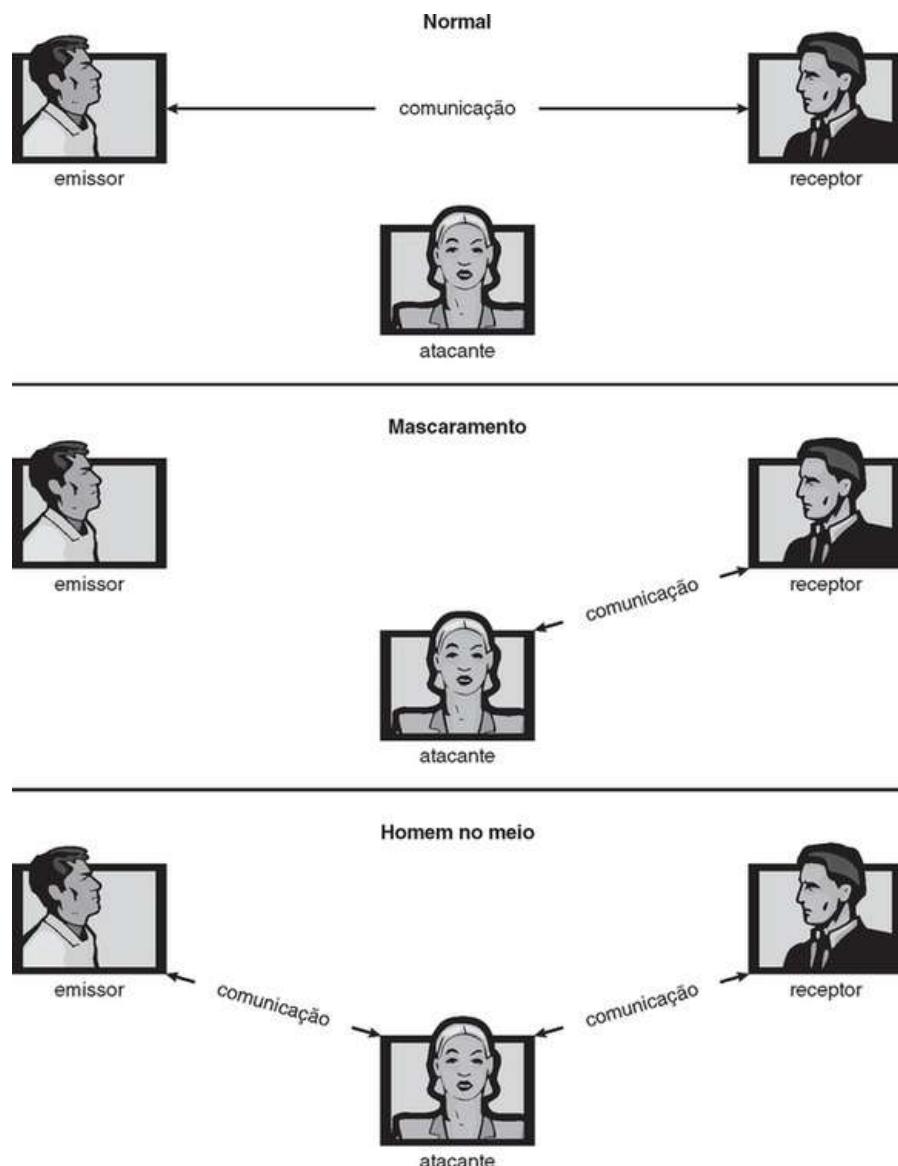


FIGURA 15.1 Ataques de segurança comuns.

Como já sugerimos, a proteção absoluta do sistema contra abuso malicioso não é possível, mas o custo para o intruso pode se tornar suficientemente alto para impedir a maioria deles. Em alguns casos, como em um ataque de negação de serviço, é preferível impedir o ataque, mas é suficiente detectar o ataque para que contramedidas possam ser tomadas.

Para proteger o sistema, temos de tomar medidas de segurança em quatro níveis:

- Físico.** As instalações contendo os sistemas computadorizados precisam ser fisicamente protegidas contra a entrada armada ou sorrateira de intrusos. As salas de sistemas e os terminais ou estações de trabalho que têm acesso às máquinas precisam ser protegidos.
- Humano.** A autorização precisa ser feita cuidadosamente, para garantir que somente os usuários apropriados tenham acesso ao sistema. Contudo, até mesmo os usuários autorizados podem ser “encorajados” a permitir que outros usem seu acesso (em troca de um suborno, por exemplo). Eles também podem ser enganados para permitir o acesso via **engenharia social**. Um tipo de ataque de engenharia social é o **phishing**. Aqui, um correio eletrônico ou página Web de aparência legítima induz um usuário a entrar com informações confidenciais. Outra técnica é chamada de **dumpster diving**, um termo genérico para a tentativa de colher informações a fim de obter acesso não autorizado ao computador (procurando no lixo, encontrando catálogos ou encontrando notas com senhas, por exemplo). Esses problemas de segurança são questões de gerenciamento e pessoais, e não problemas pertencentes aos sistemas operacionais.
- Sistema operacional.** O sistema precisa se proteger contra brechas de segurança accidentais ou propositais. Um processo descontrolado poderia se constituir em um ataque accidental de negação de serviço. Uma consulta a um serviço poderia revelar senhas. Um estouro de pilha poderia permitir a partida de um processo não autorizado. A lista de brechas possíveis é quase infinita.

4. **Rede.** Muitos dados de computador nos sistemas modernos atravessam linhas alugadas privadas, linhas compartilhadas como a Internet, conexões sem fio ou linhas discadas. A interceptação desses dados poderia ser tão prejudicial quanto a invasão de um computador; e a interrupção das comunicações poderia constituir um ataque de negação de serviço (denial-of-service) remoto, diminuindo o uso dos usuários e a confiança no sistema.

A segurança nos dois primeiros níveis precisa ser mantida se a segurança do sistema operacional tiver de ser garantida. Uma fraqueza em um alto nível de segurança (físico ou humano) permite frustrar as medidas de segurança de baixo nível estritas (sistema operacional). Assim, o velho adágio de que uma cadeia é tão fraca quanto seu elo mais fraco é especialmente verdadeiro na segurança do sistema. Todos esses aspectos precisam ser enfatizados para que a segurança seja mantida.

Além do mais, o hardware do sistema precisa prover proteção ([Capítulo 14](#)) para permitir a implementação das medidas de segurança. Sem a capacidade de autorizar usuários e processos, controlar seu acesso e registrar suas atividades em log, seria impossível para um sistema operacional implementar medidas de segurança ou executar de forma segura. Os recursos de proteção do hardware são necessários para dar suporte a um esquema de proteção geral. Por exemplo, um sistema sem proteção de memória não pode ser seguro. Novos recursos de hardware estão permitindo que os sistemas se tornem mais seguros, conforme discutiremos.

Infelizmente, pouca coisa na segurança é simples. Enquanto os intrusos exploram as vulnerabilidades de segurança, contramedidas de segurança são criadas e implantadas. Isso torna os intrusos mais sofisticados em seus ataques. Por exemplo, incidentes de segurança recentes incluem o uso de spyware para fornecer um canal para spam nos sistemas inocentes (discutimos essa prática na [Seção 15.2](#)). Esse jogo de gato e rato provavelmente continuará, com mais ferramentas de segurança necessárias para bloquear a escalada de técnicas e atividades dos intrusos.

No restante deste capítulo, focalizamos a segurança nos níveis de rede e de sistema operacional. A segurança nos níveis físico e humano, embora importante, está além do escopo deste texto. A segurança dentro do sistema operacional e entre os sistemas operacionais é implementada de várias maneiras, variando de senhas para autenticação pela proteção contra vírus até detecção de intrusões. Começaremos com uma exploração das ameaças à segurança.

15.2 Ameaças ao programa

Os processos, junto com o kernel, são os únicos meios de realizar trabalho em um computador. Portanto, escrever um programa que crie uma brecha de segurança ou fazer um processo normal mudar seu comportamento e criar uma brecha é um objetivo comum dos crackers. Na verdade, até mesmo a maioria dos eventos de segurança que não são programas tem como objetivo causar uma ameaça ao programa. Por exemplo, embora seja útil efetuar o login em um sistema sem autorização, é muito mais útil deixar para trás um daemon de **back-door** que forneça informações ou permita o acesso fácil mesmo que a exploração original seja bloqueada. Nesta seção, descrevemos os métodos comuns pelos quais os programas causam brechas de segurança. Observe que existe uma grande variação nas convenções de nomeação das brechas de segurança e que usamos os termos mais comuns ou descritivos.

15.2.1 Cavalo de Troia

Muitos sistemas possuem mecanismos para permitir que programas escritos pelos usuários sejam executados por outros usuários. Se esses programas forem executados em um domínio que provê os direitos de acesso do usuário executando, os outros usuários podem abusar desses direitos. Um programa editor de textos, por exemplo, pode incluir código para procurar o arquivo a ser editado em busca de certas palavras-chave. Se for encontrada alguma, o arquivo inteiro poderá ser copiado para uma área especial, acessível ao criador do editor de textos. Um segmento de código que abusa do seu ambiente é chamado de **cavalo de Troia** (trojan horse). Caminhos de busca longos, como são comuns nos sistemas UNIX, aumentam o problema do cavalo de Troia. O caminho de busca lista o conjunto de diretórios a buscar quando é informado um nome de programa ambíguo. O caminho é pesquisado em busca de um arquivo com esse nome, e o arquivo é executado. Todos os diretórios no caminho de busca precisam ser protegidos ou, então, um cavalo de Troia poderia ser colocado no caminho do usuário e executado acidentalmente.

Por exemplo, considere o uso do caractere “.” em um caminho de busca. O “.” diz ao shell para incluir o diretório atual na busca. Assim, se um usuário possui “.” em seu caminho de busca, definiu seu diretório atual para o diretório de um amigo e entra com o nome de um comando normal do sistema, o comando pode ser executado do diretório do amigo. O programa executaria dentro do domínio do usuário, permitindo que fizesse qualquer coisa que o usuário tem permissão para fazer, inclusive excluir os arquivos do usuário, por exemplo.

Uma variação do cavalo de Troia é um programa que simula um programa de login. Um usuário confiante começa a se conectar em um terminal e observa que aparentemente errou ao digitar sua senha. Ele tenta novamente e não tem sucesso. O que aconteceu é que sua chave de autenticação e senha foram roubadas pelo simulador de login, que foi deixado executando no terminal pelo ladrão. O simulador armazenou a senha, imprimiu uma mensagem de erro de login e saiu; o usuário, então, recebe um aviso de login genuíno. Esse tipo de ataque pode ser impedido fazendo o sistema operacional imprimir uma mensagem de uso no final de uma sessão interativa ou por uma sequência de teclas não interceptáveis, como a combinação Ctrl-Alt-Del usada por todos os sistemas operacionais Windows.

Outra variação do cavalo de Troia é o **spyware**. O spyware às vezes acompanha um programa que o usuário decidiu instalar. Frequentemente, ele vem com programas freeware ou shareware, mas às vezes está incluído no software comercial. O objetivo do spyware é fazer o download de anúncios e exibir no sistema do usuário, criar **janelas pop-up no navegador** quando certos sites forem visitados ou capturar informações do sistema do usuário e retorná-las a um site central. Esta última prática é um exemplo de uma categoria geral de ataques conhecida como **canais cobertos**, em que ocorre uma comunicação sorrateira. Por exemplo, a instalação de um programa aparentemente inocente em um sistema Windows poderia resultar na carga de um daemon de spyware. O spyware poderia entrar em contato com um site central, receber uma mensagem e uma lista de endereços de destinatário e entregar a mensagem de spam para esses usuários a partir da máquina Windows. Esse processo continua até que o usuário descubra o spyware. Normalmente, o spyware não é descoberto. Em 2004, estimou-se que 80% dos spams estavam sendo entregues por esse método. Esse roubo de serviço nem sequer é considerado crime na maioria dos países!

O spyware é um exemplo micro de um macroproblema: violação do princípio de menor privilégio. Sob a maioria das circunstâncias, um usuário de um sistema operacional não precisa instalar daemons de rede. Esses daemons são instalados por meio de dois erros. Primeiro, um usuário pode executar com mais privilégios do que o necessário (por exemplo, como administrador), permitindo que os programas executados tenham mais acesso ao sistema do que é necessário. Esse é um caso de erro humano - um ponto fraco comum na segurança. Segundo, um sistema operacional pode permitir, como padrão, mais privilégios do que o usuário normal precisa. Esse é um caso de decisões fracas no projeto do sistema operacional. Um sistema operacional (e, na realidade, o software em geral) deve permitir um controle minucioso do acesso e segurança, mas também precisa ser fácil de gerenciar e entender. Medidas de segurança inconvenientes ou inadequadas tendem a ser

contornadas, causando uma fraqueza geral da segurança que elas foram criadas para implementar.

15.2.2 Porta de armadilha

O projetista de um programa ou sistema poderia deixar um furo no software que somente ele é capaz de usar. Esse tipo de brecha de segurança, ou porta de armadilha (**trap door**), foi mostrado no filme *War Games (Jogos de Guerra)*. Por exemplo, o código poderia procurar um ID de usuário ou senha específica e contornar os procedimentos de segurança normais. Programadores têm sido presos por extraviar dinheiro dos bancos, incluindo erros de arredondamento em seu código, fazendo o meio centavo ocasional ser creditado em suas contas. Esse crédito na conta pode se acumular e chegar a uma grande soma de dinheiro, considerando a quantidade de transações que um grande banco executa.

Uma porta de armadilha inteligente poderia ser incluída em um compilador. Um compilador poderia gerar código objeto padrão e também uma porta de armadilha, independentemente do código-fonte compilado. Essa atividade é particularmente nefasta, porque uma busca do código-fonte do programa não revelará problema algum. Somente o código-fonte do compilador teria a informação.

As portas de armadilha impõem um problema porque, para detectá-las, temos de analisar todo o código-fonte para todos os componentes de um sistema. Dado que os sistemas de software podem consistir em milhões de linhas de código, essa análise não é feita com frequência e, em geral, nem sequer é feita!

15.2.3 Bomba lógica

Considere um programa que inicia um incidente de segurança apenas sob determinadas circunstâncias. Ele seria difícil de detectar, pois sob operações normais não haveria brecha de segurança. Contudo, quando um conjunto de parâmetros predefinido fosse atendido, a brecha de segurança seria criada. Esse cenário é conhecido como **bomba lógica**. Um programador, por exemplo, poderia escrever um código para detectar se ainda está empregado; se essa verificação falhar, um daemon poderia ser gerado para permitir o acesso remoto ou um código poderia ser iniciado para causar danos no site.

15.2.4 Estouro de pilha e buffer

O ataque por estouro de pilha ou buffer é o modo mais comum para um atacante fora do sistema ou uma conexão de rede ou dial-up obter acesso não autorizado ao sistema de destino. Um usuário autorizado do sistema também pode usar essa exploração para escalada de privilégios.

Basicamente, o ataque explora um bug em um programa. O bug pode ser um caso simples de má programação, em que o programador deixou de codificar a verificação de limites em um campo de entrada. Nesse caso, o atacante envia mais dados do que o programa estava esperando. Usando tentativa e erro ou examinando o código-fonte do programa atacado, se estiver disponível, o atacante determina a vulnerabilidade e escreve um programa para fazer o seguinte:

1. Estourar um campo de entrada, argumento da linha de comandos ou buffer de entrada - por exemplo, em um daemon de rede - até que escreva sobre a pilha.
2. Modificar o endereço de retorno atual na pilha pelo endereço do código de exploração carregado na etapa 3.
3. Escrever um código simples para o próximo espaço na pilha, que inclui os comandos que o atacante deseja executar - por exemplo, iniciar um shell.

O resultado da execução desse programa de ataque será a execução de um shell root ou outro comando privilegiado.

Por exemplo, se um formulário de página Web espera que um nome de usuário seja inserido em um campo, o atacante poderia enviar o nome do usuário, mais caracteres extras para estourar o buffer e alcançar a pilha, mais um novo endereço de retorno para carregar na pilha, mais o código que o atacante deseja executar. Quando a sub-rotina de leitura de buffer retorna da execução, o endereço de retorno é o código de exploração, e o código é executado.

Vejamos uma exploração de estouro de buffer com mais detalhes. Considere o programa C simples mostrado na [Figura 15.2](#). Esse programa cria um array de caracteres de tamanho `BUFFER_SIZE` e copia o conteúdo do parâmetro fornecido na linha de comandos - `argv[1]`. Desde que o tamanho desse parâmetro seja menor que `BUFFERS_SIZE` (precisamos de um byte para armazenar o término nulo), esse programa funciona corretamente. Mas considere o que acontece se o parâmetro fornecido na linha de comandos for maior que `BUFFER_SIZE`. Nesse cenário, a função `strcpy()` começará a copiar a partir de `argv[1]` até encontrar um término nulo (\0) ou até que o programa falhe. Assim, esse programa sofre de um problema de estouro de buffer em potencial, em que os dados copiados estouram o array `buffer`.

```

#include <stdio.h>
#define BUFFER_SIZE 256

int main(int argc, char *argv[ ])
{
    char buffer[BUFFER_SIZE];

    if (argc < 2)
        return -1;
    else {
        strcpy(buffer, argv[1]);
        return 0;
    }
}

```

FIGURA 15.2 Programa C com condição de estouro de buffer.

Observe que um programador cuidadoso poderia ter realizado a verificação de limites sobre o tamanho de `argv[1]` usando a função `strncpy()` em vez de `strcpy()`, substituindo a linha “`strcpy(buffer, argv[1]);`” por “`strncpy(buffer, argv[1], sizeof(buffer)-1);`”. Infelizmente, uma boa verificação de limites é uma exceção, em vez da norma.

Além do mais, a falta de verificação de limites não é a única causa possível do comportamento do programa na [Figura 15.2](#). O programa, em vez disso, poderia ter sido projetado cuidadosamente para comprometer a integridade do sistema. Agora, considere as vulnerabilidades de segurança possíveis de um estouro de buffer.

Quando uma função é invocada em uma arquitetura de computador típica, as variáveis definidas localmente à função (também conhecidas como **variáveis automáticas**), os parâmetros passados à função e o endereço ao qual o controle retorna depois que a função termina são armazenados em um **quadro de pilha**. O layout de um quadro de pilha típico aparece na [Figura 15.3](#). Examinando o quadro de pilha de cima para baixo, primeiro definimos os parâmetros passados à função, seguidos por quaisquer variáveis automáticas declaradas na função. Em seguida, vemos o **ponteiro de quadro**, que é o endereço do início do quadro de pilha. Finalmente, temos o endereço de retorno, que especifica onde retornar o controle quando a função termina. O ponteiro de frame precisa ser salvo na pilha, pois o valor do ponteiro de pilha pode variar durante a chamada de função; o ponteiro de quadro salvo permite um acesso relativo aos parâmetros e variáveis automáticas.

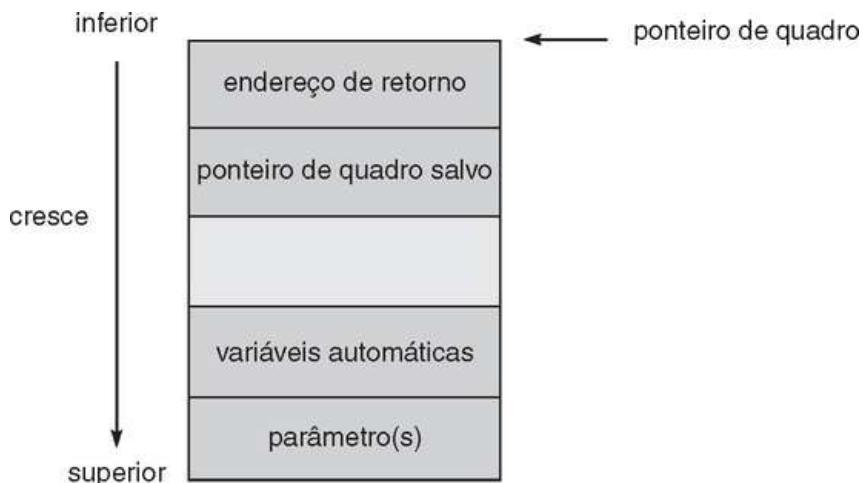


FIGURA 15.3 O layout de um quadro de pilha típico.

Dado o layout de memória-padrão, um cracker poderia executar um ataque de estouro de buffer. Seu objetivo é substituir o endereço de retorno no quadro de pilha de modo que agora aponte para o segmento de código contendo o programa de ataque.

O programador primeiro escreve um segmento de código curto, como o seguinte:

```

#include <stdio.h>

int main(int argc, char *argv[ ])
{
    execvp('\\bin\\sh', '\\bin \\sh', NULL);
    return 0;
}

```

Usando a chamada de sistema `execvp()`, esse segmento de código cria um processo shell. Se o programa sendo atacado for executado com permissões em nível de sistema, esse shell recém-criado ganhará acesso completo ao sistema. Naturalmente, o segmento de código poderia fazer qualquer coisa permitida pelos privilégios do processo atacado. Esse segmento de código é então compilado, de modo que as instruções em linguagem assembly podem ser modificadas. A principal modificação é remover recursos desnecessários no código, reduzindo, assim, o tamanho do código para que possa caber em um quadro de pilha. Esse fragmento de código montado agora é uma sequência binária que estará no centro do ataque.

Veja novamente o programa mostrado na [Figura 15.2](#). Vamos supor que, quando a função `main()` é chamada nesse programa, o quadro de pilha aparece conforme mostra a [Figura 15.4\(a\)](#). Usando o depurador, o programador então encontra o endereço de `buffer[0]` na pilha. Esse endereço é o local do código que o atacante deseja executar, de modo que a sequência binária é anexada com a quantidade necessária de instruções NO_OP (nenhuma operação) para preencher o quadro de pilha até o local do endereço de retorno; e o local de `buffer[0]`, o novo endereço de retorno, é acrescentado. O ataque termina quando o atacante indica essa sequência binária construída como entrada para o processo. O processo, então, copia a sequência binária de `argv[1]` para a posição `buffer[0]` no quadro de pilha. Agora, quando o controle retorna de `main()`, em vez de retornar para o local especificado pelo antigo valor do endereço de retorno, retornamos ao código shell modificado, que é executado com direitos de acesso do processo atacado! A [Figura 15.4\(b\)](#) contém o código do shell modificado.

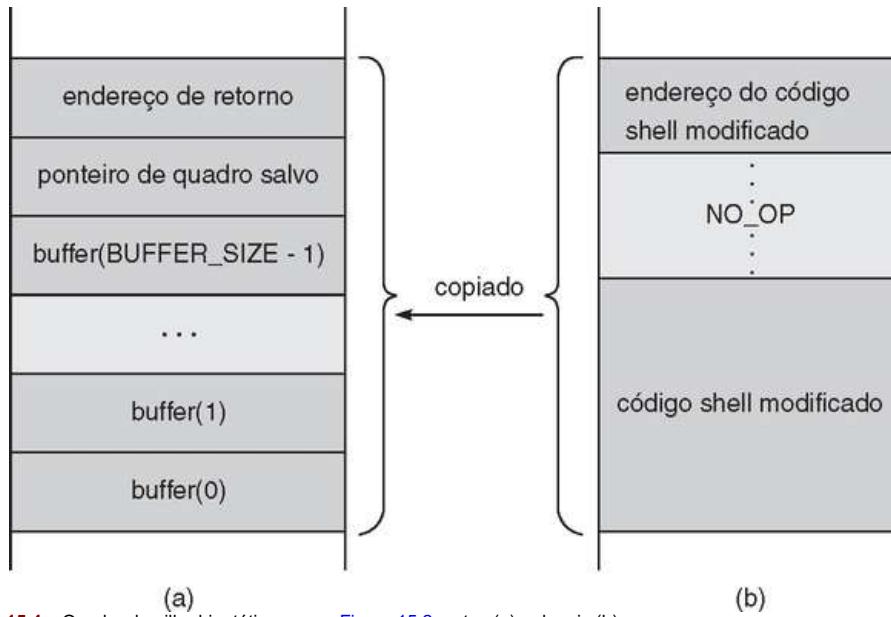


FIGURA 15.4 Quadro de pilha hipotético para a [Figura 15.2](#), antes (a) e depois (b).

Existem muitas maneiras de explorar problemas em potencial de estouro de buffer. Neste exemplo, consideraremos a possibilidade de que o programa sendo atacado – o código mostrado na [Figura 15.2](#) – executou com permissões em nível de sistema. Contudo, o segmento de código executado quando o valor do endereço de retorno foi modificado poderia realizar qualquer tipo de ato malicioso, como excluir arquivos, abrir portas de rede para mais exploração, e assim por diante.

Este ataque de exemplo de estouro de buffer revela que consideráveis conhecimentos e habilidades de programação são necessários para reconhecer o código explorável e depois explorá-lo. Infelizmente, não é preciso ser um grande programador para iniciar ataques à segurança. Em vez disso, um cracker pode determinar o bug e depois escrever uma exploração. Qualquer um com habilidades rudimentares em computação e acesso à exploração – um chamado **script kiddie** – pode tentar iniciar o ataque nos sistemas-alvo.

Um ataque de estouro de buffer é especialmente pernicioso, pois pode ser executado entre sistemas e pode atravessar canais de comunicação permitidos. Esses ataques podem ocorrer dentro de protocolos utilizados para a comunicação com a máquina-alvo e, portanto, podem ser difíceis de detectar e evitar. Eles podem ainda contornar a segurança acrescentada pelos firewalls ([Seção 15.7](#)).

Uma solução para esse problema é que a CPU tenha um recurso para desativar a execução do código em uma seção de pilha da memória. As versões recentes do chip SPARC da Sun incluem essa opção, e versões recentes do Solaris a habilitam. O endereço de retorno da rotina estourada ainda pode ser modificado; mas quando o endereço de retorno está dentro da pilha e o código de lá tenta executar, uma exceção é gerada, e o programa é interrompido com um erro.

Versões recentes dos chips AMD e Intel x86 incluem o recurso NX para impedir esse tipo de ataque. O uso do recurso tem o suporte de vários sistemas operacionais x86, incluindo Linux e Windows XP SP2. A implementação de hardware envolve o uso de um novo bit nas tabelas de página das CPUs. Esse bit marca a página associada como não executável, de modo que as instruções não possam ser lidas e executadas. À medida que esse recurso se tornar mais prevalente, os ataques de estouro de buffer deverão diminuir bastante.

15.2.5 Vírus

Outra forma de ataque ao computador é o **vírus**. Um vírus é um fragmento de código embutido em um programa legítimo. Os vírus são autorreplicantes e projetados para “infectar” outros programas. Eles podem causar destruição em um sistema, modificando ou destruindo arquivos e causando falhas no sistema e defeitos em programas. Assim como a maioria dos ataques de penetração, os vírus são bastante específicos das arquiteturas, sistemas operacionais e aplicações. Os vírus são um problema específico para os usuários de PC. Os sistemas operacionais UNIX e outros computadores multiusuário não costumam ser suscetíveis a vírus, pois os programas executáveis são protegidos contra escrita pelo sistema operacional. Mesmo que um vírus infecte tal programa, seus poderes são limitados, porque outros aspectos do sistema estão protegidos.

Os vírus normalmente nascem do correio eletrônico, com o spam sendo o vetor mais comum. Eles também podem se espalhar quando os usuários baixam programas com vírus de serviços de compartilhamento de arquivos da Internet ou trocam discos infectados.

Outra forma comum de transmissão de vírus usa arquivos do Microsoft Office, como documentos do Microsoft Word. Esses documentos podem conter *macros* (ou programas do Visual Basic) que os programas do pacote Office (Word, PowerPoint ou Excel) executarão automaticamente. Como esses programas executam sob a própria conta do usuário, as macros podem ser executadas de forma bastante irrestrita (por exemplo, excluindo arquivos do usuário à vontade). Normalmente, os vírus também são enviados por correio eletrônico à lista de contatos do usuário. Aqui está um exemplo de código que mostra a simplicidade da escrita de uma macro do Visual Basic que um vírus poderia usar para formatar o disco rígido de um computador Windows assim que o arquivo contendo a macro fosse aberto:

```
Sub AutoOpen( )
Dim oFS
Set oFS = CreateObject("Scripting.FileSystemObject")
vs = Shell("c:command.com /k format c:",vbHide)
End Sub
```

Como o vírus funciona? Quando um vírus atinge uma máquina-alvo, um programa conhecido como **colocador de vírus** insere o vírus no sistema. O colocador de vírus normalmente é um cavalo de Troia, executado por outros motivos, mas instalando o vírus como sua atividade básica. Uma vez instalado, o vírus pode fazer diversas coisas. Existem literalmente milhares de vírus, mas todos caem em diversas categorias principais. Observe que muitos vírus pertencem a mais de uma categoria.

■ **Arquivo.** Um vírus de arquivo-padrão infecta um sistema anexando-se a um arquivo. Ele muda o início do programa para que a execução salte para o seu código. Depois que for executado, ele retorna o controle para o programa, para que sua execução não seja observada. Os vírus de arquivo às vezes são conhecidos como vírus parasita, pois não deixam arquivos inteiros para trás e deixam os programas hospedeiros ainda funcionando.

■ **Boot.** Um vírus de boot infecta o setor de boot do sistema, executando toda vez que o sistema é inicializado e antes que o sistema operacional seja carregado. Ele procura outros meios inicializáveis (ou seja, disquetes) e os infecta. Esses vírus também são conhecidos como vírus de memória, pois não aparecem no sistema de arquivos. A [Figura 15.5](#) mostra como funciona um vírus de boot.

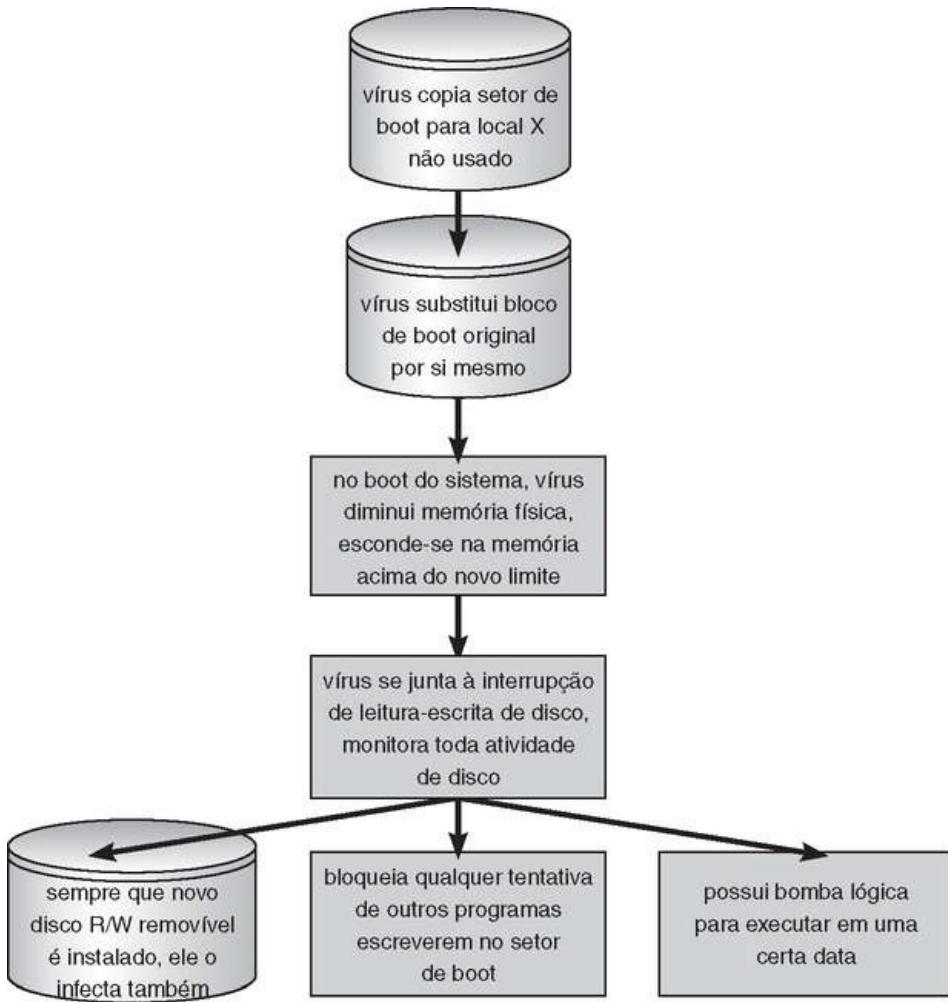


FIGURA 15.5 Um vírus de computador do setor de boot.

- **Macro.** A maioria dos vírus é escrita em uma linguagem de baixo nível, como assembly ou C. Os vírus de macro são escritos em uma linguagem de alto nível, como Visual Basic. Esses vírus são disparados quando um programa capaz de executar a macro é executado. Por exemplo, um vírus de macro poderia estar contido em um arquivo de planilha.
- **Código-fonte.** Um vírus de código-fonte procura o código-fonte e o modifica para incluir o vírus e ajudar a espalhar o vírus.
- **Polimórfico.** Um vírus polimórfico muda toda vez que é instalado, para evitar detecção por um software antivírus. As mudanças não afetam a funcionalidade do vírus, mas mudam a assinatura do vírus. Uma **assinatura de vírus** é um padrão que pode ser usado para identificar um vírus, normalmente uma série de bytes que compõem o código do vírus.
- **Criptografado.** Um vírus criptografado inclui o código de decriptografia junto com o vírus criptografado, novamente para evitar detecção. O vírus primeiro é decriptografado e depois é executado.
- **Furtivo.** Esse vírus enganoso tenta evitar detecção modificando partes do sistema que poderiam ser usadas para detectá-lo. Por exemplo, ele poderia modificar a chamada de sistema `read` para que, se um arquivo que ele modificou for lido, a forma original do código seja retornada, em vez do código infectado.
- **Tunelamento.** Esse vírus tenta evitar a detecção por um analisador de antivírus, instalando-se na cadeia do tratador de interrupção. Vírus semelhantes se instalam nos drivers de dispositivo.
- **Multipartite.** Um vírus desse tipo é capaz de infectar várias partes de um sistema, incluindo setores de boot, memória e arquivos. Isso dificulta sua detecção e contenção.
- **Blindado.** Um vírus blindado é codificado para tornar difícil para os pesquisadores de antivírus o desemaranharem e entenderem. Ele também pode ser compactado, para evitar detecção e desinfecção. Além disso, os disseminadores de vírus e outros arquivos cheios que fazem parte de uma infestação de vírus normalmente são escondidos por meio de atributos de arquivo ou nomes de arquivo que não podem ser vistos.

A grande variedade de vírus provavelmente continuará crescendo. De fato, em 2004, um vírus novo e bastante espalhado foi detectado. Ele explorava três bugs separados para a sua operação. Esse vírus começava infectando centenas de servidores Windows (incluindo muitos sites confiáveis) executando o Microsoft Internet Information Server (IIS). Qualquer navegador Web Microsoft

Explorer vulnerável visitando esses sites recebia um vírus de navegador com qualquer download. O vírus de navegador instalava vários programas de back door, incluindo um **registrator de toques de tecla**, que registra tudo o que for digitado no teclado (incluindo senhas e números de cartão de crédito). Ele também instalava um daemon para permitir acesso remoto ilimitado por um intruso e outro que permitia que um intruso enviasse spam pelo computador desktop infectado.

Geralmente, os vírus são os ataques de segurança que mais destroem e, por serem eficazes, continuarão a ser escritos e a se espalhar. Entre os debates ativos dentro da comunidade de computação existe um sobre se uma **monocultura**, em que muitos sistemas executam o mesmo hardware, sistema operacional e/ou software de aplicação, está aumentando a ameaça e o dano causado pelas intrusões de segurança. Essa monocultura supostamente consiste em produtos da Microsoft, e parte do debate é sobre se tal monocultura ainda existe hoje.

15.3 Ameaças ao sistema e à rede

As ameaças ao programa normalmente utilizam um desmembramento nos mecanismos de proteção de um sistema aos programas de ataque. Por outro lado, as ameaças ao sistema e à rede envolvem o abuso de serviços e conexões de rede. Ameaças ao sistema e à rede criam uma situação em que os recursos do sistema operacional e os arquivos do usuário são utilizados indevidamente. Às vezes, um ataque ao sistema e à rede é usado para iniciar um ataque ao programa, e vice-versa.

Quanto mais **aberto** for um sistema operacional - mais serviços ele habilita e mais funções ele permite -, mais provável é que um bug esteja disponível para exploração. Cada vez mais, os sistemas operacionais lutam para ser **seguros por padrão**. Por exemplo, Solaris 10 passou de um modelo em que muitos serviços (FTP, telnet e outros) eram habilitados como padrão quando o sistema fosse instalado para um modelo em que quase todos os serviços são desabilitados no momento da instalação e precisam ser habilitados especificamente pelos administradores do sistema. Essas mudanças reduzem a **superfície de ataque** do sistema - o conjunto de maneiras como um atacante pode tentar invadir o sistema.

No restante desta seção, discutimos alguns exemplos de ameaças ao sistema e à rede, incluindo vermes, varredura de porta e ataques de negação de serviço. É importante observar que os ataques de mascaramento e reprodução também são comuns pelas redes entre os sistemas. De fato, esses ataques são mais eficazes e mais difíceis de contra-atacar quando vários sistemas estão envolvidos. Por exemplo, dentro de um computador, o sistema operacional normalmente pode determinar o emissor e o receptor de uma mensagem. Mesmo que o emissor mude para a ID de mais alguém, pode haver um registro dessa mudança de ID. Quando vários sistemas estão envolvidos, especialmente sistemas controlados por atacantes, então esse rastreamento é muito mais difícil.

Em geral, podemos dizer que o compartilhamento de segredos (para provar a identidade e como chaves para criptografia) é necessário para a autenticação e a criptografia, e o compartilhamento de segredos é mais fácil em ambientes (como um único sistema operacional) em que existem métodos de compartilhamento seguros. A criação da comunicação e autenticação segura é discutida nas [Seções 15.4 e 15.5](#).

15.3.1 Vermes

Um **verme** (worm) é um processo que usa o mecanismo de **procriação** para se duplicar. O verme procria cópias de si mesmo, usando os recursos do sistema e talvez bloqueando todos os outros processos. Em redes de computador, os vermes são particularmente potentes, pois podem se reproduzir entre os sistemas e, assim, acabar com a rede inteira. Esse evento ocorreu em 1988 nos sistemas UNIX na Internet, ocasionando perda de sistemas e tempo de administrador de sistemas que valem milhões de dólares.

No final do expediente de 2 de novembro de 1988, Robert Tappan Morris Jr., um aluno do primeiro ano na Cornell University, disparou um programa de verme em um ou mais hosts conectados à Internet. Visando às estações de trabalho Sun 3 da Sun Microsystems e computadores VAX rodando variantes do BSD UNIX versão 4, o verme rapidamente se espalhou por grandes distâncias; dentro de algumas horas de sua liberação, ele tinha consumido recursos do sistema a ponto de parar todas as máquinas infectadas.

Embora Robert Morris tenha projetado o programa de autorreplicação para reprodução e distribuição rápidas, alguns dos recursos do ambiente de rede do UNIX forneceram os meios para propagar o verme por todo o sistema. É provável que Morris tenha escolhido para infecção inicial um host da Internet que ficou aberto e acessível a usuários externos. De lá, o programa de verme explorou falhas nas rotinas de segurança do sistema operacional UNIX e tirou proveito de utilitários do UNIX que simplificam o compartilhamento de recursos em redes locais para obter acesso não autorizado a milhares de outras instalações conectadas. Os métodos de ataque de Morris são esboçados a seguir.

O verme era composto de dois programas, um programa de **gancho de atracação** (também chamado **bootstrap** ou **vítor**) e o programa principal. Com o nome *l1.c*, o gancho de atracação consistia em 99 linhas de código C compilado e executado em cada máquina que ele acessava. Uma vez estabelecido no sistema computadorizado sob ataque, o ganho de atracação se conectava à máquina onde foi originado e fazia o upload de uma cópia do verme principal para o sistema *atracado* ([Figura 15.6](#)). O programa principal prosseguia procurando outras máquinas para as quais o sistema recém-infectado poderia se conectar facilmente. Nessas ações, Morris explorava o utilitário de rede do UNIX, *rsh*, para facilitar a execução remota da tarefa. Configurando arquivos especiais que listam os pares de nomes de host-login, os usuários podem omitir a entrada de uma senha cada vez que acessam uma conta remota na lista emparelhada. O verme procurava esses arquivos especiais em busca de nomes de site que permitiriam a execução remota sem uma senha. Onde os shells remotos eram estabelecidos, o programa de verme era carregado e começava a execução novamente.

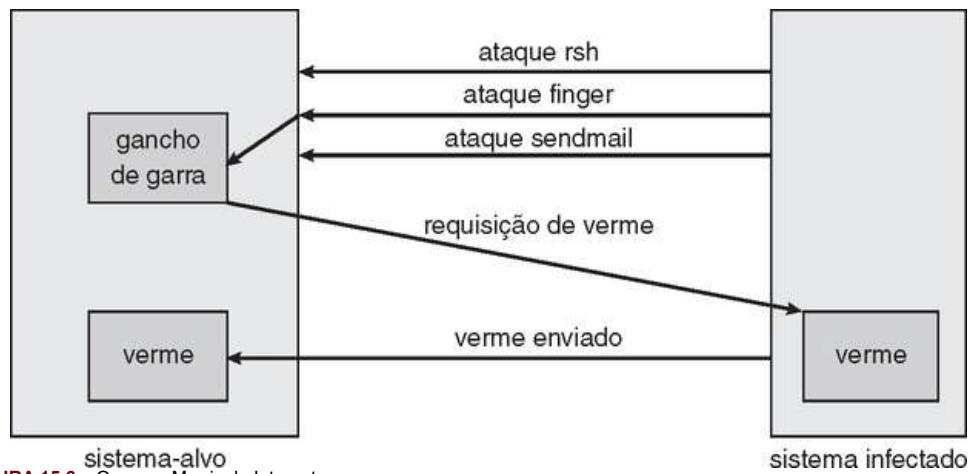


FIGURA 15.6 O verme Morris da Internet.

O ataque via acesso remoto era um dos três métodos de infecção embutidos no verme. Os outros dois métodos envolviam bugs do sistema operacional nos programas finger e sendmail do UNIX.

O utilitário finger funciona como um catálogo telefônico eletrônico; o comando retorna os nomes real e de login de uma pessoa, junto com outras informações que o usuário pode ter fornecido, como endereço do escritório e residencial, números de telefone, plano de pesquisa ou citação inteligente. O finger é executado como um processo de segundo plano (ou daemon) em cada site BSD e responde a consultas por meio da Internet. O verme executava um ataque de estouro de buffer no finger. O programa de Morris consultava o finger com uma string de 536 bytes preparada para ultrapassar o buffer alocado para entrada e escrever sobre o quadro de pilha. Em vez de retornar à rotina principal, o daemon finger era roteado para um procedimento dentro da sequência invasora de 536 bytes, agora residindo na pilha. O novo procedimento executava /bin/sh, que, se tivesse sucesso, dava ao verme um shell remoto na máquina sob ataque.

`finger nome-usuário@nome-host`

O bug explorado no sendmail também envolvia o uso de um processo daemon para entrada maliciosa. O sendmail envia, recebe e roteia o correio eletrônico. O código de depuração no utilitário permite que os testadores verifiquem e mostrem o estado do sistema de correio. A opção de depuração foi útil para administradores do sistema e normalmente foi deixada ativa como processo em segundo plano. Morris incluiu em seu arsenal de ataque uma chamada a debug que, em vez de especificar um endereço do usuário, como seria normal no teste, emitia um conjunto de comandos que remetiam por correio e executavam uma cópia do programa do gancho de atração.

Uma vez no lugar, o verme principal passava por tentativas sistemáticas de descobrir senhas do usuário. Ele começava tentando casos simples de nenhuma senha ou de senhas construídas de combinações de conta e nome de usuário, depois usava comparações com um dicionário interno das 432 opções favoritas de senha, e depois seguia para o estágio final, experimentando cada palavra no dicionário on-line padrão do UNIX, como uma possível senha. Esse algoritmo de descoberta de senha em três estágios, elaborado e eficiente, permitia que o verme tivesse acesso a outras contas de usuário no sistema infectado. O verme, então, procurava arquivos de dados rsh nessas contas recém-invadidas e os usava conforme descrevemos anteriormente para obter acesso a contas de usuário em sistemas remotos.

A cada novo acesso, o programa de verme procurava cópias já ativas de si mesmo. Se encontrasse uma, a nova cópia saía, exceto a cada 17^a ocorrência. Se o verme saísse em todas as duplicatas, ele poderia ter permanecido não detectado. Permitir que cada 17^a duplicata prosseguisse (talvez para confundir os esforços para interromper sua divulgação, disfarçando-se de falso verme) criou uma infestação em atacado dos sistemas Sun e VAX na Internet.

Os próprios recursos do ambiente de rede do UNIX que auxiliavam na propagação do verme também ajudaram a interromper seu avanço. A facilidade de comunicação eletrônica, mecanismos para copiar arquivos-fonte e binário para máquinas remotas, e acesso ao código-fonte e à perícia humana permitiram esforços cooperativos para desenvolver soluções rapidamente. Na noite do dia seguinte, 3 de novembro, métodos de interromper o programa invasor circulavam entre os administradores de sistemas por meio da Internet. Dentro de alguns dias, correções de software específicas para as falhas de segurança exploradas já estavam à disposição.

Por que Morris lançou o verme? A ação foi caracterizada tanto como um trote inofensivo cujo resultado foi infeliz quanto como uma ofensa criminosa séria. Com base na complexidade do ataque, é pouco provável que o lançamento do verme ou o escopo de sua divulgação não fosse intencional. O programa de verme tomava medidas elaboradas para cobrir seus rastros e repelir os esforços para impedir sua divulgação. Mesmo assim, o programa não tinha código visando danificar ou destruir os sistemas em que era executado. O autor certamente tinha condições de incluir tais comandos; de fato, estruturas de dados estavam presentes no código de partida que poderiam ter sido usadas para

transferir programas de cavalo de Troia ou vírus. O comportamento do programa pode levar a observações interessantes, mas não provê uma base sadia para chegar ao motivo. Contudo, o que não está aberto à especulação é o resultado legal: uma corte federal condenou Morris e estabeleceu uma sentença de três anos de liberdade condicional, 400 horas de serviços comunitários e uma fiança de US\$10.000. Os custos legais de Morris provavelmente ultrapassaram os US\$100.000.

Os especialistas em segurança continuam a avaliar métodos para diminuir ou eliminar vermes. Um evento mais recente mostra que os vermes ainda são um fato da vida na Internet. Ele também mostra que, com o crescimento da Internet, o dano que até mesmo vermes “inofensivos” podem causar também cresce e pode ser significativo. Este exemplo ocorreu durante o mês de agosto de 2003. A versão 5 do verme “Sobig”, mais conhecida como “W32.Sobig.F@mm”, foi lançada por pessoas até o momento desconhecidas. Ele é o verme que se espalhou mais rapidamente até o momento, no seu auge infectando centenas de milhares de computadores e uma em cada 17 mensagens de correio eletrônico na Internet. Ele encheu caixas de entrada de correio, retardou redes e exigiu um grande número de homens-hora para fazer a limpeza.

O Sobig.F foi lançado por meio de um upload para um newsgroup de pornografia, por meio de uma conta criada usando um cartão de crédito roubado. Ele foi disfarçado como uma foto. O vírus visava os sistemas Microsoft Windows e usava seu próprio mecanismo de SMTP para ser remetido por correio eletrônico para todos os endereços encontrados em um sistema infectado. Ele usava diversas linhas de assunto para ajudar a evitar detecção, incluindo “Thank You!”, “Your details” e “Re: Approved”. Ele também usava um endereço aleatório no host como o endereço “From:”, tornando difícil determinar por meio da mensagem qual máquina foi a fonte infectada. O Sobig.F incluía um anexo para o leitor de correio eletrônico do alvo para que fosse clicado, novamente com diversos nomes. Quando o payload era executado, ele armazenava um programa chamado WINPPR32.EXE no diretório-padrão do Windows, junto com um arquivo de texto. Ele também modificava o registro do Windows.

O código incluído no anexo também era programado para tentar periodicamente se conectar a um dentre 20 servidores, baixando e executando um programa deles. Felizmente, os servidores foram desativados antes de o código poder ser baixado. O conteúdo do programa desses servidores ainda não foi determinado. Se o código fosse malévolos, poderia haver danos incalculáveis a uma grande quantidade de máquinas.

15.3.2 Varredura de porta

Varredura de porta não é um ataque, mas sim um meio para um cracker detectar as vulnerabilidades de um sistema a atacar. A varredura de porta normalmente é automatizada, envolvendo uma ferramenta que tenta criar uma conexão TCP/IP para uma porta específica ou um intervalo de portas. Por exemplo, suponha que exista uma vulnerabilidade (ou bug) em `sendmail`. Um cracker poderia iniciar uma varredura de porta para tentar se conectar, digamos, à porta 25 de determinado sistema ou intervalo de sistemas. Se a conexão tiver sucesso, o cracker (ou a ferramenta) poderá tentar se comunicar com o serviço de resposta para determinar se realmente foi o `sendmail`, se for, se foi a versão com o bug.

Agora imagine uma ferramenta em que cada bug de cada serviço de cada sistema operacional fosse codificado. A ferramenta poderia tentar se conectar a cada porta de um ou mais sistemas. Para cada serviço que respondesse, ela poderia tentar usar cada bug conhecido. Frequentemente, os bugs são estouros de buffer, permitindo a criação de um shell de comando privilegiado no sistema. A partir daí, logicamente, o cracker poderia instalar cavalos de Troia, programas de back-door e assim por diante.

Não existe uma ferramenta assim, mas existem ferramentas que realizam subconjuntos dessa funcionalidade. Por exemplo, `nmap` (de <http://www.insecure.org/nmap/>) é um utilitário open-source muito versátil para exploração de rede e auditoria de segurança. Quando apontado para um alvo, ele determina quais serviços estão sendo executados, incluindo nomes de aplicações e suas versões. Ele pode determinar o sistema operacional host. Ele também pode fornecer informações sobre as defesas, como quais firewalls estão defendendo o alvo. Ele não explora quaisquer bugs conhecidos.

O Nessus (de <http://www.nessus.org/>) realiza uma função semelhante, mas tem um banco de dados de bugs e suas explorações. Ele pode varrer um intervalo de sistemas, determinar os serviços executando nesses sistemas e tentar atacar todos os bugs apropriados. Ele gera relatórios sobre os resultados. Ele não realiza a etapa final de explorar os bugs encontrados, mas um cracker com conhecimento ou um script kiddie poderia.

Como as varreduras de porta são detectáveis (Seção 15.6.3), elas constantemente são iniciadas a partir de **sistemas zumbis**, que são sistemas previamente comprometidos, independentes, servindo aos seus proprietários legítimos enquanto são usados por outros para finalidades nefastas, incluindo ataques de negação de serviço e repasse de spam. Os zumbis tornam os crackers particularmente difíceis de perseguir, pois determinar a origem do ataque e a pessoa que o iniciou é algo desafiador. Esse é um dos muitos motivos para proteger sistemas “inconsequentes”, e não apenas os sistemas contendo informações ou serviços “valiosos”.

15.3.3 Negação de serviço

Como mencionado, a negação de serviço (Denial of Service) não visa a obter informações ou roubar recursos, mas sim interromper o uso legítimo de um sistema ou instalação. A maioria dos ataques de negação de serviço envolve sistemas que o atacante não penetrou. Em vez disso, o disparo de um ataque que impede o uso legítimo frequentemente é mais fácil do que invadir uma máquina ou instalação.

Os ataques de negação de serviço geralmente são baseados em rede. Eles podem ser de duas categorias. Os ataques na primeira categoria utilizam tantos recursos da instalação que, basicamente, nenhum trabalho útil pode ser feito. Por exemplo, um clique no site poderia baixar um applet Java que prossegue usando todo o tempo disponível da CPU ou fazendo surgir janelas pop-up infinitamente. A segunda categoria envolve a interrupção da rede da instalação. Tem havido vários ataques de negação de serviço bem-sucedidos desse tipo contra os principais sites. Esses ataques são resultantes do abuso de alguma funcionalidade fundamental do TCP/IP. Por exemplo, se o atacante enviar a parte do protocolo que diz "Eu quero iniciar uma conexão TCP", mas nunca acompanhá-la do padrão "A conexão agora está completa", o resultado podem ser várias sessões TCP parcialmente iniciadas. Se várias dessas sessões forem iniciadas, elas poderão consumir todos os recursos de rede do sistema, desativando quaisquer outras conexões TCP legítimas. Esses ataques, que podem durar horas ou dias, têm diminuído ou impedido tentativas legítimas de usar a instalação-alvo. Esses ataques normalmente são impedidos em nível de rede até os sistemas operacionais poderem ser atualizados para reduzir sua vulnerabilidade.

Em geral, é impossível impedir ataques de negação de serviço. Os ataques utilizam os mesmos mecanismos da operação normal. Ainda mais difícil de prever e resolver são os **ataques de negação de serviço distribuídos (DDOS)**. Esses ataques são disparados de vários sites ao mesmo tempo, em direção a um alvo comum, normalmente por zumbis. Os ataques de DDOS têm se tornado mais comuns e às vezes estão associados a tentativas de chantagem. Um site é atacado e os atacantes se oferecem para interromper o ataque em troca de dinheiro.

Às vezes, um site nem sequer sabe que está sob ataque. Pode ser difícil determinar se uma lentidão no sistema é apenas um surto no uso do sistema ou um ataque. Imagine que uma campanha publicitária bem-sucedida, que aumente bastante o tráfego para um site, pode ser considerada um DDOS.

Existem outros aspectos interessantes dos ataques de DOS. Por exemplo, se um algoritmo de autenticação bloquear uma conta por um período após várias tentativas incorretas de acessar a conta, então um atacante poderia fazer toda a autenticação ser bloqueada fazendo propositadamente tentativas incorretas de acessar todas as contas. De modo semelhante, um firewall que bloqueia automaticamente alguns tipos de tráfego poderia ser induzido a bloquear esse tráfego quando não deveria. Esses exemplos sugerem que programadores e gerentes de sistemas precisam compreender totalmente os algoritmos e as tecnologias que estão empregando. Finalmente, as aulas de ciência de computação são fontes notórias de ataques acidentais de DOS ao sistema. Considere os primeiros exercícios de programação em que os alunos aprendem a criar subprocessos ou threads. Um bug comum envolve a criação de subprocessos indefinidamente. A memória livre do sistema e os recursos da CPU não têm chance alguma.

15.4 Criptografia como ferramenta de segurança

Existem muitas defesas contra os ataques ao computador, variando de metodologia e de tecnologia. A ferramenta mais à disposição dos projetistas e usuários do sistema é a criptografia. Nesta seção, discutimos os detalhes da criptografia e seu uso na segurança do computador.

Em um computador isolado, o sistema operacional pode determinar com confiança o emissor e o destinatário de toda a comunicação entre processos, pois controla todos os canais de comunicação no computador. Em uma rede de computadores, a situação é bem diferente. Um computador em rede recebe bits *pelos fios* sem uma forma imediata e confiável de determinar qual máquina ou aplicação enviou esses bits. De modo semelhante, o computador envia bits para a rede sem um modo de saber quem poderia recebê-los mais cedo ou mais tarde.

Normalmente, os endereços da rede são usados para deduzir os emissores e receptores em potencial das mensagens na rede. O pacote da rede chega com um endereço de origem, como um endereço IP. E quando um computador envia uma mensagem, ele indica o receptor desejado especificando um endereço de destino. Todavia, para aplicações em que a segurança é importante, estaremos chamando problemas se considerarmos que o endereço de origem ou destino de um pacote determina de forma confiável quem enviou ou recebeu esse pacote. Um computador fraudulento pode enviar uma mensagem com um endereço de origem falsificado, e vários computadores além daquele especificado pelo endereço de destino podem receber (e normalmente recebem) um pacote. Por exemplo, todos os roteadores no caminho até o destino também receberão o pacote. Logo, como um sistema operacional pode decidir se concederá uma requisição quando não pode confiar na origem indicada da requisição? E como ele poderia prover proteção para uma requisição ou dados se não pode determinar quem receberá a resposta ou o conteúdo da mensagem que envia pela rede?

Em geral, é considerado inviável montar uma rede de qualquer tamanho em que os endereços de origem e destino dos pacotes podem ser confiáveis nesse sentido. Portanto, a única alternativa é eliminar de alguma forma a necessidade de confiar na rede. Essa é a tarefa da criptografia. De forma abstrata, a **criptografia** é usada para restringir os emissores e/ou receptores em potencial de uma mensagem. A criptografia moderna é baseada em segredos, chamados **chaves**, que são distribuídos seletivamente aos computadores em uma rede e usados para processar mensagens. A criptografia permite a um destinatário de uma mensagem verificar se a mensagem foi criada por algum computador que possui determinada chave - a chave é a *origem* da mensagem. De modo semelhante, um emissor pode codificar sua mensagem de modo que somente um computador com determinada chave possa decodificar a mensagem, de modo que a chave se torna o *destino*. No entanto, ao contrário dos endereços da rede, as chaves são criadas de modo que não seja computacionalmente viável derivá-las das mensagens que foram usadas para gerar ou a partir de qualquer outra informação pública. Assim, elas proveem um meio muito mais confiável de restringir os emissores e receptores das mensagens. Observe que a criptografia é um campo de estudo por si só, com complexidades e sutilezas grandes e pequenas. Aqui, exploramos os aspectos mais importantes das partes da criptografia que pertencem aos sistemas operacionais.

15.4.1 Codificação

Por solucionar uma grande variedade de problemas de segurança de comunicação, a **codificação** é usada frequentemente em muitos aspectos da computação moderna. A codificação é um meio de restringir os possíveis receptores de uma mensagem. Um algoritmo de codificação permite ao emissor de uma mensagem garantir que somente um computador possuindo determinada chave possa ler a mensagem. A codificação de mensagens é uma prática antiga, é claro, e existem muitos algoritmos de codificação, desde os tempos mais antigos. Nesta seção, descrevemos os princípios de codificação e algoritmos mais importantes atualmente.

A [Figura 15.7](#) mostra um exemplo de dois usuários se comunicando de forma segura por um canal inseguro. Vamos nos referir a essa figura durante esta seção. Observe que a troca de chave pode ocorrer diretamente entre as duas partes ou por meio de um terceiro confiável (ou seja, uma autoridade de certificação), conforme discutimos na [Seção 15.4.1.4](#).

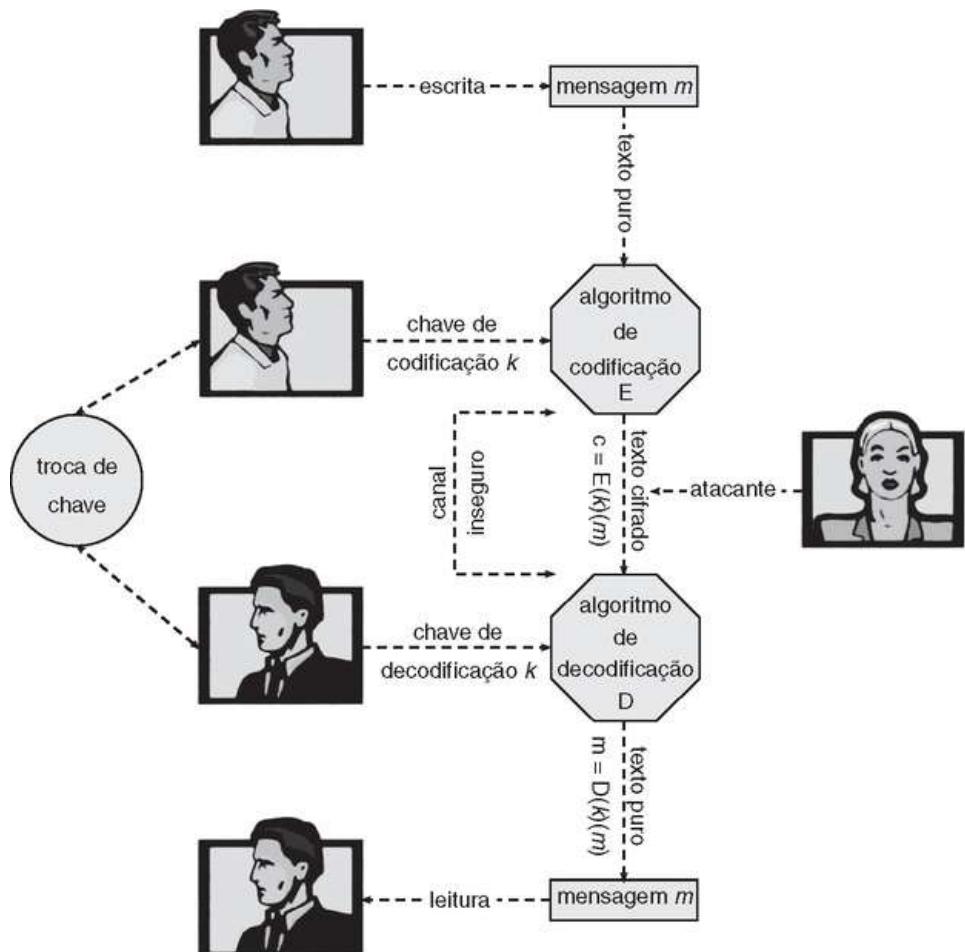


FIGURA 15.7 Uma comunicação segura por um meio inseguro.

Um algoritmo de codificação consiste nos seguintes componentes:

- Um conjunto K de chaves.
- Um conjunto M de mensagens.
- Um conjunto C de cifras.
- Uma função $E : K \rightarrow (M \rightarrow C)$. Ou seja, para cada $k \in K$, $E(k)$ é uma função para gerar cifras das mensagens. Tanto E quanto $E(k)$ para qualquer k devem ser funções que possam ser calculadas de forma eficiente.
- Uma função $D : K \rightarrow (C \rightarrow M)$. Ou seja, para cada $k \in K$, $D(k)$ é uma função para gerar mensagens de cifras. Tanto D quanto $D(k)$ para qualquer k devem ser funções que possam ser calculadas de forma eficiente.

Um algoritmo de codificação precisa prover essa propriedade essencial: dada uma cifra $c \in C$, um computador deve poder calcular m de modo que $E(k)(m) = c$ somente se possuir $D(k)$. Assim, um computador mantendo $D(k)$ pode decodificar cifras para os textos limpos usados para produzi-las, mas um computador que não mantém $D(k)$ não pode decodificar as cifras. Como as cifras são expostas (por exemplo, elas são enviadas na rede), é importante que seja inviável derivar $D(k)$ das cifras.

Existem dois tipos principais de algoritmos de codificação: simétrico e assimétrico. Discutimos os dois tipos nas próximas seções.

15.4.1.1 Codificação simétrica

No **algoritmo de codificação simétrico**, a mesma chave é usada para codificar e decodificar. Isto é, $E(k)$ pode ser derivado de $D(k)$ e vice-versa. Portanto, o segredo de $E(k)$ precisa ser protegido ao mesmo ponto que o de $D(k)$.

Durante as últimas décadas, o algoritmo de codificação simétrica mais utilizado nos Estados Unidos para aplicações civis tem sido o **padrão de codificação de dados (DES - Data Encryption Standard)**, adotado pelo National Institute of Standards and Technology (NIST). O DES funciona apanhando um valor de 64 bits e uma chave de 56 bits, e realizando uma série de transformações. Essas transformações são baseadas em operações de substituição e permutação, como geralmente acontece para transformações de codificação simétrica. Algumas das transformações são **transformações de caixa-preta**, pois seus algoritmos são ocultos. Na verdade, essas chamadas "S-boxes" são classificadas pelo governo dos Estados Unidos. As mensagens maiores do que 64 bits são

desmembradas em pedaços de 64 bits. Como o DES funciona em um pedaço de bits de cada vez, ele é conhecido como **cifra em bloco**. Se a mesma chave for usada para codificar uma quantidade estendida de dados, ela se torna vulnerável a ataque. Considere, por exemplo, que o mesmo bloco de origem resultaria no mesmo texto cifrado se a mesma chave e algoritmo de criptografia fossem usados. Portanto, os pedaços não são apenas codificados, mas também XORados com o bloco de texto cifrado anterior antes da codificação. Isso é conhecido como **encadeamento de bloco cifrado**.

O DES agora é considerado inseguro para muitas aplicações, pois suas chaves podem ser exaustivamente pesquisadas com recursos de computador moderados. Contudo, em vez de abandonar o DES, o NIST criou uma modificação chamada **triplo DES**, em que o algoritmo DES é repetido três vezes (duas codificações e uma decodificação) no mesmo texto puro usando duas ou três chaves - por exemplo, $c = E(k_3)(D(k_2)(E(K_1)(m)))$. Quando três chaves são usadas, o tamanho de chave efetivo é de 168 bits. O triplo DES é bastante utilizado hoje.

Em 2001, o NIST adotou um novo algoritmo de codificação, chamado **padrão de codificação avançado (Advanced Encryption Standard - AES)**, para substituir o DES. O AES é outra cifra em bloco simétrico. Ele pode usar tamanhos de chave de 128, 192 e 256 bits e trabalha em blocos de 128 bits. Ele funciona realizando 10 a 14 rodadas de transformações em uma matriz formada a partir de um bloco. Geralmente, o algoritmo é compacto e eficiente.

Existem vários outros algoritmos de criptografia em bloco simétricos em uso atualmente, que precisam ser mencionados. O algoritmo **twofish** é rápido, compacto e fácil de implementar. Ele pode usar um tamanho de chave variável de até 256 bits e trabalha em blocos de 128 bits. O **RC5** pode variar em tamanho de chave, número de transformações e tamanho de bloco. Por usar apenas operações computacionais básicas, ele pode ser executado em uma grande variedade de CPUs.

O **RC4** talvez seja a cifra de stream mais comum. Uma **cifra de stream** é criada para codificar e decodificar um stream de bytes ou bits, em vez de um bloco. Isso é útil quando o tamanho de uma comunicação tornaria uma cifra em bloco muito lenta. A chave é inserida em um gerador de bits pseudoaleatório, que é um algoritmo que tenta produzir bits aleatórios. A saída do gerador quando recebe uma chave é um **keystream**. Um **keystream** é um conjunto infinito de chaves que podem ser usadas para o stream de texto puro inserido. O RC4 é usado na codificação de streams de dados, como WEP, o protocolo de LAN sem fio. Ele também é usado nas comunicações entre os navegadores Web e os servidores Web, conforme discutimos a seguir. Infelizmente, descobriu-se que o RC4 usado no WEP (padrão IEEE 802.11) pode ser quebrado com uma quantidade razoável de tempo do computador. Na verdade, o próprio RC4 possui vulnerabilidades.

15.4.1.2 Codificação assimétrica

Em um **algoritmo de codificação assimétrico**, existem várias chaves de codificação e decodificação. Aqui, descrevemos um algoritmo desse tipo, conhecido como **RSA**, que são as iniciais dos seus inventores (Rivest, Shamir e Adleman). A cifra RSA é um algoritmo de chave pública com cifra por bloco e é o algoritmo assimétrico mais utilizado. Os algoritmos assimétricos baseados em curvas elípticas estão ganhando terreno, pois o tamanho de chave de tal algoritmo pode ser menor para o mesmo peso criptográfico.

É computacionalmente inviável derivar $D(k_d, N)$ de $E(k_e, N)$ e, por isso, $E(k_e, N)$ não precisa ser mantido secreto e pode ser disseminado abertamente; assim, $E(k_e, N)$ (ou apenas k_e) é a **chave pública** e $D(k_d, N)$ (ou apenas k_d) é a **chave privada**. N é o produto de dois números primos grandes, escolhidos aleatoriamente, p e q (por exemplo, p e q possuem 512 bits cada). O algoritmo de codificação é $E(k_e, N)(m) = m^{k_e} \text{ mod } N$, onde k_e satisfaz $k_e k_d \text{ mod } (p-1)(q-1) = 1$. O algoritmo de decodificação é, então, $D(k_d, N)(c) = c^{k_d} \text{ mod } N$.

Um exemplo usando valores pequenos aparece na [Figura 15.8](#). Nesse exemplo, tornamos $p = 7$ e $q = 13$. Depois, calculamos $N = 7*13 = 91$ e $(p-1)(q-1) = 72$. Em seguida, selecionamos k_e relativamente primo de 72 e < 72, gerando 5. Finalmente, calculamos k_d de modo que $k_e k_d \text{ mod } 72 = 1$, gerando 29. Agora temos nossas chaves: a chave pública, $k_e, N = 5, 91$, e a chave privada, $k_d, N = 29, 91$. A codificação da mensagem 69 com a chave pública resulta na mensagem 62, que é depois decodificada pelo receptor por meio da chave privada.

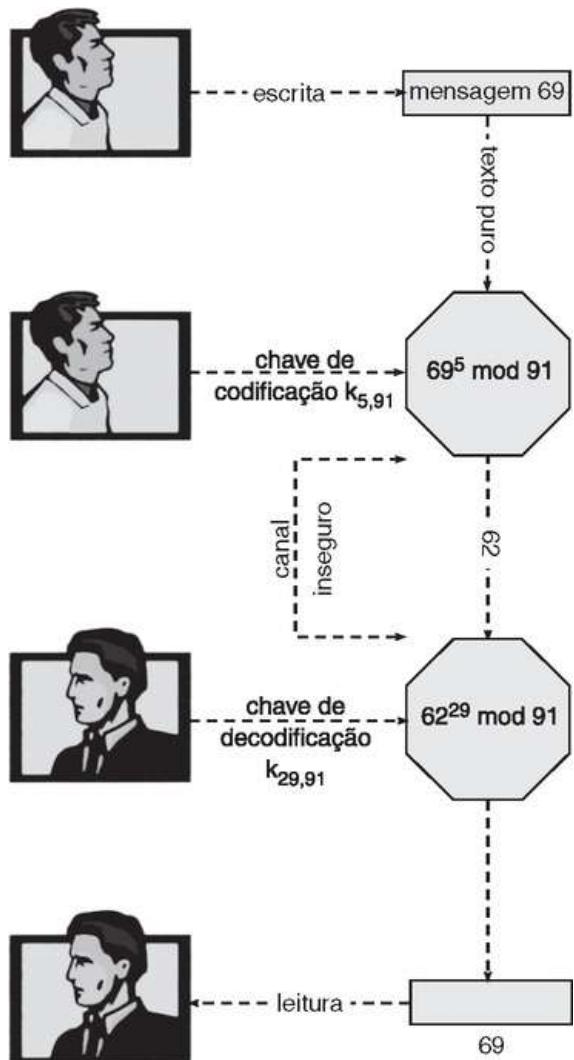


FIGURA 15.8 Codificação e decodificação usando a criptografia simétrica RSA.

O uso da codificação assimétrica começa com a publicação da chave pública do destino. Para a comunicação bidirecional, a origem também precisa publicar sua chave pública. A “publicação” pode ser tão simples quanto entregar uma cópia eletrônica da chave ou pode ser mais complexa. A chave privada (ou “chave secreta”) precisa ser protegida com cuidado, pois qualquer um que mantém essa chave poderá decodificar qualquer mensagem criada pela chave pública correspondente.

Temos que observar que a diferença aparentemente pequena no uso da chave entre a criptografia assimétrica e a simétrica é muito grande na prática. A criptografia assimétrica é baseada em funções matemáticas, em vez de transformações, tornando muito mais dispendioso computacionalmente de se executar. É muito mais rápido para um computador codificar e decodificar texto cifrado usando os algoritmos simétricos normais do que usando os algoritmos assimétricos. Por que, então, usamos um algoritmo assimétrico? Na verdade, esses algoritmos não são usados para a codificação de uso geral de grande quantidade de dados. Contudo, eles são usados não apenas para a codificação de pequenas quantidades de dados, mas também para autenticação, confidencialidade e distribuição de chaves, conforme mostramos nas próximas seções.

15.4.1.3 Autenticação

Vimos que a codificação fornece um modo de restringir o conjunto de receptores possíveis de uma mensagem. Restringir o conjunto de emissores em potencial de uma mensagem é algo chamado **autenticação**. A autenticação é, assim, complementar à codificação. Na verdade, às vezes suas funções se sobrepõem. Considere que uma mensagem codificada também pode provar a identidade do emissor. Por exemplo, se $D(k_d, N)(E(k_e, N)(m))$ produzir uma mensagem válida, então sabemos que o criador da mensagem deverá manter k_e . A autenticação também é útil para provar que uma mensagem não foi modificada. Nesta seção, discutimos a autenticação como uma restrição nos receptores possíveis de uma mensagem. Observe que esse tipo de autenticação é semelhante, mas distinta, da autenticação do usuário, que discutimos na [Seção 15.5](#).

Um algoritmo de autenticação consiste nos seguintes componentes:

- Um conjunto K de chaves.
- Um conjunto M de mensagens.
- Um conjunto A de autenticadores.
- Uma função $S : K \rightarrow (M \rightarrow A)$. Ou seja, para cada $k \in K$, $S(k)$ é uma função para gerar autenticadores das mensagens. Tanto S quanto $S(k)$ para qualquer k devem ser funções que possam ser calculadas de forma eficiente.
- Uma função $V : K \rightarrow (M \times A \rightarrow \{\text{true}, \text{false}\})$. Ou seja, para cada $k \in K$, $V(k)$ é uma função para verificar os autenticadores nas mensagens. Tanto V quanto $V(k)$ para qualquer k devem ser funções que possam ser calculadas de forma eficiente.

A propriedade crítica que um algoritmo de autenticação precisa possuir é esta: para uma mensagem m , um computador pode gerar um autenticador $a \in A$ de modo que $V(k)(m, a) = \text{true}$ somente se possuir $S(k)$. Assim, um computador mantendo $S(k)$ pode gerar autenticadores sobre mensagens de modo que qualquer outro computador possuindo $V(k)$ possa verificá-los. Contudo, um computador que não possui $S(k)$ não poderá gerar autenticadores sobre as mensagens que possam ser verificados usando $V(k)$. Como os autenticadores ficam expostos (por exemplo, eles são enviados na rede com as próprias mensagens), não deverá ser viável derivar $S(k)$ dos autenticadores.

Assim como existem dois tipos de algoritmos de codificação, existem duas variedades principais de algoritmos de autenticação. O primeiro passo para entender esses algoritmos é explorar as funções hash. Uma **função hash** $H(m)$ cria um pequeno bloco de dados de tamanho fixo, conhecido como **resumo de mensagem** ou **valor de hash**, a partir de uma mensagem m . As funções hash funcionam apanhando uma mensagem em blocos de n bits e processando os blocos para produzir um hash de n bits. H precisa ser resistente à colisão em m , isto é, precisa ser inviável encontrar um $m' \neq m$ tal que $H(m) = H(m')$. Agora, se $H(m) = H(m')$, sabemos que $m = m'$ - isto é, sabemos que a mensagem não foi modificada. Funções comuns de resumo de mensagem incluem **MD5**, que produz um hash de 128 bits, e **SHA-1**, que gera um hash de 160 bits. Os resumos de mensagem são úteis para detectar mensagens alteradas, mas não são úteis como autenticadores. Por exemplo, $H(m)$ pode ser enviado junto com uma mensagem, mas se H for conhecido alguém poderia modificar m e recalcular $H(m)$, e a modificação da mensagem não seria detectada. Portanto, um algoritmo de autenticação apanha o resumo da mensagem e o codifica.

O primeiro tipo principal de algoritmo de autenticação utiliza a criptografia simétrica. Em um **código de autenticação de mensagem (Message-Authentication Code - MAC)**, uma soma de verificação criptográfica é gerada a partir da mensagem usando uma chave secreta. O conhecimento de $V(k)$ e o conhecimento de $S(k)$ são equivalentes: um pode ser derivado do outro; por esse motivo k deve ser mantido secreto. Um exemplo simples de um MAC define $S(k)(m) = f(k, H(m))$, onde f é uma função que é unidirecional no seu primeiro argumento - ou seja, k não pode ser derivado de $f(k, H(m))$. Devido à resistência à colisão na função hash, temos uma razoável garantia de que nenhuma outra mensagem poderia criar o mesmo MAC. Um algoritmo de verificação adequado é, então, $V(k)(m, a) \equiv (f(k, m) = a)$. Observe que k é necessário para calcular tanto $S(k)$ quanto $V(k)$, ou seja, qualquer um capaz de calcular um pode calcular o outro.

O segundo tipo principal de algoritmo de autenticação é um **algoritmo de assinatura digital**, e os autenticadores produzidos dessa forma são chamados **assinaturas digitais**. Em um algoritmo de assinatura digital, é computacionalmente inviável derivar $S(k_s)$ de $V(k_v)$; em particular, V é uma função unidirecional. Assim, k_v é denominada chave pública e k_s é a chave privada.

Considere, como exemplo, o algoritmo de assinatura digital RSA. Ele é semelhante ao algoritmo de codificação RSA, mas o uso da chave é revertido. A assinatura digital de uma mensagem é derivada do cálculo de $S(k_s)(m) = H(m)^{k_s} \pmod{N}$. A chave k_s novamente é um par $\langle d, N \rangle$, onde N é o produto de dois números primos grandes, escolhidos aleatoriamente, p e q . O algoritmo de verificação é, então, $V(k_v)(m, a) \equiv (a^{k_v} \pmod{N} = H(m))$, onde k_v satisfaz $k_v k_s \pmod{(p-1)(q-1)} = 1$.

Se a codificação pode provar a identidade do emissor de uma mensagem, então por que precisamos de algoritmos de autenticação? Existem três motivos principais:

- Os algoritmos de autenticação geralmente exigem menos cálculos (com a notável exceção das assinaturas digitais RSA). Por grandes quantidades de texto puro, essa eficiência pode fazer uma enorme diferença no uso de recursos e no tempo necessário para autenticar uma mensagem.
- Um autenticador de mensagem é quase sempre mais curto do que a mensagem, e é texto cifrado. Isso melhora o uso de espaço e a eficiência do tempo de transmissão.
- Às vezes, queremos autenticação, mas não confidencialidade. Por exemplo, uma empresa poderia fornecer um patch de software e poderia "assinar" o patch para provar que ele veio da empresa e que não foi modificado.

A autenticação é um componente de muitos aspectos da segurança. Por exemplo, ela é o núcleo do **não repúdio**, que fornece uma prova de que uma entidade realizou uma ação. Um exemplo típico de não repúdio envolve o preenchimento de formulários eletrônicos como uma alternativa à assinatura de contratos em papel. O não repúdio garante que uma pessoa preenchendo um formulário eletrônico não pode negar que fez isso.

15.4.1.4 Distribuição de chave

Certamente, uma boa parte da batalha entre os criptógrafos (aqueles que inventam as cifras) e os criptoanalistas (aqueles tentando quebrar as cifras) envolve as chaves. Com os algoritmos simétricos, as duas partes precisam da chave, e ninguém mais deverá tê-la. A entrega da chave simétrica é um desafio enorme. Às vezes, ela é realizada **off-line** - digamos, por meio de um documento em papel ou uma conversa. Contudo, esses métodos não funcionam muito bem em grande escala. Considere também o desafio do gerenciamento de chaves. Suponha que um usuário queira se comunicar com N outros usuários privadamente. Esse usuário precisaria de N chaves e, por mais segurança, precisaria mudar essas chaves com frequência.

Esses são os próprios motivos para os esforços na criação dos algoritmos de chave assimétrica. Não apenas as chaves podem ser trocadas em público, mas um usuário específico só precisa de uma chave privada, não importa com quantas outras pessoas ele queira se comunicar. Há ainda a questão de gerenciar uma chave pública por parte a ser comunicada, mas como as chaves públicas não precisam ser protegidas o armazenamento simples pode ser usado para esse **chaveiro**.

Infelizmente, até mesmo a distribuição de chaves públicas exige algum cuidado. Considere o ataque de homem no meio, mostrado na [Figura 15.9](#). Aqui, a pessoa que deseja receber a mensagem codificada envia sua chave pública, mas um atacante também envia sua chave pública “má” (que corresponde à sua chave privada). A pessoa que deseja enviar a mensagem codificada não sabe disso, e portanto usa a chave má para codificar a mensagem. O atacante, então, a decodifica sem problemas.

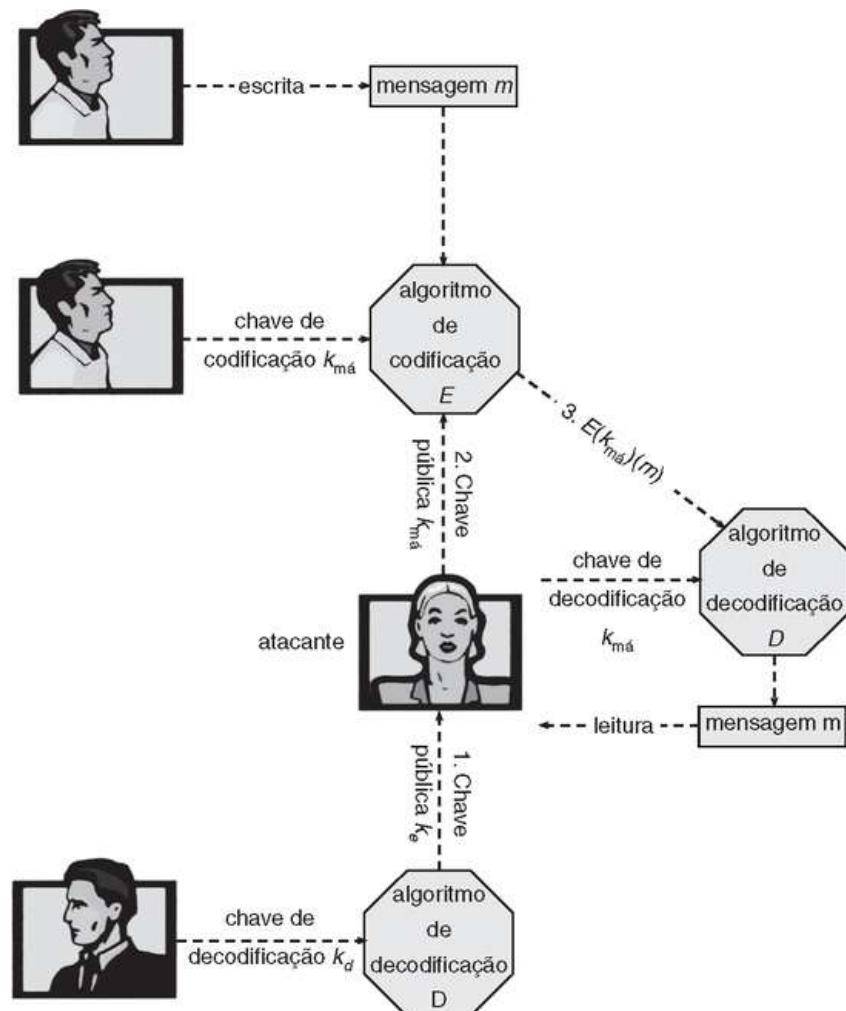


FIGURA 15.9 Ataque de homem no meio na criptografia assimétrica.

O problema é de autenticação - o que precisamos é de uma prova de quem (ou o quê) possui uma chave pública. Um modo de resolver esse problema envolve o uso de certificados digitais. Um **certificado digital** é uma chave pública assinada digitalmente por uma parte confiável. A parte confiável recebe prova de identificação de alguma entidade e certifica que a chave pública pertence a essa entidade. Mas como saber que podemos confiar no certificador? Essas **autoridades de certificado** possuem suas chaves públicas incluídas dentro dos navegadores Web (e outros consumidores de certificados) antes que sejam distribuídas. Essas autoridades de certificado podem então atestar outras autoridades (assinando digitalmente as chaves públicas dessas outras

autoridades), e assim por diante, criando uma teia de confiança. Os certificados podem ser distribuídos em um formato-padrão de certificado digital X.509, que pode ser analisado pelo computador. Esse esquema é usado para a comunicação segura na Web, conforme discutimos na [Seção 15.4.3](#).

15.4.2 Implementação da criptografia

Os protocolos de rede normalmente são organizados em **camadas**, e cada camada atua como um cliente para uma camada abaixo dela, ou seja, quando um protocolo gera uma mensagem para enviar ao seu protocolo equivalente em outra máquina, ele entrega sua mensagem ao protocolo abaixo dele, na pilha de protocolos de rede, para remessar ao seu par na outra máquina. Por exemplo, em uma rede IP, o TCP (um protocolo da *camada de transporte*) atua como um cliente do IP (um protocolo da *camada de rede*): os pacotes TCP são passados para o IP para entrega ao par TCP no outro extremo da conexão TCP. O IP encapsula o pacote TCP em um pacote IP, que ele passa para a *camada de enlace de dados*, para ser transmitido pela rede ao seu par IP no computador de destino. Esse par IP, então, sobe com o pacote TCP para o par TCP nessa máquina. Na realidade, o **modelo de referência ISO**, que tem sido adotado quase universalmente como modelo para redes de dados, define sete dessas camadas de protocolos. (Você lerá mais sobre o modelo de rede ISO no [Capítulo 16](#); a [Figura 16.6](#) mostra um diagrama do modelo.)

A criptografia pode ser inserida em quase todos os níveis nesse modelo. O SSL ([Seção 15.4.3](#)), por exemplo, provê segurança na camada de transporte. A segurança da camada de rede em geral tem sido padronizada como **IPSec**, que define os formatos de pacote IP que permitem a inserção de autenticadores e a codificação do conteúdo do pacote. Ele usa criptografia simétrica e usa o protocolo **IKE** para a troca de chaves. O IPSec está se tornando muito utilizado como base para as **redes privadas virtuais (VPNs)**, em que todo o tráfego entre duas extremidades IPSec é codificado para criar uma rede privada a partir de outra que pode ser pública. Diversos protocolos também têm sido desenvolvidos para uso pelas aplicações, mas depois as próprias aplicações precisam ser desenvolvidas para implementar a segurança.

Onde a proteção criptográfica é mais bem colocada em uma pilha de protocolos? Em geral, não existe uma resposta definitiva. Por um lado, mais protocolos se beneficiam com proteções feitas mais abaixo na pilha. Por exemplo, como os pacotes IP encapsulam pacotes TCP, a codificação de pacotes IP (usando IPSec, por exemplo) também esconde o conteúdo dos pacotes TCP encapsulados. De modo semelhante, os autenticadores nos pacotes IP detectam a modificação da informação contida no cabeçalho TCP.

Por outro lado, a proteção em camadas mais baixas na pilha de protocolos pode dar proteção insuficiente para os protocolos de camadas mais altas. Por exemplo, um servidor de aplicações que trabalha em cima de IPSec poderia ser capaz de autenticar os computadores clientes dos quais as requisições são recebidas. No entanto, para autenticar um usuário em um computador cliente, o servidor pode ter de usar um protocolo no nível de aplicação – por exemplo, o usuário pode ser requisitado a digitar uma senha. Considere também o problema do correio eletrônico. O correio eletrônico entregue por meio do protocolo SMTP padrão do setor é armazenado e encaminhado, geralmente várias vezes, antes de ser entregue. Cada uma dessas transmissões poderia passar por uma rede protegida ou desprotegida. Para que a mensagem seja segura, a mensagem de correio eletrônico precisa ser codificada de modo que sua segurança seja independente dos transportes que a carregam.

15.4.3 Um exemplo: SSL

O SSL 3.0 é um protocolo criptográfico que permite que dois computadores se comuniquem com segurança, ou seja, de modo que cada um possa limitar o emissor e receptor das mensagens ao outro. Talvez ele seja o protocolo criptográfico mais utilizado na Internet hoje, pois é o protocolo-padrão pelo qual os navegadores Web se comunicam com segurança com servidores Web. Para completar, temos que observar que o SSL foi projetado pela Netscape e que evoluiu para um protocolo TLS padrão do setor. Nesta discussão, usamos SSL para indicar tanto SSL quanto TLS.

O SSL é um protocolo complexo, com muitas opções. Aqui apresentamos apenas uma única variação dele, e mesmo assim de uma forma muito simplificada e abstrata, para manter o foco em seu uso de primitivas criptográficas. O que estamos para ver é uma complexa dança em que a criptografia assimétrica é usada para que cliente e servidor possam estabelecer uma **chave de sessão** segura, que possa ser usada para a codificação simétrica da sessão entre os dois – tudo isso enquanto se evitam os ataques de homem no meio e reprodução. Para maior força criptográfica, as chaves de sessão são esquecidas quando uma sessão é concluída. Outra comunicação entre os dois exigiria a geração de novas chaves de sessão.

O protocolo SSL é iniciado por um **cliente** c para se comunicar com segurança com um **servidor**. Antes do uso do protocolo, considera-se que o servidor s obteve um certificado, indicado por cert, de outra parte chamada autoridade de certificação, ou CA. Esse certificado é uma estrutura contendo o seguinte:

- Diversos atributos attrs do servidor, como seu nome *distinto* exclusivo e seu nome *comum* (DNS).
- A identidade de um algoritmo de codificação pública $E(\cdot)$ para o servidor2.
- A chave pública k_e para esse servidor.
- Um intervalo de validade interval , durante o qual o certificado precisa ser considerado válido.
- Uma assinatura digital a sobre a informação acima, dada pela CA, ou seja, $a = S(k_{CA}) (\langle \text{attrs}, E(k_e), \text{interval} \rangle)$.

Além disso, antes do uso do protocolo, presume-se que o cliente tenha obtido o algoritmo de verificação público $V(k_{CA})$ para CA. No caso da Web, o navegador do usuário é remetido pelo fornecedor com os algoritmos de verificação e as chaves públicas de determinadas autoridades de certificação. O usuário pode acrescentar ou excluir esses itens para as autoridades de certificação como desejar.

Quando c se conecta a s , ele envia um valor aleatório de 28 bytes n_c ao servidor, que responde com um valor aleatório próprio, n_s , mais seu certificado cert_s . O cliente verifica se $V(k_{CA}) (\langle \text{attrs}, E(k_e), \text{interval} \rangle, a) = \text{true}$ e se a hora atual está no intervalo de validade interval . Se os dois testes forem satisfeitos, o servidor provou sua identidade. Então o cliente gera um **segredo pré-mestre** aleatório de 46 bytes, pms , e envia $\text{cpms} = E(k_s)(\text{pms})$ ao servidor. O servidor recupera $\text{pms} = D(k_o)(\text{cpms})$. Agora, tanto o cliente quanto o servidor estão de posse de n_c , n_s e pms , e cada um pode calcular um **segredo mestre** compartilhado de 48 bytes $\text{ms} = f(n_c, n_s, \text{pms})$, onde f é uma função unidirecional e resistente à colisão. Apenas o servidor e o cliente podem calcular ms , pois somente eles conhecem pms . Além do mais, a dependência de ms sobre n_c e n_s garante que ms é um valor *fresco*, ou seja, uma chave de sessão que não foi usada em uma comunicação anterior. Nesse ponto, o cliente e o servidor calculam as seguintes chaves do ms :

- Uma chave de codificação simétrica k_{cs}^{crypt} para codificar mensagens do cliente ao servidor.
- Uma chave de codificação simétrica k_{sc}^{crypt} para codificar mensagens do servidor ao cliente.
- Uma chave de geração MAC k_{cs}^{mac} para gerar autenticadores sobre mensagens do cliente ao servidor.
- Uma chave de geração MAC k_{sc}^{mac} para gerar autenticadores sobre mensagens do servidor ao cliente.

Para enviar uma mensagem m ao servidor, o cliente envia

$$c = E(k_{cs}^{\text{crypt}})(\langle m, S(k_{cs}^{\text{mac}})(m) \rangle).$$

Ao receber c , o servidor recupera

$$\langle m, a \rangle = D(k_{cs}^{\text{crypt}})(c)$$

e aceita m se $V(k_{cs}^{\text{mac}})(m, a) = \text{true}$. De modo semelhante, para enviar uma mensagem m ao cliente, o servidor envia

$$c = E(k_{sc}^{\text{crypt}})(\langle m, S(k_{sc}^{\text{mac}})(m) \rangle)$$

e o cliente recupera

$$\langle m, a \rangle = D(k_{sc}^{\text{crypt}})(c)$$

e aceita m se $V(k_{sc}^{\text{mac}})(m, a) = \text{true}$.

Esse protocolo permite que o servidor limite os destinatários de suas mensagens ao cliente que

gerou pms e limite os emissores das mensagens que ele aceita a esse mesmo cliente. De modo semelhante, o cliente pode limitar os destinatários das mensagens que envia e o emissor das mensagens que aceita ao parceiro que conhece $S(k_d)$ (ou seja, o parceiro que pode decodificar $cpms$). Em muitas aplicações, como transações Web, o cliente precisa verificar a identidade do parceiro que conhece $S(k_d)$. Essa é uma finalidade do certificado $cert_s$; em particular, o campo $attrs$ contém informações que o cliente pode usar para determinar a identidade - por exemplo, o nome de domínio - do servidor com o qual está se comunicando. Para aplicações em que o servidor também precisa de informações sobre o cliente, o SSL aceita uma opção pela qual um cliente pode enviar um certificado ao servidor.

Além do seu uso na Internet, o SSL está sendo usado para uma grande variedade de tarefas. Por exemplo, VPNs IPSec agora têm um competidor nas VPNs SSL. O IPSec é bom para a criptografia ponto a ponto do tráfego - digamos, entre dois escritórios da empresa. As VPNs SSL são mais flexíveis, mas não tão eficientes, de modo que poderiam ser usadas entre um funcionário individual trabalhando remotamente e o escritório corporativo.

15.5 Autenticação do usuário

A discussão anterior sobre autenticação envolve mensagens e sessões. Mas, e os usuários? Se um sistema não pode autenticar um usuário, então não tem sentido autenticar que uma mensagem veio desse usuário. Um problema de segurança importante para sistemas operacionais é a **autenticação do usuário**. O sistema de proteção depende da capacidade de identificar os programas e os processos em execução, que, por sua vez, depende da capacidade de identificar cada usuário no sistema. Os usuários normalmente se identificam. Como determinamos se a identidade de um usuário é autêntica? Em geral, a autenticação do usuário é baseada em um ou mais de três itens: posse do usuário de algo (uma chave ou cartão), conhecimento do usuário de algo (um identificador de usuário e senha) e/ou um atributo do usuário (impressão digital, padrão de retina ou assinatura).

15.5.1 Senhas

A técnica mais comum para autenticar a identidade de um usuário é o uso de **senhas**. Quando o usuário se identifica por ID de usuário ou nome de conta, ele precisa informar uma senha. Se a senha fornecida pelo usuário combinar com a senha armazenada no sistema, o sistema considera que a conta está sendo acessada pelo proprietário dessa conta.

As senhas são usadas para proteger objetos no sistema computadorizado, na ausência de esquemas de proteção mais completos. Elas podem ser consideradas um caso especial de chaves ou capacidades. Por exemplo, uma senha pode estar associada a cada recurso (como um arquivo). Sempre que for feita uma requisição para usar o recurso, a senha precisa ser fornecida. Se ela estiver correta, o acesso é concedido. Diferentes senhas podem estar associadas a diferentes direitos de acesso. Por exemplo, diferentes senhas podem ser usadas para ler arquivos, acrescentar dados a arquivos e atualizar arquivos.

Na prática, a maioria dos sistemas exige apenas uma senha para um usuário ganhar todos os direitos. Embora mais senhas teoricamente seja mais seguro, esses sistemas costumam não ser implementados devido à escolha clássica entre segurança e conveniência. Se a segurança tornar algo inconveniente, então a segurança constantemente é contornada ou totalmente evitada.

15.5.2 Vulnerabilidades da senha

As senhas são bastante comuns porque são fáceis de entender e usar. Infelizmente, essas senhas podem ser adivinhadas, expostas de forma acidental, farejadas ou transferidas ilegalmente de um usuário autorizado para outro não autorizado, como mostramos em seguida.

Existem duas maneiras comuns de adivinhar uma senha. Uma delas é quando o intruso (seja humano ou programa) conhece o usuário ou tem informações sobre ele. Constantemente, as pessoas utilizam informações óbvias (como os nomes de seus gatos ou cônjuges) como senhas. A outra maneira é usar a força bruta, tentar a enumeração ou todas as combinações possíveis de caracteres de senha válidos (letras, números e sinais de pontuação), até a senha ser descoberta. Senhas curtas são bem vulneráveis nesse método. Por exemplo, uma senha de quatro caracteres numéricos provê apenas 10.000 variações. Na média, adivinhar 5.000 vezes produziria uma senha correta. Um programa que pudesse tentar uma senha a cada milissegundo levaria cerca de 5 segundos para adivinhar uma senha de quatro caracteres numéricos. A enumeração tem menos sucesso onde sistemas permitem senhas maiores, que incluem maiúsculas e minúsculas, além do uso de números e todos os caracteres de pontuação. Naturalmente, os usuários precisam tirar proveito do maior espaço de senhas e não devem, por exemplo, usar apenas letras minúsculas.

Além de serem adivinhadas, as senhas podem ser expostas como resultado de monitoramento visual ou eletrônica. Um intruso pode olhar sobre os ombros de um usuário (**shoulder surfing**) quando ele estiver se conectando e descobrir a senha observando o teclado. Como alternativa, qualquer pessoa com acesso à rede em que um computador reside poderia acrescentar tranquilamente um monitor de rede, permitindo observar todos os dados transferidos na rede (**sniffing**), incluindo os IDs de usuário e senhas. O uso da criptografia do fluxo de dados contendo a senha resolve esse problema. Contudo, até mesmo esse sistema poderia ter senhas roubadas. Por exemplo, se um arquivo for usado para conter as senhas, ele poderia ser copiado para análise fora do sistema. Ou, então, considere um programa de cavalo de Troia instalado no sistema que capture cada toque de tecla antes de enviá-lo para a aplicação.

A exposição é um problema muito grave se a senha estiver escrita onde puder ser lida ou perdida. Conforme veremos, alguns sistemas forçam os usuários a selecionar senhas difíceis de lembrar ou longas, o que pode fazer um usuário registrar a senha ou a reutilizar. Como resultado, esses sistemas proveem muito menos segurança do que os sistemas que permitem que os usuários selecioneem senhas fáceis!

O último método de comprometimento de senha, a transferência ilegal, é o resultado da natureza humana. A maioria das instalações de computador possui uma regra que proíbe os usuários de compartilhar contas. Essa regra às vezes é implementada por motivos contábeis, mas visa a

melhorar a segurança. Por exemplo, suponha que um ID de usuário seja compartilhado por vários usuários e ocorra uma brecha na segurança a partir desse ID de usuário. É impossível saber quem estava usando esse ID de usuário no momento em que houve a invasão no sistema ou mesmo se o usuário estava autorizado. Com um usuário por ID de usuário, qualquer usuário pode ser questionado diretamente sobre o uso de sua conta; além disso, o usuário poderia observar algo diferente sobre a conta e detectar a invasão. Às vezes, os usuários quebram regras de compartilhamento de senhas para ajudar amigos ou para contornar a contabilidade, e esse comportamento pode resultar em um sistema acessado por usuários não autorizados - possivelmente maldosos.

As senhas podem ser geradas pelo sistema ou selecionadas por um usuário. As senhas geradas pelo sistema podem ser difíceis de lembrar e, assim, os usuários podem escrevê-las. Entretanto, como mencionado, as senhas selecionadas pelo usuário costumam ser fáceis de adivinhar. Alguns sistemas verificarão uma senha proposta por facilidade de descoberta antes de aceitá-la. Em algumas instalações, os administradores verificam as senhas do usuário e notificam um usuário se a senha for fácil de adivinhar. Alguns sistemas também envelhecem senhas, forçando os usuários a trocarem suas senhas em intervalos regulares (a cada três meses, por exemplo). Esse método também não é à prova de falhas, pois os usuários podem alternar entre duas senhas. A solução implementada em alguns sistemas é registrar um histórico de senhas para cada usuário. Por exemplo, o sistema poderia registrar as últimas N senhas e não permitir sua reutilização.

Diversas variantes sobre esses esquemas de senha simples podem ser usadas. Por exemplo, a senha pode ser alterada com frequência. No caso extremo, a senha é alterada de uma sessão para outra. Uma nova senha é selecionada (seja pelo sistema ou pelo usuário) no final de *cada* sessão, e essa senha precisa ser usada para a próxima seção. Nesse caso, mesmo que uma senha seja mal utilizada, ela só pode ser usada uma vez. Quando o usuário legítimo tentar usar uma senha agora inválida na próxima sessão, ele descobrirá a violação na segurança. Nesse caso, podem ser tomadas medidas para reparar a falha na segurança.

15.5.3 Senhas criptografadas

Um problema com todas essas técnicas é a dificuldade de manter a senha secreta dentro do computador. Como o sistema pode armazenar uma senha com segurança e permitir seu uso para autenticação quando o usuário apresentar sua senha? O sistema UNIX utiliza a criptografia para evitar a necessidade de manter sua lista de senhas secretas. Cada usuário possui uma senha. O sistema contém uma função bem difícil - os projetistas esperam que seja impossível - de inverter, mas simples de calcular, ou seja, dado um valor x , é fácil calcular o valor da função $f(x)$. Todavia, dado o valor da função $f(x)$, é impossível calcular x . Essa função é usada para codificar todas as senhas. Somente senhas codificadas são armazenadas. Quando um usuário apresenta uma senha, ela é codificada e comparada com a senha codificada armazenada. Mesmo que a senha codificada armazenada seja vista, ela não poderá ser decodificada, de modo que a senha não poderá ser determinada. Assim, o arquivo de senhas não precisa ser mantido em segredo. A função $f(x)$ normalmente é um algoritmo de criptografia projetado e testado rigorosamente.

A falha nesse método é que o sistema não tem mais o controle sobre as senhas. Embora as senhas sejam criptografadas, qualquer um com uma cópia do arquivo de senhas pode executar rápidas rotinas de criptografia contra ela, criptografando cada palavra em um dicionário, por exemplo, e comparando os resultados com as senhas. Se o usuário tiver selecionado uma senha que também é uma palavra no dicionário, a senha estará decifrada. Em computadores suficientemente velozes ou ainda em grupos de computadores lentos, tal comparação pode levar apenas algumas horas. Além disso, como os sistemas UNIX utilizam um algoritmo de criptografia bem conhecido, um cracker poderia manter um cache de senhas decifradas anteriormente. Por esse motivo, novas versões do UNIX armazenam as entradas de senha criptografadas em um arquivo que só pode ser lido pelo **superusuário**. Os programas que compararam uma senha apresentada e a senha armazenada executam `setuid` para root; assim, eles podem ler esse arquivo, mas outros usuários não podem. Eles também incluem um "sal", ou número aleatório registrado, no algoritmo de codificação. O sal é acrescentado à senha para garantir que, se duas senhas em texto puro forem iguais, elas resultem em diferentes textos cifrados.

Outro ponto fraco nos métodos de senha do UNIX é que muitos sistemas UNIX tratam apenas os oito primeiros caracteres como significativos. Portanto, é extremamente importante que os usuários tirem proveito do espaço de senha disponível. Para evitar o método de criptografia do dicionário, alguns sistemas não permitem o uso de palavras do dicionário como senhas. Uma boa técnica é gerar sua senha usando a primeira letra de cada palavra de uma frase de fácil lembrança, usando caracteres maiúsculos e minúsculos com um número ou sinal de pontuação incluído, por medida de segurança. Por exemplo, a frase "O nome da minha mãe é Catarina" poderia gerar a senha "OndmmeC.".! A senha é difícil de ser descoberta, mas fácil para o usuário lembrar.

15.5.4 Senhas de única vez

Para evitar os problemas de farejamento (sniffing) de senha e espreita sobre os ombros, um sistema pode usar um conjunto de **senhas emparelhadas**. Quando uma sessão é iniciada, o sistema seleciona aleatoriamente e apresenta uma parte de um par de senhas; o usuário precisa fornecer a outra parte. Nesse sistema, o usuário é **desafiado** e precisa **responder** com a resposta correta a esse desafio.

Essa técnica pode ser generalizada para o uso de um algoritmo como uma senha. O algoritmo poderia ser uma função com inteiros, por exemplo. O sistema seleciona um inteiro aleatório e o apresenta ao usuário. O usuário aplica a função e responde com o resultado correto. O sistema também aplica a função. Se os dois resultados combinarem, o acesso é permitido.

Essas senhas algorítmicas não são suscetíveis à exposição, ou seja, um usuário pode digitar uma senha e nenhuma entidade que intercepte essa senha poderá reutilizá-la. Nessa variação, o sistema e o usuário compartilham um segredo. O segredo nunca é transmitido por um meio que permita exposição. Em vez disso, o segredo é usado como entrada para a função, junto com uma semente compartilhada. Uma **semente** é uma sequência numérica ou alfanumérica aleatória. A semente é o desafio de autenticação do computador. O segredo e a semente são usados como entrada para a função $f(\text{segredo}, \text{semente})$. O resultado dessa função é transmitido como senha para o computador. Como o computador também sabe o segredo e a semente, ele pode realizar o mesmo cálculo. Se os resultados combinarem, o usuário é autenticado. Da próxima vez que o usuário precisar ser autenticado, outra semente é gerada, e as mesmas etapas acontecem. Dessa vez, a senha é diferente.

Nesse sistema de **senha de única vez (one-time password)**, a senha é diferente em cada ocasião. Qualquer um que capture a senha de uma sessão e tente reutilizá-la em outra sessão falhará. As senhas de única vez estão entre as únicas maneiras de evitar a autenticação imprópria devido à exposição de senha.

Os sistemas de senha de única vez são implementados de várias maneiras. As implementações comerciais, como SecurID, utilizam calculadoras de hardware. A maioria dessas calculadoras está na forma de um cartão de crédito, chaveiro ou dispositivo USB; elas incluem uma tela e podem ou não ter um teclado. Algumas utilizam a hora atual como a semente aleatória. Outras exigem que o usuário utilize o teclado para informar o segredo compartilhado, também conhecido como **número de identificação pessoal (Personal Identification Number - PIN)**. O visor mostra a senha de única vez. O uso de um gerador de senha de única vez e um PIN é uma forma de **autenticação de fator dois**. Dois tipos diferentes de componentes são necessários nesse caso. A autenticação de fator dois provê uma proteção por autenticação muito melhor do que a autenticação de fator único.

Outra variação das senhas de única vez é o uso de um **livro-código**, ou **bloco de única vez (one-time pad)**, que é uma lista de senhas de único uso. Cada senha na lista é usada uma única vez, e depois é riscada ou apagada. O sistema S/Key, bastante utilizado, usa uma calculadora de software ou um livro-código baseado nesses cálculos como fonte das senhas de única vez. Logicamente, o usuário precisa proteger seu livro-código.

15.5.5 Biometria

Outra variação no uso de senhas para autenticação envolve o uso de medições biométricas. Leitores de palma de mão normalmente são usados para proteger o acesso físico – por exemplo, o acesso a um centro de dados. Esses leitores combinam parâmetros armazenados com o que está sendo lido dos aparelhos leitores de mão. Os parâmetros podem incluir um mapa de temperatura, além de tamanho do dedo, largura do dedo e padrões de linhas. Esses dispositivos atualmente são muito grandes e caros para serem usados para autenticação normal no computador.

Os leitores de impressão digital se tornaram precisos e econômicos e deverão se tornar mais comuns no futuro. Esses dispositivos leem os padrões de sulcos do dedo e os convertem em uma sequência de números. Com o tempo, eles podem armazenar um conjunto de sequências para se ajustar ao local do dedo no dispositivo de leitura e outros fatores. O software pode, então, digitalizar um dedo no aparelho e comparar suas características com aquelas sequências armazenadas, para determinar se elas correspondem. Naturalmente, vários usuários podem ter perfis armazenados, e o scanner pode diferenciar entre eles. Um esquema de autenticação de fator dois muito preciso pode resultar em exigir uma senha além de um nome de usuário e leitura da impressão digital. Se essa informação for criptografada em trânsito, o sistema poderá ser bastante resistente a ataques de falsificação ou reprodução.

A **autenticação multifator** é ainda melhor. Considere como a autenticação pode ser forte com um dispositivo USB que seja conectado ao sistema, um PIN e uma varredura de impressão digital. Exceto pelo fato de que o usuário precisa colocar seu dedo sobre um dispositivo e conectar o USB no sistema, esse método de autenticação não é menos conveniente do que usar as senhas normais. Lembre-se, porém, que a autenticação forte por si só não é suficiente para garantir a ID do usuário. Uma sessão de autenticação ainda pode ser sequestrada, se não for codificada.

15.6 Implementando defesas de segurança

Assim como existem milhares de ameaças à segurança do sistema e da rede, existem muitas soluções de segurança. As soluções variam desde maior treinamento aos usuários, tecnologia, até escrita de software sem bugs. A maioria dos profissionais de segurança adere à teoria da **defesa em profundidade**, que afirma que mais camadas de defesa é melhor do que menos camadas. Naturalmente, essa teoria se aplica a qualquer tipo de segurança. Considere a segurança de uma casa sem uma fechadura, com uma fechadura e com uma fechadura e um alarme. Nesta seção, examinamos os principais métodos, ferramentas e técnicas que podem ser usados para melhorar a resistência às ameaças.

15.6.1 Política de segurança

O primeiro passo em direção à melhoria da segurança de qualquer aspecto de computação é ter uma **política de segurança**. As políticas variam bastante, mas geralmente incluem uma declaração do que está sendo protegido. Por exemplo, uma política poderia indicar que todas as aplicações acessíveis de fora da empresa devem ter uma revisão de código antes de serem implantadas ou que os usuários não devem compartilhar suas senhas, ou que todos os pontos de conexão entre uma empresa e o exterior devem ter varreduras de porta executadas a cada seis meses. Sem uma política, é impossível que usuários e administradores saibam o que é permitido, o que é obrigatório e o que não é permitido. A política é um mapa de estrada para a segurança, e se um site estiver tentando passar de menos seguro para mais seguro ele precisa de um mapa para saber como chegar lá.

Quando a política de segurança estiver pronta, as pessoas que ela afeta deverão conhecê-la bem. Ela deverá ser um guia. A política também deverá ser um **documento vivo** que seja visto e atualizado periodicamente, para garantir que ainda seja pertinente e continua sendo seguido.

15.6.2 Avaliação de vulnerabilidade

Como podemos determinar se uma política de segurança foi implementada corretamente? A melhor maneira é executar uma avaliação de vulnerabilidade. Essas avaliações podem cobrir um grande terreno, desde a engenharia social até a avaliação de risco para as varreduras de porta. Por exemplo, a **avaliação de risco** valoriza os bens da entidade em questão (um programa, uma equipe de gerência, um sistema ou uma instalação) e determina as chances de que um incidente de segurança afete a entidade e diminua seu valor. Quando as chances de sofrer uma perda e a quantidade de perda em potencial são conhecidas, um valor pode ser determinado para tentar proteger a entidade.

A principal atividade da maioria das avaliações de vulnerabilidade é um **teste de penetração**, em que a entidade é varrida em busca de vulnerabilidades conhecidas. Como este livro se refere a sistemas operacionais e o software que é executado neles, vamos nos concentrar nos aspectos da avaliação de vulnerabilidade.

As varreduras de vulnerabilidade normalmente são feitas em ocasiões em que o uso do computador é relativamente baixo, para minimizar seu impacto. Quando apropriado, elas são feitas em sistemas de teste, em vez de sistemas em produção, pois podem induzir um comportamento impróprio dos sistemas-alvo ou dispositivos de rede.

Uma varredura dentro de um sistema individual pode verificar diversos aspectos do sistema:

- Senhas curtas ou fáceis de descobrir.
- Programas privilegiados não autorizados, como programas *setuid*.
- Programas não autorizados nos diretórios do sistema.
- Processos de longa duração não previstos.
- Proteções de diretório impróprias nos diretórios do usuário e do sistema.
- Proteções impróprias nos arquivos de dados do sistema, como arquivo de senhas, drivers de dispositivos ou até mesmo o próprio kernel do sistema operacional.
- Entradas perigosas no caminho de busca de programas (por exemplo, um cavalo de Troia, discutido na [Seção 15.2.1](#)).
- Mudanças nos programas do sistema detectadas via checksum.
- Daemons de rede inesperados ou ocultos.

Quaisquer problemas encontrados por uma sondagem de segurança poderão ser consertados automaticamente ou informados aos gerentes do sistema.

Os computadores em rede são muito mais suscetíveis a ataques de segurança do que os sistemas independentes. Em vez de ataques de um conjunto conhecido de pontos de acesso, como a conexão direta de terminais, encaramos ataques de um conjunto amplo e desconhecido de pontos de acesso – um problema de segurança potencialmente grave. Até certo ponto, os sistemas conectados a linhas telefônicas via modem também estão mais expostos.

De fato, o governo dos Estados Unidos considera um sistema tão seguro quanto sua conexão mais

distante. Por exemplo, um sistema altamente secreto só pode ser acessado de dentro de um prédio também considerado altamente secreto. O sistema perde sua classificação de altamente secreto se qualquer forma de comunicação puder ocorrer fora desse ambiente. Algumas instalações do governo tomam precauções de segurança extremas. Os conectores que encaixam um terminal ao computador seguro são trancados em um cofre no escritório quando o terminal não está em uso. Uma pessoa precisa ter ID apropriada para obter acesso ao prédio e ao seu escritório, precisa saber uma combinação de cofre físico, além de informações de autenticação para o próprio computador, para obter acesso ao computador - um exemplo de autenticação multifator.

Infelizmente, para administradores de sistemas e profissionais de segurança de computador, é impossível trancar uma máquina em uma sala e desativar todo o acesso remoto. Por exemplo, a Internet conecta milhões de computadores. Ela está se tornando um recurso de missão crítica indispensável para muitas empresas e indivíduos. Se você considerar a Internet um clube, então, como em qualquer clube com milhões de membros, existem muitos membros bons e membros maus. Os membros maus possuem muitas ferramentas que podem usar para tentar obter acesso às comunicações interligadas, assim como Morris fez com seu worm.

As varreduras de vulnerabilidade podem ser aplicadas às redes para resolver alguns dos problemas com a segurança da rede. As varreduras pesquisam uma rede em busca de portas que respondem a uma requisição. Se serviços forem ativados mas não deveriam ser, o acesso a eles pode ser bloqueado ou eles podem ser desativados. As varreduras, então, determinam os detalhes da aplicação escutando nessa porta e tentam determinar se cada uma possui alguma vulnerabilidade conhecida. O teste dessas vulnerabilidades pode determinar se o sistema está mal configurado ou se estão faltando patches necessários.

Finalmente, considere o uso de ferramentas de varredura de porta nas mãos de um cracker, em vez de alguém tentando melhorar a segurança. Essas ferramentas poderiam ajudar os crackers a encontrar vulnerabilidades para atacar. (Felizmente, é possível detectar varreduras de porta por meio da detecção de anomalia, conforme discutimos a seguir.) É um desafio geral à segurança que as mesmas ferramentas possam ser usadas para o bem e para o mal. De fato, algumas pessoas defendem a **segurança pela obscuridade**, afirmando que nenhuma ferramenta deveria ser escrita para testar a segurança, pois essas ferramentas podem ser usadas para encontrar (e explorar) brechas na segurança. Outros acreditam que essa técnica de segurança não é válida, indicando, por exemplo, que os crackers poderiam escrever suas próprias ferramentas. Parece razoável que a segurança pela obscuridade seja considerada uma das camadas de segurança, desde que não seja a única camada. Por exemplo, uma empresa poderia publicar sua informação inteira de configuração de rede; mas manter essa informação secreta torna mais difícil para os intrusos saberem o que atacar ou determinarem o que poderia ser detectado. Contudo, mesmo aqui, uma empresa supondo que essa informação continuará um segredo terá uma falsa sensação de segurança.

15.6.3 Detecção de intrusão

Proteger sistemas e instalações está intimamente ligado à detecção de intrusão. A **detecção de intrusão**, como o nome sugere, se esforça para detectar intrusões tentadas ou realizadas nos sistemas computadorizados e para iniciar respostas apropriadas às intrusões. A detecção de intrusão compreende grande variedade de técnicas, que variam por diversos eixos, incluindo os seguintes:

- O momento em que a detecção ocorre. A detecção pode ocorrer em tempo real (enquanto a intrusão está ocorrendo) ou depois do fato.
- Os tipos de entradas examinadas para detectar a atividade intrusiva. Podem incluir comandos de shell do usuário, chamadas de sistema feitas pelo processo e cabeçalhos ou conteúdo do pacote de rede. Algumas formas de intrusão só podem ser detectadas pela correlação de informações de várias dessas fontes.
- O intervalo de capacidades de resposta. Formas simples de resposta incluem alerta de um administrador à intrusão em potencial ou interrupção de alguma forma da atividade potencialmente intrusiva - por exemplo, encerrando um processo engajado em tal atividade. Em uma forma sofisticada de resposta, um sistema poderia desviar a atividade de um intruso para um **pote de mel (honeypot)** - um recurso falso, uma isca para o atacante. Para o invasor, o recurso parece ser real e permite ao sistema monitorar e obter informações sobre o ataque.

Esses graus de liberdade no espaço do projeto para detectar intrusões geraram uma grande variedade de soluções, conhecidas como **sistemas de detecção de intrusão (Intrusion Detection Systems - IDSs)** e **sistemas de prevenção de intrusão (Intrusion Prevention Systems -- IDPs)**. Os IDSs disparam um alarme quando uma intrusão é detectada, enquanto os IDPs atuam como roteadores, passando o tráfego a menos que uma intrusão seja detectada (no ponto em que o tráfego é bloqueado).

Em que constitui uma intrusão? A definição de uma especificação adequada de intrusão é muito difícil, e IDSs e IDPs automáticos hoje em dia utilizam uma de duas técnicas menos ambiciosas. Na primeira, chamada **detecção baseada em assinatura**, a entrada do sistema ou o tráfego da rede é examinado em busca de padrões de comportamento específicos (ou **assinaturas**), reconhecidos por

indicar ataques. Um exemplo simples de detecção baseada em assinatura é a busca de pacotes de rede com a sequência `/etc/passwd` visando a um sistema UNIX. Outro exemplo é um software de detecção de vírus, que varre os binários ou pacotes de rede em busca de vírus conhecidos por sua assinatura.

A segunda técnica, denominada **deteção de anomalia**, por meio de diversas técnicas, tenta detectar um comportamento anômalo dentro dos sistemas computadorizados. Naturalmente, nem toda atividade anômala no sistema indica uma intrusão, mas presume-se que as intrusões induzem um comportamento anômalo. Um exemplo de detecção de anomalia é o monitoramento de chamadas de sistema de um processo daemon para detectar se o comportamento da chamada de sistema é diferente dos padrões normais, possivelmente indicando que um estouro de buffer foi explorado no daemon para adulterar seu comportamento. Outro exemplo é o monitoramento de comandos do shell para detectar comandos anômalos para determinado usuário ou detectar um horário de login estranho para o usuário, ambos indicando que um atacante conseguiu obter acesso à conta desse usuário.

A detecção baseada em assinatura e a detecção de anomalia podem ser vistas como dois lados da mesma moeda: a detecção baseada em assinatura tenta caracterizar comportamentos perigosos e detectar quando um desses comportamentos ocorre, enquanto a detecção de anomalia tenta caracterizar os comportamentos normais (ou não perigosos) e detectar quando está ocorrendo algo diferente desses comportamentos.

Contudo, essas técnicas diferentes geram IDSs e IDPs com propriedades muito diferentes. Em particular, a detecção de anomalia pode detectar métodos de intrusão previamente desconhecidos (chamados **ataques do dia zero**). Por sua vez, a detecção baseada em assinatura identificará apenas ataques conhecidos, que possam ser codificados em um padrão reconhecível. Assim, novos ataques não contemplados quando as assinaturas foram geradas escaparão da detecção baseada em assinatura. Esse problema é muito bem conhecido dos fornecedores de software de detecção de vírus, que precisam lançar novas assinaturas com muita frequência, à medida que novos vírus são detectados manualmente.

No entanto, a detecção de anomalia não é necessariamente superior à detecção baseada em assinatura. Na verdade, um desafio significativo para os sistemas que tentam detectar anomalias é avaliar o comportamento “normal” do sistema com precisão. Se o sistema já tiver sido invadido quando for avaliado, então a atividade invasora pode ser incluída na avaliação “normal”. Mesmo que o sistema seja avaliado de forma limpa, sem a influência de algum comportamento intrusivo, a avaliação precisa prover uma imagem razoavelmente completa do comportamento normal. Caso contrário, a quantidade de **falsos positivos** (alarmes falsos) ou, pior, **falsos negativos** (intrusões perdidas) será excessiva.

Para ilustrar o impacto de até mesmo uma taxa não muito alta de alarmes falsos, considere uma instalação que consiste em centenas de estações de trabalho UNIX, nas quais eventos relevantes à segurança são registrados para fins de detecção de intrusão. Uma pequena instalação como essa poderia gerar um milhão de registros de auditoria por dia. Somente um ou dois poderiam merecer a investigação de um administrador. Se supusermos, de forma otimista, que cada ataque desse tipo é refletido em dez registros de auditoria, podemos então calcular aproximadamente a taxa de ocorrência de registros de auditoria refletindo a atividade intrusiva como

$$\frac{2 \frac{\text{intrusões}}{\text{dia}} \cdot 10 \frac{\text{registros}}{\text{intrusão}}}{10^6 \frac{\text{registros}}{\text{dia}}} = 0,00002$$

Interpretando isso como uma “probabilidade da ocorrência de registros de intrusão”, indicamos o valor como $P(I)$, ou seja, o evento I é a ocorrência de um registro refletindo o comportamento verdadeiramente intrusivo. Como $P(I) = 0,00002$, também sabemos que $P(\neg I) = 1 - P(I) = 0,99998$. Agora, considere A o evento que levanta um alarme pelo IDS. Um IDS preciso deverá maximizar tanto $P(I|A)$ quanto $P(\neg I|\neg A)$ – ou seja, as probabilidades de que um alarme indique uma intrusão e que nenhum alarme indique nenhuma intrusão. Focalizando $P(I|A)$ por enquanto, podemos calculá-lo usando o **teorema de Bayes**:

$$P(I|A) = \frac{P(I) \cdot P(A|I)}{P(I) \cdot P + P(\neg I) \cdot P(\neg A|\neg I)}$$

$$= \frac{0,00002 \cdot P(A|I)}{0,00002 \cdot P(A|I) + (0,99998 \cdot P(A|\neg I))}$$

Agora, considere o impacto da taxa de alarme falso $P(A|\neg I)$ sobre $P(I|A)$. Mesmo com uma taxa de alarme verdadeiro muito boa $P(A|I) = 0,8$, uma taxa de alarme falso aparentemente boa $P(A|\neg I) = 0,0001$ gera $P(I|A) \approx 0,14$, ou seja, menos de um em cada sete alarmes indica uma intrusão real! Em sistemas em que um administrador de segurança investiga cada alarme, uma alta taxa de alarmes falsos - chamada "efeito árvore de Natal" - é um grande desperdício e logo ensinaria o administrador a ignorar os alarmes.

Esse exemplo ilustra um princípio geral para os IDSs e IDPS: para ser útil, um IDS precisa prover uma taxa de alarme falso muito baixa. Alcançar uma taxa de alarme falso suficientemente baixa é um desafio sério para sistemas de detecção de anomalia, como mencionado, devido às dificuldades da avaliação adequada do comportamento normal do sistema. Todavia, a pesquisa continua a melhorar as técnicas de detecção de anomalia. O software de detecção de intrusão está evoluindo para implementar assinaturas, algoritmos de anomalia e outros algoritmos, e combinar os resultados para chegar a uma taxa de detecção de anomalia mais precisa.

15.6.4 Proteção contra vírus

Como já vimos, os vírus podem danificar e realmente danificam os sistemas. A proteção contra vírus, portanto, é um aspecto de segurança importante. Os programas antivírus normalmente são utilizados para fornecer essa proteção. Alguns desses pacotes antivírus comerciais é eficaz apenas contra vírus conhecidos em particular. Eles atuam pesquisando todos os programas em um sistema em busca do padrão específico de instruções conhecidas por compor o vírus. Quando eles encontram um padrão conhecido, removem as instruções, **desinfetando** o programa. Os programas antivírus possuem catálogos de milhares de vírus pelos quais procuram.

Os vírus e o software antivírus continuam a se tornar mais sofisticados. Alguns vírus se modificam enquanto infectam outro software, para evitar a técnica básica de combinação de padrão do software antivírus. O software antivírus, por sua vez, agora procura famílias de padrões, em vez de um único padrão, para identificar um vírus. De fato, alguns programas antivírus implementam diversos algoritmos de detecção. Eles podem descompactar os vírus compactados antes de procurar uma assinatura. Alguns procuram anomalias de processo. Um processo abrindo um arquivo executável para escrita é suspeito, por exemplo, a menos que seja um compilador. Outra técnica popular é executar um programa em uma **caixa de brita**, que é uma seção do sistema controlada e simulada. O software antivírus analisa o comportamento do código na caixa de brita antes de deixar que ele seja executado sem monitoramento. Alguns programas antivírus também montam um escudo completo, em vez de apenas varrer os arquivos dentro de um sistema de arquivos. Eles pesquisam setores de boot, memória, correio eletrônico recebido e emitido, arquivos enquanto são baixados, arquivos em dispositivos ou mídia removível, e assim por diante.

A melhor proteção contra vírus de computador é a prevenção ou a prática de **computação segura**. Comprar software não aberto de fornecedores e evitar cópias gratuitas ou pirateadas de fontes públicas ou trocar discos fornecem o caminho mais seguro para impedir a infecção. Entretanto, até mesmo cópias novas de aplicações de software legítimas não estão imunes à infecção do vírus: tem havido casos em que funcionários descontentes de uma empresa de software infectaram as cópias mestras dos programas de software para causar prejuízo econômico à empresa. Para os vírus de macro, uma defesa é trocar documentos do Word em um formato de arquivo alternativo, chamado **Rich Text Format (RTF)**. Ao contrário do formato nativo do Word, o RTF não inclui a capacidade de anexar macros.

Outra defesa é evitar a abertura de quaisquer anexos de correio eletrônico de usuários desconhecidos. Infelizmente, a história tem mostrado que as vulnerabilidades do correio eletrônico aparecem tão rapidamente quanto são consertadas. Por exemplo, em 2000, o vírus *do amor* (*love bug*) foi bastante difundido por parecer ser uma nota de amor enviada por um amigo do receptor. Quando o script do Visual Basic anexado era aberto, o vírus se propagava pelo envio de si mesmo aos primeiros endereços na lista de contatos de correio eletrônico do receptor. Felizmente, exceto por entupir os sistemas de correio eletrônico e as caixas de entrada dos usuários, ele era relativamente inofensivo. Contudo, ele efetivamente negava a estratégia defensiva de abrir anexos apenas de pessoas conhecidas do receptor. Um método de defesa mais eficaz é evitar abrir qualquer anexo de correio eletrônico que contenha código executável. Algumas empresas agora impõem isso como política, removendo todos os anexos que chegam com as mensagens de correio eletrônico.

Um exemplo de ferramenta de detecção de anomalia é a ferramenta de verificação de integridade do **sistema de arquivos Tripwire** para UNIX, desenvolvida na Purdue University. O Tripwire opera na premissa de que muitas intrusões resultam na modificação dos diretórios e arquivos do sistema. Por exemplo, um atacante poderia modificar os programas do sistema, talvez inserindo cópias com cavalos de Troia, ou poderia inserir novos programas em diretórios normalmente encontrados nos caminhos de busca do shell do usuário. Ou, então, um intruso poderia remover arquivos de log do sistema para encobrir seus rastros. O Tripwire é uma ferramenta para monitorar os sistemas de arquivos em busca de arquivos acrescentados, excluídos ou alterados, e alertar os administradores de sistemas sobre essas modificações.

A operação do Tripwire é controlada por um arquivo de configuração tw.config, que enumera os diretórios e arquivos a serem monitorados em busca de mudanças, exclusões e acréscimos. Cada entrada nesse arquivo de configuração inclui uma máscara de seleção para especificar os atributos do arquivo (atributos de inode) que será monitorada para as mudanças. Por exemplo, a máscara de seleção poderia especificar que as permissões de um arquivo sejam monitoradas, mas seu tempo de acesso seja ignorado. Além disso, a máscara de inserção pode instruir que o arquivo seja monitorado para mudanças. O monitoramento do hash de um arquivo para as mudanças é tão bom quanto o monitoramento do próprio arquivo, mas o armazenamento de hashes de arquivos exige muito menos espaço do que copiar os próprios arquivos.

Quando executado inicialmente, o Tripwire toma como entrada o arquivo tw.config e calcula uma assinatura para cada arquivo ou diretório, consistindo em seus atributos monitorados (atributos de inode e valores de hash). Essas assinaturas são armazenadas em um banco de dados. Quando executado mais tarde, o Tripwire apanha tanto tw.config quanto o banco de dados previamente armazenado, recalcula a assinatura para cada arquivo ou diretório nomeado em tw.config e compara essa assinatura com a assinatura (se houver) no banco de dados calculado anteriormente. Os eventos informados a um administrador incluem qualquer arquivo ou diretório monitorado cuja assinatura difere daquela no banco de dados (um arquivo alterado), qualquer arquivo ou diretório em um diretório monitorado para o qual não existe uma assinatura no banco de dados (um arquivo acrescentado) e qualquer assinatura no banco de dados para a qual o arquivo ou diretório correspondente não existe mais (um arquivo excluído).

Embora eficaz para uma grande classe de ataques, o Tripwire possui suas limitações. Talvez a mais óbvia seja a necessidade de proteger o programa Tripwire e seus arquivos associados, especialmente o arquivo de banco de dados, contra modificação não autorizada. Por esse motivo, o Tripwire e seus arquivos associados devem ser armazenados em algum meio à prova de modificação, como um disco protegido contra escrita ou um servidor seguro, onde os logins possam ser controlados de perto. Infelizmente, isso o torna menos conveniente para atualizar o banco de dados após as atualizações autorizadas nos diretórios e arquivos monitorados. Uma segunda limitação é que alguns arquivos relevantes à segurança – por exemplo, arquivos de log do sistema – *deveriam* mudar com o tempo, e o Tripwire não provê um meio de distinguir entre uma mudança autorizada e outra não autorizada. Assim, por exemplo, um ataque que modificasse (sem excluir) um log do sistema que, de qualquer forma, também seria alterado, escaparia das capacidades de detecção do Tripwire. O melhor que o Tripwire pode fazer nesse caso é detectar algumas inconsistências óbvias (por exemplo, o encolhimento do arquivo de log). Versões gratuitas e comerciais do Tripwire estão disponíveis em <http://tripwire.org> e <http://tripwire.com>.

Outra defesa, embora não impeça a infecção, permite a detecção antecipada. Um usuário precisa começar reformatando completamente seu disco rígido, especialmente o setor de boot, que normalmente é alvo do ataque de vírus. Somente software seguro é carregado, e uma assinatura de cada programa é realizada por meio de um cálculo seguro de síntese de mensagem. A lista resultante de nome de arquivo e síntese de mensagem associada precisa ser mantida protegida contra acesso não autorizado. Periodicamente, ou toda vez que o programa for executado, o sistema operacional recalcula a assinatura e a compara com a assinatura na lista original; qualquer diferença serve como uma advertência de possível infecção. Essa técnica pode ser combinada com outras. Por exemplo, uma varredura de antivírus de alto overhead, como uma caixa de brita, poderá ser utilizada; e se um programa passar no teste, uma assinatura pode ser criada para ele. Se as assinaturas combinarem na próxima vez que o programa for executado, ele não precisará ser varrido novamente em busca de vírus.

15.6.5 Auditoria, contabilidade e logging

Auditoria, contabilidade e logging podem diminuir o desempenho do sistema, mas são úteis em diversas áreas, incluindo a segurança. O logging pode ser geral ou específico. Todas as execuções de chamada de sistema podem ser registradas para análise do comportamento (ou mau comportamento) do programa. Normalmente, os eventos suspeitos são registrados em log. As falhas de autenticação e autorização podem nos dizer muito sobre tentativas de invasão.

A contabilidade é outra ferramenta de potencial no kit de um administrador de segurança. Ela

pode ser usada para encontrar mudanças de desempenho, que por sua vez podem revelar problemas de segurança. Uma das primeiras invasões de computador UNIX foi detectada por Cliff Stoll, quando estava examinando logs de contabilidade e localizou uma anomalia.

15.7 Uso de firewalls para proteger sistemas e redes

Agora vamos nos voltar para a questão como o computador confiável pode ser conectado com segurança a uma rede não confiável. Uma solução é usar um firewall para separar sistemas confiáveis e não confiáveis. Um **firewall** é um computador, aparelho ou roteador que fica entre o confiável e o não confiável. Um firewall em rede limita o acesso à rede entre os dois **domínios de segurança** e monitora e registra todas as conexões. Ele também pode limitar as conexões com base no endereço de origem ou destino, porta de origem ou destino, ou direção da conexão. Por exemplo, os servidores Web utilizam HTTP para se comunicarem com navegadores Web. Um firewall, portanto, pode permitir que somente HTTP passe de todos os hosts fora do firewall para o servidor Web dentro do firewall. O verme Morris da Internet usava o protocolo finger para entrar nos computadores, de modo que o finger não teria permissão para passar, por exemplo.

Na verdade, um firewall em rede pode separar uma rede em vários domínios. Uma implementação comum tem a Internet como o domínio não confiável; uma rede semiconfiável ou semissegura, denominada **zona desmilitarizada (Demilitarized Zone - DMZ)**, como outro domínio; e os computadores de uma empresa, como um terceiro domínio (Figura 15.10). As conexões são permitidas da Internet para os computadores da DMZ e dos computadores da empresa para a Internet, mas não da Internet ou dos computadores da DMZ para os computadores da empresa. Opcionalmente, comunicações controladas podem ser permitidas entre a DMZ e um ou mais computadores da empresa. Por exemplo, um servidor Web na DMZ pode ter de consultar um servidor de banco de dados na rede da empresa. Com um firewall, todo o acesso é restrito e quaisquer sistemas da DMZ que atravessam o firewall ainda serão incapazes de acessar os computadores da empresa.

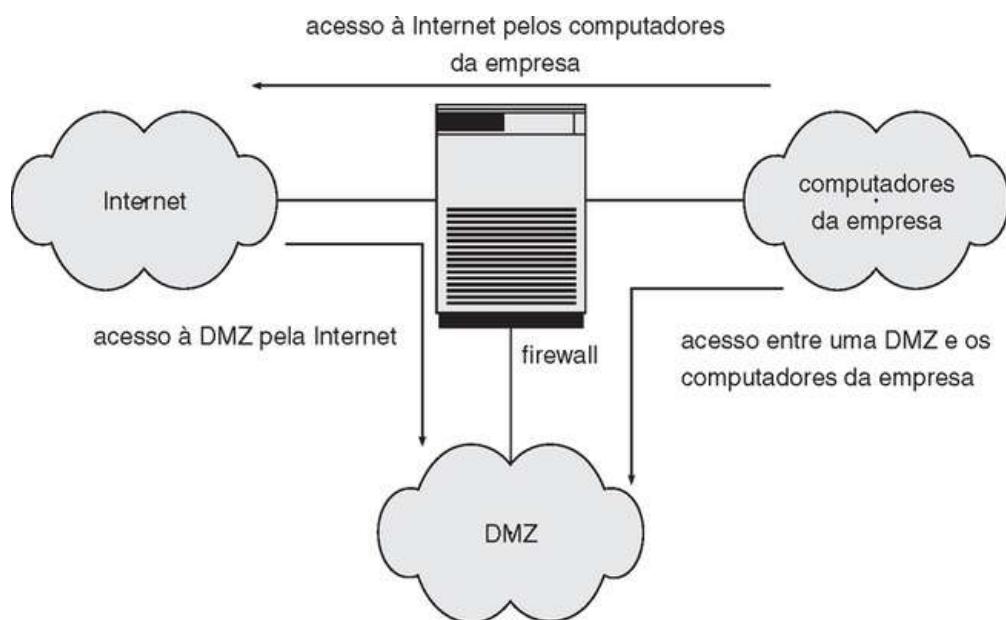


FIGURA 15.10 Separação de domínio via firewall.

Naturalmente, o próprio firewall precisa estar protegido e ser à prova de ataques; caso contrário, sua capacidade de proteger as conexões poderá ser comprometida. Além disso, os firewalls não impedem ataques via **túnel** ou aqueles que são transportados dentro dos protocolos ou conexões que o firewall autoriza. Um ataque de estouro de buffer em um servidor Web não será impedido pelo firewall, por exemplo, porque a conexão HTTP é permitida; é o conteúdo da conexão HTTP que abriga o ataque. De modo semelhante, os ataques de negação de serviço podem afetar os firewalls tanto quanto quaisquer outras máquinas. Outra vulnerabilidade dos firewalls é o **spoofing**, em que um host não autorizado finge ser um host autorizado, atendendo a algum critério de autorização. Por exemplo, se uma regra do firewall permite uma conexão de um host e identifica esse host por seu endereço IP, então outro host poderia enviar pacotes usando esse mesmo endereço e ter permissão para passar pelo firewall.

Além dos firewalls de rede mais comuns, existem outros tipos de firewall, mais recentes, cada um com seus prós e contras. Um **firewall pessoal** é uma camada de software incluída com o sistema operacional ou acrescentada como uma aplicação. Em vez de limitar a comunicação entre os domínios de segurança, ele limita a comunicação para (e possivelmente de) determinado host. Um usuário poderia acrescentar um firewall pessoal ao seu PC de modo que um cavalo de Troia tivesse

acesso negado à rede à qual o PC está conectado, por exemplo. Um **firewall de proxy de aplicação** entende os protocolos que as aplicações falam pela rede. Por exemplo, o SMTP é usado para a transferência de correio. Um proxy de aplicação aceita uma conexão, assim como um servidor SMTP aceitaria, e depois inicia uma conexão com o servidor SMTP de destino original. Ele pode monitorar o tráfego enquanto encaminha a mensagem, procurando e desativando comandos ilegais, tentativas de explorar bugs, e assim por diante. Alguns firewalls são projetados para um protocolo específico. Um **firewall XML**, por exemplo, possui a finalidade específica de analisar tráfego XML e bloquear XML não permitida ou mal formada. **Firewalls de chamada de sistema** ficam entre as aplicações e o kernel, monitorando a execução de chamada de sistema. Por exemplo, no Solaris 10, o recurso de “menor privilégio” implementa uma lista de mais de 50 chamadas de sistema que os processos podem ou não ter permissão para fazer. Um processo que não precisa gerar outros processos pode ter essa capacidade retirada, por exemplo.

15.8 Classificações de segurança de computador

Os critérios de avaliação de computadores confiáveis do Departamento de Defesa dos Estados Unidos especificam quatro divisões de segurança nos sistemas: A, B, C e D. Essa especificação é bastante utilizada para determinar a segurança de uma instalação e modelar soluções e segurança, e por isso a explicamos aqui. A classificação de nível mais baixo é a divisão D, ou proteção mínima. A divisão D compreende apenas uma classe e é usada para sistemas que deixaram de atender aos requisitos de qualquer uma das outras classes de segurança. Por exemplo, MS-DOS e Windows 3.1 estão na divisão D.

A divisão C, o próximo nível de segurança, provê proteção discricionária e contabilidade dos usuários e de suas ações, por meio do uso de capacidades de auditoria. A divisão C possui dois níveis: C1 e C2. Um sistema de classe C1 incorpora alguma forma de controle, que permite aos usuários protegerem informações privadas e evitarem que outros usuários accidentalmente leiam ou destruam seus dados. Um ambiente C1 é aquele em que os usuários em cooperação acessam dados nos mesmos níveis de sensibilidade. A maioria das versões do UNIX é de classe C1.

O total de todos os sistemas de proteção dentro de um sistema computadorizado (hardware, software, firmware) que impõe uma política de segurança é conhecido como **base de computador confiável (Trusted Computer Base - TCB)**. A TCB de um sistema C1 controla o acesso entre os usuários e os arquivos, permitindo que o usuário especifique e controle o compartilhamento de objetos por indivíduos nomeados ou grupos definidos. Além disso, a TCB exige que os usuários se identifiquem antes de iniciarem quaisquer atividades que a TCB espera mediar. Essa identificação é realizada por meio de um mecanismo protegido ou senha; a TCB protege os dados de autenticação, de modo que sejam inacessíveis a usuários não autorizados.

Um sistema de classe C2 acrescenta um controle de acesso de nível individual aos requisitos de um sistema C1. Por exemplo, os direitos de acesso de um arquivo podem ser especificados no nível de um único indivíduo. Além disso, o administrador do sistema pode realizar a auditoria seletiva das ações de qualquer um ou mais usuários com base na identidade individual. A TCB também se protege contra modificação de seu código ou estruturas de dados. Além disso, nenhuma informação produzida por um usuário anterior está disponível a outro usuário que acessa um objeto de armazenamento liberado de volta ao sistema. Algumas versões especiais e seguras do UNIX foram certificadas no nível C2.

Os sistemas de proteção obrigatória da divisão B possuem todas as propriedades de um sistema de classe C2; além disso, eles reúnem um rótulo de sensibilidade a cada objeto. A TCB de classe B1 mantém o rótulo de segurança de cada objeto no sistema; o rótulo é usado para decisões referentes ao controle de acesso obrigatório. Por exemplo, um usuário no nível confidencial não poderia acessar um arquivo no nível secreto mais sensível. A TCB também indica o nível de sensibilidade no início e no final de cada página de qualquer saída legível ao ser humano. Além das informações de autenticação normais de nome de usuário e senha, a TCB também mantém as autorizações dos usuários individuais e aceita pelo menos dois níveis de segurança. Esses níveis são hierárquicos, de modo que um usuário pode acessar quaisquer objetos que transportam rótulos de sensibilidade iguais ou menores do que sua autorização de segurança. Por exemplo, um usuário de nível secreto poderia acessar um arquivo no nível confidencial na ausência de outros controles de acesso. Os processos também estão isolados por meio do uso de espaços de endereços distintos.

Um sistema de classe B2 estende os rótulos de sensibilidade a cada recurso do sistema, como objetos de armazenamento. Os dispositivos físicos recebem níveis de segurança mínimo e máximo, que o sistema utiliza para obrigar as restrições impostas pelos ambientes físicos em que os dispositivos estão localizados. Além disso, um sistema B2 aceita canais secretos e a auditoria de eventos que possam levar à exploração de um canal secreto.

Um sistema de classe B3 permite a criação de listas de controle de acesso que indicam usuários ou grupos que *não* têm acesso a determinado objeto nomeado. A TCB também contém um mecanismo para monitorar eventos que possam indicar uma violação da política de segurança. O mecanismo notifica o administrador de segurança e, se necessário, termina o evento da maneira que menos atrapalhe.

A classificação de nível mais alto é a divisão A. Arquitetonicamente, um sistema de classe A1 é funcionalmente equivalente a um sistema B3, mas utiliza especificações de projeto formais e técnicas de verificação, concedendo um alto grau de garantia de que a TCB foi implementada de maneira correta. Um sistema além da classe A1 pode ser projetado e desenvolvido em uma instalação confiável pelo pessoal de confiança.

O uso de uma TCB garante que o sistema pode impor aspectos de uma política de segurança; a TCB não especifica qual deverá ser a política. Em geral, determinado ambiente de computação desenvolve uma política de segurança para **certificação** e tem o plano **autorizado** por uma agência de segurança, como o National Computer Security Center. Certos ambientes de computação podem exigir outra certificação, como da TEMPEST, que protege contra espionagem eletrônica. Por exemplo, um sistema com certificado da TEMPEST possui terminais blindados para impedir que os campos eletromagnéticos escapem. Essa blindagem garante que um equipamento fora da sala ou do

prédio onde o terminal se encontra não poderá detectar as informações que estão sendo exibidas pelo terminal.

15.9 Um exemplo: Windows XP

O Microsoft Windows XP é um sistema operacional de uso geral, projetado para dar suporte a diversos recursos e métodos de segurança. Nesta seção, examinamos os recursos que o Windows XP utiliza para realizar as funções de segurança. Para obter mais informações e fundamentos sobre o Windows XP, consulte o [Capítulo 22](#).

O modelo de segurança do Windows XP é baseado na noção de **contas do usuário**. O Windows XP permite a criação de qualquer quantidade de contas do usuário, que podem ser agrupadas de qualquer maneira. O acesso aos objetos do sistema pode, então, ser permitido ou negado como for desejado. Os usuários são identificados ao sistema por um Security ID (identificador de segurança) exclusivo. Quando um usuário se conecta, o Windows XP cria um **token de acesso de segurança** que inclui o Security ID para o usuário, security IDs para quaisquer grupos dos quais o usuário seja membro e uma lista de quaisquer privilégios especiais que o usuário tenha. Alguns exemplos de privilégios especiais incluem o backup de arquivos e diretórios, o desligamento do computador, a conexão interativa e a mudança do relógio do sistema. Cada processo que o Windows XP executa em favor de um usuário receberá uma cópia do token de acesso. O sistema utiliza os Security IDs na ficha de acesso para permitir ou negar o acesso aos objetos do sistema sempre que o usuário, ou um processo em favor do usuário, tentar acessar o objeto. A autenticação de uma conta do usuário normalmente é realizada por meio de um nome de usuário e senha, embora o projeto modular do Windows XP permita o desenvolvimento de pacotes de autenticação personalizados. Por exemplo, um scanner da retina (ou olho) poderia ser usado para verificar se o usuário é quem ele diz ser.

O Windows XP usa o conceito de um súdito para garantir que os programas executados por um usuário não recebam acesso maior ao sistema do que o usuário está autorizado a ter. Um **súdito** é usado para acompanhar e gerenciar as permissões de cada programa que um usuário executa; ele é composto do token de acesso do usuário e do programa do usuário. Como o Windows XP opera com um modelo cliente-servidor, duas classes de súditos são usadas para controlar o acesso: súdito simples e súdito servidor. Um exemplo de **súdito simples** é o programa de aplicação típico que um usuário executa depois de se conectar. O súdito simples recebe um **contexto de segurança** baseado no token de acesso de segurança do usuário. Um **súdito servidor** é um processo implementado como um servidor protegido que usa o contexto de segurança do cliente quando estiver atuando em favor do cliente.

Conforme mencionamos na [Seção 15.7](#), a auditoria é uma técnica de segurança útil. O Windows XP possui auditoria embutida, o que permite que muitas ameaças de segurança comuns sejam monitoradas. Alguns exemplos incluem auditoria de falha para eventos de login e logoff, para detectar invasões de senha aleatórias; auditoria de sucesso para eventos de login e logoff, para detectar atividade de login em horários estranhos; auditoria de acesso para escrita com sucesso e falha para arquivos executáveis, para rastrear uma invasão de vírus; e auditoria de sucesso e falha para acesso ao arquivo, para detectar o acesso a arquivos sensíveis.

Os atributos de segurança de um objeto no Windows XP são descritos por um **descritor de segurança**. O descritor de segurança contém o security ID do proprietário do objeto (que pode mudar as permissões de acesso), um Security ID de grupo usado apenas pelo subsistema POSIX, uma lista de controle de acesso discricionário, que identifica quais usuários ou grupos têm acesso permitido (e quais não são permitidos), e uma lista de controle de acesso do sistema, que controla quais mensagens de auditoria o sistema gerará. Por exemplo, o descritor de segurança do arquivo *foo.bar* poderia ter o proprietário *avi* e esta lista de controle de acesso discricionário:

- *avi* - acesso total.
- *grupo cs* - acesso para leitura-escrita.
- *usuário cliff* - sem acesso.

Além disso, ele poderia ter uma lista de controle de acesso do sistema com escritas de auditoria feitas por todos.

Uma lista de controle de acesso é composta de entradas de controle de acesso que contêm o Security ID do indivíduo e uma máscara de acesso que define todas as ações possíveis sobre o objeto, com um valor de AccessAllowed ou AccessDenied para cada ação. Os arquivos no Windows XP podem ter os seguintes tipos de acesso: ReadData, WriteData, AppendData, Execute, ReadExtendedAttribute, WriteExtendedAttribute, ReadAttributes e WriteAttributes. Podemos ver como isso permite um grau de controle minucioso sobre o acesso aos objetos.

O Windows XP classifica os objetos como objetos contêiner ou objetos não contêiner. Os **objetos contêiner**, como diretórios, podem conter logicamente outros objetos. Como padrão, quando um objeto é criado dentro de um objeto contêiner, o novo objeto herda as permissões do objeto pai. De maneira similar, se o usuário copiar um arquivo de um diretório para um novo diretório, o arquivo herdará as permissões do diretório de destino. **Objetos não contêiner** não herdam outras permissões. Além disso, se uma permissão for alterada em um diretório, as novas permissões não se aplicam automaticamente aos arquivos e subdiretórios existentes; o usuário pode aplicá-las explicitamente, se assim desejar.

O administrador do sistema pode proibir a impressão em uma impressora no sistema por todo ou

parte do dia e pode usar o Performance Monitor do Windows XP para ajudá-lo a localizar problemas se aproximando. Em geral, o Windows XP realiza um bom trabalho provendo recursos para ajudar a garantir um ambiente de computação seguro. Contudo, muitos desses recursos não estão ativados como padrão, o que pode ser um motivo para as muitas brechas de segurança nos sistemas Windows XP. Outro motivo é a grande quantidade de serviços que o Windows XP inicia na hora do boot do sistema e a quantidade de aplicações que normalmente são instaladas em um sistema Windows XP. Para um ambiente multiusuário real, o administrador do sistema deve formular um plano de segurança e implementá-lo, usando os recursos que o Windows XP provê e outras ferramentas de segurança.

15.10 Resumo

A proteção é um problema interno. A segurança, ao contrário, precisa considerar o sistema computadorizado e o ambiente - pessoas, prédios, empresas, objetos valiosos e ameaças - dentro do qual o sistema é utilizado.

Os dados armazenados no sistema computadorizado precisam ser protegidos contra acesso não autorizado, destruição ou alteração maliciosa e introdução accidental de incoerência. É mais fácil proteger contra perda accidental da coerência de dados do que proteger contra acesso malicioso aos dados. A proteção absoluta das informações armazenadas em um sistema computadorizado contra abuso malicioso não é possível, mas o custo para o invasor pode se tornar suficientemente alto para desencorajar a maioria, se não todas as tentativas de acessar essas informações sem a devida autorização.

Vários tipos de ataques podem ser disparados contra programas, computadores individuais ou as massas. As técnicas de estouro de pilha e buffer permitem que atacantes mudem seu nível de acesso ao sistema com sucesso. Os vírus e os vermes (worms) são autoperpetuáveis, às vezes infectando milhares de computadores. Os ataques por negação de serviço impedem o uso legítimo dos sistemas-alvo.

A codificação limita o domínio dos receptores de dados, enquanto a autenticação limita o domínio dos emissores. A codificação é usada para prover confidencialidade dos dados que estão sendo armazenados ou transferidos. A codificação simétrica exige uma chave compartilhada, enquanto a codificação assimétrica fornece uma chave pública e uma chave privada. A autenticação, quando combinada com o hashing, pode provar que os dados não foram alterados.

Os métodos de autenticação de usuário são usados para identificar usuários legítimos de um sistema. Além da proteção-padrão de nome de usuário e senha, vários métodos de autenticação são utilizados. Senhas de única vez, por exemplo, mudam de uma sessão para outra, para evitar ataques de reprodução. A autenticação de fator dois exige duas formas de autenticação, como uma calculadora de hardware com um PIN de ativação. A autenticação multifator utiliza três ou mais formas. Esses métodos diminuem bastante as chances de autenticação forjada.

Os métodos de prevenção ou detecção de incidentes de segurança incluem sistemas de detecção de intrusão, software antivírus, auditoria e logging de eventos do sistema, monitoramento de mudanças de software do sistema, monitoramento de chamada de sistema e firewall.

Exercícios

- 15.1. Ataques de estouro de buffer podem ser evitados adotando-se uma metodologia de programação melhor ou usando-se suporte especial do hardware. Discuta essas soluções.
- 15.2. Uma senha pode se tornar conhecida de outros usuários de diversas maneiras. Existe algum método simples para detectar que tal evento aconteceu? Explique sua resposta.
- 15.3. Qual é a finalidade de usar um "sal" junto com a senha fornecida pelo usuário? Onde o "sal" deve ser armazenado e como ele deve ser usado?
- 15.4. A lista de todas as senhas é mantida dentro do sistema operacional. Assim, se um usuário conseguir ler essa lista, a proteção da senha não será mais fornecida. Sugira um esquema que evite esse problema. (Sugestão: Use representações internas e externas diferentes.)
- 15.5. Um acréscimo experimental do UNIX permite ao usuário conectar um programa **watchdog** (**cão de guarda**) a um arquivo. O watchdog é chamado sempre que um programa requisita acesso ao arquivo. Ele concede ou nega o acesso ao arquivo. Discuta dois prós e dois contras do uso de watchdog como medida de segurança.
- 15.6. O programa COPS do UNIX varre determinado sistema em busca de possíveis furos de segurança e alerta o usuário quanto a possíveis problemas. Cite dois riscos em potencial do uso de tal sistema para a segurança. Como esses problemas podem ser limitados ou eliminados?
- 15.7. Discuta um meio pelo qual os gerentes de sistemas conectados à Internet poderiam ter projetado seus sistemas para limitar ou eliminar os danos feitos por um verme (worm). Quais são as desvantagens de fazer a mudança que você sugere?
- 15.8. Argumente contra ou a favor da sentença judicial dada contra Robert Morris Jr., por sua criação e execução do verme (worm) da Internet discutido na [Seção 15.3.1](#).
- 15.9. Faça uma lista de seis aspectos de segurança para o sistema computadorizado de um banco. Para cada item na sua lista, informe se esse problema está relacionado com a segurança física, humana ou do sistema operacional.
- 15.10. Quais são duas vantagens de criptografar dados armazenados em um sistema computadorizado?
- 15.11. Que programas de computador usados normalmente são passíveis de ataques de homem no meio? Discuta as soluções para evitar essa forma de ataque.
- 15.12. Compare os esquemas de codificação simétrico e assimétrico, e discuta sob quais circunstâncias um sistema distribuído usaria um ou o outro.
- 15.13. Por que $D(k_d, N)(E(k_e, N)(m))$ não fornece autenticação do emissor? Quais usos uma codificação desse tipo poderia ter?
- 15.14. Discuta como o algoritmo de codificação assimétrica pode ser usado para conseguir os objetivos a seguir.
- Autenticação: o receptor sabe que somente o emissor poderia ter gerado a mensagem.
 - Segredo: somente o receptor pode decodificar a mensagem.
 - Autenticação e segredo: somente o receptor pode decodificar a mensagem, e o receptor sabe que somente o emissor poderia ter gerado a mensagem.
- 15.15. Considere um sistema que gera 10 milhões de registros de auditoria por dia. Suponha também que existem em média 10 ataques por dia nesse sistema e que cada ataque desse tipo seja refletido em 20 registros. Se o sistema de detecção de intrusão tem uma taxa de alarme verdadeiro de 0,6 e uma taxa de alarme falso de 0,0005, que porcentagem dos alarmes gerados pelo sistema corresponde a intrusões reais?

Notas bibliográficas

As discussões gerais referentes à segurança são dadas por [Hsiao e outros \[1979\]](#), [Landwehr \[1981\]](#), [Denning \[1982\]](#), [Pfleeger e Pfleeger \[2003\]](#), [Tanenbaum \[2003\]](#) e [Russell e Gangemi \[1991\]](#). O texto de [Lobel \[1986\]](#) também é de interesse geral. As redes de computadores são discutidas por [Kurose e Ross \[2005\]](#).

Questões relativas ao projeto e verificação de sistemas seguros são discutidas por [Rushby \[1981\]](#) e por [Silverman \[1983\]](#). Um kernel de segurança para um microcomputador multiprocessado é descrito por [Schell \[1983\]](#). Um sistema distribuído seguro é descrito por [Rushby e Randell \[1983\]](#).

[Morris e Thompson \[1979\]](#) discutem a segurança da senha. [Morsedian \[1986\]](#) apresenta métodos para combater piratas de senhas. A autenticação por senha com comunicações desprotegidas é considerada por [Lamport \[1981\]](#). A questão de invasão de senhas é examinada por [Seely \[1989\]](#). Invasões de computador são discutidas por [Lehmann \[1987\]](#) e por [Reid \[1987\]](#). Questões relacionadas com a confiança de programas de computador são discutidas por [Thompson \[1984\]](#).

Discussões referentes à segurança no UNIX são providas por [Grampp e Morris \[1984\]](#), [Wood e Kochan \[1985\]](#), [Farrow \[1986a\]](#), [Farrow \[1986b\]](#), [Filipski e Hanko \[1986\]](#), [Hecht e outros \[1988\]](#), [Kramer \[1988\]](#) e [Garfinkel e outros \[2003\]](#). [Bershad e Pinkerton \[1988\]](#) apresentam a extensão de watchdog ao BSD UNIX. O pacote de verificação de segurança COPS para UNIX foi escrito por Farmer na Purdue University. Ele está disponível aos usuários na Internet por meio do programa FTP do host `ftp.uu.net`, no diretório `/pub/security/cops`.

[Spafford \[1989\]](#) apresenta uma discussão técnica detalhada do verme da Internet. O artigo de Spafford aparece com três outros em uma seção especial sobre vermes de Morris da Internet em *Communications of the ACM* (Volume 32, Número 6, junho de 1989).

Problemas de segurança associados ao conjunto de protocolos TCP/IP são descritos em [Bollovin \[1989\]](#). Os mecanismos normalmente usados para impedir tais ataques são discutidos em [Cheswick e outros \[2003\]](#). Outra técnica para proteger as redes contra ataques internos é proteger a descoberta da topologia ou da rota. [Kent e outros \[2000\]](#), [Hu e outros \[2002\]](#), [Zapata e Asokan \[2002\]](#) e [Hu e Perrig \[2004\]](#) apresentam soluções para o roteamento seguro. [Savage e outros \[2000\]](#) examinam o ataque de negação de serviço distribuído e propõem soluções de trace-back do IP para resolver o problema. [Perlman \[1988\]](#) propõe uma técnica para diagnosticar falhas quando a rede contém roteadores maliciosos.

Informações adicionais sobre vírus e vermes poderão ser encontradas em <http://www.viruslist.com> e também em [Ludwig \[1998\]](#) e [Ludwig \[2002\]](#). Outros sites contendo informações de segurança atualizadas são <http://www.trusecure.com> e <http://www.eeye.com>. Um artigo sobre os perigos de uma monocultura de computador pode ser encontrado em <http://www.cccanet.org/papers/cyberinsecurity.pdf>.

[Diffie e Hellman \[1976\]](#) e [Diffie e Hellman \[1979\]](#) foram os primeiros pesquisadores a propor o uso do esquema de codificação por chave pública. O algoritmo apresentado na [Seção 15.4.1](#) é baseado no esquema de codificação por chave pública; ele foi desenvolvido por [Rivest e outros \[1978\]](#). [Lempel \[1979\]](#), [Simmons \[1979\]](#), [Denning e Denning \[1979\]](#), [Gifford \[1982\]](#), [Denning \[1982\]](#) e [Ahituv e outros \[1987\]](#), [Schneier \[1996\]](#) e [Stallings \[2003\]](#) exploram o uso da criptografia nos sistemas computadorizados. Discussões referentes à proteção de assinaturas digitais são providas por [Akl \[1983\]](#), [Davies \[1983\]](#), [Denning \[1983\]](#) e [Denning \[1984\]](#).

O governo dos Estados Unidos se preocupa com a segurança. O *Department of Defense Trusted Computer System Evaluation Criteria*, [DoD \[1985\]](#), também conhecido *Orange Book*, descreve um conjunto de níveis de segurança e os recursos que um sistema operacional precisa ter para se qualificar para cada classificação de segurança. Sua leitura é um bom ponto de partida para entender questões de segurança. O *Microsoft Windows NT Workstation Resource Kit* ([Microsoft \[1996\]](#)) descreve o modelo de segurança do NT e como usar esse modelo.

O algoritmo RSA é apresentado em [Rivest e outros \[1978\]](#). Informações sobre as atividades com AES do NIST podem ser encontradas em <http://www.nist.gov/aes/>; informações sobre outros padrões criptográficos para os Estados Unidos também podem ser encontradas nesse site. Uma cobertura mais completa do SSL 3.0 pode ser encontrada em <http://home.netscape.com/eng/ssl3/>. Em 1999, o SSL 3.0 foi modificado ligeiramente e apresentado em um Request for Comments (RFC) do IETF, sob o nome TLS.

O exemplo na [Seção 15.6.3](#), ilustrando o impacto da taxa de alarmes falsos sobre a eficiência dos IDSs, é baseado em [Axelsson \[1999\]](#). A descrição do Tripwire, na [Seção 15.6.5](#), é baseada em [Kim e Spafford \[1993\]](#). A pesquisa sobre detecção de anomalia baseada em chamada de sistema é descrita em [Forrest e outros \[1996\]](#).

PARTE VI

SISTEMAS DISTRIBUÍDOS

ESBOÇO

Capítulo 21: Introdução a Sistemas distribuídos
Capítulo 22: Estruturas de sistemas distribuídos
Capítulo 23: Sistemas de arquivos distribuídos
Capítulo 24: Coordenação distribuída

Introdução a Sistemas distribuídos

Um sistema distribuído é uma coleção de processadores que não compartilham memória ou relógio. Em vez disso, cada processador tem sua própria memória local e os processadores se comunicam entre si por meio de linhas de comunicação, como redes locais ou remotas. Os processadores em um sistema distribuído variam em tamanho e em função. Tais sistemas podem incluir pequenos dispositivos portáteis ou de tempo real, computadores pessoais, estações de trabalho e computadores de grande porte.

Um sistema de arquivos distribuído é um sistema de serviço de arquivo cujos usuários, servidores e dispositivos de armazenamento estão espalhados pelas instalações de um sistema distribuído. Sendo assim, a atividade do serviço precisa ser executada por meio da rede; em vez de um único repositório de dados centralizado, existem dispositivos de armazenamento múltiplos e independentes.

Os benefícios de um sistema distribuído incluem acesso do usuário aos recursos mantidos pelo sistema e, portanto, agilidade na computação e melhor disponibilidade e confiabilidade dos dados. Entretanto, como um sistema está distribuído, ele precisa prover mecanismos para o sincronismo de processo e comunicação, para lidar com o problema de deadlock e para lidar com falhas, as quais não são encontradas em um sistema centralizado.

CAPÍTULO 16

Estruturas de sistemas distribuídos

Um sistema distribuído é uma coleção de processadores que não compartilham memória ou relógio. Na verdade, cada processador possui sua própria memória local. Os processadores se comunicam uns com os outros por meio de diversas redes de comunicação, como barramentos de alta velocidade e linhas telefônicas. Neste capítulo, vamos discutir a estrutura geral dos sistemas distribuídos e as redes que os interconectam. Comparamos as principais diferenças no projeto do sistema operacional entre esses sistemas e os sistemas centralizados. No [Capítulo 17](#), continuamos discutindo sobre sistemas de arquivos distribuídos. Depois, no [Capítulo 18](#), descrevemos os métodos necessários para os sistemas operacionais distribuídos coordenarem suas ações.

OBJETIVOS DO CAPÍTULO

- Fornecer uma visão de alto nível dos sistemas distribuídos e as redes que os interconectam.
- Discutir a estrutura geral dos sistemas operacionais distribuídos.

16.1 Motivação

Um **sistema distribuído** é uma coleção de processadores pouco acoplados, interconectados por uma rede de comunicação. Do ponto de vista de um processador específico em um sistema distribuído, o restante dos processadores e seus respectivos recursos são remotos, enquanto seus próprios recursos são locais.

Os processadores em um sistema distribuído podem variar em tamanho e função. Eles podem incluir pequenos processadores, estações de trabalho, minicomputadores e grandes computadores de uso geral. Esses processadores são chamados de vários nomes, como *instalações*, *nós*, *computadores*, *máquinas* ou *hosts*, dependendo do contexto em que são mencionados. Usaremos principalmente *instalação* para indicar o local de uma máquina e *host* para nos referirmos a um sistema específico em uma instalação. Em geral, um host em uma instalação, o *servidor*, possui um recurso que outro host em outra instalação, o cliente (ou usuário) gostaria de usar. Uma estrutura geral de um sistema distribuído pode ser vista na [Figura 16.1](#).

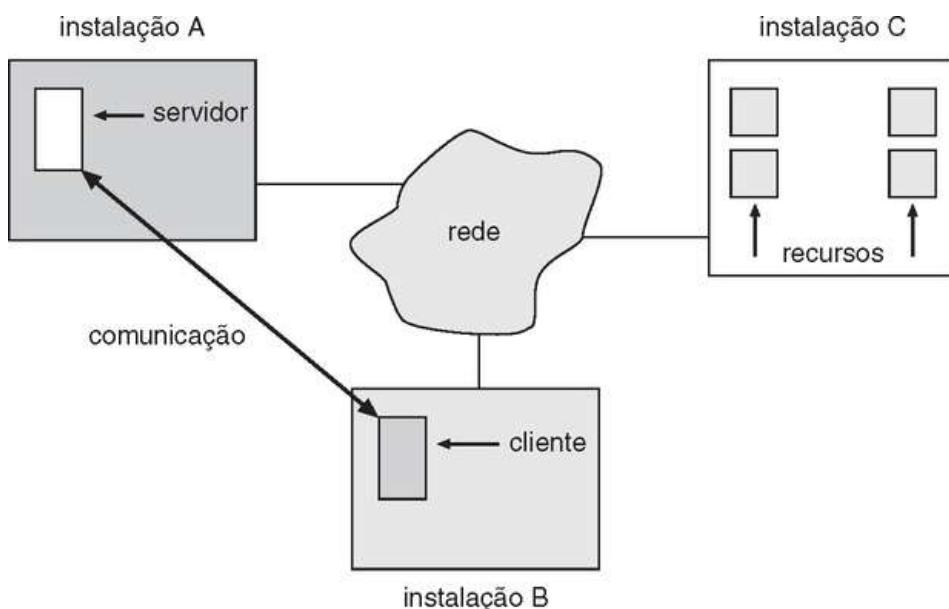


FIGURA 16.1 Um sistema distribuído.

Os quatro motivos principais para a criação de sistemas distribuídos são: *compartilhamento de recursos*, *aumento da velocidade de computação*, *confiabilidade* e *comunicação*. Nesta seção, vamos discutir cada um deles resumidamente.

16.1.1 Compartilhamento de recursos

Se várias instalações diferentes (com capacidades diferentes) estiverem conectadas entre si, então um usuário em uma instalação pode ser capaz de usar os recursos disponíveis em outra instalação. Por exemplo, um usuário na instalação A pode usar uma impressora a laser localizada na instalação B. Enquanto isso, um usuário em B pode acessar um arquivo que reside em A. Em geral, o **compartilhamento de recursos** em um sistema distribuído provê mecanismos para compartilhar arquivos nas instalações remotas, processar informações em um banco de dados distribuído, imprimir arquivos em instalações remotas, usar dispositivos de hardware remotos especializados (como um processador de array de alta velocidade) e realizar outras operações.

16.1.2 Aumento da velocidade de computação

Se uma computação em particular puder ser particionada em subcomputações que possam ser executadas concorrentemente, então um sistema distribuído nos permite distribuir as subcomputações entre as diversas instalações; as subcomputações podem ser executadas de maneira concorrente e, assim, prover um **aumento da velocidade de computação (computation speedup)**. Além disso, se determinada instalação estiver sobrecarregada de tarefas, algumas podem ser movidas para outras instalações pouco carregadas. Esse movimento de tarefas é chamado de **compartilhamento de carga (load sharing)**. O compartilhamento de carga automatizado, no qual o sistema operacional distribuído move tarefas automaticamente, ainda não é comum nos sistemas comerciais.

16.1.3 Confiabilidade

Se uma instalação falhar em um sistema distribuído, as instalações restantes podem continuar operando, dando ao sistema melhor confiabilidade. Se o sistema for composto de várias instalações autônomas grandes (ou seja, computadores de uso geral), deixar de usar uma delas não deverá afetar o restante. No entanto, se o sistema for composto de pequenas máquinas, cada uma responsável por alguma função crucial do sistema (como a E/S de caracteres de terminal ou o sistema de arquivos), então uma única falha pode interromper a operação do sistema inteiro. Em geral, com redundância suficiente (tanto no hardware quanto nos dados), o sistema pode continuar a operação, mesmo que algumas de suas instalações tenham falhado.

A falha de uma instalação precisa ser detectada pelo sistema, e a ação apropriada pode ser necessária para a recuperação da falha. O sistema não precisa mais usar os serviços dessa instalação. Além disso, se a função da instalação que falhou puder ser assumida por outra instalação, o sistema deverá garantir que a transferência da função ocorrerá corretamente. Por fim, quando a instalação que falhou se recuperar ou for reparada, é preciso haver mecanismos para integrá-la de volta ao sistema. Conforme veremos nos [Capítulos 17 e 18](#), essas ações apresentam problemas difíceis, que possuem muitas soluções possíveis.

16.1.4 Comunicação

Quando várias instalações estão conectadas umas às outras por uma rede de comunicação, os usuários em diferentes instalações podem trocar informações. Em um nível baixo, **mensagens** são trocadas entre os sistemas, assim como as mensagens são trocadas entre os processos no sistema de mensagens em um único computador, discutido na [Seção 3.4](#). Com a troca de mensagens, toda a funcionalidade de nível mais alto encontrada nos sistemas independentes pode ser expandida para abranger o sistema distribuído. Essas funções incluem transferência de arquivo, login, correio e remote procedure call (RPC).

A vantagem de um sistema distribuído é que essas funções podem ser executadas por grandes distâncias. Duas pessoas em instalações geograficamente separadas podem colaborar em um projeto, por exemplo. Transferindo os arquivos do projeto, conectando-se aos sistemas remotos um do outro para executar programas e trocando correspondência para coordenar o trabalho, os usuários reduzem as limitações inerentes ao trabalho a longa distância. Escrevemos este livro colaborando dessa maneira.

As vantagens dos sistemas distribuídos resultaram em uma tendência de todo o setor em direção ao **downsizing**. Muitas empresas estão substituindo seus computadores de grande porte por redes de estações de trabalho ou computadores pessoais. As empresas ganham mais pelo dinheiro investido (ou seja, obtêm melhor funcionalidade pelo custo), mais flexibilidade na localização de recursos e facilidades de expansão, melhores interfaces de usuário e manutenção mais fácil.

16.2 Tipos de sistemas operacionais em rede

Nesta seção, descrevemos as duas categorias gerais de sistemas operacionais orientados para rede: sistemas operacionais de rede e sistemas operacionais distribuídos. Os sistemas operacionais de rede são mais simples de implementar, mas mais difíceis para os usuários acessarem e utilizarem do que os sistemas operacionais distribuídos, que proveem mais recursos.

16.2.1 Sistemas operacionais de rede

Um **sistema operacional de rede** provê um ambiente em que os usuários, que estão cientes da multiplicidade de máquinas, podem acessar recursos remotos efetuando o login na máquina remota apropriada ou transferindo dados da máquina remota para suas próprias máquinas.

16.2.1.1 Login remoto

Uma função importante de um sistema operacional de rede é permitir aos usuários efetuarem o login remotamente. A Internet provê a facilidade **telnet** para essa finalidade. Para ilustrar como isso funciona, vamos supor que um usuário na Westminster College queira realizar alguma computação em “cs.yale.edu”, um computador localizado na Universidade de Yale. Para isso, o usuário precisa ter uma conta válida nessa máquina. Para efetuar o login remotamente, o usuário emite o comando

```
telnet cs.yale.edu
```

Esse comando resulta na formação de uma conexão de socket, entre a máquina local na Westminster College e o computador “cs.yale.edu”. Depois de essa conexão ter sido estabelecida, o software de rede cria um enlace transparente, bidirecional, de modo que todos os caracteres digitados pelo usuário são enviados a um processo em “cs.yale.edu” e toda a saída desse processo é enviada de volta ao usuário. O processo na máquina remota pede um nome de login e uma senha ao usuário. Quando a informação correta tiver sido recebida, o processo atua como um proxy para o usuário, que pode computar na máquina remota como se fosse um usuário local qualquer.

16.2.1.2 Transferência de arquivo remoto

Outra função importante de um sistema operacional de rede é prover um mecanismo para a **transferência de arquivo remoto** de uma máquina para outra. Nesse ambiente, cada computador mantém seu próprio sistema de arquivos local. Se um usuário em uma instalação (digamos, “cs.uvm.edu”) quiser acessar um arquivo localizado em outro computador (digamos, “cs.yale.edu”), então o arquivo precisa ser copiado explicitamente do computador em Yale para o computador na Universidade de Vermont.

A Internet provê um mecanismo para esse tipo de transferência com o programa File Transfer Protocol (FTP). Suponha que um usuário em “cs.uvm.edu” queira copiar um programa Java, Server.java, que reside em “cs.yale.edu”. O usuário primeiro precisa chamar o programa FTP, executando

```
ftp cs.yale.edu
```

O programa, então, pede o nome de login do usuário e uma senha. Quando a informação correta for recebida, o usuário terá de se conectar ao subdiretório em que o arquivo Server.java reside e depois copiar o arquivo executando

```
get Server.java
```

Nesse esquema, o local do arquivo não é transparente ao usuário; os usuários precisam saber exatamente onde se encontra cada arquivo. Além do mais, não existe compartilhamento de arquivos real, pois um usuário só pode *copiar* um arquivo de uma instalação para outra. Assim, pode haver várias cópias do mesmo arquivo, resultando em um desperdício de espaço. Além disso, se essas cópias forem modificadas, as diversas cópias estarão incoerentes.

Observe que, no nosso exemplo, o usuário na Universidade de Vermont precisa ter permissão de login em “cs.yale.edu”. O FTP também provê um meio de permitir a um usuário que não tenha uma conta no computador de Yale copiar arquivos remotamente. Essa cópia remota é feita por meio do método de “FTP anônimo”, que funciona da seguinte maneira. O arquivo a ser copiado (ou seja, Server.java) precisa ser colocado em um subdiretório especial (digamos, *ftp*) com a proteção configurada para permitir que o público leia o arquivo. Um usuário que deseja copiar o arquivo usa o comando *ftp*, como antes. Quando o usuário tiver de informar o nome de login, ele fornece o nome “anonymous” e uma senha qualquer.

Quando o login anônimo é realizado, o sistema precisa ter o cuidado de garantir que esse usuário parcialmente autorizado não acessará arquivos impróprios. Em geral, o usuário tem permissão para acessar apenas os arquivos que estão na árvore de diretórios do usuário “anonymous”. Quaisquer arquivos colocados aqui são acessíveis a quaisquer usuários anônimos, sujeitos ao esquema normal de proteção de arquivos usado nessa máquina. Entretanto, os usuários anônimos não podem acessar arquivos fora dessa árvore de diretórios.

A implementação do mecanismo de FTP é semelhante à implementação do telnet. Existe um daemon na instalação remota que observa requisições de conexão com a porta FTP do sistema. A

autenticação de login é realizada, e o usuário tem permissão para executar comandos remotamente. Ao contrário do daemon telnet, que executa qualquer comando para o usuário, o daemon FTP só responde a um conjunto predefinido de comandos relacionados com arquivo. Entre eles estão:

- get – Transfere um arquivo da máquina remota para a máquina local.
- put – Transfere da máquina local para a máquina remota.
- ls ou dir – Lista arquivos no diretório atual da máquina remota.
- cd – Muda o diretório atual na máquina remota.

Há também vários comandos para mudar os modos de transferência (para arquivos binários ou ASCII) e para determinar o status da conexão.

Um ponto importante sobre o telnet e o FTP é que eles exigem que o usuário mude de paradigma. O FTP exige que o usuário conheça um conjunto de comandos inteiramente diferente dos comandos normais do sistema operacional. O telnet exige uma mudança menor: o usuário precisa conhecer os comandos apropriados no sistema remoto. Por exemplo, um usuário em uma máquina Windows que realiza um telnet com uma máquina UNIX precisa passar para comandos do UNIX durante a sessão telnet. As facilidades são mais convenientes para os usuários se eles não precisarem usar um conjunto de comandos diferente. Os sistemas operacionais distribuídos são projetados para minimizar esse problema.

16.2.2 Sistemas operacionais distribuídos

Em um sistema operacional distribuído, os usuários acessam recursos remotos da mesma maneira como fazem com recursos locais. A migração de dados e processos de uma instalação para outra está sob o controle do sistema operacional distribuído.

16.2.2.1 Migração de dados

Suponha que um usuário na instalação A queira acessar dados (como um arquivo) que residem na instalação B. O sistema pode transferir os dados com dois métodos básicos. Uma técnica para a **migração de dados** é transferir o arquivo inteiro para a instalação A. Desse ponto em diante, todo o acesso ao arquivo é local. Quando o usuário não precisar mais acessar o arquivo, uma cópia do arquivo (se ele tiver sido modificado) é enviada de volta à instalação B. Mesmo que apenas uma mudança simples tenha sido feita a um arquivo grande, todos os dados precisam ser transferidos. Esse mecanismo pode ser imaginado como um sistema de FTP automatizado. Essa técnica foi usada no sistema de arquivos Andrew, conforme discutimos no [Capítulo 17](#), mas descobriu-se que era muito ineficaz.

A outra técnica é transferir para a instalação A somente as partes do arquivo *necessárias* para a tarefa imediata. Se outra parte for exigida mais tarde, outra transferência ocorrerá. Quando o usuário não quiser mais acessar o arquivo, qualquer parte modificada precisará ser enviada de volta à instalação B. (Observe a semelhança com a paginação por demanda.) O protocolo Network File System (NFS) da Sun Microsystems utiliza esse método ([Capítulo 17](#)), assim como as versões mais recentes do Andrew. O protocolo SMB da Microsoft (executando em cima de TCP/IP ou do protocolo NetBEUI da Microsoft) também permite o compartilhamento de arquivos por uma rede.

É claro que, se apenas uma pequena parte de um arquivo grande estiver sendo acessada, essa última técnica deve ser usada preferencialmente. Contudo, se partes significativas do arquivo estiverem sendo acessadas, é mais eficiente copiar o arquivo inteiro. Nos dois métodos, a migração de dados inclui mais do que uma simples transferência de dados de uma instalação para outra. O sistema também precisa realizar diversas traduções de dados se as duas instalações envolvidas não forem diretamente compatíveis (por exemplo, se usarem representações de código de caracteres diferentes ou se representarem inteiros com uma quantidade de bits ou ordem diferente).

16.2.2.2 Migração de computação

Em algumas circunstâncias, poderemos querer transferir a computação, em vez dos dados, do sistema; essa técnica é chamada **migração de computação**. Por exemplo, considere uma tarefa que precisa acessar vários arquivos grandes que residem em instalações diferentes, para obter um resumo desses arquivos. Seria mais difícil acessar os arquivos nas instalações onde residem e retornar os resultados desejados para a instalação que iniciou a computação. Em geral, se o tempo para transferir os dados for maior do que o tempo para executar o comando remoto, o comando remoto deverá ser usado.

Essa computação pode ser executada de diferentes maneiras. Suponha que o processo P queira acessar um arquivo na instalação A. O acesso ao arquivo é executado na instalação A e poderia ser iniciado por uma RPC. Uma RPC usa um **protocolo de datagrama** (UDP na Internet) para executar uma rotina em um sistema remoto ([Seção 3.6.2](#)). O processo P invoca um procedimento predefinido na instalação A. O procedimento executa corretamente e depois retorna os resultados para P.

Como alternativa, o processo P pode enviar uma mensagem à instalação A. O sistema operacional na instalação A, então, cria um novo processo Q, cuja função é executar a tarefa designada. Quando o processo Q terminar sua execução, ele envia o resultado necessário de volta para P por meio do

sistema de mensagem. Nesse esquema, o processo P pode ser executado concorrentemente com o processo Q e, na verdade, pode ter vários processos executando ao mesmo tempo em várias instalações.

Os dois métodos poderiam ser usados para acessar vários arquivos residindo em várias instalações. Uma RPC poderia resultar na chamada de outra RPC ou ainda na transferência de mensagens para outra instalação. De modo semelhante, o processo Q poderia, durante o curso de sua execução, enviar uma mensagem para outra instalação, que, por sua vez, criaria outro processo. Esse processo poderia enviar uma mensagem de volta para Q ou repetir o ciclo.

16.2.2.3 Migração de processo

Uma extensão lógica da migração de computação é a **migração de processo**. Quando um processo é submetido para execução, ele nem sempre é executado na instalação em que é iniciado. O processo inteiro, ou partes dele, pode ser executado em diferentes instalações. Esse esquema pode ser usado por vários motivos:

- **Balanceamento de carga.** Os processos (ou subprocessos) podem ser distribuídos por meio da rede para uniformizar a carga de trabalho.
- **Aumento da velocidade de computação.** Se um único processo puder ser dividido em uma série de subprocessos que podem ser executados simultaneamente em diferentes instalações, então o tempo de retorno total do processo pode ser reduzido.
- **Preferência de hardware.** O processo pode ter características que o tornam mais adequado para execução em algum processador especializado (como a inversão de matriz em um array processador, em vez de um microprocessador).
- **Preferência de software.** O processo pode exigir algum software que esteja disponível apenas em determinada instalação, e o software não pode ser movido ou é menos dispendioso para mover o processo.
- **Acesso aos dados.** Assim como na migração da computação, se os dados usados na computação forem numerosos, pode ser mais eficiente fazer um processo ser executado remotamente do que transferir todos os dados localmente.

Usamos duas técnicas complementares para mover processos em uma rede de computadores. Na primeira, o sistema pode tentar esconder o fato de o processo ter migrado do cliente. Esse esquema tem a vantagem de o usuário não precisar codificar seu programa explicitamente para realizar a migração. Esse método é empregado para conseguir o balanceamento de carga e o aumento de velocidade de computação entre sistemas homogêneos, já que não precisam da entrada do usuário para ajudá-los a executar os programas remotamente.

A outra técnica é permitir (ou exigir) que o usuário especifique explicitamente como o processo deve ser migrado. Esse método é empregado quando o processo precisa ser movido para satisfazer uma preferência de hardware ou software.

Você já deve ter observado que a Web possui muitos aspectos de um ambiente de computação distribuída. Certamente, ela provê migração de dados (entre um servidor Web e um cliente Web). Ela também provê migração de computação. Por exemplo, um cliente Web poderia acionar uma operação de banco de dados em um servidor Web. Por fim, com Java, ela provê uma forma de migração de processo: applets Java são enviados do servidor para o cliente, onde são executados. Um sistema operacional de rede provê a maior parte desses recursos, mas um sistema operacional distribuído os torna transparentes e acessíveis. O resultado é uma facilidade poderosa e de fácil utilização - um dos motivos para o enorme crescimento da World Wide Web.

16.3 Estrutura de rede

Existem basicamente dois tipos de redes: **redes locais** (**Local Area Network - LAN**) e **redes de longa distância** (**Wide Area Network - WAN**). A principal diferença entre as duas é a forma como são distribuídas geograficamente. As redes locais são compostas por computadores distribuídos por pequenas áreas geográficas (como um único prédio ou uma série de prédios adjacentes); já as redes de longa distância são compostas por uma série de computadores autônomos distribuídos em uma grande área geográfica (como os estados do país). Essas diferenças implicam grandes variações na velocidade e na confiabilidade da rede de comunicações e são refletidas no projeto do sistema operacional distribuído.

16.3.1 Redes locais

As redes locais surgiram no início da década de 1970, como substitutos para os grandes mainframes. Para muitas empresas, é mais econômico ter diversos computadores pequenos, cada um com suas próprias aplicações autocontidas, do que um único sistema gigantesco. Como é provável que cada computador pequeno precise de um complemento total de dispositivos periféricos (como discos e impressoras), e como alguma forma de compartilhamento de dados deve ocorrer em uma única empresa, conectar esses pequenos sistemas em uma rede foi um passo natural.

Como foi dito, as redes locais normalmente são projetadas para cobrir uma área geográfica pequena (como um único prédio ou alguns prédios adjacentes) e, em geral, as LANs são usadas em um ambiente de escritório. Todos os locais nesses sistemas estão próximos um do outro, de modo que os enlaces (links) de comunicação costumam ter uma velocidade maior e uma taxa de erro menor do que seus equivalentes nas redes de longa distância. Cabos de alta qualidade (mais caros) são necessários para conseguir esses níveis mais altos de velocidade e confiabilidade. É possível também usar os cabos de LAN exclusivamente para o tráfego da rede de dados. Para distâncias maiores, o custo do uso de cabo de alta qualidade é muito grande, e o uso exclusivo do cabo costuma ser proibitivamente caro.

Os enlaces mais comuns em uma rede local são o cabeamento de par trançado e fibra óptica. As configurações mais comuns são redes com barramento de múltiplo acesso, estrela e anel. As velocidades de comunicação variam de 1 megabit por segundo, para redes como AppleTalk e Bluetooth, até 10 gigabits por segundo, para rede Ethernet a 10 gigabits. Dez megabits por segundo é a velocidade da **10BaseT Ethernet**. A **100BaseT Ethernet** exige um cabo de maior qualidade, mas está se tornando comum. Também está aumentando o uso de redes FDDI baseadas em fibra óptica. A rede FDDI é baseada em token e trabalha a mais de 100 megabits por segundo.

Uma LAN típica consiste em uma série de computadores (de mainframes a laptops ou PDAs), diversos dispositivos periféricos compartilhados (como impressoras a laser e unidades de fita magnética) e um ou mais gateways (processadores especializados) que fornecem acesso a outras redes ([Figura 16.2](#)). Um esquema Ethernet é usado para construir LANs. Uma rede Ethernet não possui controlador central, pois é um barramento de múltiplo acesso, de modo que novos hosts podem ser acrescentados facilmente à rede. O protocolo Ethernet é definido pelo padrão IEEE 802.3.

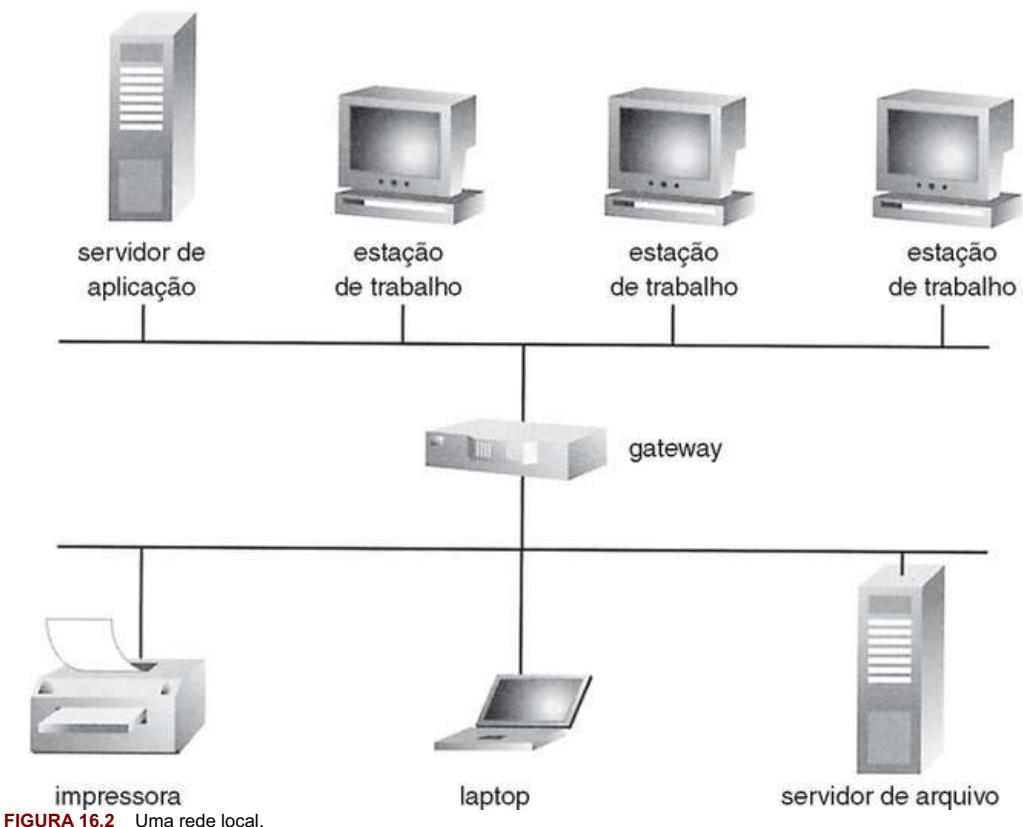


FIGURA 16.2 Uma rede local.

Tem havido um crescimento significativo no uso do espectro sem fio para o projeto de redes locais. Redes sem fio (wireless ou WiFi) permitem a construção de uma rede usando apenas um roteador sem fio para a transmissão de sinais entre os hosts. Cada host possui uma placa de rede de adaptador sem fio que lhe permite ingressar na rede sem fio e utilizá-la. Porém, enquanto os sistemas Ethernet normalmente trabalham a 100 megabits por segundo, as redes WiFi normalmente possuem velocidades menores. Existem diversos padrões IEEE para as redes sem fio: 802.11g teoricamente pode funcionar a 54 megabits por segundo, embora, na prática, as taxas de dados costumam ser menores do que a metade disso. O padrão mais recente 802.11n oferece teoricamente taxas de dados muito mais altas do que o 802.11g, embora, na prática, redes 802.11n possuam taxas de dados típicas em torno de 75 megabits por segundo. As taxas de dados das redes sem fio são bastante influenciadas pela distância entre o roteador sem fio e o host, bem como pela interferência no espectro sem fio. As redes sem fio geralmente possuem uma vantagem física em relação às redes Ethernet com fio, pois nenhum cabeamento precisa ser feito para conectar os hosts que se comunicam. Por conseguinte, as redes sem fio são populares nas residências, além de áreas públicas como bibliotecas e lanchonetes.

16.3.2 Redes remotas

As redes remotas surgiram no final da década de 1960, principalmente como um projeto acadêmico de pesquisa para fornecer comunicação eficiente entre as instalações, permitindo que o hardware e o software fossem compartilhados de modo conveniente e econômico por uma grande comunidade de usuários. A primeira WAN a ser projetada e desenvolvida foi a *Arpanet*. Iniciada em 1968, a Arpanet cresceu de uma rede experimental de quatro locais para uma rede mundial de redes, a Internet, compreendendo milhões de computadores.

Como as instalações em uma WAN são distribuídas fisicamente por uma grande área geográfica, os enlaces de comunicação, como padrão, são relativamente lentos e pouco confiáveis. Os enlaces típicos são linhas telefônicas, linhas de dados dedicadas, enlaces de micro-ondas e canais de satélite. Esses enlaces de comunicação são controlados por **processadores de comunicação** especiais (Figura 16.3), responsáveis por definir a interface por meio da qual as instalações se comunicam pela rede, bem como transferir informações entre as diversas instalações.

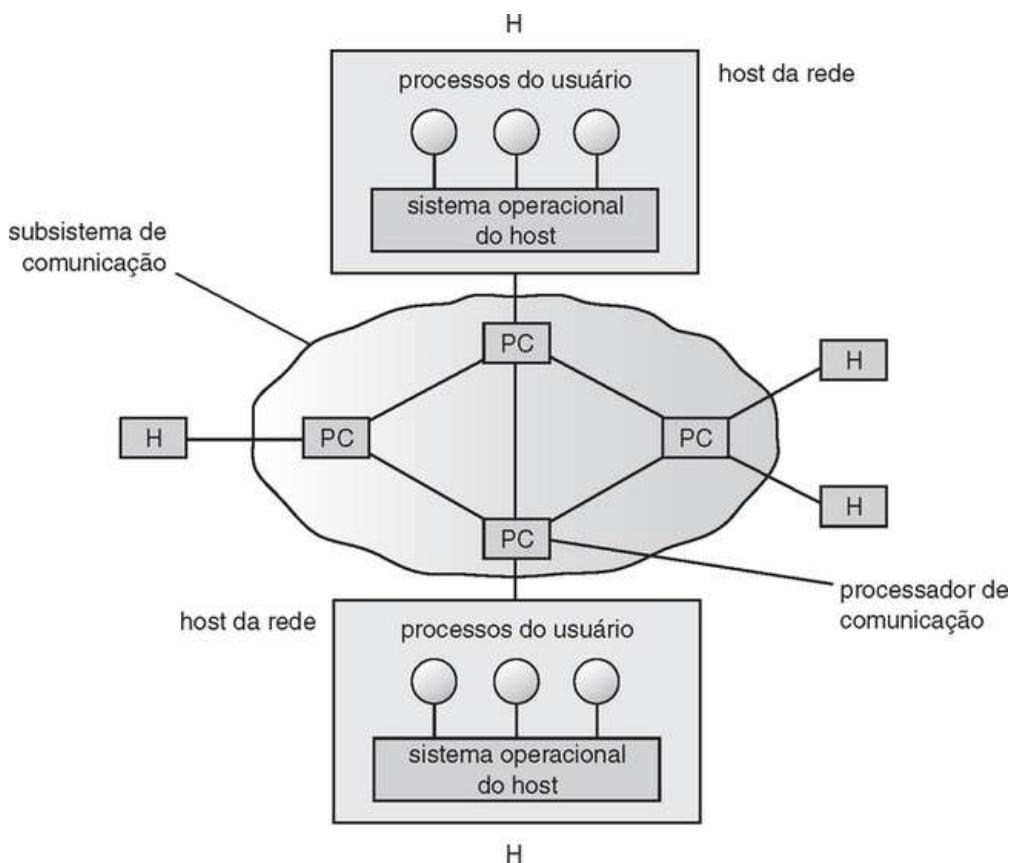


FIGURA 16.3 Processadores de comunicação em uma rede remota.

Por exemplo, a WAN da Internet fornece a capacidade de os hosts em sites separados geograficamente se comunicarem entre si. Os computadores host em geral diferem entre si em tipo, velocidade, tamanho de word, sistema operacional e assim por diante. Os hosts estão em LANs, que são, por sua vez, conectadas à Internet por meio de redes regionais. As redes regionais, como NSFnet no nordeste dos Estados Unidos, são interligadas com **roteadores** (Seção 16.5.2) para formar uma rede mundial. As conexões entre as redes utilizam um serviço do sistema telefônico chamado T1, que fornece uma taxa de transferência de 1,544 megabit por segundo por meio de uma linha alugada. Para instalações que exigem acesso mais rápido à Internet, T1s são conectadas em múltiplas unidades T1, que atuam em paralelo, para fornecer maior vazão. Por exemplo, uma T3 é composta de 28 conexões T1 e possui uma taxa de transferência de 45 megabits por segundo. Os roteadores controlam o caminho tomado por cada mensagem pela rede. Esse roteamento pode ser dinâmico, para aumentar a eficiência das comunicações, ou estático, para reduzir os riscos de segurança ou permitir o cálculo das cobranças pela comunicação.

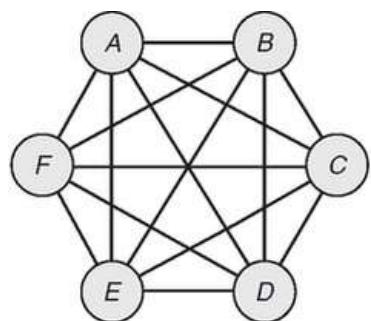
Outras WANs utilizam linhas telefônicas padrão como seu principal meio de comunicação. Os **modems** são dispositivos que aceitam dados digitais do lado do computador e os convertem para sinais analógicos que o sistema telefônico utiliza. Um modem na instalação de destino converte o sinal analógico de volta para o formato digital, e o destino recebe os dados. A rede de notícias do UNIX, UUCP, permite que os sistemas se comuniquem entre si em horários predeterminados, via modems, para trocar mensagens. As mensagens são, então, roteadas por outros sistemas vizinhos e, dessa maneira, são propagadas a todos os hosts na rede (mensagens públicas) ou são transferidas para destinos específicos (mensagens privadas). As WANs geralmente são mais lentas que as LANs; suas taxas de transmissão variam de 1.200 bits por segundo até mais de 1 megabit por segundo. O UUCP foi substituído pelo PPP, o Point-to-Point Protocol. O PPP funciona por meio de conexões de modem, permitindo que computadores domésticos sejam totalmente conectados à Internet.

16.4 Topologia de rede

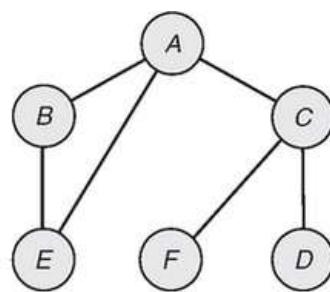
As instalações em um sistema distribuído podem ser conectadas fisicamente de diversas maneiras. Cada configuração possui vantagens e desvantagens. Podemos comparar as configurações usando os seguintes critérios:

- **Custo de instalação.** O custo da ligação física das instalações no sistema.
- **Custo de comunicação.** O custo em tempo e dinheiro para enviar uma mensagem da instalação A para a instalação B.
- **Disponibilidade.** A extensão à qual os dados podem ser acessados apesar da falha de alguns enlaces ou instalações.

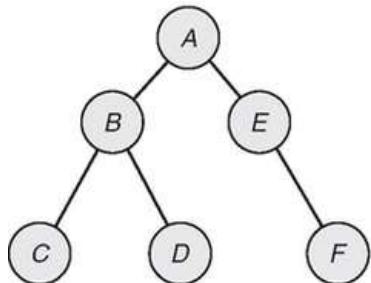
As diversas topologias são representadas na [Figura 16.4](#) como gráficos cujos nós correspondem às instalações. Uma aresta do nó A para o nó B corresponde a um enlace de comunicação direto entre as duas instalações. Em uma rede totalmente conectada, cada instalação está conectada a cada outra instalação. Contudo, o número de enlaces cresce como o quadrado do número de instalações, resultando em um custo de instalação enorme. Portanto, as redes totalmente conectadas não são práticas em nenhum sistema grande.



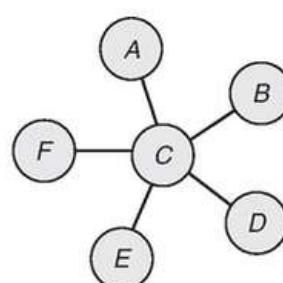
rede totalmente conectada



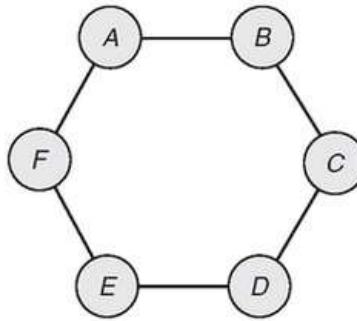
rede parcialmente conectada



rede estruturada em árvore



rede em estrela



rede em anel

FIGURA 16.4 Topologias de rede.

Em uma **rede parcialmente conectada** existem enlaces diretos entre alguns – mas não todos – pares de instalações. Logo, o custo da instalação dessa configuração é menor do que o custo da rede totalmente conectada. Todavia, se duas instalações A e B não estiverem conectadas diretamente, as mensagens de uma para a outra precisarão ser **roteadas** por uma sequência de enlaces de comunicação. Esse requisito resulta em um custo de comunicação maior.

Se um enlace de comunicação falhar, as mensagens que teriam sido transmitidas pelo enlace precisam ser roteadas novamente. Em alguns casos, outra rota pela rede poderá ser encontrada, de modo que as mensagens são capazes de alcançar seu destino. Em outros casos, uma falha pode significar que não existe uma conexão entre algum par (ou pares) de instalações. Quando um sistema está dividido em dois (ou mais) subsistemas não conectados, ele está particionado. Sob essa definição, um subsistema (ou partição) pode consistir em um único nó.

Algumas das topologias de rede mais comuns são redes estruturadas em árvore, redes de anel e redes de estrela, como mostra a [Figura 16.4](#). Esses tipos possuem diferentes características de falha e custos de instalação e comunicação. Os custos de instalação e comunicação são relativamente baixos para uma rede estruturada em árvore. No entanto, a falha de um único enlace em tal rede pode resultar na repartição da rede. Em uma rede de anel, pelo menos dois enlaces precisam falhar para haver repartição. Assim, a rede de anel possui um maior grau de disponibilidade do que a rede estruturada em árvore. Contudo, o custo de comunicação é alto, pois uma mensagem pode ter de atravessar uma grande quantidade de enlaces. Em uma rede em estrela, a falha de um único enlace resulta em uma repartição na rede, mas uma das partições tem apenas uma única instalação. Essa partição pode ser tratada como uma falha de única instalação. A rede em estrela também possui baixo custo de comunicação, pois cada instalação está no máximo dois enlaces de distância de qualquer outra instalação. Entretanto, se a instalação central falhar, todas as instalações no sistema ficarão desconectadas.

16.5 Estrutura de comunicação

Agora que já discutimos os aspectos físicos das redes, voltemos para o funcionamento interno. O projetista de uma rede de comunicação precisa enfrentar cinco aspectos básicos:

- **Nomeação e tradução de nomes.** Como dois processos localizam um ao outro para se comunicar?
- **Estratégias de roteamento.** Como as mensagens são enviadas pela rede?
- **Estratégias de pacote.** Os pacotes são enviados individualmente ou como uma sequência?
- **Estratégias de conexão.** Como dois processos enviam uma sequência de mensagens?
- **Disputa.** Como resolvemos demandas em conflito para o uso das redes, visto que esse é um recurso compartilhado?

Nas seções a seguir, vamos detalhar cada um desses aspectos.

16.5.1 Nomeação e tradução de nomes

O primeiro componente da comunicação na rede é a nomeação dos sistemas na rede. Para um processo na instalação A trocar informações com um processo na instalação B, cada um precisa ser capaz de especificar o outro. Dentro de um computador, cada processo tem um identificador de processo, e as mensagens podem ser endereçadas com o identificador de processo. Porém, como os sistemas em rede não compartilham memória, um host dentro do sistema não possui conhecimento sobre os processos em outros hosts.

Para resolver esse problema, os processos em sistemas remotos são identificados pelo par *<nome de host, identificador>*, onde *nome de host* é um nome exclusivo dentro da rede e *identificador* pode ser um identificador de processo ou outro número exclusivo dentro desse host. Um *nome de host* costuma ser um identificador alfanumérico, em vez de um número, para facilitar a especificação do usuário. Por exemplo, a instalação A poderia ter hosts chamados *homer*, *marge*, *bart* e *lisa*. *Bart* certamente é mais fácil de lembrar do que *12814831100*.

Os nomes são convenientes para uso humano, mas os computadores preferem números, por velocidade e simplicidade. Por esse motivo, é preciso haver um mecanismo para **traduzir** o nome de host para um **identificador de host**, que descreva o sistema de destino para o hardware de rede. Esse mecanismo é semelhante à associação de nome e endereço que ocorre durante a compilação, vínculo, carregamento e execução de um programa ([Capítulo 8](#)). No caso dos nomes de host, existem duas possibilidades. Primeiro, cada host pode ter um arquivo de dados contendo os nomes e endereços de todos os outros hosts que podem ser alcançados na rede (semelhante ao vínculo em tempo de compilação). O problema com esse modelo é que a inclusão ou remoção de um host da rede exige a atualização dos arquivos de dados em todos os hosts. A alternativa é distribuir as informações entre os sistemas na rede. A rede precisa, então, usar um protocolo para distribuir e apanhar as informações. Esse esquema é como o vínculo em tempo de execução. O primeiro método foi aquele usado originalmente na Internet; porém, com o crescimento da Internet, ele se tornou insustentável, de modo que o segundo método, o **sistema de nome de domínio (Domain-Name System - DNS)**, entrou em uso e permanece até hoje.

O DNS especifica a estrutura de nomes dos hosts, além da tradução de nomes para endereços. Os hosts na Internet são endereçados logicamente com um nome em múltiplas partes, conhecidos como endereços IP. As partes de um endereço IP seguem da parte mais específica para a parte mais genérica do endereço, com pontos separando os campos. Por exemplo, *bob.cs.brown.edu* refere-se ao host *bob* no Departamento de Ciência da Computação (*cs*: computer science) da Universidade Brown dentro do domínio de alto nível *edu*. (Outros domínios de alto nível são *com* para instalações comerciais, *org* para organizações, assim como o domínio para cada país conectado à rede, para sistemas especificados por país, em vez de tipo de organização.) Em geral, o sistema traduz endereços examinando os componentes de nome de host em ordem inversa. Cada componente possui um **servidor de nomes** – um processo em um sistema – que aceita um nome e retorna o endereço do servidor de nomes responsável por esse nome. Como etapa final, o servidor de nomes para o host em questão é contatado e um identificador de host é retornado. Por exemplo, uma requisição feita por um processo no sistema A para se comunicar com *bob.cs.brown.edu* resultaria nas seguintes etapas:

1. O kernel do sistema A emite uma requisição ao servidor de nomes para o domínio *edu*, pedindo o endereço do servidor de nomes para *brown.edu*. O servidor de nomes para o domínio *edu* precisa estar em um endereço conhecido, de modo a poder ser consultado.
2. O servidor de nomes *edu* retorna o endereço do host em que o servidor de nomes *brown.edu* reside.
3. O kernel no sistema A, então, consulta o servidor de nomes nesse endereço e pergunta a respeito de *cs.brown.edu*.
4. Um endereço é retornado, e agora uma requisição para o endereço *bob.cs.brown.edu*, finalmente, retorna um identificador de host com um **endereço da Internet** para esse host (por exemplo, 128.148.31.100).

Esse protocolo pode parecer ineficaz, mas caches locais normalmente são mantidos em cada servidor de nomes para agilizar o processo. Por exemplo, o servidor de nomes *edu* teria *brown.edu* em seu cache e informaria ao sistema A que ele poderia traduzir duas partes do nome, retornando um ponteiro para o servidor de nomes *cs.brown.edu*. Naturalmente, o conteúdo desses caches precisa ser atualizado com o tempo, caso o servidor de nomes seja movido ou seu endereço mude. Na verdade, esse serviço é tão importante que muitas otimizações ocorreram no protocolo, além de muitas proteções. Considere o que aconteceria se o servidor de nomes *edu* principal falhasse. É possível que nenhum host *edu* pudesse ter seus endereços traduzidos, tornando-os inalcançáveis! A solução é usar servidores de nomes secundários, de reserva, que duplicam o conteúdo dos servidores principais.

Antes de o serviço de nome de domínio ser introduzido, todos os hosts da Internet precisavam ter cópias de um arquivo que continha os nomes e endereços de cada host na rede. Todas as mudanças nesse arquivo tinham de ser registradas em uma instalação (host SRI-NIC), e periodicamente todos os hosts tinham de copiar o arquivo atualizado do SRI-NIC para poderem contatar novos sistemas ou encontrar hosts cujos endereços tinham mudado. Sob o serviço de nome de domínio, a instalação de cada servidor de nomes é responsável por atualizar as informações do host para esse domínio. Por exemplo, quaisquer mudanças no host da Universidade Brown são de responsabilidade do servidor de nomes para *brown.edu* e não precisam ser informadas em qualquer outro lugar. As pesquisas de DNS apanharão automaticamente a informação atualizada, porque *brown.edu* é contatado de forma direta. Dentro dos domínios, pode haver subdomínios autônomos para distribuir ainda mais a responsabilidade pelas mudanças de nome de host e identificador de host.

A Java provê a API necessária para projetar um programa que mapeia nomes IP em endereços IP. O programa mostrado na [Figura 16.5](#) recebe um nome IP (como *bob.cs.brown.edu*) na linha de comandos e gera o endereço IP do host ou retorna uma mensagem indicando que o nome do host não pôde ser traduzido. Um *InetAddress* é uma classe Java que representa um nome ou endereço IP. O método estático *getByName()*, pertencente à classe *InetAddress*, recebe uma representação em string de um nome IP, e retorna o *InetAddress* correspondente. O programa, então, chama o método *getHostAddress()*, que internamente utiliza DNS para pesquisar o endereço IP do host designado.

```
/*
 * Uso:    Java DNSLookUp <nome IP>
 * Exemplo: Java DNSLookUp www.wiley.com
 */
public class DNSLookUp {
    public static void main(String[ ] args) {
        InetAddress hostAddress;

        try {
            hostAddress = InetAddress.getByName(args[0]);
            System.out.println(hostAddress.getHostAddress());
        }
        catch (UnknownHostException uhe) {
            System.err.println("host desconhecido: " + args[0]);
        }
    }
}
```

FIGURA 16.5 Programa Java ilustrando uma pesquisa de DNS.

Em geral, o sistema operacional é responsável por aceitar dos seus processos uma mensagem destinada para <nome host, identificador> e por transferi-la para o host apropriado. O kernel no host de destino é responsável por transferir a mensagem para o processo nomeado pelo identificador. Essa troca, de forma alguma, é trivial; ela é descrita na [Seção 16.5.4](#).

16.5.2 Estratégias de roteamento

Quando um processo na instalação A deseja se comunicar com um processo na instalação B, como a mensagem é enviada? Se houver apenas um caminho físico de A para B (como em uma rede estruturada em estrela ou árvore), a mensagem precisa ser enviada por esse caminho. Todavia, se houver vários caminhos físicos de A para B, então existem várias opções de roteamento. Cada instalação possui uma **tabela de roteamento** indicando os caminhos alternativos que podem ser

usados para enviar uma mensagem para outras instalações. A tabela pode incluir informações sobre a velocidade e o custo dos diversos caminhos de comunicação e pode ser atualizada conforme a necessidade, seja manualmente seja por programas que trocam informações de roteamento. Os três esquemas de roteamento mais comuns são **roteamento fixo**, **roteamento virtual** e **roteamento dinâmico**.

■ **Roteamento fixo.** Um caminho de A para B é especificado em avanço e não muda a menos que uma falha do hardware o desative. Normalmente, o caminho mais curto é o escolhido, de modo que os custos de comunicação são minimizados.

■ **Roteamento virtual.** Um caminho de A para B é fixo pela duração de uma **sessão**. Diferentes sessões envolvendo mensagens de A para B podem usar diferentes caminhos. Uma sessão poderia ser tão curta quanto uma transferência de arquivo ou tão longa quanto um período de login remoto.

■ **Roteamento dinâmico.** O caminho usado para enviar uma mensagem da instalação A para a instalação B é escolhido somente quando uma mensagem é enviada. Como a decisão é feita dinamicamente, mensagens separadas podem ser atribuídas a diferentes caminhos. A instalação A tomará uma decisão de enviar a mensagem para a instalação C; C, por sua vez, decidirá enviá-la para a instalação D, e assim por diante. Por fim, uma instalação entregará a mensagem a B. Em geral, uma instalação envia uma mensagem para outra instalação, qualquer que seja o enlace menos utilizado nesse determinado momento.

Existem prós e contras entre esses três esquemas. O roteamento fixo não pode se adaptar a falhas de enlace ou mudanças de carga. Em outras palavras, se um caminho tiver sido estabelecido entre A e B, as mensagens precisam ser enviadas por esse caminho, mesmo que ele esteja interrompido ou esteja sendo mais usado do que outro caminho possível. Podemos remediar em parte esse problema usando o roteamento virtual e evitá-lo completamente usando o roteamento dinâmico. O roteamento fixo e o roteamento virtual garantem que as mensagens de A para B serão entregues na ordem em que foram enviadas. No roteamento dinâmico, as mensagens podem chegar fora de ordem. Podemos remediar esse problema acrescentando um número de sequência a cada mensagem.

O roteamento dinâmico é o mais complicado de configurar e executar; porém, essa é a melhor maneira de gerenciar o roteamento em ambientes complicados. O UNIX provê o roteamento fixo para usar nos hosts dentro de redes simples e o roteamento dinâmico para ambientes de rede mais complicados. Também é possível misturar os dois. Dentro de uma instalação, os hosts podem precisar saber apenas como alcançar o sistema que conecta a rede local a outras redes (como as redes da empresa ou a Internet). Esse nó é conhecido como **gateway**. Cada host individual possui uma rota estática para o gateway, embora o próprio gateway utilize roteamento dinâmico para alcançar qualquer host no restante da rede.

Um roteador é uma entidade dentro da rede de computadores responsável por rotear mensagens. Um roteador pode ser um computador com software de roteamento ou um dispositivo de uso especial. De qualquer forma, um roteador precisa ter pelo menos duas conexões de rede ou não terá um lugar para rotear as mensagens. Um roteador decide se a mensagem precisa ser passada da rede em que é recebida para qualquer outra rede conectada ao roteador. Ele faz essa determinação examinando o endereço Internet do destino da mensagem. O roteador verifica suas tabelas para determinar o local do host de destino ou, pelo menos, da rede para a qual enviará a mensagem para o host de destino. No caso do roteamento estático, essa tabela é alterada apenas pela atualização manual (um novo arquivo é carregado no roteador). Com o roteador dinâmico, um **protocolo de roteamento** é usado entre os roteadores para informá-los das mudanças da rede e para permitir que atualizem suas tabelas de roteamento automaticamente. Os gateways e os roteadores são dispositivos de hardware dedicados que executam código a partir de um firmware.

16.5.3 Estratégias de pacote

As mensagens geralmente variam em tamanho. Para simplificar o projeto do sistema, implementamos a comunicação com mensagens de tamanho fixo, chamadas **pacotes**, **quadros** ou **datagramas**. Uma comunicação implementada em um pacote pode ser enviada para seu destino em uma **mensagem sem conexão**. Uma mensagem sem conexão pode ser **não confiável** quando o emissor não tem garantias de que (e não pode saber se) o pacote chegou ao seu destino. Como alternativa, o pacote pode ser **confiável**; nesse caso, um pacote é retornado do destino para indicar que o outro pacote chegou. (Naturalmente, o pacote de retorno poderia se perder no caminho.) Se uma mensagem for muito longa para caber em um pacote ou se os pacotes precisarem fluir de um lado para outro entre os dois comunicadores, uma conexão é estabelecida, para permitir a troca confiável de vários pacotes.

16.5.4 Estratégias de conexão

Quando as mensagens são capazes de alcançar seus destinos, os processos podem instituir **sessões de comunicações** para trocar informações. Pares de processos que querem se comunicar pela rede podem ser conectados de diversas maneiras. Os três esquemas mais comuns são **comutação de**

circuitos, troca de mensagens e comutação de pacotes.

■ **Comutação de circuitos.** Se dois processos desejam se comunicar, um enlace físico permanente é estabelecido entre eles. Esse enlace é alocado pela duração da sessão de comunicação, e nenhum outro processo pode usar esse enlace durante esse período (mesmo que os dois processos não estejam se comunicando ativamente por um tempo). Esse esquema é semelhante ao usado no sistema telefônico. Quando uma linha de comunicação tiver sido aberta entre as duas partes (ou seja, a parte A chama a parte B), ninguém mais pode usar esse circuito até a comunicação ser terminada (por exemplo, quando as duas partes colocam o fone no gancho).

■ **Troca de mensagens.** Se dois processos quiserem se comunicar, um enlace temporário será estabelecido pela duração de uma transferência de mensagem. Os enlaces físicos são alocados dinamicamente entre os correspondentes, conforme a necessidade, e são alocados somente por períodos curtos. Cada mensagem é um bloco de dados com informações do sistema - como a origem, o destino e os códigos de proteção de erro (ECC) -, que permitem à rede de comunicação entregar a mensagem corretamente ao destino. Esse esquema é semelhante ao sistema de correspondência da agência de correios. Cada carta é considerada uma mensagem que contém o endereço de destino e a origem (remetente). Muitas mensagens (de diferentes usuários) podem ser enviadas pelo mesmo enlace.

■ **Comutação de pacotes.** Uma mensagem lógica pode ter de ser dividida em uma série de pacotes. Cada pacote pode ser enviado ao destino separadamente e, por isso, cada um precisa incluir um endereço de origem e destino com seus dados. Além do mais, cada pacote pode seguir um caminho diferente pela rede. Os pacotes precisam ser remontados em mensagens à medida que chegam. Observe que não é prejudicial para os dados serem divididos em pacotes, possivelmente roteados de forma separada e remontados no destino. Ao contrário, o desmembramento de um sinal de áudio (digamos, uma comunicação telefônica) poderia causar grande confusão se não fosse feito com cuidado.

Existem compensações óbvias entre esses esquemas. A comutação de circuitos exige um tempo de preparação substancial e pode desperdiçar largura de banda de rede, mas gera menos custo adicional para enviar cada mensagem. Como consequência, a troca de mensagens e a comutação de pacotes exigem menos tempo de preparação, mas ocasiona um custo adicional maior por mensagem. Além disso, na comutação de pacotes cada mensagem precisa ser dividida em pacotes e remontada mais tarde. A comutação de pacotes é o método mais utilizado nas redes de dados, pois faz o melhor uso da largura de banda da rede.

16.5.5 Disputa

Dependendo da topologia da rede, um enlace pode conectar mais de duas instalações na rede de computadores, e várias dessas instalações podem querer transmitir informações por um enlace ao mesmo tempo. Essa situação ocorre principalmente em uma rede com barramento em anel ou multiacesso. Nesse caso, a informação transmitida pode se tornar embaralhada. Se isso acontecer, ela precisa ser descartada, e as instalações precisam ser notificadas sobre o problema para poderem retransmitir as informações. Se nenhuma provisão especial for feita, essa situação pode ser repetida, resultando em desempenho prejudicado. Várias técnicas foram desenvolvidas para evitar colisões repetidas, incluindo detecção de colisão (Collision Detection - CD) e passagens token.

■ **CSMA/CD.** Antes de transmitir uma mensagem por um enlace, uma instalação precisa escutar para determinar se outra mensagem está sendo transmitida por esse enlace; essa técnica é denominada **Carrier Sense with Multiple Access (CSMA)**. Se o enlace estiver livre, a instalação poderá começar a transmitir. Caso contrário, ela terá de esperar (e continuar escutando) até o enlace estar livre. Se duas ou mais instalações começarem a transmitir exatamente ao mesmo tempo (cada uma pensando que nenhuma outra instalação está usando o enlace), então elas registrarão uma **detecção de colisão (Collision Detection - CD)** e deixarão de transmitir. Cada instalação tentará mais uma vez após algum intervalo de tempo aleatório. O problema principal com essa técnica é que, quando o sistema está muito ocupado, muitas colisões podem ocorrer e, com isso, o desempenho pode ser prejudicado. Apesar disso, o CSMA/CD tem sido usado com sucesso no sistema Ethernet, o sistema de rede mais comum. Uma estratégia para limitar a quantidade de colisões é limitar a quantidade de hosts por rede Ethernet. A inclusão de mais hosts a uma rede congestionada poderia resultar em baixo throughput de rede. À medida que os sistemas se tornam mais rápidos, eles podem enviar mais pacotes por segmento de tempo. Como resultado, a quantidade de sistemas por segmento Ethernet está diminuindo para o desempenho da rede ser razoável.

■ **Passagem de tokens.** Um tipo de mensagem exclusivo, conhecido como **token**, circula continuamente no sistema (em geral, uma estrutura de anel). Uma instalação que queira transmitir informações precisa esperar até o token chegar. Ela então remove a ficha do anel e começa a transmitir suas mensagens. Quando a instalação termina sua vez de passar mensagens, ela retransmite o token. Essa ação, por sua vez, permite que outra instalação receba e remova o token e inicie sua transmissão de mensagem. Se o token se perder, então os sistemas precisam detectar a perda e gerar um novo token. Eles fazem isso declarando uma **eleição** para escolher

uma instalação exclusiva onde um novo token será gerado. Mais adiante, na [Seção 18.6](#), apresentamos um algoritmo de eleição. Um esquema de passagem de tokens foi adotado pelos sistemas IBM e HP/Apollo. O benefício de uma rede de passagem de tokens é que o desempenho é constante. A inclusão de novos sistemas em uma rede pode aumentar o tempo de espera para o token, mas isso não causará grande diminuição no desempenho, como pode acontecer nas redes Ethernet. Contudo, em redes pouco carregadas, a Ethernet é mais eficiente, pois os sistemas podem enviar mensagens a qualquer momento.

16.6 Protocolos de comunicação

Quando estamos projetando uma rede de comunicação, temos de lidar com a complexidade inerente de coordenar as operações assíncronas comunicando em um ambiente potencialmente lento e passível de erros. Além disso, os sistemas na rede precisam combinar sobre um protocolo ou um conjunto de protocolos para determinar nomes de host, localizar hosts na rede, estabelecer conexões, e assim por diante. Podemos simplificar o problema de projeto (e a implementação relacionada) dividindo o problema em várias camadas. Cada camada em um sistema se comunica com a camada equivalente nos outros sistemas. Normalmente, cada camada possui seus próprios protocolos e a comunicação ocorre entre camadas emparelhadas, usando um protocolo específico. Os protocolos podem ser implementados no hardware ou no software. Por exemplo, a [Figura 16.6](#) mostra as comunicações lógicas entre dois computadores, com as três camadas de nível mais baixo implementadas no hardware. Seguindo a International Standards Organization (ISO), referimo-nos às camadas com as seguintes descrições:

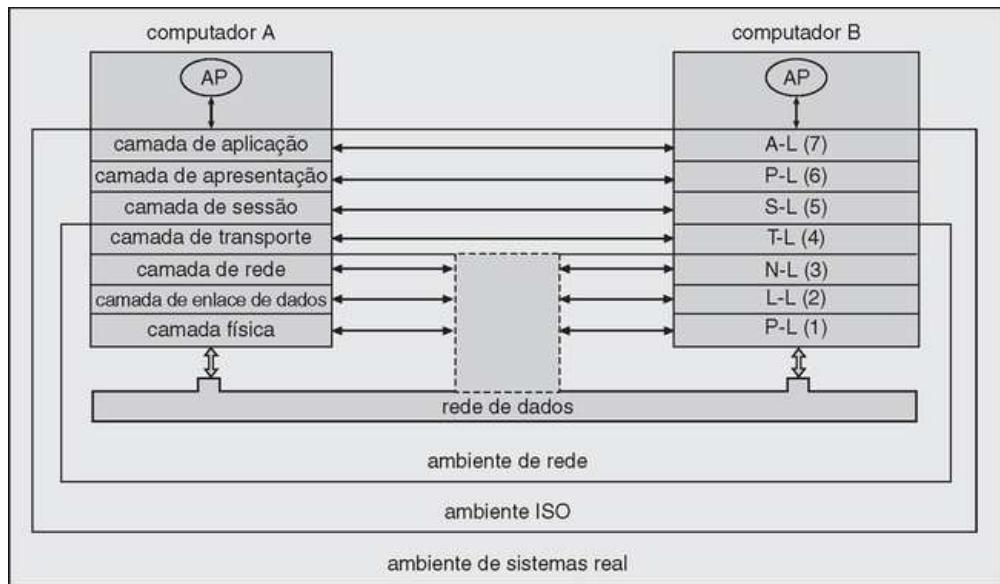


FIGURA 16.6 Dois computadores se comunicando por meio do modelo de rede ISO.

- 1. Camada física.** A camada física é responsável por tratar dos detalhes mecânicos e elétricos da transmissão física de um fluxo de bits. No nível físico, o sistema se comunicando precisa combinar com a representação elétrica de um binário 0 e 1, de modo que, quando os dados são enviados como um fluxo de sinais elétricos, o receptor seja capaz de interpretar os dados corretamente como dados binários. Essa camada é implementada no hardware do dispositivo de rede.
- 2. Camada de enlace de dados.** A camada de enlace de dados é responsável por tratar dos *frames* (*quadros*) ou partes de pacotes de tamanho fixo, incluindo qualquer detecção e recuperação de erro que ocorra na camada física.
- 3. Camada de rede.** A camada de rede é responsável por fornecer conexões e por rotear pacotes na rede de comunicação, incluindo o tratamento do endereço dos pacotes de saída, decodificação do endereço dos pacotes que chegam e manutenção de informações de roteamento para a resposta apropriada em níveis de carga variáveis. Os roteadores trabalham nessa camada.
- 4. Camada de transporte.** A camada de transporte é responsável pelo acesso em baixo nível para a rede e pela transferência de mensagens entre os clientes, incluindo o desmembramento de mensagens em pacotes, manutenção da ordem dos pacotes, controle de fluxo e geração de endereços físicos.
- 5. Camada de sessão.** A camada de sessão é responsável por implementar sessões ou protocolos de comunicação de processo a processo. Em geral, esses protocolos são as comunicações reais para logins remotos e para transferências de arquivo e correspondência.
- 6. Camada de apresentação.** A camada de apresentação é responsável por resolver as diferenças nos formatos entre as diversas instalações na rede, incluindo conversões de caracteres e modos half-duplex e full-duplex (eco de caracteres).
- 7. Camada de aplicação.** A camada de aplicação é responsável por interagir diretamente com os usuários. Essa camada trata de transferência de arquivo, protocolos de login remoto e correio

eletrônico, bem como esquemas para bancos de dados distribuídos.

A [Figura 16.7](#) resume a **pilha de protocolos ISO** – um conjunto de protocolos de cooperação –, mostrando o fluxo de dados físico. Como dissemos, logicamente, cada camada de uma pilha de protocolos se comunica com a camada equivalente em outros sistemas. Todavia, fisicamente, uma mensagem começa na camada de aplicação ou acima, e passa em cada nível por sua vez. Cada camada pode modificar a mensagem e incluir dados de cabeçalho de mensagem para a camada equivalente no lado receptor. Por fim, a mensagem chega à camada de rede de dados e é transferida como um ou mais pacotes ([Figura 16.8](#)). A camada de enlace de dados do sistema de destino recebe esses dados e a mensagem sobe pela pilha de protocolos; ela é analisada, modificada e tem seus cabeçalhos removidos enquanto prossegue. Por fim, ela alcança a camada de aplicação para ser usada pelo processo receptor.

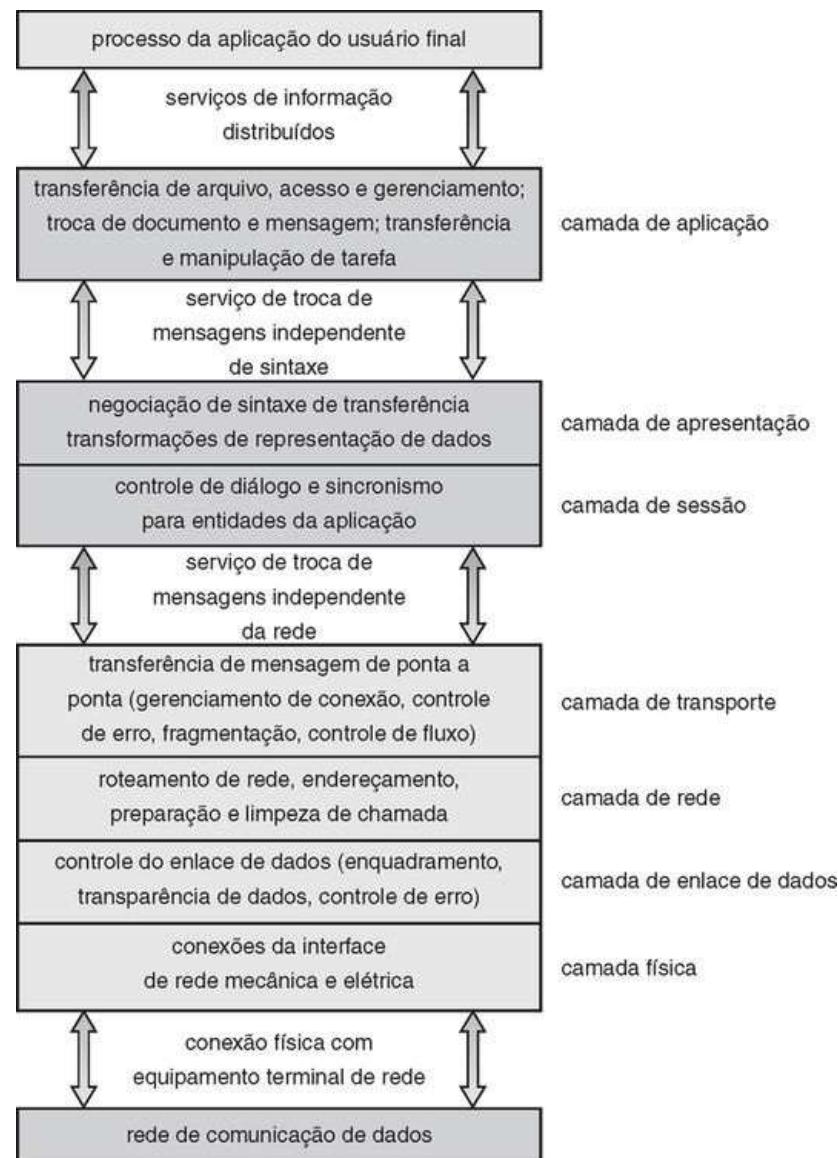


FIGURA 16.7 A pilha de protocolos ISO.

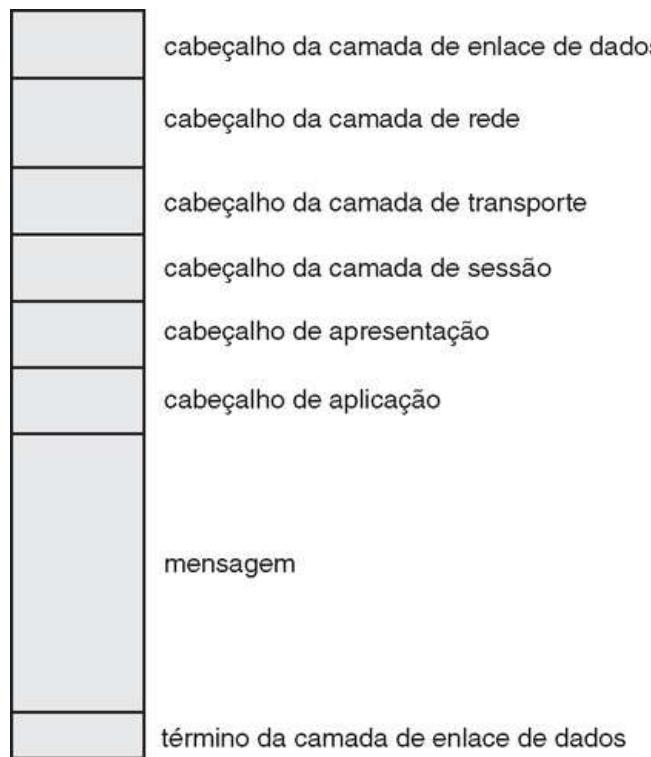


FIGURA 16.8 Uma mensagem de rede ISO.

O modelo ISO formaliza parte do trabalho mais antigo feito nos protocolos de rede, mas foi desenvolvido no final da década de 1970 e atualmente não está em uso generalizado. Talvez a pilha de protocolos mais adotada seja o modelo TCP/IP, adotado por praticamente todas as instalações da Internet. A pilha de protocolos TCP/IP possui menos camadas do que o modelo ISO. Teoricamente, por combinar várias funções em cada camada, ela é mais difícil de implementar, porém mais eficiente do que as redes ISO. O relacionamento entre os modelos ISO e TCP/IP aparece na [Figura 16.9](#). A camada de aplicação TCP/IP identifica vários protocolos em uso atualmente na Internet, incluindo HTTP, FTP, Telnet, DNS e SMTP. A camada de transporte identifica o protocolo não confiável e sem conexão **User Datagram Protocol (UDP)** e o protocolo confiável e orientado a conexão **Transmission Control Protocol (TCP)**. O Internet Protocol (**IP**) é responsável por rotear datagramas IP pela Internet. O modelo TCP/IP não identifica formalmente uma camada de enlace ou física, permitindo que o tráfego TCP/IP atravessesse qualquer rede física. Na [Seção 16.9](#), consideraremos o modelo TCP/IP trabalhando em cima de uma rede Ethernet.

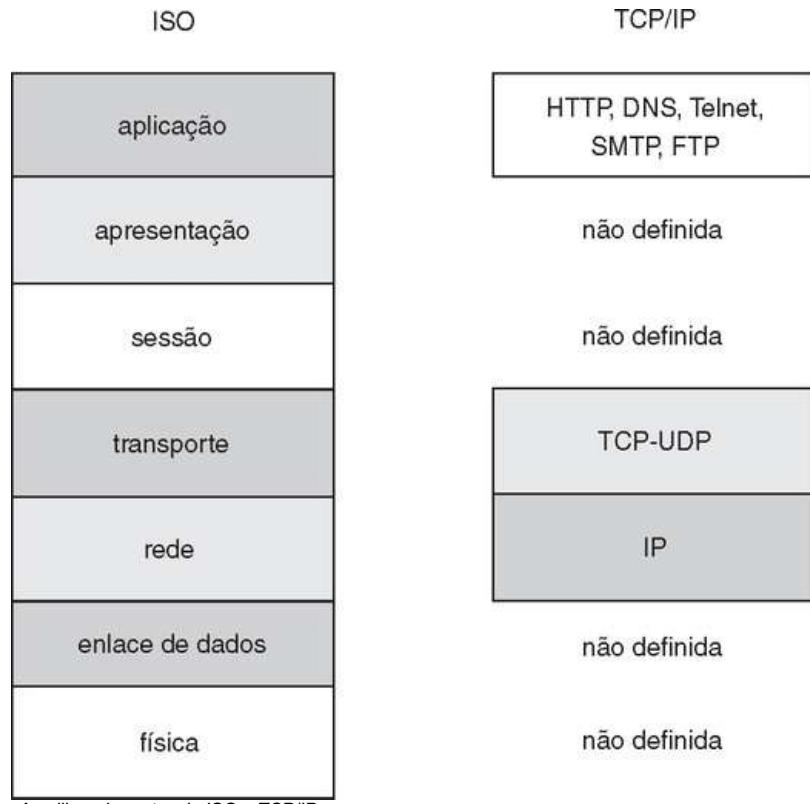


FIGURA 16.9 As pilhas de protocolo ISO e TCP/IP.

A segurança deverá ser uma preocupação no projeto e na implementação de qualquer protocolo de comunicação moderno. Tanto a autenticação forte quanto a criptografia forte são necessárias para a comunicação segura. Autenticação forte garante que o emissor e o receptor de uma comunicação sejam quem ou o que eles deveriam ser. A criptografia protege o conteúdo da comunicação contra bisbilhotagem. No entanto, uma autenticação fraca e comunicações em texto claro ainda são muito comuns, por diversos motivos. Quando a maioria dos protocolos comuns foi projetada, a segurança costumava ser menos importante do que o desempenho, a simplicidade e a eficiência.

A autenticação forte requer um protocolo de handshake em várias etapas ou dispositivos de autenticação, aumentando a complexidade de um protocolo. As CPUs modernas podem realizar a criptografia de modo eficiente, e os sistemas frequentemente deixam a criptografia para processadores de criptografia separados, de modo que o desempenho do sistema não é comprometido. A comunicação em longa distância pode se tornar segura pela autenticação das extremidades e a criptografia do fluxo de pacotes em uma rede privada virtual, conforme discutimos na [Seção 15.4.2](#). A comunicação pela rede local continua sendo não criptografada na maioria das instalações, mas protocolos como NFS versão 4, que incluem forte autenticação e criptografia nativa, deverão ajudar a melhorar até mesmo a segurança da rede local.

16.7 Robustez

Um sistema distribuído pode sofrer de vários tipos de falhas de hardware. A falha de um enlace, a falha de uma instalação e a perda de uma mensagem são as falhas mais comuns. Para garantir que o sistema seja robusto, precisamos detectar qualquer uma dessas falhas, reconfigurar o sistema de modo que a computação possa continuar e recuperar quando uma instalação ou um enlace for reparado.

16.7.1 Detecção de falhas

Em um ambiente sem memória compartilhada, em geral somos incapazes de diferenciar entre falha no enlace, falha na instalação e perda de mensagem. Normalmente, podemos detectar apenas que uma dessas falhas ocorreu. Quando uma falha tiver sido detectada, a ação apropriada precisa ser realizada. A ação apropriada depende da aplicação.

Para detectar a falha de enlace e instalação, usamos um procedimento de **confirmação inicial (handshaking)**. Suponha que as instalações A e B tenham um enlace físico direto entre elas. Em intervalos fixos, as instalações enviam entre si uma mensagem *estou ativa*. Se a instalação A não receber essa mensagem dentro de um período predeterminado, ela pode considerar que a instalação B falhou, que o enlace entre A e B falhou ou que a mensagem de B foi perdida. Nesse ponto, a instalação A tem duas opções. Ela pode esperar outro período para receber uma mensagem *estou ativa* de B ou pode enviar uma mensagem “*você está ativa?*” para B.

Se o tempo passar e a instalação A ainda não tiver recebido uma mensagem *estou ativa* ou se a instalação A tiver enviado uma mensagem “*você está ativa?*” e não tiver recebido uma resposta, o procedimento pode ser repetido. Novamente, a única conclusão que a instalação A pode tirar com segurança é que ocorreu algum tipo de falha.

A instalação A pode tentar diferenciar entre a falha do enlace e a falha da instalação, enviando uma mensagem “*você está ativa?*” para B por outra rota (se houver). Se e quando B receber essa mensagem, ela responde positivamente. Essa resposta positiva diz a A que B está ativa e que a falha está no enlace direto entre elas. Como não sabemos antecipadamente quanto tempo levará a mensagem para atravessar de A para B e voltar, temos de usar um **esquema de time-out**. No momento em que A envia a mensagem “*você está ativa?*”, ela especifica um intervalo de tempo durante o qual deseja esperar pela resposta de B. Se A receber a mensagem de resposta dentro desse intervalo de tempo, então ela pode seguramente concluir que B está ativa. Contudo, se não receber (ou seja, se ocorrer o tempo limite), então A pode concluir apenas que ocorreu uma ou mais das situações a seguir:

- A instalação B está parada.
- O enlace direto (se existir) de A para B está interrompido.
- O caminho alternativo de A para B está interrompido.
- A mensagem foi perdida.

Contudo, a instalação A não pode determinar qual desse eventos ocorreu.

16.7.2 Reconfiguração

Suponha que a instalação A tenha descoberto, por meio do mecanismo descrito na seção anterior, que ocorreu uma falha. Ela precisa, então, iniciar um procedimento que permitirá ao sistema reconfigurar e continuar seu modo de operação normal.

- Se um enlace direto de A para B tiver falhado, essa informação precisa ser transmitida por broadcast a cada instalação no sistema, de modo que as diversas tabelas de roteamento possam ser atualizadas corretamente.
- Se o sistema acredita que uma instalação falhou (porque essa instalação não pode mais ser alcançada), então cada instalação no sistema precisa ser notificada, para não tentarem mais usar os serviços da instalação que falhou. A falha de uma instalação que atende como um coordenador central para alguma atividade (como detecção de deadlock) exige a escolha de um novo coordenador. Da mesma forma, se a instalação que falhou faz parte de um anel lógico, então um novo anel lógico precisa ser construído. Observe que, se a instalação não tiver falhado (ou seja, se estiver ativa, mas não puder ser alcançada), então podemos ter a situação indesejável em que duas instalações servem como coordenadores. Quando a rede é particionada, os dois coordenadores (cada um para sua própria partição) podem iniciar ações conflitantes. Por exemplo, se os coordenadores forem responsáveis por implementar a exclusão mútua, podemos ter uma situação em que dois processos estão executando concorrentemente em suas seções críticas.

16.7.3 Recuperação da falha

Quando um enlace ou instalação que falhou é reparado, ele precisa ser integrado ao sistema de

forma controlada e tranquila.

- Suponha que um enlace entre A e B tenha falhado. Quando ele for reparado, tanto A quanto B precisam ser notificados. Podemos conseguir essa notificação repetindo continuamente o procedimento de confirmação inicial descrito na [Seção 16.7.1](#).
- Suponha que a instalação B tenha falhado. Quando ela se recuperar, terá de notificar a todas as outras instalações de que está ativa novamente. A instalação B, então, pode ter de receber informações das outras instalações para atualizar suas tabelas locais; por exemplo, ela pode precisar de informações da tabela de roteamento, uma lista das instalações que estão paradas ou mensagens e correspondência não entregue. Se a instalação não tiver falhado, mas apenas não puder ser alcançada, então essa informação ainda é necessária.

16.7.4 Tolerância a falhas

Um sistema distribuído precisa tolerar um nível de falha e continuar a funcionar normalmente quando enfrentar diversos tipos de falhas. Para tornar uma instalação tolerante a falhas, é preciso começar no nível de protocolo, conforme já descrevemos, mas continuar por todos os aspectos do sistema. Usamos o termo *tolerância a falhas* em um sentido amplo. Falhas de comunicação, falhas de máquina (do tipo *falhou, parou*, onde a máquina para antes de realizar uma operação errônea, visível a outros processadores), desastres com dispositivo de armazenamento e deterioração de mídia de armazenamento devem ser tolerados até certo ponto. Um **sistema tolerante a falhas** deverá continuar funcionando, talvez de uma forma degradada, quando encarado com esses tipos de falhas. A degradação pode ser no desempenho, na funcionalidade ou em ambos. Entretanto, ela deverá ser proporcional às falhas que a causaram. Um sistema que chega a parar quando somente alguns de seus componentes falharam certamente não é tolerante a falhas.

Infelizmente, a tolerância a falhas é difícil e dispendiosa de implementar. Na camada de rede, diversos caminhos de comunicação e dispositivos de rede redundantes, como switches e roteadores, são necessários para evitar uma falha de comunicação. Uma falha de armazenamento pode causar perda do sistema operacional, aplicações ou dados. As unidades de armazenamento podem incluir componentes de hardware redundantes, que assumem automaticamente um no lugar do outro em caso de falha. Além disso, sistemas RAID podem garantir o acesso continuado aos dados, mesmo no evento de uma ou mais falhas de disco ([Seção 12.7](#)).

Uma falha no sistema sem redundância pode fazer uma aplicação ou uma instalação inteira parar de operar. A falha de sistema mais simples envolve um sistema que executa apenas aplicações sem estado. Essas aplicações podem ser reiniciadas sem comprometer a operação; portanto, desde que as aplicações possam ser executadas em mais de um computador (nó), a operação pode continuar. Essa instalação normalmente é conhecida como **cluster de computação**, pois gira em torno na computação.

Por outro lado, os sistemas que giram em torno de dados envolvem a execução de aplicações paralelas que acessam e modificam dados compartilhados. Como resultado, as instalações de computação centralizadas nos dados são mais difíceis de se tornarem tolerantes a falhas. Elas exigem software de monitoramento de falhas e infraestrutura especial. Por exemplo, **clusters de alta disponibilidade**, como Veritas Cluster e Sun Cluster, incluem dois ou mais computadores e um conjunto de discos compartilhados. Qualquer aplicação pode ser armazenada nos computadores ou no disco compartilhado, mas os dados precisam ser armazenados no disco compartilhado. O nó da aplicação em execução possui acesso exclusivo aos dados da aplicação no disco. A aplicação é monitorada pelo software de cluster e, se ela falhar, é automaticamente reiniciada. Se ela não puder ser reiniciada, ou se o computador inteiro falhar, o acesso exclusivo do nó aos dados da aplicação é terminado e entregue a outro nó no cluster. A aplicação é reiniciada nesse novo nó. A aplicação perde qualquer informação de estado que esteja na memória do sistema que falhou, mas pode continuar com base no estado salvo por último no disco compartilhado. Do ponto de vista de um usuário, um serviço foi interrompido e depois reiniciado, possivelmente com algum dado faltando.

Aplicações específicas podem melhorar essa funcionalidade implementando o gerenciamento de lock juntamente com o uso de clusters. Com o gerenciamento de lock ([Seção 18.4.1](#)), a aplicação pode ser executada em vários nós e pode usar os mesmos dados em discos compartilhados simultaneamente. Os bancos de dados em clusters constantemente implementam essa funcionalidade. Se um nó falha, as transações podem continuar em outros nós, e os usuários não notam qualquer interrupção de serviço, desde que o cliente possa localizar automaticamente os outros nós no cluster. Quaisquer transações não confirmadas (ou “commitadas”) no nó que falhou são perdidas, porém, mas uma vez, as aplicações clientes podem ser projetadas para repetir as transações não confirmadas se detectarem uma falha do seu nó de banco de dados.

16.8 Aspectos de projeto

Tornar a multiplicidade de processadores e dispositivos de armazenamento **transparente** aos usuários tem sido um desafio importante para muitos projetistas. O ideal é que um sistema distribuído pareça a seus usuários um sistema convencional, centralizado. A interface com o usuário de um sistema distribuído não deverá distinguir entre recursos locais e remotos, ou seja, os usuários deverão ser capazes de acessar sistemas distribuídos remotos como se fossem locais, e o sistema distribuído deverá ser responsável por localizar os recursos e arrumar a interação apropriada.

Outro aspecto da transparência é a mobilidade do usuário. Seria conveniente permitir aos usuários se conectar a qualquer máquina no sistema, em vez de forçá-los a usar uma máquina específica. Um sistema distribuído transparente facilita a mobilidade do usuário, trazendo o ambiente do usuário (por exemplo, o diretório home) para onde quer que ele esteja conectado. Tanto o sistema de arquivos Andrew da CMU quanto o Project Athena do MIT proveem essa funcionalidade em grande escala. O NFS pode prover essa transparência em uma escala menor.

Outra questão é a **escalabilidade** - a capacidade de um sistema adaptar-se a uma carga de serviço aumentada. Os sistemas possuem recursos limitados e podem se tornar saturados sob uma carga aumentada. Por exemplo, com relação a um sistema de arquivos, a saturação ocorre quando a CPU de um servidor executa em uma alta taxa de utilização ou quando os discos estão quase cheios. A escalabilidade é uma propriedade relativa, mas pode ser medida com precisão. Um sistema escalável reage de forma mais controlada ao aumento de carga do que um sistema não escalável. Primeiro, seu desempenho degrada mais moderadamente; segundo, seus recursos atingem um estado saturado mais tarde. Até mesmo o projeto perfeito não consegue acomodar uma carga em crescimento contínuo. A inclusão de mais recursos pode solucionar o problema, mas pode gerar carga indireta adicional sobre outros recursos (por exemplo, a inclusão de máquinas a um sistema distribuído pode encher a rede e aumentar as cargas de serviço). Pior ainda, a expansão do sistema pode ocasionar modificações de projeto dispendiosas. Um sistema escalável deverá ter o potencial de crescer sem esses problemas. Em um sistema distribuído, a capacidade de escalar de forma controlada é de importância especial, pois a expansão da rede pela inclusão de novas máquinas ou a interconexão de duas redes é muito comum. Resumindo, um projeto escalável deverá suportar alta carga de serviço, acomodar o crescimento da comunidade de usuários e permitir a integração simples de recursos adicionais.

A tolerância a falhas e a escalabilidade estão relacionadas, como já discutimos. Um componente bastante carregado pode se tornar paralisado e se comportar como um componente falho. Além disso, deslocar a carga de um componente com falha para o backup desse componente pode saturar este último. Em geral, ter recursos de reserva é essencial para garantir a confiabilidade e também tratar de cargas de pico de forma controlada. Uma vantagem inerente a um sistema distribuído é um potencial para tolerância a falhas e escalabilidade devido à multiplicidade de recursos. Todavia, o projeto impróprio pode esconder esse potencial. Considerações de tolerância a falhas e escalabilidade exigem um projeto demonstrando a distribuição de controle e dados.

Sistemas distribuídos em escala muito grande, até certo ponto, ainda são apenas teóricos. Nenhuma orientação mágica garante a escalabilidade de um sistema. É mais fácil apontar por que os projetos atuais *não* são escaláveis. Em seguida, discutimos vários projetos que impõem problemas e propomos soluções possíveis, tudo no contexto da escalabilidade.

Um princípio para projetar sistemas em escala muito grande é que a demanda por serviço de qualquer componente do sistema deve ser limitada por uma constante, independentemente do número de nós no sistema. Qualquer mecanismo de serviço cuja demanda de carga é proporcional ao tamanho do sistema é destinado a tornar-se obstruído quando crescer além de determinado tamanho. A inclusão de mais recursos não aliviaria tal problema. A capacidade desse mecanismo limita o crescimento do sistema.

Outro princípio diz respeito à centralização. Os esquemas de controle centrais e os recursos centrais não devem ser usados para a criação de sistemas escaláveis (e tolerantes a falhas). Alguns exemplos de entidades centralizadas são servidores de autenticação centrais, servidores de nomes centrais e servidores de arquivos centrais. A centralização é uma forma de assimetria funcional entre as máquinas constituintes do sistema. A alternativa ideal é uma configuração funcionalmente simétrica, ou seja, todas as máquinas componentes possuem um papel igual na operação do sistema e, por isso, cada máquina tem algum grau de autonomia. Na prática, é quase impossível obedecer a esse princípio. Por exemplo, a incorporação de máquinas sem disco infringe a simetria funcional, pois as estações de trabalho dependem de um disco central. No entanto, a autonomia e a simetria são objetivos importantes, aos quais devemos aspirar.

A decisão sobre a estrutura de processos do servidor é um problema importante no projeto de qualquer serviço. Os servidores deveriam operar de forma eficiente em períodos de pico, quando centenas de clientes ativos precisam ser atendidos concorrentemente. Um servidor de único processo não é uma boa escolha, pois sempre que uma requisição precisar de E/S de disco, o serviço inteiro será bloqueado. A atribuição de um processo para cada cliente é uma opção melhor; porém, o custo das trocas de contexto frequentes entre os processos precisa ser considerado. Um problema

relacionado ocorre porque todos os processos do servidor precisam compartilhar informações.

Uma das melhores soluções para a arquitetura do servidor é o uso de processos leves, ou threads, sobre os quais discutimos no [Capítulo 4](#). Podemos pensar em um grupo de processos leves como várias threads de controle associadas a alguns recursos compartilhados. Em geral, um processo leve não está preso a determinado cliente. Em vez disso, ele atende a requisições isoladas de diferentes clientes. O escalonamento de threads pode ser preemptivo ou não preemptivo. Se as threads tiverem permissão para executar até o final (não preemptivos), então seus dados compartilhados não precisam ser protegidos explicitamente. Caso contrário, algum mecanismo de lock explícito precisa ser usado. Alguma forma de esquema de processos leves é essencial se os servidores tiverem de ser escaláveis.

16.9 Um exemplo: redes

Agora, vamos retornar à questão de tradução de nomes levantada na [Seção 16.5.1](#) e examinar sua operação com relação à pilha de protocolos TCP/IP na Internet. Consideraremos o processamento necessário para transferir um pacote entre hosts em diferentes redes Ethernet.

Em uma rede TCP/IP, cada host possui um nome e um endereço IP (ou identificador de host) associado. Essas duas strings precisam ser exclusivas, e para o espaço de nomes poder ser gerenciado elas são segmentadas. O nome é hierárquico (conforme explicamos na [Seção 16.5.1](#)), descrevendo o nome de host e depois a organização com a qual o host está associado. O identificador de host é dividido em um número de rede e um número de host. A proporção da divisão varia, dependendo do tamanho da rede. Quando os administradores da Internet atribuem um número de rede, a instalação com esse número fica livre para atribuir identificadores de host.

O sistema que envia verifica suas tabelas de roteamento para localizar um roteamento para enviar o quadro ao seu modo. Os roteadores utilizam a parte de rede do identificador de host para transferir o pacote da sua rede de origem para a rede de destino. O sistema de destino, então, recebe o pacote. O pacote, então, pode ser uma mensagem completa ou apenas um componente de uma mensagem, com outros pacotes sendo necessários antes de a mensagem poder ser novamente montada e passada à camada TCP/UDP, para transmissão ao processo de destino.

Agora, sabemos como um pacote passa de sua rede de origem para o seu destino. Dentro de uma rede, como um pacote passa do emissor (host ou roteador) para o receptor? Cada dispositivo Ethernet possui um número de byte exclusivo, chamado **endereço MAC (Medium Access Control)**, atribuído a ele para o endereçamento. Dois dispositivos em uma LAN se comunicam apenas com esse número. Se um sistema precisa enviar dados para outro sistema, o software de rede gera um pacote **Address Resolution Protocol (ARP)** contendo o endereço IP do sistema de destino. Esse pacote é transmitido por **broadcast** para todos os outros sistemas nessa rede Ethernet.

Um broadcast utiliza um endereço de rede especial (normalmente, o endereço máximo) para sinalizar que todos os hosts deverão receber e processar o pacote. O broadcast não é reenviado pelos gateways, de modo que somente os sistemas na rede local o recebem. Somente o sistema cujo endereço IP combina com o endereço IP da requisição ARP responde e envia de volta seu endereço MAC ao sistema que iniciou a consulta. Por questão de eficiência, o host coloca em cache o par de endereços IP-MAC em uma tabela interna. As entradas do cache são **envelhecidas**, de modo que, por fim, serão removidas do cache se um acesso a esse sistema não for exigido no tempo indicado. Assim, os hosts removidos de uma rede são **esquecidos**. Para aumentar o desempenho, as entradas do ARP para os hosts mais usados podem ficar rigidamente fixadas no cache ARP.

Quando um dispositivo Ethernet tiver anunciado seu identificador de host e endereço, a comunicação poderá ser iniciada. Um processo pode especificar o nome de um host com o qual irá se comunicar. O software de rede apanha esse nome e determina o número da Internet do destino, usando uma pesquisa de DNS. A mensagem é passada da camada da aplicação, passando pelas camadas de software e indo para a camada de hardware. Na camada de hardware, o pacote (ou pacotes) tem o endereço Ethernet no seu início; um término indica o final do pacote e contém uma **checksum** para detecção dos danos do pacote ([Figura 16.10](#)). O pacote é colocado na rede pelo dispositivo Ethernet. A seção de dados do pacote pode conter alguns ou todos os dados da mensagem original, mas também pode conter alguns dos cabeçalhos de nível superior que compõem a mensagem. Em outras palavras, todas as partes da mensagem original precisam ser enviadas da origem ao destino, e todos os cabeçalhos acima da camada 802.3 (camada de enlace de dados) são incluídos como dados nos pacotes Ethernet.

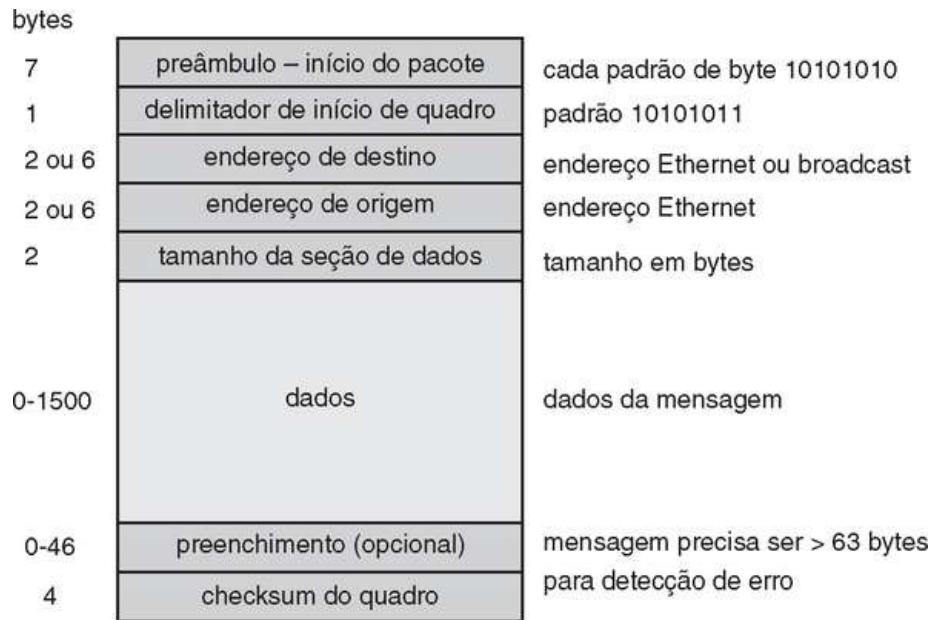


FIGURA 16.10 Um pacote Ethernet.

Se o destino estiver na mesma rede local da origem, o sistema poderá pesquisar em seu cache ARP, encontrar o endereço Ethernet do host e levar o pacote para os fios. O dispositivo Ethernet de destino vê seu endereço no pacote e lê o pacote, passando-o para cima na pilha de protocolos.

Se o sistema de destino estiver em uma rede diferente da rede da origem, o sistema de origem encontra um roteador apropriado em sua rede e envia o pacote para lá. Os roteadores, então, passam o pacote adiante na WAN até ele alcançar sua rede de destino. O roteador que conecta a rede de destino verifica seu cache ARP, encontra o número Ethernet do destino e envia o pacote para esse host. Durante todas essas transferências, o cabeçalho da camada de enlace de dados pode mudar enquanto o endereço Ethernet do próximo roteador na cadeia é utilizado, mas os outros cabeçalhos do pacote permanecem inalterados até que o pacote seja recebido e processado pela pilha de protocolos e finalmente passado para o processo receptor pelo kernel.

16.10 Resumo

Um sistema distribuído é uma coleção de processadores que não compartilham memória ou relógio. Em vez disso, cada processador tem sua própria memória local, e os processadores se comunicam entre si por várias linhas de comunicação, como barramentos de alta velocidade ou linhas telefônicas. Os processadores em um sistema distribuído variam em tamanho e função. Eles podem incluir pequenos microprocessadores, estações de trabalho, minicomputadores e grandes computadores de uso geral.

Os processadores no sistema são conectados por meio de uma rede de comunicação, que pode ser configurada de diversas maneiras. A rede pode estar conectada total ou parcialmente. Ela pode ser uma árvore, uma estrela, um anel ou um barramento de multiacesso. O projeto da rede de comunicação precisa incluir estratégias de roteamento e conexão, e precisa solucionar os problemas de disputa e segurança.

Um sistema distribuído dá ao usuário acesso aos recursos que o sistema provê. O acesso a um recurso compartilhado pode ser fornecido por migração de dados, migração de computação ou migração de processo.

As pilhas de protocolo, conforme especificadas pelos modelos de camadas de rede, trabalham com a mensagem, acrescentando informações a ela, para garantir que ela alcance seu destino. Um sistema de nomes (como o DNS) precisa ser usado para traduzir de um nome de host para endereço de rede, e outro protocolo (como o ARP) pode ser necessário para traduzir o número de rede para um endereço de dispositivo de rede (um endereço Ethernet, por exemplo). Se os sistemas estiverem localizados em redes separadas, roteadores são necessários para passar os pacotes da rede de origem para a rede de destino.

Um sistema distribuído pode sofrer de vários tipos de falha de hardware. Para um sistema distribuído ser tolerante a falhas, ele precisa detectar as falhas de hardware e reconfigurar o sistema. Quando a falha é reparada, o sistema precisa ser reconfigurado novamente.

Exercícios práticos

- 16.1. A maioria das WANs emprega uma topologia apenas parcialmente conectada. Por que isso acontece?
- 16.2. Sob quais circunstâncias uma rede de passagem de tokens é mais eficaz do que uma rede Ethernet?
- 16.3. Por que seria uma má ideia para os gateways passar pacotes de *broadcast* entre as redes? Quais seriam as vantagens de se fazer isso?
- 16.4. Discuta as vantagens e as desvantagens do caching de traduções de nome para os computadores localizados em domínios remotos.
- 16.5. Quais são as vantagens e desvantagens do uso da comutação de circuitos? Para que tipos de aplicações a comutação de circuitos é uma estratégia viável?
- 16.6. Quais são dois problemas notáveis que os projetistas precisam resolver para implementar um sistema transparente à rede?
- 16.7. A migração de processos dentro de uma rede heterogênea normalmente é impossível, dadas as diferenças nas arquiteturas e nos sistemas operacionais. Descreva um método para a migração de processos por diferentes arquiteturas executando:
- a. O mesmo sistema operacional
 - b. Sistemas operacionais diferentes
- 16.8. Para se montar um sistema distribuído robusto, é preciso conhecer os tipos de falhas que podem ocorrer.
- a. Indique três tipos possíveis de falha em um sistema distribuído.
 - b. Especifique quais das entradas na sua lista também se aplicam a um sistema centralizado.
- 16.9. É sempre decisivo saber que a mensagem que você enviou chegou ao seu destino com segurança? Se a sua resposta for *sim*, explique por quê. Se a sua resposta for *não*, dê exemplos apropriados.
- 16.10. Apresente um algoritmo para reconstruir um anel lógico depois que um processo no anel falhar.
- 16.11. Considere um sistema distribuído com duas instalações, A e B. Considere se a instalação A pode distinguir entre o seguinte:
- a. B está parada.
 - b. O enlace entre A e B está interrompido.
 - c. B está extremamente sobrecarregada, e seu tempo de resposta é 100 vezes maior que o normal.
- Quais implicações a sua resposta terá para a recuperação em sistemas distribuídos?

Exercícios

16.12. Qual é a diferença entre migração de computação e migração de processo? Qual é mais fácil de implementar e por quê?

- 16.13. Compare as diversas topologias de rede em termos dos seguintes atributos:
- Confiabilidade
 - Largura de banda disponível para comunicações concorrentes
 - Custo de instalação
 - Balanceamento de carga nas responsabilidades de roteamento

16.14. Embora o modelo ISO de rede especifique sete camadas de funcionalidade, a maioria dos sistemas computadorizados utiliza menos camadas para implementar uma rede. Por que eles usam menos camadas? Quais problemas poderiam ser causados por isso?

16.15. Explique por que a duplicação da velocidade dos sistemas em um segmento Ethernet pode resultar em menor desempenho na rede. Quais mudanças poderiam ajudar a resolver esse problema?

16.16. Quais são as vantagens de usar dispositivos de hardware dedicados para roteadores e gateways? Quais são as desvantagens de usar esses dispositivos em comparação com o uso de computadores de uso geral?

16.17. De que maneiras o uso de um servidor de nomes é melhor do que o uso de tabelas de host estáticas? Quais problemas ou complicações estão associados aos servidores de nome? Quais métodos você poderia usar para descrever a quantidade de tráfego que os servidores de nome geram para satisfazer as requisições de tradução?

16.18. Servidores de nomes são organizados de uma maneira hierárquica. Qual é a finalidade de usar uma organização hierárquica?

16.19. Considere uma camada de rede que sente colisões e retransmite imediatamente na detecção de uma colisão. Quais problemas poderiam surgir com essa estratégia? Como eles poderiam ser retificados?

16.20. As camadas mais baixas do modelo de rede ISO fornecem serviço de datagrama, sem garantias de entrega para as mensagens. Um protocolo da camada de transporte, como o TCP, é usado para fornecer confiabilidade. Discuta as vantagens e desvantagens do suporte à remessa de mensagem confiável na camada mais baixa possível.

16.21. Como o uso de uma estratégia de roteamento dinâmico afeta o comportamento da aplicação? Para que tipo de aplicações é benéfico usar o roteamento virtual em vez do roteamento dinâmico?

16.22. Execute o programa mostrado na Figura 16.5 e determine os endereços IP dos seguintes nomes de host:

- www.wiley.com
- www.cs.yale.edu
- www.apple.com
- www.westminstercollege.edu
- www.ietf.org

16.23. O protocolo HTTP original utiliza TCP/IP como protocolo de rede subjacente. Para cada página, gráfico ou applet, uma sessão TCP separada foi construída, usada e desfeita. Devido ao custo adicional de criar e destruir conexões TCP/IP, problemas de desempenho sobrevieram com esse método de implementação. Seria uma boa alternativa usar UDP em vez de TCP? Quais outras mudanças você poderia fazer para melhorar o desempenho do HTTP?

16.24. Qual é a utilidade de um protocolo de tradução de endereços? Por que é melhor usar esse protocolo do que fazer cada host ler cada pacote para determinar o destino desse pacote? Uma rede de passagem de tokens precisa de tal protocolo? Explique sua resposta.

16.25. Quais são as vantagens e as desvantagens de tornar a rede de computadores transparente ao usuário?

Projetos de programação

Criando um servidor Web

Este projeto consiste em criar e implementar um servidor Web multithreads simples, de acordo com o protocolo HTTP 1.1. Um servidor Web opera escutando uma porta (normalmente, a porta 80), esperando por conexões de clientes. Quando um cliente (um navegador Web) se conecta ao servidor, ele requisita um recurso, como uma página HTML, imagem ou documento, do servidor. Se for feita uma requisição válida do cliente, o servidor retorna o recurso requisitado ao cliente. Os clientes também podem requisitar outros serviços de um servidor Web e até mesmo pedir para colocar um recurso no servidor. Não consideramos esses cenários neste projeto.

Cada solicitação de cliente é ser atendido por um thread separado no servidor. Você pode atender cada solicitação de cliente criando um novo thread ou usando um pool de threads ([Seção 4.5.4](#)). Se optar por usar um pool de threads, você pode usar a API do Java para pools de threads, descritos na [Seção 4.5.4](#) ou o pool de segmentos que você criou no projeto de programação no [Capítulo 6](#).

Para entender melhor a interação entre um cliente e um servidor Web, primeiro examinamos como um cliente faz uma requisição a um servidor e depois consideramos como um servidor responde a uma requisição do cliente. Porém, antes de prosseguirmos, temos que explicar que estamos examinando apenas uma pequena parte do protocolo HTTP. Os que estiverem interessados nos detalhes completos do HTTP deverão ler a RFC para o HTTP 1.1, fornecida na bibliografia.

Cliente para servidor

Quando um cliente se conecta a um servidor Web, ele envia uma requisição usando uma **mensagem de requisição** HTTP. Essa mensagem de requisição se parece com:

```
<linha de requisição>
<cabeçalhos gerais>
<cabeçalhos de requisição>
<linha vazia>
```

A **linha de requisição** consiste em (1) um método HTTP; (2) o nome do recurso solicitado; e (3) o protocolo HTTP utilizado. Os **cabeçalhos gerais** incluem a data e se a conexão TCP deve permanecer aberta ou fechada quando a requisição tiver sido atendida. Os **cabeçalhos de requisição** incluem (1) o nome de host do servidor; (2) o tipo (ou tipos) de documento que o cliente aceitará; e (3) o agente (o navegador Web) que faz a requisição.

Como exemplo, suponha que um cliente (um navegador Safari executando no Mac OS X) requisite o seguinte URL:

<http://localhost/index.html>

A mensagem de requisição aparece na [Figura 16.11](#).

```
GET /index.html HTTP/1.1
Accept: /*
Accept-Language: en-usuário
Accept-Encoding: gzip, deflate
User-Agent: Safari/417.9.2
Connection: keep-alive
Host: localhost
```

FIGURA 16.11 A mensagem de requisição.

Embora o servidor Web totalmente funcional responda a todos os cabeçalhos que recebe, este projeto exige o desmembramento apenas da linha de requisição - *GET /index.html HTTP/1.1*. O método GET especifica que o cliente está requisitando um recurso do servidor; o recurso requisitado é */index.html*. (Os nomes de recurso aparecem após a barra inicial “/”.) Finalmente, o protocolo HTTP utilizado pelo cliente é *HTTP/1.1*. Existem outros métodos especificados no protocolo HTTP, como HEAD, POST e PUT. Para este projeto, você só precisa lidar com o método GET.

Como já dissemos, os nomes de recurso aparecem após a barra inicial “/”. Suponha que o cliente requisite o seguinte URL:

<http://www.wiley.com>

Aqui, a linha de requisição apareceria como *GET/HTTP/1.1*, ou seja, sem nome de recurso após a

barra inicial. Quando isso acontece, um servidor Web é configurado para retornar um documento default. Para este projeto, retorne *index.html* como documento default.

Fornecemos um programa Java *HttpRequestEcho.java* para explicar melhor a mensagem de requisição enviada dos navegadores clientes aos servidores Web. Consulte o arquivo README para obter os detalhes de como executar esse programa.

Servidor para cliente

Quando o servidor tiver recebido uma mensagem de requisição, ele envia de volta uma **mensagem de resposta** HTTP ao cliente. Uma mensagem de resposta se parece com:

```
<linha de status>
<cabeçalhos gerais>
<cabeçalhos de resposta>
<cabeçalhos de entidade>
<linha vazia>
<corpo da mensagem>
```

A **linha de status** é estruturada da seguinte maneira:

HTTP/1.1 <código de status> <frase de motivo>

O primeiro campo da linha de status é a versão HTTP especificada na linha de requisição do cliente. O **código de status** é um código numérico indicando o status da mensagem de resposta. A **frase de motivo** é uma rápida descrição do código de status. Seu servidor Web precisa ter suporte para os três códigos de status a seguir e suas frases de motivo associadas, conforme a [Figura 16.12](#).

Código de status	Frase de motivo
200	OK
400	Bad Request
404	Not Found

FIGURA 16.12 Três códigos de status e suas frases de motivo associadas.

(Em breve, discutiremos qual código de status seu servidor Web retornará ao cliente.)

Os **cabeçalhos gerais** incluem a data e se a conexão deve ser fechada ou permanecer aberta. Os **cabeçalhos de resposta** incluem o nome do servidor (ou seja, um servidor Web *Apache*). **Cabeçalhos de entidade** incluem o tipo de conteúdo do recurso (como html ou jpeg) e o tamanho (em bytes) do recurso sendo retornado.

Fornecemos um programa Java *HttpHeader.java* para explicar melhor a mensagem de resposta enviada de um servidor Web para um navegador cliente. Consulte o arquivo README para obter os detalhes de como executar esse programa.

O servidor Web

Crie um servidor Web que escute na porta 2880 e implemente o comando GET do protocolo HTTP 1.1. Seu servidor escutará conexões do cliente, atendendo cada uma em uma thread separada. Quando uma requisição do cliente for recebida, a thread desmembrará a requisição, lerá o recurso solicitado no servidor, construirá os cabeçalhos HTTP necessários e retornará a mensagem de resposta e o conteúdo do recurso solicitado.

A mensagem de resposta consistirá nos seguintes cabeçalhos:

1. Status line
2. Date
3. Connection: close
4. Server
5. Content-Type
6. Content-Length

Ao construir a linha de status, você é responsável por três códigos de status: 200, 400 e 404. Os códigos devem ser enviados nas seguintes situações:

- 200 é enviado se a requisição for válida e o recurso estiver sendo enviado ao cliente.
- 400 é enviado se o cliente enviou uma requisição que o servidor não consegue entender.
- 404 é enviado se o recurso solicitado não puder ser encontrado no servidor.

A data precisa ser formatada de acordo com um protocolo (RFC 1123). Fornecemos o arquivo Java *DateFormatter.java* para ilustrar como formatar a data enviada no cabeçalho de resposta. O cabeçalho

Connection: close indica que o seu servidor não tem suporte para conexões persistentes e a conexão TCP é fechada quando o servidor responde ao cliente. O valor do cabeçalho do servidor é o nome que você escolheu para o seu servidor Web.

Os tipos de conteúdo que o seu servidor Web precisa aceitar aparecem a seguir:

Tipo de conteúdo	Descrição
text/html	arquivo HTML
image/gif	imagem GIF
image/jpeg	imagem JPEG
text/plain	documento de texto puro

Finalmente, o tamanho do conteúdo é o tamanho em bytes do recurso que você está retornando ao cliente.

Logging

Todas as requisições ao servidor Web precisam ser registradas em um log usando o formato de arquivo de log comum. Uma linha usando esse formato aparece como:

- **Remote host.** O endereço IP do host remoto conectando ao seu servidor Web.
- **Date.** A data da requisição, que aparece entre colchetes ("[]").
- **Request.** A linha de requisição HTTP, que é colocada entre aspas, pois pode conter espaços.
- **Status.** O código de status: 200, 400 ou 404.
- **Bytes.** O tamanho em bytes do recurso retornado ao cliente.

As duas entradas possíveis em um arquivo de log são:

```
10.0.0.2 [Thu Jul 2 17:24:07 EDT 2009]
"GET /screen.jpg HTTP/1.1" 200 82961
```

```
10.0.0.2 [Thu Jul 2 17:24:07 EDT 2009]
"GET / HTTP/1.1" 200 22
```

Configuração da XML

Existem três parâmetros para a configuração do servidor Web:

- O nome e o local do arquivo de log.
- O nome do servidor Web.
- O local dos documentos servidos pelo servidor Web.

Esses parâmetros devem ser especificados em um arquivo XML. No início, seu servidor lerá os parâmetros de configuração no arquivo config.xml. O exemplo a seguir em Java ilustra como obter o valor dos três parâmetros de configuração especificados em config.xml:

```
Configuration configurator = new
Configuration("config.xml");
.
.
.
configurator.getLogFile();
configurator.getServerName();
configurator.getDocBase();
```

O método `getLogFile()` retorna o nome do arquivo de log onde o servidor deve registrar as requisições, e o método `getServerName()` fornece o nome do servidor. O método `getDocBase()` é usado para obter o nome do diretório do qual as páginas Web devem ser servidas. Por exemplo, se a pasta onde um site é armazenado se chamar

C:\web-server\docs
todos os documentos serão apanhados abaixo desse diretório e `getDocBase()` retornará esse valor.

O URL

http://127.0.0.1/index.html
indica que o recurso index.html está armazenado como

C:\web-server\docs\index.html.

O URL

http://127.0.0.1/pictures/jvm.gif

refere-se ao arquivo

C:\web-server\docs\pictures\jvm.gif

e assim por diante.

Fornecemos um arquivo XML de exemplo, além de três programas Java ilustrando como obter os valores especificados no arquivo XML. Esses arquivos são: (1) config.xml, (2) Configuration.java, (3) ConfigurationException.java e (4) Test.java, e estão disponíveis on-line.

Notas bibliográficas

Tanenbaum [2003], Stallings [2000a] e Kurose e Ross [2003] proveem introduções gerais sobre redes de computadores. Williams [2001] provê cobertura de redes de computadores de um ponto de vista da arquitetura do computador.

A Internet e seus protocolos foram descritos em Comer [1999] e Comer [2000]. A cobertura do pequeno modelo TCP/IP pode ser encontrada em Stevens [1994] e Stevens [1995]. A programação em rede no UNIX foi descrita por completo em Stevens [1997] e Stevens [1998].

As discussões referentes a estruturas do sistema operacional distribuído foram oferecidas por Coulouris e outros [2001] e Tanenbaum e van Steen [2002].

O balanceamento de carga e o compartilhamento de carga foram discutidos por Harchol-Balter e Downey [1997] e Vee e Hsu [2000]. Harish e Owens [1999] descreveram os servidores de DNS de balanceamento de carga. A migração de processos foi discutida por Jul e outros [1988], Douglis e Ousterhout [1991], Han e Ghosh [1998] e Milojicic e outros [2000]. As questões relacionadas com uma máquina virtual distribuída para sistemas distribuídos foram examinadas em Sirer e outros [1999].

CAPÍTULO 17

Sistemas de arquivos distribuídos

No capítulo anterior, discutimos a construção da rede e os protocolos de baixo nível necessários para a transferência de mensagens entre os sistemas. Agora, discutimos um uso dessa infraestrutura. Um **sistema de arquivos distribuído** (**Distributed File System - DFS**) é uma implementação distribuída do modelo clássico de tempo compartilhado de um sistema de arquivos, no qual vários usuários compartilham arquivos e recursos de armazenamento ([Capítulo 11](#)). A finalidade de um DFS é dar suporte ao mesmo tipo de compartilhamento quando os arquivos estão fisicamente dispersos entre as instalações de um sistema distribuído.

Neste capítulo, discutimos as maneiras como um DFS pode ser projetado e implementado. Primeiro, discutimos os conceitos comuns sobre os quais se baseiam. Depois, ilustramos nossos conceitos examinando um DFS influente – o **Andrew File System (AFS)**.

OBJETIVOS DO CAPÍTULO

- Explicar o mecanismo de nomes que fornece transparência e independência de local.
- Descrever os diversos métodos para acessar arquivos distribuídos.
- Comparar os servidores de arquivo distribuídos com e sem estado.
- Mostrar como a replicação de arquivos em diferentes máquinas em um sistema de arquivos distribuídos é uma redundância útil para melhorar a disponibilidade.
- Introduzir o sistema de arquivos Andrew (AFS) como um exemplo de um sistema de arquivos distribuídos.

17.1 Aspectos básicos

Como vimos no capítulo anterior, um sistema distribuído é uma coleção de máquinas livremente acopladas e interconectadas por uma rede de comunicação. Esses computadores podem compartilhar arquivos fisicamente dispersos usando um sistema de arquivos distribuídos (DFS). Neste capítulo, usamos o termo *DFS* para indicar sistemas de arquivos distribuídos em geral, e não o produto comercial Transarc DFS. Referimo-nos a este último como *Transarc DFS*. Além disso, NFS refere-se ao NFS versão 3, a menos que seja indicado de outra maneira.

Para explicar a estrutura de um DFS, precisamos definir os termos *serviço*, *servidor* e *cliente*. Um **serviço** é uma entidade de software executada em uma ou mais máquinas e fornecendo um tipo específico de função a clientes. Um **servidor** é um software de serviço executado em uma única máquina. Um **cliente** é um processo que pode chamar um serviço usando um conjunto de operações que forma sua **interface de cliente**. Às vezes, uma interface de nível mais baixo é definida para a interação real entre as máquinas; essa é a **interface entre máquinas**.

Usando essa terminologia, dizemos que um sistema de arquivos provê serviços de arquivo aos clientes. Uma interface de cliente para um serviço de arquivo é formada por um conjunto de operações de arquivo primitivas, como criar um arquivo, excluir um arquivo, ler de um arquivo e escrever em um arquivo. O componente de hardware principal que um servidor de arquivos controla é um conjunto de dispositivos locais de armazenamento secundário (normalmente, discos magnéticos) nos quais os arquivos são armazenados e dos quais são apanhados de acordo com as requisições do cliente.

Um DFS é um sistema de arquivos cujos clientes, servidores e dispositivos de armazenamento estão dispersos entre as máquinas de um sistema distribuído. Como consequência, a atividade do serviço precisa ser executada pela rede e, em vez de um único repositório de dados centralizado, o sistema possui vários dispositivos de armazenamento independentes. Como você verá, a configuração e a implementação concreta de um DFS podem variar de um sistema a outro. Em algumas configurações, os servidores são executados em máquinas dedicadas; em outros, uma máquina pode ser um servidor e um cliente. Um DFS pode ser implementado como parte de um sistema operacional distribuído ou, como alternativa, por uma camada de software cuja tarefa é gerenciar a comunicação entre os sistemas operacionais convencionais e os sistemas de arquivos. Os recursos distintos de um DFS são a multiplicidade e a autonomia dos clientes e servidores no sistema.

O ideal é que um DFS pareça aos seus clientes um sistema de arquivos convencional, centralizado. A multiplicidade e a dispersão de seus servidores e dispositivos de armazenamento devem se tornar invisível, ou seja, a interface de cliente de um DFS não deverá distinguir entre arquivos locais e remotos. Fica a cargo do DFS localizar os arquivos e arrumar o transporte dos dados. Um DFS **transparente** facilita a mobilidade do usuário trazendo o ambiente do usuário, ou seja, o diretório principal para onde quer que um usuário efetue o login.

A medida de desempenho mais importante de um DFS é a quantidade de tempo necessária para satisfazer as requisições de serviço. Nos sistemas convencionais, esse tempo consiste em tempo de acesso ao disco e uma pequena quantidade de tempo de processamento de CPU. Contudo, em um DFS, um acesso remoto tem o custo adicional atribuído à estrutura distribuída. Esse custo adicional inclui o tempo para entregar a requisição a um servidor, bem como o tempo para enviar a resposta, pela rede, de volta ao cliente. Para cada direção, além da transferência de informações, existe o custo adicional de CPU para executar o software do protocolo de comunicação. O desempenho de um DFS pode ser visto como outra dimensão da transparência do DFS, ou seja, o desempenho de um DFS ideal seria comparável ao de um sistema de arquivos convencional.

O fato de um DFS gerenciar um conjunto de dispositivos de armazenamento dispersos é o fator-chave de distinção do DFS. O espaço de armazenamento geral gerenciado por um DFS é composto de espaços de armazenamento menores, diferentes e localizados remotamente. Em geral, esses espaços de armazenamento constituintes correspondem a um conjunto de arquivos. Uma **unidade componente** é o menor conjunto de arquivos que pode ser armazenado em uma única máquina, independentemente de outras unidades. Todos os arquivos pertencentes à mesma unidade componente precisam residir no mesmo local.

17.2 Nomeação e transparência

Nomeação é o mapeamento entre objetos lógicos e físicos. Por exemplo, os usuários lidam com objetos de dados lógicos representados por nomes de arquivo, enquanto o sistema manipula os blocos de dados físicos, armazenados nas trilhas do disco. Normalmente, um usuário se refere a um arquivo pelo nome textual, associado a um identificador numérico de nível inferior que, por sua vez, é associado a blocos de disco. O mapeamento multinível provê aos usuários uma abstração de um arquivo que esconde detalhes de como e onde no disco o arquivo é armazenado.

Em um DFS transparente, uma nova dimensão é acrescentada à abstração: a de esconder onde o arquivo está localizado na rede. Em um sistema de arquivos convencional, o intervalo de mapeamento de nomeação é um endereço dentro de um disco. Em um DFS, esse intervalo é expandido para incluir a máquina específica em cujo disco o arquivo está armazenado. Seguindo um pouco à frente com o conceito de tratar arquivos como abstrações, chegamos à possibilidade de **replicação de arquivos**. Dado um nome de arquivo, o mapeamento retorna um conjunto de locais das réplicas desse arquivo. Nessa abstração, a existência de várias cópias e seu local é escondida.

17.2.1 Estruturas de nomeação

Precisamos diferenciar duas noções relacionadas com relação aos mapeamentos de nomes em um DFS:

1. **Transparência de local.** O nome de um arquivo não revela qualquer informação relativa ao local de armazenamento físico do arquivo.
2. **Independência de local.** O nome de um arquivo não precisa ser modificado quando o local de armazenamento físico do arquivo mudar.

As duas definições são relativas no nível de nomeação discutido anteriormente, pois os arquivos possuem diferentes nomes em diferentes níveis (ou seja, os nomes textuais no nível do usuário e os identificadores numéricos no nível do sistema). Um esquema de nomeação independente de local é um mapeamento dinâmico, pois pode mapear o mesmo nome de arquivo a diferentes locais em diferentes momentos. Portanto, a independência de local é uma propriedade mais forte do que a transparência de local.

Na prática, a maioria dos DFSs atuais provê um mapeamento estático e transparente ao local para os nomes no nível do usuário. Contudo, esses sistemas não admitem **migração de arquivos**, ou seja, mudar o local de um arquivo automaticamente é impossível. Logo, a noção de independência de local é irrelevante para esses sistemas. Os arquivos são associados permanentemente a um conjunto específico de blocos de disco. Arquivos e discos podem ser movidos entre as máquinas manualmente, mas a migração de arquivo implica uma ação automática, iniciada pelo sistema operacional. Somente o AFS e alguns sistemas de arquivos experimentais admitem independência de local e mobilidade de arquivos. O AFS admite a mobilidade de arquivos para fins administrativos. Um protocolo provê migração das unidades componentes do AFS para satisfazer as requisições de alto nível do usuário, sem mudar os nomes no nível do usuário ou os nomes reais dos arquivos correspondentes.

Alguns aspectos podem diferenciar ainda mais a independência de local e a transparência do local estático:

- A separação entre dados e local, conforme exibida pela independência do local, provê melhor abstração para arquivos. Um nome de arquivo deverá indicar os atributos mais significativos do arquivo, que são seu conteúdo em vez do seu local. Os arquivos independentes de local podem ser vistos como contêineres de dados lógicos, que não estão ligados a um local de armazenamento específico. Se apenas a transparência de local estática for admitida, o nome de arquivo ainda indica um conjunto específico, embora oculto, de blocos físicos no disco.
- A transparência de local estático provê aos usuários um modo conveniente de compartilhar dados. Os usuários podem compartilhar arquivos remotos nomeando os arquivos de maneira transparente ao local, como se os arquivos fossem locais. Apesar disso, o compartilhamento do espaço de armazenamento é problemático, pois os nomes lógicos ainda são ligados estaticamente aos dispositivos de armazenamento físico. A independência de local promove o compartilhamento do próprio espaço de armazenamento, bem como os objetos de dados. Quando os arquivos podem ser mobilizados, o espaço de armazenamento geral, no nível do sistema, se parece com um único recurso virtual. Um benefício possível de tal visão é a capacidade de balancear a utilização de discos por meio do sistema.
- A independência de local separa a hierarquia de nomes da hierarquia de dispositivos de armazenamento e da estrutura entre computadores. Por outro lado, se a transparência de local estática for usada (embora os nomes sejam transparentes), podemos facilmente expor a correspondência entre as unidades e máquinas componentes. As máquinas são configuradas em um padrão semelhante à estrutura de nomeação. Essa configuração pode restringir a arquitetura do sistema desnecessariamente e entrar em conflito com outras considerações. Um servidor encarregado de um diretório raiz é um exemplo de uma estrutura ditada pela hierarquia de

nomes e contradiz as orientações de descentralização.

Quando a separação de nome e local tiver sido completa, os clientes poderão acessar arquivos residentes em sistemas servidores remotos. Na verdade, esses clientes podem não ter **discos** e contar com os servidores para prover todos os arquivos, incluindo o kernel do sistema operacional. Entretanto, protocolos especiais são necessários para a sequência de boot. Considere o problema de apanhar o kernel para uma estação de trabalho sem disco. A estação de trabalho sem disco não possui kernel, de modo que não pode usar o código do DFS para apanhar o kernel. Em vez disso, é chamado um protocolo de boot especial, armazenado na memória somente de leitura (ROM) no cliente. Ele ativa a rede e apanha somente um arquivo especial (o código do kernel ou de boot) em um local fixo. Quando o kernel é copiado pela rede e carregado na memória, seu DFS faz tornar todos os outros arquivos do sistema operacional disponíveis. As vantagens de clientes sem disco são muitas, incluindo menor custo (porque a máquina cliente não precisa de discos) e maior conveniência (quando houver uma atualização do sistema operacional, somente o servidor terá de ser modificado). As desvantagens são a maior complexidade dos protocolos de boot e a perda de desempenho resultante do uso de uma rede em vez de um disco local.

A tendência atual é usar clientes com discos locais e servidores de arquivos remotos. Os sistemas operacionais e o software de rede são armazenados localmente; os sistemas de arquivos contendo dados do usuário – e possivelmente aplicações – são armazenados em sistemas de arquivos remotos. Alguns sistemas clientes podem armazenar aplicações usadas comumente, como processadores de textos e navegadores Web, também no sistema de arquivos local. Outras aplicações, menos utilizadas, podem ser **empurradas** do servidor de arquivos remoto para o cliente sob demanda. O motivo principal para fornecer aos clientes sistemas de arquivos locais em vez de sistemas sem disco puros é que as unidades de disco estão aumentando em capacidade e diminuindo em custo, com novas gerações aparecendo a cada ano. O mesmo não pode ser dito das redes, que evoluem há apenas alguns anos. Em geral, os sistemas estão crescendo mais rapidamente do que as redes, de modo que um trabalho extra é necessário para limitar o acesso à rede para melhorar o throughput do sistema.

17.2.2 Esquemas de nomeação

Existem três técnicas principais para os esquemas de nomeação em um DFS. Na técnica mais simples, o arquivo é identificado por alguma combinação de seu nome de host e nome local, o que garante um nome exclusivo no nível do sistema. No Ibis, por exemplo, um arquivo é identificado exclusivamente pelo nome *host:nome-local*, onde *nome-local* é um caminho tipo UNIX. Esse esquema de nomeação não é transparente ao local nem independente do local. Apesar disso, as operações com nome de arquivo podem ser usadas para arquivos locais e remotos. O DFS é estruturado como uma coleção de unidades componentes, cada qual um sistema de arquivos convencionais inteiros. Nessa primeira técnica, as unidades componentes permanecem isoladas, embora haja meios para nos referirmos a um arquivo remoto. Não consideraremos mais esse esquema neste texto.

A segunda técnica foi popularizada pelo Network File System (NFS) da Sun. O NFS é o componente do sistema de arquivos do ONC+, um pacote de redes com o suporte de muitos fornecedores UNIX. O NFS provê meios de conectar diretórios remotos a diretórios locais, dando, assim, a aparência de uma árvore de diretórios coerente. Antigas versões do NFS permitiam apenas que diretórios remotos previamente montados fossem acessados de forma transparente. Com o advento do recurso **automount**, as montagens são feitas por demanda, com base em uma tabela de pontos de montagem e nomes da estrutura de arquivos. Os componentes são integrados para dar suporte ao compartilhamento transparente, embora essa integração seja limitada e não uniforme, pois cada máquina pode conectar diretórios remotos diferentes da sua árvore. A estrutura resultante é versátil.

Podemos alcançar a integração total dos sistemas de arquivos componentes usando a terceira técnica. Aqui, uma única estrutura global de nomes cobre todos os arquivos no sistema. O ideal é que a estrutura do sistema de arquivos composto seja a mesma que a estrutura de um sistema de arquivos convencional. No entanto, na prática, os muitos arquivos especiais (por exemplo, arquivos de dispositivo UNIX e diretórios binários específicos da máquina) tornam esse objetivo difícil de alcançar.

Para avaliar as estruturas de nomes, olhamos sua **complexidade administrativa**. A estrutura mais complexa e mais difícil de manter é a estrutura do NFS. Como qualquer diretório remoto pode estar conectado a qualquer lugar na árvore de diretórios local, a hierarquia resultante pode ser bastante desestruturada. Se um servidor se tornar indisponível, qualquer conjunto de diretórios em diferentes máquinas se tornará indisponível. Além disso, um mecanismo de aprovação separado controla qual máquina tem permissão para conectar qual diretório à sua árvore. Assim, um usuário poderia ter permissão para acessar uma árvore de diretório remota em um cliente, mas ter o acesso negado em outro cliente.

17.2.3 Técnicas de implementação

A implementação da nomeação transparente requer uma provisão para o mapeamento de um nome de arquivo ao local associado. Para manter esse mapeamento de forma controlada, temos de agregar conjuntos de arquivos a unidades componentes e prover o mapeamento com base em cada unidade componente, e não com base em um único arquivo. Essa agregação também serve a finalidades administrativas. Sistemas tipo UNIX utilizam a árvore de diretórios hierárquica para prover mapeamento de nome e local e agrregar arquivos recursivamente aos diretórios.

Para melhorar a disponibilidade das informações cruciais do mapeamento, podemos usar métodos como replicação, caching local ou ambos. Conforme observamos, a independência de local significa que o mapeamento muda com o tempo; logo, a replicação do mapeamento torna impossível uma atualização simples, porém coerente, dessa informação. Uma técnica para contornar esse obstáculo é introduzir **identificadores de arquivo independentes do local**, em baixo nível. Os nomes de arquivo textuais são associados a identificadores de arquivo de nível mais baixo, que indicam a que unidade componente o arquivo pertence. Esses identificadores ainda são independentes de local. Eles podem ser replicados e colocados em cache livremente, sem serem invalidados pela migração de unidades componentes. O preço inevitável é um segundo nível de mapeamento, que mapeia unidades componentes a locais e precisa de um mecanismo de atualização simples, porém coerente. A implementação de árvores de diretório tipo UNIX usando esses identificadores em baixo nível, independentes de local, torna a hierarquia inteira invariante sob a migração da unidade componente. O único aspecto que muda é o mapeamento de local da unidade componente.

Uma forma comum de implementar esses identificadores em baixo nível é usar nomes estruturados. Esses nomes são sequências de bits que possuem duas partes. A primeira parte identifica a unidade componente à qual o arquivo pertence; a segunda parte identifica o arquivo em particular dentro da unidade. Variantes com mais partes são possíveis. A invariante dos nomes estruturados, porém, é que as partes individuais do nome são exclusivas em todos os momentos apenas dentro do contexto do restante das partes. Podemos obter a exclusividade em todos os momentos com o cuidado de não reutilizar um nome já usado, acrescentando mais bits suficientes (esse método é usado no AFS) ou usando uma estampa de tempo como parte do nome (como é feito no Apollo Domain). Outra maneira de ver esse processo é que estamos apanhando um sistema transparente ao local, como o Ibis, e acrescentando outro nível de abstração, para produzir um esquema de nomeação independente de local.

Agregar arquivos em unidades componentes e usar identificadores de arquivos independentes de local em nível mais baixo são técnicas exemplificadas no AFS.

17.3 Acesso a arquivo remoto

Considere um usuário que requisita acesso a um arquivo remoto. O servidor que está armazenando o arquivo foi localizado pelo esquema de nomeação, e agora é preciso ocorrer a transferência de dados real.

Um modo de conseguir essa transferência é por meio de um **mecanismo de serviço remoto**, pelo qual as requisições para acessos são entregues ao servidor, a máquina do servidor realiza os acessos e seus resultados são encaminhados de volta ao usuário. Uma das maneiras mais comuns de implementar o serviço remoto é o paradigma Remote Procedure Call (RPC), que discutimos no [Capítulo 3](#). Existe uma analogia direta entre os métodos de acesso ao disco nos sistemas de arquivos convencionais e o método de serviço remoto em um DFS: o uso de um método de serviço remoto é semelhante à realização de um acesso ao disco para cada requisição de acesso.

Para garantir um desempenho razoável de um mecanismo de serviço remoto, podemos usar uma forma de caching. Nos sistemas de arquivos convencionais, a razão para o uso de caching é reduzir a E/S em disco (aumentando assim o desempenho), enquanto nos DFSs o objetivo é reduzir o tráfego de rede e a E/S de disco. A seguir, discutimos a implementação do caching em um DFS e o comparamos com o paradigma básico de serviço remoto.

17.3.1 Esquema básico de caching

O conceito de caching é simples. Se os dados necessários para satisfazer a requisição de acesso ainda não estiverem no cache, então uma cópia desses dados é trazida do servidor para o sistema cliente. Os acessos são realizados na cópia em cache. A ideia é reter em cache os blocos de disco acessados recentemente, para os acessos repetidos à mesma informação poderem ser tratados no local, sem tráfego de rede adicional. Uma política de substituição (por exemplo, o algoritmo LRU) mantém o tamanho do cache limitado. Não existe uma correspondência direta entre os acessos e o tráfego para o servidor. Os arquivos ainda são identificados com uma cópia mestra residindo na máquina servidora, mas cópias (ou partes) do arquivo estão espalhadas em diferentes caches. Quando uma cópia em cache é modificada, as mudanças precisam ser refletidas na cópia mestra, para preservar a semântica de consistência relevante. O problema de manter as cópias em cache consistentes com o arquivo-mestre é o **problema da consistência de cache**, discutido na [Seção 17.3.4](#). O caching do DFS poderia ser chamado **memória virtual de rede**: ele atua de forma semelhante à memória virtual paginada por demanda, exceto que o armazenamento de apoio normalmente não é um disco local, mas um servidor remoto. O NFS permite que o espaço de swap seja montado remotamente, de modo que realmente possa implementar a memória virtual por uma rede, embora disso resulte uma penalidade de desempenho.

A granularidade dos dados em cache em um DFS pode variar desde blocos de um arquivo a um arquivo inteiro. Normalmente, mais dados são colocados em cache do que o necessário para satisfazer a um único acesso, para muitos acessos poderem ser atendidos pelos dados em cache. Esse procedimento é muito semelhante à leitura antecipada do disco ([Seção 11.6.2](#)). O AFS coloca os arquivos no cache em grandes pedaços (64 KB). Os outros sistemas discutidos neste capítulo admitem o caching de blocos individuais controlados pela demanda do cliente. Aumentar a unidade de caching aumenta a taxa de acertos, mas também aumenta a penalidade por falha, pois cada falha exige a transferência de mais dados. Isso também aumenta o potencial para problemas de consistência. A seleção da unidade de caching envolve a consideração de parâmetros como a unidade de transferência de rede e a unidade de serviço do protocolo RPC (no caso de um protocolo RPC ser utilizado). A unidade de transferência de rede (para Ethernet, um pacote) tem cerca de 1,5 KB, de modo que unidades maiores de dados em cache precisam ser desmontadas para entrega e remontadas no recebimento.

O tamanho do bloco e o tamanho total do cache obviamente são importantes para os esquemas de caching de bloco. Nos sistemas tipo UNIX, os tamanhos de bloco comuns são 4 KB e 8 KB. Para caches grandes (mais de 1 MB), grandes tamanhos de bloco (mais de 8 KB) são benéficos. Para caches menores, grandes tamanhos de bloco são menos benéficos, pois resultam em menos blocos no cache e uma taxa de acertos menor.

17.3.2 Localização do cache

Onde os dados em cache deverão ser armazenados? Os caches de disco possuem uma vantagem clara em relação ao cache da memória principal: são confiáveis. As modificações para os dados em cache são perdidas em uma falha se o cache for mantido na memória volátil. Além do mais, se os dados em cache forem mantidos no disco, eles ainda existem durante a recuperação e não é preciso buscá-los novamente. Entretanto, os caches da memória principal possuem diversas vantagens próprias:

- Os caches da memória principal permitem que as estações de trabalho sejam unidades sem disco.
- Os dados podem ser acessados mais rapidamente a partir de um cache na memória principal do

que de um disco.

- A tecnologia evolui em direção de memórias maiores e mais baratas. É previsto que o aumento alcançado na velocidade de desempenho ultrapasse as vantagens dos caches de disco.
- Os caches do servidor (usados para agilizar a E/S do disco) estarão na memória principal independentemente de onde os caches do usuário estejam localizados; se também usarmos caches da memória principal na máquina do usuário, poderemos montar um único mecanismo de caching para uso por servidores e usuários.

Muitas implementações de acesso remoto podem ser consideradas híbridos de caching e serviço remoto. No NFS, por exemplo, a implementação é baseada no serviço remoto, mas é aumentada com o caching de memória no cliente e no servidor, para melhorar o desempenho. De maneira similar, a implementação do Sprite é baseada no caching, mas, sob certas circunstâncias, um método de serviço remoto é adotado. Assim, para avaliar os dois métodos, avaliamos até que ponto qualquer um deles é enfatizado.

O protocolo NFS e a maioria das implementações não proveem caching de disco. As implementações recentes do NFS no Solaris (Solaris 2.6 em diante) incluem uma opção de caching de disco no cliente, o sistema de arquivos **cachefs**. Quando o cliente NFS lê blocos de um arquivo a partir do servidor, ele os coloca no cache da memória e também no disco. Se a cópia da memória for esvaziada, ou mesmo se o sistema reiniciar, o cache de disco é referenciado. Se um bloco necessário não estiver na memória ou no cache de disco cachefs, uma RPC é enviada ao servidor para apanhar o bloco, e ele é armazenado no cache de disco e também no cache de memória, para uso do cliente.

17.3.3 Política de atualização de cache

A política utilizada para escrever blocos de dados modificados na cópia mestra do servidor tem um efeito crítico sobre o desempenho e a confiabilidade do sistema. A política mais simples é escrever dados no disco assim que são colocados em qualquer cache. A vantagem de uma **política de escrita direta (write-through policy)** é a confiabilidade: pouca informação é perdida quando um sistema cliente falha. No entanto, essa política exige que cada acesso de escrita espere até a informação ser enviada ao servidor, de modo que gera um desempenho de escrita fraco. O caching com escrita direta é equivalente ao uso do serviço remoto para os acessos de escrita e a exploração do caching apenas para os acessos de leitura.

Uma alternativa é a **política de escrita adiada (delayed-write policy)**, também conhecida como **caching write-back**, na qual adiamos as atualizações à cópia mestra. As modificações são escritas no cache e depois são transferidas para o servidor, em outro momento. Essa política possui duas vantagens em relação à escrita direta. Na primeira, como as escritas são feitas ao cache, os acessos para escrita são concluídos muito mais rapidamente. Na segunda, os dados podem ser modificados antes de serem armazenados, quando somente a última atualização precisará ser escrita. Infelizmente, os esquemas de escrita adiada introduzem problemas de confiabilidade, pois os dados não escritos são perdidos sempre que uma máquina do usuário falhar.

Algumas variações da política de escrita adiada diferem no momento em que os blocos de dados modificados são transferidos para o servidor. Uma alternativa é transferir um bloco quando ele está para ser ejetado do cache do cliente. Essa opção pode resultar em um bom desempenho, mas alguns blocos podem residir no cache do cliente por muito tempo antes de serem escritos no servidor. Um meio-termo entre essa alternativa e a política de escrita direta é varrer o cache em intervalos regulares e esvaziar os blocos que foram modificados desde a varredura mais recente, assim como o UNIX varre seu cache local. O Sprite utiliza essa política com um intervalo de 30 segundos. O NFS também utiliza essa política para os dados de arquivo, mas quando uma escrita é emitida ao servidor durante o esvaziamento do cache a escrita precisa alcançar o disco do servidor antes de ser considerada completa. O NFS trata os metadados (dados de diretório e dados de atributo de arquivo) de forma diferente. Quaisquer mudanças nos metadados são emitidas de forma síncrona ao servidor. Assim, a perda de arquivos e a adulteração da estrutura de diretório são evitadas quando um cliente ou servidor falha.

Para o NFS com cachefs, as escritas também são feitas na área de cache de disco local quando são escritas no servidor, para manter todas as cópias consistentes. Assim, o NFS com cachefs melhora o desempenho em relação ao NFS-padrão em uma requisição de leitura com uma taxa de acertos cachefs, mas diminui o desempenho para requisições de leitura ou escrita com uma falha de cache. Assim como em todos os caches, é vital haver uma alta taxa de acertos do cache para o ganho de desempenho. A [Figura 17.1](#) mostra como cachefs utiliza caches write-through e write-back.

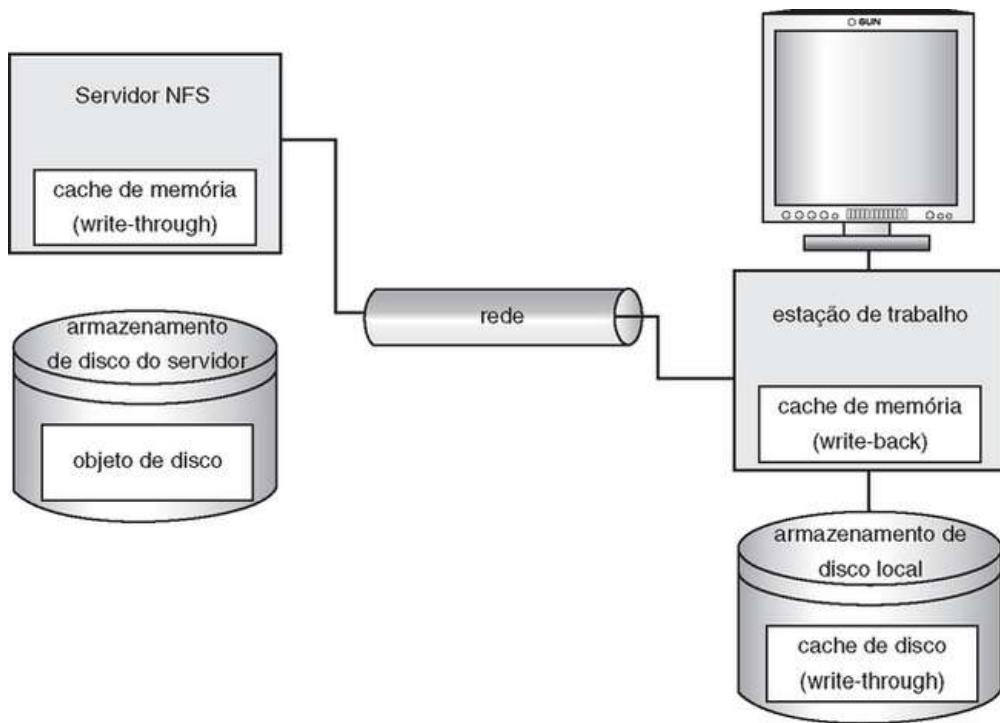


FIGURA 17.1 Cachês e seu uso do caching.

Outra variação da escrita adiada é escrever os dados no servidor quando o arquivo for fechado. Essa **política de escrita no fechamento (write-on-close policy)** é usada no AFS. No caso de arquivos abertos por curtos períodos ou raramente modificados, essa política não reduz o tráfego da rede de forma significativa. Além disso, a política de escrita no fechamento exige um atraso no processo de fechamento enquanto o arquivo está sendo escrito, o que reduz as vantagens de desempenho das escritas adiadas. As vantagens no desempenho dessa política em relação à escrita adiada com esvaziamento mais frequente são aparentes para os arquivos que ficam abertos por longos períodos e são modificados com frequência.

17.3.4 Consistência

Uma máquina cliente tem pela frente o problema de decidir se uma cópia dos dados no cache local é consistente com a cópia mestra (e, portanto, pode ser usada). Se a máquina cliente determinar que seus dados em cache estão desatualizados, os acessos não podem mais ser atendidos por esses dados em cache. Uma cópia atualizada dos dados precisa ser colocada em cache. Existem duas técnicas para verificar a validade dos dados em cache:

1. **Enfoque Client Initiated (iniciado no cliente).** O cliente inicia uma verificação de validade em que contata o servidor e verifica se os dados lógicos são consistentes com a cópia mestra. A frequência da verificação de validade é o ponto crucial dessa técnica e determina a semântica de consistência resultante. Ela pode variar desde uma verificação antes de cada acesso até uma verificação apenas no primeiro acesso a um arquivo (basicamente, na abertura do arquivo). Cada acesso acoplado a uma verificação de validade é adiado, em comparação com um acesso atendido imediatamente pelo cache. Como alternativa, uma verificação pode ser iniciada a cada intervalo de tempo fixo. Dependendo de sua frequência, a verificação de validade pode sobrecarregar a rede e o servidor.
2. **Enfoque Server Initiated (iniciado no servidor).** O servidor registra, para cada cliente, os arquivos (ou partes de arquivos) mantidos em cache. Quando o servidor detecta uma inconsistência em potencial, ele precisa realizar alguma ação. Um potencial para inconsistência ocorre quando dois clientes diferentes, com modos em conflito, colocam um arquivo em cache. Se a semântica do UNIX ([Seção 10.5.3](#)) estiver sendo implementada, podemos resolver a inconsistência em potencial fazendo o servidor desempenhar um papel ativo. O servidor precisa ser notificado sempre que um arquivo for aberto, e o modo intencionado (modo de leitura ou escrita) precisa ser indicado para cada abertura. Supondo haver tal notificação, o servidor pode atuar quando detectar que um arquivo foi aberto simultaneamente em modos em conflito, desativando o caching para esse arquivo em particular. Na realidade, a desativação do caching resulta na troca para um modo de operação de serviço remoto.

17.3.5 Comparação entre caching e serviços remotos

Basicamente, a opção entre caching e serviço remoto é uma questão de aumentar o desempenho e

diminuir a simplicidade. Avaliamos essa questão listando as vantagens e desvantagens dos dois métodos:

- O cache local pode tratar de uma quantidade substancial dos acessos remotos de forma eficiente quando o caching for usado. Aproveitar a localidade nos padrões de acesso a arquivo torna o caching ainda mais atraente. Assim, a maioria dos acessos remotos será atendida tão rapidamente quanto os locais. Além do mais, os servidores são contatados apenas ocasionalmente, e não para cada acesso. Como consequência, a carga do servidor e o tráfego da rede são reduzidos, e o potencial para escalabilidade é aumentado. Por outro lado, cada acesso remoto é tratado pela rede quando o método de serviço remoto é utilizado. A penalidade no tráfego da rede, carga do servidor e desempenho é óbvia.
- O custo adicional total da rede é menor para a transmissão de grandes trechos de dados (como é feito no caching) do que para a transmissão de séries de respostas a requisições específicas (como no método de serviço remoto). Além do mais, as rotinas de acesso ao disco no servidor podem ser mais bem otimizadas se for conhecido que as requisições são sempre para segmentos de dados grandes e contíguos, em vez de blocos de disco aleatórios.
- O problema de consistência do cache é a principal desvantagem do caching. Nos padrões de acesso que exibem escritas pouco frequentes, o caching é superior. Contudo, quando as escritas são frequentes, os mecanismos empregados para contornar o problema de consistência incorrem em grande custo adicional em termos de desempenho, tráfego de rede e carga do servidor.
- Para o caching occasionar um benefício, a execução deve ser executada em máquinas que possuem discos locais ou grandes memórias principais. O acesso remoto em máquinas sem disco, com pequena memória, deve ser feito por meio do método de serviço remoto.
- No caching, como os dados são transferidos em massa entre o servidor e o cliente, e não em resposta às necessidades específicas de uma operação de arquivo, a interface inferior entre máquinas é diferente da interface com o usuário superior. O paradigma do serviço remoto, por outro lado, é apenas uma extensão da interface do sistema de arquivos local pela rede. Assim, a interface entre máquinas espelha a interface do sistema de arquivos do usuário local.

17.4 Serviço Stateful e serviço Stateless

Existem duas técnicas para informações no servidor quando o cliente acessa arquivos remotos: ou o servidor acompanha cada arquivo que está sendo acessado por cada cliente ou ele provê blocos à medida que são requisitados pelo cliente, sem conhecimento de como esses blocos foram usados. No primeiro caso, o serviço fornecido é *com estado(stateful)*; no segundo, ele é *sem estado(stateless)*.

Vejamos, em seguida, o cenário típico envolvendo um **stateful file service (serviço de arquivo com estado)**. Um cliente precisa realizar uma operação `open()` sobre um arquivo antes de acessá-lo. O servidor apanha informações sobre o arquivo a partir do seu disco, armazena-as em sua memória e provê ao cliente um identificador de conexão exclusivo ao cliente e arquivo aberto. (Em termos de UNIX, o servidor apanha o inode e dá ao cliente um descritor de arquivo, que serve como índice para uma tabela de inodes na memória.) Esse identificador é usado para acessos subsequentes, até a sessão terminar. Um serviço stateful é caracterizado como uma conexão entre o cliente e o servidor durante uma sessão. Seja no fechamento do arquivo ou por um mecanismo de coleta de lixo, o servidor precisa retomar o espaço na memória principal usado pelos clientes que não estão mais ativos. O ponto-chave com relação à tolerância a falhas em uma técnica de serviço stateful é que o servidor mantenha as informações na memória principal sobre seus clientes. O AFS é um serviço de arquivos stateful.

Um **stateless file service (serviço de arquivo sem estado)** evita informações de estado tornando cada requisição autocontida, ou seja, cada requisição identifica o arquivo e a posição no arquivo (para acessos de leitura e escrita) totalmente. O servidor não precisa manter uma tabela de arquivos abertos na memória principal, embora faça isso por motivos de eficiência. Além do mais, não é preciso estabelecer e terminar uma conexão por operações `open()` e `close()`. Elas são redundantes, pois cada operação de arquivo é independente e não é considerada parte de uma sessão. Um processo cliente abriria um arquivo, e essa abertura não resultaria no envio de uma mensagem remota. Leituras e escritas ocorreriam como mensagens remotas (ou pesquisas de cache). O fechamento final pelo cliente resultaria em apenas uma operação local. O NFS é um serviço de arquivo stateless.

A vantagem que um serviço stateful tem em relação a um serviço stateless é o maior desempenho. As informações de arquivo são mantidas em cache na memória principal e podem ser acessadas com facilidade por meio do identificador da conexão, economizando assim os acessos ao disco. Além disso, um servidor stateful sabe se um arquivo é aberto para acesso sequencial e, portanto, pode ler os blocos seguintes antecipadamente. Os servidores stateless não podem fazer isso, pois não têm conhecimento da finalidade das requisições do cliente.

A distinção entre serviço stateful e stateless se torna mais evidente quando consideramos os efeitos de uma falha ocorrendo durante uma atividade de serviço. Um servidor stateful perde todo o seu estado em uma falha. A garantia de uma recuperação controlada de tal servidor envolve a restauração desse estado, normalmente por um protocolo de recuperação baseado em um diálogo com clientes. Uma recuperação menos controlada exige o cancelamento das operações que estavam em andamento quando ocorreu a falha. Um problema diferente é causado por falhas no cliente. O servidor precisa se tornar ciente de tais falhas, para poder retomar o espaço alocado para registrar o estado dos processos do cliente que falharam. Esse fenômeno às vezes é conhecido como **detecção e eliminação de órfãos**.

Um servidor stateless evita esses problemas, pois um servidor recém-reactivado pode responder a uma requisição autocontida sem qualquer dificuldade. Portanto, os efeitos das falhas e recuperação do servidor quase não serão observados. Não existe diferença entre um servidor lento e um servidor em recuperação, do ponto de vista de um cliente. O cliente continua retransmitindo sua requisição se não receber resposta.

A penalidade para o uso do serviço stateless robusto são mensagens de requisição maiores e processamento mais lento das requisições, já que não existem informações na memória para agilizar o processamento. Além disso, o serviço stateless impõe restrições adicionais sobre o projeto do DFS. Primeiro, como cada requisição identifica o arquivo de destino, um esquema de nomeação uniforme no nível do sistema e de baixo nível deverá ser usado. A tradução de nomes remotos para local para cada requisição causaria um processamento ainda mais lento das requisições. Segundo, como os clientes retransmitem requisições para operações de arquivo, essas operações precisam ser consistentes, ou seja, cada operação precisa ter o mesmo efeito e retornar a mesma saída se for executada várias vezes consecutivamente. Os acessos para leitura e escrita autocontidos são também poderosos, desde que usem uma contagem de bytes absoluta para indicar a posição dentro do arquivo que acessam e não contem com um deslocamento incremental (como é feito nas chamadas de sistema `read()` e `write()` do UNIX). No entanto, devemos ter cuidado ao implementar operações destrutivas (como a exclusão de um arquivo) para torná-las poderosas.

Em alguns ambientes, um serviço stateful é uma necessidade. Se o servidor empregar o método iniciado pelo servidor para validação do cache, ele não pode prover serviço stateless, pois mantém um registro de quais arquivos estão em cache e por quais clientes.

O modo como o UNIX utiliza descritores de arquivos e deslocamentos implícitos é inherentemente

stateful. Os servidores precisam manter tabelas para associar os descritores de arquivo aos inodes e precisam armazenar o deslocamento atual dentro de um arquivo. Esse requisito é o motivo pelo qual o NFS, que emprega um serviço stateless, não usa descritores de arquivos e inclui um deslocamento explícito em cada acesso.

17.5 Replicação de arquivos

A replicação de arquivos em diferentes máquinas em um sistema de arquivo distribuído é uma redundância útil para melhorar a disponibilidade. A replicação em múltiplas máquinas também pode beneficiar o desempenho: a seleção de uma réplica nas proximidades para atender a uma requisição de acesso resulta em menor tempo de atendimento.

O requisito básico de um esquema de replicação é que diferentes réplicas do mesmo arquivo residem em máquinas independentes de falha, ou seja, a disponibilidade de uma réplica não é afetada pela disponibilidade do restante das réplicas. Esse requisito óbvio implica que o gerenciamento da replicação é inherentemente uma atividade opaca ao local. Provisões para fazer uma réplica em uma máquina específica precisam estar disponíveis.

É desejável esconder os detalhes da replicação dos usuários. É a tarefa do esquema de nomeação mapear um nome de arquivo replicado para uma réplica específica. A existência de réplicas deve ser invisível em níveis mais altos. Entretanto, em níveis mais baixos, as réplicas precisam ser distinguidasumas das outras por nomes diferentes no nível inferior. Outro requisito de transparência é melhorar o controle de replicação nos níveis superiores. O controle de replicação inclui a determinação do grau de replicação e do posicionamento das réplicas. Sob determinadas circunstâncias, podemos querer expor esses detalhes aos usuários. O sistema de arquivos Locus, por exemplo, provê aos usuários e administradores do sistema mecanismos para controlar o esquema de replicação.

O principal problema associado às réplicas é sua atualização. Do ponto de vista de um usuário, as réplicas de um arquivo indicam a mesma entidade lógica, e assim uma atualização a qualquer réplica precisa ser refletida em todas as outras réplicas. Mais precisamente, a semântica de consistência relevante precisa ser preservada quando os acessos às réplicas são vistos como acessos virtuais aos arquivos lógicos dessas réplicas. Se a consistência não for de importância fundamental, ela pode ser sacrificada por disponibilidade e desempenho. Nessa escolha fundamental na área de tolerância a falhas, a escolha é entre preservar a consistência a todo custo, criando assim um potencial para bloqueio indefinido, e sacrificar a consistência sob algumas circunstâncias (esperamos que sejam raras) para o bem do progresso garantido. O Locus, por exemplo, emprega bastante replicação e sacrifica a consistência no caso de partição de rede, para o bem da disponibilidade de arquivos nos acessos de leitura e escrita.

O Ibis utiliza uma variação da técnica de cópia principal. O domínio do mapeamento de nomes é um par *<identificador de réplica principal, identificador de réplica local>*. Se não houver uma réplica, um valor especial é utilizado. Assim, o mapeamento é relativo à máquina. Se a réplica local for a principal, o par terá dois identificadores idênticos. O Ibis admite replicação por demanda, uma política de controle automático de replicação, semelhante ao caching de arquivo inteiro. Sob a replicação por demanda, a leitura de uma réplica não local a faz ser colocada no cache local, gerando assim uma nova réplica não principal. As atualizações são realizadas apenas na cópia principal e fazem todas as outras réplicas serem invalidadas pelo envio de mensagens apropriadas. A invalidação atômica e seriada de todas as réplicas não principais não é garantida. Logo, uma réplica antiga poderia ser considerada válida. Para satisfazer os acessos de escrita remota, migramos a cópia principal para a máquina requisitante.

17.6 Um exemplo: AFS

O Andrew é um ambiente de computação distribuído, projetado e implementado na Universidade Carnegie Mellon a partir de 1983. O Andrew File System (AFS) constitui o mecanismo básico de compartilhamento de informações entre os clientes do ambiente. A Transarc Corporation assumiu o desenvolvimento do AFS e depois foi adquirida pela IBM. A IBM, desde então, tem produzido várias implementações comerciais do AFS. O AFS, mais tarde, foi escolhido como o DFS para uma coalizão no setor, e o resultado foi o **Transarc DFS**, parte do ambiente de computação distribuída (**Distributed Computing Environment - DCE**) da organização OSF.

NFS V4

Nossa explicação sobre NFS até aqui só considerou a versão 3 (V3) do NFS. O padrão mais recente do NFS é a versão 4 (V4), e ela difere fundamentalmente das versões anteriores. A mudança mais significativa é que o protocolo agora é *com estado (stateful)*, significando que o servidor mantém o estado da sessão do cliente do momento em que o arquivo remoto é aberto até ele ser fechado. Assim, o protocolo NFS agora provê operações `open()` e `close()`; as versões anteriores do NFS (que são sem estado) não fornecem tais operações. Além do mais, as versões anteriores especificam protocolos separados para montar sistemas de arquivos remotos e lock de arquivos remotos. O V4 provê todos esses recursos sob um único protocolo. Em particular, o protocolo `mount` foi eliminado, permitindo que o NFS trabalhe com firewalls de rede. O protocolo `mount` foi uma brecha de segurança notória nas implementações do NFS.

Além disso, o V4 melhorou a capacidade de os clientes colocarem dados do arquivo em cache local. Esse recurso melhora o desempenho do sistema de arquivo distribuído, pois os clientes podem resolver mais acessos a arquivo a partir do cache local, em vez de terem que passar para o servidor. O V4 permite que os clientes também requisitem locks de arquivo dos servidores. Se o servidor concede a requisição, o cliente mantém o lock até que ele seja liberado ou sua locação expire. (Os clientes também têm permissão para renovar as locações existentes.) Os sistemas operacionais Windows utilizam o locking obrigatório. Para permitir que o NFS trabalhe bem com sistemas não UNIX, o V4 agora também provê locking obrigatório. Os novos mecanismos de locking e caching são baseados no conceito de **delegação**, pelo qual o servidor delega responsabilidades para o lock de um arquivo e o conteúdo ao cliente que requisitou o lock. Esse cliente delegado mantém em cache a versão atual do arquivo, e outros clientes podem pedir a esse cliente delegado o acesso por lock e o conteúdo de arquivo até que o cliente delegado libere o lock e a delegação.

Finalmente, onde as versões anteriores do NFS são baseadas no protocolo de rede UDP, o V4 é baseado no TCP, o que lhe permite ajustar melhor as cargas de tráfego variadas na rede. A delegação dessas responsabilidades aos clientes reduz a carga no servidor e melhora a coerência do cache.

Em 2000, o Transarc Lab da IBM anunciou que o AFS seria um produto de código-fonte aberto (denominado OpenAFS) disponível sob a licença pública da IBM e o Transarc DFS foi descontinuado como produto comercial. O OpenAFS está disponível sob a maioria das versões comerciais do UNIX, além dos sistemas Linux e Microsoft Windows. Muitos fornecedores de UNIX, bem como a Microsoft, suportam o sistema DCE e seu DFS (baseado no AFS), e os trabalhos continuam para tornar o DCE-DFS uma plataforma para diversos sistemas, aceito universalmente. Como o AFS e o Transarc DFS são muito semelhantes, descrevemos o AFS no decorrer desta seção, fora os casos em que o Transarc DFS é citado especificamente.

O AFS busca solucionar muitos dos problemas dos DFSs mais simples, como o NFS, e é comprovadamente o DFS não experimental mais cheio de recursos. Ele possui um espaço de nomes uniforme, compartilhamento de arquivos independente de local, caching no cliente com consistência de cache e autenticação segura via Kerberos. Ele também inclui caching no servidor, na forma de réplicas, com alta disponibilidade, por meio da passagem automática para uma réplica se o servidor de origem não estiver disponível. Um dos atributos mais formidáveis do AFS é a escalabilidade: o sistema Andrew é direcionado para cobrir 5.000 estações de trabalho. Entre o AFS e o Transarc DFS, existem centenas de implementações em todo o mundo.

17.6.1 Visão geral

O AFS distingue entre *máquinas clientes* (às vezes chamadas *estações de trabalho*) e *máquinas servidoras* dedicadas. Servidores e clientes originalmente executavam apenas o UNIX 4.2 BSD, mas o AFS foi portado para muitos sistemas operacionais. Os clientes e os servidores estão interconectados por uma rede de LANs ou WANs.

Os clientes são apresentados com um espaço partitionado de nomes de arquivos: um **espaço de nomes local (local name space)** e um **espaço de nomes compartilhado (shared name space)**.

Os servidores dedicados, coletivamente chamados de *Vice*, devido ao nome do software que executam, apresentam o shared name space aos clientes como uma hierarquia de arquivos homogênea, idêntica e transparente ao local. O local name space é o sistema de arquivos raiz de uma estação de trabalho, do qual o shared name space descende. As estações de trabalho executam o protocolo *Virtue* para a comunicação com o Vice, e cada um precisa ter discos locais, onde armazenam seu local name space. Os servidores, coletivamente, são responsáveis pelo armazenamento e gerenciamento do shared name space. O local name space é pequeno, distinto para cada estação de trabalho e contém programas do sistema essenciais para a operação autônoma e melhor desempenho. Também locais são os arquivos temporários e os arquivos que o proprietário da estação de trabalho, por motivos particulares, desejam armazenar explicitamente no local.

Vistos em sua granularidade, os clientes e os servidores são estruturados em clusters interconectados por uma WAN. Cada cluster consiste em uma coleção de estações de trabalho em uma LAN e um representante do Vice chamado **servidor de clusters (cluster server)**; cada cluster é conectado à WAN por um roteador. A decomposição em clusters é feita para resolver o problema de escala. Para um desempenho ideal, as estações de trabalho deveriam usar o servidor em seu próprio cluster na maior parte do tempo, tornando as referências de arquivo entre clusters bem pouco frequentes.

A arquitetura do sistema de arquivos também foi baseada na consideração da escala. A heurística básica é de aliviar o trabalho dos servidores para os clientes, devido à experiência que indica que a velocidade da CPU do servidor é o gargalo do sistema. Seguindo essa heurística, o mecanismo-chave para as operações de arquivo remoto é colocar os arquivos em cache em grandes pedaços (64 KB). Esse recurso reduz a latência de abertura de arquivo e permite que leituras e escritas sejam direcionadas para a cópia em cache sem envolver os servidores com frequência.

Resumidamente, aqui estão alguns aspectos adicionais no projeto do AFS:

- **Mobilidade do cliente.** Os clientes são capazes de acessar qualquer arquivo no shared name space a partir de qualquer estação de trabalho. O cliente pode observar alguma degradação inicial no desempenho, devido ao caching de arquivos ao acessar os arquivos de alguma estação de trabalho diferente das suas habituais.
- **Segurança.** A interface Vice é considerada o limite da confiabilidade, pois nenhum programa cliente é executado nas máquinas Vice. Funções de autenticação e transmissão segura são fornecidas como parte de um pacote de comunicação baseado em conexão, com base no paradigma da RPC. Após a autenticação mútua, o servidor Vice e um cliente se comunicam por meio de mensagens codificadas. A criptografia é realizada por dispositivos de hardware ou (mais lentamente) no software. As informações sobre clientes e grupos são armazenadas em bancos de dados de proteção replicados em cada servidor.
- **Proteção.** O AFS provê **listas de acesso** para proteger diretórios, e os bits normais do UNIX para proteção de arquivos. A lista de acesso pode conter informações sobre aqueles usuários com permissão para acessar um diretório, bem como informações sobre aqueles usuários *sem* permissão para acessá-lo. Assim, é simples especificar que todos, exceto, digamos, Jim, podem acessar um diretório. O AFS admite os tipos de acesso read, write, lookup, insert, administer, lock e delete.
- **Heterogeneidade.** A definição de uma interface clara com o Vice é uma chave para a integração de diversos hardwares de estação de trabalho e o sistema operacional. Para facilitar a heterogeneidade, alguns arquivos no diretório */bin* local são links simbólicos apontando para arquivos executáveis específicos da máquina, residentes no Vice.

17.6.2 O shared name space

O shared name space do AFS é constituído de unidades componentes, chamadas **volumes**. Os volumes do AFS são unidades componentes bem pequenas. Normalmente, eles são associados aos arquivos de um único cliente. Poucas escolhas residem dentro de uma única partição de disco, e elas podem crescer (até uma cota) e encolher em tamanho. Conceitualmente, os volumes são colados por um mecanismo semelhante ao mecanismo de montagem UNIX. Além do mais, a diferença de granularidade é significativa, pois no UNIX somente uma partição de disco inteira (contendo um sistema de arquivos) pode ser montada. Os volumes são uma unidade administrativa chave e desempenham um papel virtual na identificação e localização de um arquivo individual.

Um arquivo ou diretório do Vice é identificado por um identificador de baixo nível chamado **fid**. Cada entrada de diretório do AFS associa um componente de nome de caminho a um fid. Um fid possui 96 bits de extensão e três componentes de mesmo tamanho: um *número de volume*, um *número de vnode* e um *uniquificador*. O **número de vnode** é usado como um índice para um array contendo os inodes dos arquivos em um único volume. O **uniquificador** permite a reutilização de números de vnode, mantendo, assim, certas estruturas de dados compactas. Os fids são transparentes ao local; portanto, os movimentos de arquivo de um servidor para outro não invalidam o conteúdo do diretório em cache.

As informações de local são mantidas em um volume com base em um **banco de dados de volume-local (Volume-Location Database)** replicado em cada servidor. Um cliente pode

identificar o local de cada volume no sistema consultando esse banco de dados. A agregação de arquivos em volumes possibilita manter o banco de dados local em um tamanho de fácil administração.

Para equilibrar o espaço disponível em disco e a utilização dos servidores, os volumes precisam ser migrados entre partições de disco e servidores. Quando um volume é enviado para seu novo local, seu servidor original é deixado com informações de encaminhamento temporárias, para que o banco de dados local não precise ser atualizado de forma síncrona. Enquanto o volume está sendo transferido, o servidor original ainda pode lidar com atualizações, enviadas mais tarde ao novo servidor. Em algum ponto, o volume será desativado para as modificações recentes serem processadas; depois, o novo volume se torna disponível novamente na nova instalação. A operação de movimentação de volume é atômica; se um servidor falhar, a operação será abortada.

A replicação somente de leitura na granularidade de um volume inteiro é admitida para arquivos executáveis do sistema e para arquivos raramente atualizados nos níveis superiores do espaço de nomes do Vice. O banco de dados de local de volume especifica o servidor que contém a única cópia de leitura e escrita de um volume, e uma lista das instalações de replicação somente de leitura.

17.6.3 Operações de arquivo e semântica de consistência

O princípio arquitetônico fundamental no AFS é o caching de arquivos inteiros a partir dos servidores. De acordo com isso, uma estação de trabalho cliente interage com os servidores Vice apenas durante a abertura e o fechamento dos arquivos, e até mesmo essa interação nem sempre é necessária. A leitura e a escrita de arquivos não causa interação remota (ao contrário do método de serviço remoto). Essa distinção-chave possui ramificações de longo alcance para o desempenho, bem como para a semântica das operações de arquivo.

O sistema operacional em cada estação de trabalho intercepta as chamadas de sistema de arquivos e as encaminha para um processo no nível de cliente nessa estação de trabalho. Esse processo, chamado *Venus*, coloca os arquivos do Vice em cache quando são abertos e armazena cópias modificadas dos arquivos de volta aos servidores de onde vieram, quando fechados. O Venus pode contatar o Vice apenas quando o arquivo é aberto ou fechado; a leitura e a escrita dos bytes individuais de um arquivo são realizadas diretamente na cópia em cache, evitando o Venus. Como resultado, as escritas em algumas instalações não são visíveis imediatamente em outras instalações.

O caching é explorado ainda mais para aberturas futuras do arquivo em cache. O Venus considera que as entradas em cache (arquivos ou diretórios) são válidas a menos que seja notificado de outra forma. Portanto, o Venus não precisa contatar o Vice sobre um arquivo aberto para validar a cópia em cache. O mecanismo para dar suporte a essa política, chamado **callback**, reduz bastante a quantidade de requisições de validação de cache recebidas pelos servidores. Ele funciona da seguinte forma. Quando um cliente coloca um arquivo ou um diretório em cache, o servidor atualiza sua informação de estado para registrar esse caching. Digamos que o cliente tenha um callback nesse arquivo. O servidor notifica o cliente antes de permitir que outro cliente modifique o arquivo. Nesse caso, dizemos que o servidor remove o callback sobre o arquivo para o cliente anterior. Um cliente só pode usar um arquivo em cache para fins de abertura quando o arquivo tiver um callback. Se um cliente fechar um arquivo depois de modificá-lo, todos os outros clientes mantendo esse arquivo em cache perderão seus callbacks. Portanto, quando esses clientes abrirem o arquivo mais tarde, eles terão de apanhar a nova versão no servidor.

A leitura e a escrita de bytes de um arquivo são realizadas diretamente pelo kernel, sem a intervenção do Venus sobre a cópia em cache. O Venus readquire o controle quando o arquivo é fechado. Se o arquivo tiver sido modificado no local, ele atualiza o arquivo no servidor apropriado. Assim, as únicas ocasiões em que o Venus contata servidores Vice são em aberturas de arquivos que não estão no cache ou que tiveram seu callback revogado e em fechamentos de arquivos modificados localmente.

Basicamente, o AFS implementa a semântica de sessão. As únicas exceções são operações de arquivo que não sejam leitura e escrita de primitivas (tal proteção muda no nível de diretório), que são visíveis em qualquer lugar na rede, após o término da operação.

Apesar do mecanismo de callback, uma pequena quantidade de tráfego de validação em cache ainda está presente, normalmente para substituir callbacks perdidos devido a falhas de máquina ou rede. Quando uma estação de trabalho é reinicializada, o Venus considera suspeitos todos os arquivos e diretórios em cache e gera uma requisição de validação de cache para o primeiro uso de cada uma dessas entradas.

O mecanismo de callback força cada servidor a manter informações de callback e cada cliente a manter informações de validade. Se a quantidade de informações de callback mantidas por um servidor for excessiva, o servidor poderá dividir os callbacks e retomar algum armazenamento notificando clientes unilateralmente e revogando a validade de seus arquivos em cache. Se o estado de callback mantido pelo Venus ficar fora de sincronismo com o estado correspondente mantido pelos servidores, algumas inconsistências poderão ser ocasionadas.

O Venus também coloca em cache o conteúdo de diretórios e links simbólicos, para tradução de nome de caminho. Cada componente no nome do caminho é apanhado e um callback é estabelecido

para ele, se ainda não estiver em cache ou se o cliente não tiver um callback sobre ele. O Venus realiza as pesquisas localmente sobre os diretórios apanhados usando fids. Nenhuma requisição é encaminhada de um servidor para outro. No final de uma travessia de nomes de caminho, todos os diretórios intermediários e o arquivo-alvo estão no cache com callbacks sobre eles. Chamadas futuras de abertura para esse arquivo não envolverão comunicação de rede alguma, a menos que um callback esteja desfeito sobre um componente do nome de caminho.

As únicas exceções à política de caching são modificações aos diretórios que são feitas diretamente no servidor responsável por esse diretório, por questões de integridade. A interface Vice possui operações bem definidas para tais finalidades. O Venus reflete as mudanças em sua cópia em cache para evitar apanhar o diretório novamente.

17.6.4 Implementação

Os processos do cliente são interligados a um kernel do UNIX com o conjunto normal de chamadas de sistema. O kernel é modificado ligeiramente para detectar referências aos arquivos do Vice nas operações relevantes e encaminhar as requisições ao processo Venus no nível do cliente, na estação de trabalho.

O Venus executa a tradução de nome de caminho, componente por componente, conforme descrevemos anteriormente. Ele possui um cache de mapeamento que associa volumes a locais do servidor, a fim de evitar a pesquisa ao servidor em busca de um local de volume já conhecido. Se um volume não estiver presente nesse cache, o Venus contata qualquer servidor com o qual já tenha uma conexão, requisita a informação de local e entra com essa informação no cache de mapeamento. A menos que o Venus já tenha uma conexão com o servidor, ele estabelece uma nova conexão. Depois, ele usa essa conexão para buscar o arquivo ou diretório. O estabelecimento da conexão é necessário para fins de autenticação e segurança. Quando o arquivo-alvo for encontrado e mantido em cache, uma cópia será criada no disco local. O Venus, então, retorna ao kernel, que abre a cópia em cache e retorna seu descritor ao processo cliente.

O sistema de arquivos UNIX é usado como um sistema de armazenamento de baixo nível para servidores AFS e clientes. O cache cliente é um diretório local no disco da estação de trabalho. Dentro desse diretório estão arquivos cujos nomes são marcadores de lugar para entradas de cache. Tanto o Venus quanto os processos do servidor acessam arquivos do UNIX diretamente pelos inodes do servidor, para evitar a rotina dispendiosa de tradução de nome de caminho para inode (*namei*). Como a interface de inode interna não é visível aos processos no nível de cliente (tanto o Venus quanto os processos do servidor são processos no nível de cliente), um conjunto apropriado de chamadas de sistema adicionais foi acrescentado. O DFS utiliza seu próprio sistema de arquivos de diário para melhorar o desempenho e a confiabilidade em relação ao UFS.

O Venus gerencia dois caches separados: um para status e o outro para dados. Ele usa um algoritmo LRU (Least-Recently-Used) simples para manter cada um deles limitado em tamanho. Quando um arquivo é esvaziado do cache, o Venus notifica o servidor apropriado a fim de remover o callback para esse arquivo. O cache de status é mantido na memória virtual para permitir o atendimento rápido de chamadas de sistema *stat()* (status de arquivo retornando). O cache de dados é residente no disco local, mas o mecanismo de buffer de E/S do UNIX realiza algum caching de blocos de disco na memória que é transparente ao Venus.

Um único processo no nível de cliente em cada servidor de arquivos atende a todas as requisições dos clientes. Esse processo utiliza um pacote de processo leve com escalonamento sem possibilidade de preempção para atender a muitas requisições simultaneamente. O pacote da RPC é integrado ao pacote de processo leve, permitindo assim que o servidor de arquivos esteja simultaneamente criando ou atendendo a uma RPC por processo leve. O pacote de RPC é montado em cima de uma abstração de datagrama de baixo nível. A transferência do arquivo inteiro é implementada como um efeito colateral dessas chamadas de RPC. Existe uma conexão RPC por cliente, mas não existe um vínculo *a priori* de processos leves para essas conexões. Em vez disso, um banco de processos leves atende às requisições do cliente em todas as conexões. O uso de um único processo servidor multithread permite o caching das estruturas de dados necessárias para atender às requisições. No lado negativo, uma falha de um único processo servidor possui o efeito desastroso de paralisar esse servidor em particular.

17.7 Resumo

Um DFS é um sistema de serviço de arquivo cujos clientes, servidores e dispositivos de armazenamento estão dispersos entre as instalações de um sistema distribuído. De acordo com isso, a atividade de serviço precisa ser executada pela rede; em vez de um único repositório de dados centralizado, existem vários dispositivos de armazenamento independentes.

O ideal é que um DFS apareça para seus clientes como um sistema de arquivos convencional, centralizado. A multiplicidade e a dispersão de seus servidores e dispositivos de armazenamento devem se tornar transparentes, ou seja, a interface do cliente de um DFS não deve distinguir entre arquivos locais e remotos. Fica a critério do DFS localizar os arquivos e arrumar o transporte dos dados. Um DFS transparente facilita a mobilidade do cliente, trazendo o ambiente do cliente para a instalação onde ele efetua o login.

Existem várias técnicas para os esquemas de nomeação em um DFS. Na técnica mais simples, os arquivos recebem alguma combinação de seu nome de host e nome local, o que garante um nome exclusivo a todo o sistema. Outra técnica, popularizada pelo NFS, provê um meio de anexar diretórios remotos a diretórios locais, dando assim a aparência de uma árvore de diretórios coerente.

As requisições para acessar um arquivo remoto são tratadas por dois métodos complementares. Com o serviço remoto, as requisições de acesso são entregues ao servidor. A máquina servidora realiza os acessos, e seus resultados são encaminhados de volta ao cliente. Com o caching, se os dados necessários para satisfazer a requisição de acesso ainda não estiverem em cache, uma cópia desses dados é trazida do servidor para o cliente. Os acessos são realizados na cópia em cache. A ideia é reter blocos de disco acessados recentemente no cache, para que os acessos repetidos à mesma informação possam ser tratados localmente, sem o tráfego de rede adicional. Uma política de substituição é usada para manter o tamanho do cache limitado. O problema de manter as cópias em cache consistentes com o arquivo-mestre é o problema de consistência do cache.

Existem duas técnicas para as informações no servidor: ou o servidor acompanha cada arquivo que o cliente acessa ou provê blocos à medida que o cliente os requisita, sem conhecimento do seu uso. Essas técnicas são paradigmas de serviço stateful *versus* stateless.

A replicação de arquivos em diferentes máquinas é uma redundância útil para melhorar a disponibilidade. A replicação em múltiplas máquinas também pode beneficiar o desempenho, pois a seleção de uma réplica nas proximidades para atender a uma requisição de acesso resulta em menor tempo de serviço.

O AFS é um DFS cheio de recursos caracterizado pela independência de local e transparência de local. Ele também impõe semântica de consistência significativa. O caching e a replicação são usados para melhorar o desempenho.

Exercícios

- 17.1. Quais são os benefícios de um DFS quando comparado com um sistema de arquivos em um sistema centralizado?
- 17.2. Qual dos DFSs de exemplos discutidos neste capítulo trataria de uma aplicação de banco de dados grande e com múltiplos clientes da forma mais eficiente? Explique sua resposta.
- 17.3. Discuta se o AFS e o NFS fornecem o seguinte: (a) transparência de local e (b) independência de local.
- 17.4. Sob quais circunstâncias um cliente preferiria um DFS transparente ao local? Sob quais ele preferiria um DFS independente de local? Discuta os motivos para essas preferências.
- 17.5. Que aspectos de um sistema distribuído você selecionaria para um sistema executando em uma rede totalmente confiável?
- 17.6. Considere o AFS, que é um sistema de arquivos distribuído com estado. Que ações precisam ser realizadas para se recuperar de uma falha do servidor a fim de preservar a consistência garantida pelo sistema?
- 17.7. Compare as técnicas de caching de blocos de disco localmente, em um sistema cliente, e remotamente, em um servidor.
- 17.8. O AFS foi criado para dar suporte a um grande número de clientes. Discuta três técnicas usadas para tornar o AFS um sistema escalável.
- 17.9. Discuta as vantagens e desvantagens da tradução de nome de caminho na qual o cliente envia o caminho inteiro ao servidor requisitando uma tradução para o nome de caminho inteiro até o arquivo.
- 17.10. Quais são os benefícios de mapear objetos na memória virtual, como faz o Apollo Domain? Quais são os prejuízos?
- 17.11. Descreva algumas das diferenças fundamentais entre o AFS e o NFS (consulte o [Capítulo 11](#)).
- 17.12. Discuta se os clientes nos sistemas a seguir podem obter dados inconsistentes ou passados do servidor de arquivos e, nesse caso, sob que cenários isso poderia ocorrer.
 - a. AFS
 - b. Sprite
 - c. NFS

Notas bibliográficas

Controle de consistência e recuperação para arquivos replicados são examinados por [Davcev e Burkhard \[1985\]](#). O gerenciamento de arquivos replicados em um ambiente UNIX foi abordado por [Brereton \[1986\]](#) e [Purdin e outros \[1987\]](#). [Wah \[1984\]](#) discutiu a questão de posicionamento de arquivo em sistemas de computador distribuídos. Um estudo detalhado dos servidores de arquivos principalmente centralizados aparece em [Svobodova \[1984\]](#).

O Network File System (NFS) da Sun é descrito por [Callaghan \[2000\]](#) e [Sandberg e outros \[1985\]](#). O sistema AFS é discutido por [Morris e outros \[1986\]](#), [Howard e outros \[1988\]](#) e [Satyanarayanan \[1990\]](#). As informações sobre o OpenAFS estão disponíveis no endereço <http://www.openafs.org>.

Muitos DFSs diferentes e interessantes não são abordados com detalhes neste texto, incluindo UNIX United, Sprite e Locus. O UNIX United foi descrito por [Brownbridge e outros \[1982\]](#). O sistema Locus foi discutido por [Popek e Walker \[1985\]](#). O sistema Sprite foi descrito por [Ousterhout e outros \[1988\]](#). Os sistemas de arquivos distribuídos para dispositivos de armazenamento móveis foram discutidos em [Kistler e Satyanarayanan \[1992\]](#) e [Sobti e outros \[2004\]](#). Uma pesquisa considerável também tem sido realizada sobre sistemas de arquivos distribuídos com base em cluster ([Anderson e outros \[1995\]](#), [Lee e Thekkath \[1996\]](#), [Thekkath e outros \[1997\]](#) e [Anderson e outros \[2000\]](#)). Os sistemas de armazenamento distribuídos para configurações remotas em grande escala são apresentados em [Dabek e outros \[2001\]](#) e [Kubiatowicz e outros \[2000\]](#).

CAPÍTULO 18

Coordenação distribuída

No [Capítulo 6](#), descrevemos diversos mecanismos que permitem aos processos sincronizar suas ações. Também discutimos uma série de esquemas para garantir a propriedade de atomicidade de uma transação executada em isolamento ou concorrentemente com outras transações. No [Capítulo 7](#), descrevemos diversos métodos que um sistema operacional pode utilizar para lidar com o problema de deadlock. Neste capítulo, vamos examinar como os mecanismos de sincronismo centralizados podem ser estendidos para um ambiente distribuído. Também discutimos métodos para tratar deadlocks em um sistema distribuído.

OBJETIVOS DO CAPÍTULO

- Descrever diversos métodos para conseguir a exclusão mútua em um sistema distribuído.
- Explicar como as transações atômicas podem ser implementadas em um sistema distribuído.
- Mostrar como alguns dos esquemas de controle de concorrência discutidos no [Capítulo 6](#) podem ser modificados para uso em um ambiente distribuído.
- Apresentar esquemas para lidar com prevenção de deadlock, evitar deadlock e detecção de deadlock em um sistema distribuído.

18.1 Ordenação de eventos

Em um sistema centralizado, sempre podemos determinar a ordem em que dois eventos ocorreram, pois o sistema possui uma única memória e relógio comuns. Muitas aplicações podem exigir que determinemos a ordem. Por exemplo, em um esquema de alocação de recursos, especificamos que um recurso só pode ser usado *depois* de um recurso ter sido concedido. No entanto, um sistema distribuído não possui memória comum nem relógio comum. Portanto, às vezes é impossível dizer qual dentre dois eventos ocorreu primeiro. A relação *ocorreu antes*, discutida a seguir, é apenas uma ordenação parcial dos eventos nos sistemas distribuídos. Como a capacidade de definir uma ordenação total é crucial em muitas aplicações, apresentamos um algoritmo distribuído para estender a relação *ocorreu antes* a uma ordenação total coerente de todos os eventos no sistema.

18.1.1 A relação ocorreu antes

Como estamos considerando apenas os processos sequenciais, todos os eventos executados em um único processo são totalmente ordenados. Além disso, pela lei da causalidade, uma mensagem só pode ser recebida depois de ter sido enviada. Portanto, podemos definir a relação *ocorreu antes* (indicada por \rightarrow) sobre um conjunto de eventos da seguinte maneira (supondo que o envio e o recebimento de uma mensagem constituam um evento):

1. Se A e B são eventos no mesmo processo, e A foi executado antes de B, então $A \rightarrow B$.
2. Se A é o evento de envio de uma mensagem por um processo e B é o evento de recebimento dessa mensagem por outro processo, então $A \rightarrow B$.
3. Se $A \rightarrow B$ e $B \rightarrow C$, então $A \rightarrow C$.

Como um evento não pode acontecer antes dele mesmo, a relação \rightarrow é uma ordenação parcial não reflexiva.

Se dois eventos, A e B, não estiverem relacionados pela relação \rightarrow (ou seja, A não aconteceu antes de B, e B não aconteceu antes de A), então dizemos que esses dois eventos foram executados **concorrentemente**. Nesse caso, nenhum dos eventos poderá, por causalidade, afetar o outro. Contudo, se $A \rightarrow B$, então é possível que o evento A afete o evento B por causalidade.

Um diagrama espaço-tempo, como o da Figura 18.1, pode ilustrar melhor as definições da concorrência e *ocorreu antes*. A direção horizontal representa o espaço (ou seja, diferentes processos) e a direção vertical representa o tempo. As linhas verticais rotuladas indicam processos (ou processadores). Os pontos rotulados indicam eventos. Uma linha ondulada indica uma mensagem enviada de um processo para outro. Os eventos são concorrentes se e somente se não houver um caminho entre eles.

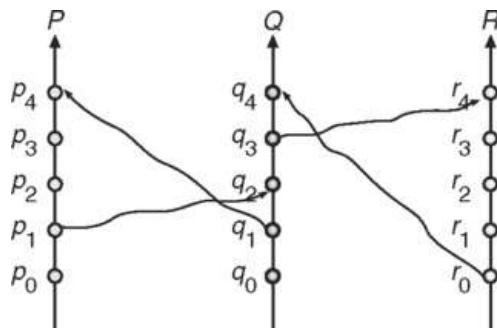


FIGURA 18.1 Tempo relativo para três processos simultâneos.

Por exemplo, esses são alguns dos eventos relacionados pela relação *ocorreu antes* na Figura 18.1:

$$\begin{aligned} p_1 &\rightarrow q_2 \\ r_0 &\rightarrow q_4 \\ q_3 &\rightarrow r_4 \\ p_1 &\rightarrow q_4 \text{ (pois } p_1 \rightarrow q_2 \text{ e } q_2 \rightarrow q_4\text{).} \end{aligned}$$

Esses são alguns dos eventos concorrentes no sistema:

$q_0 \cdot p_2$
 $r_0 \cdot q_3$
 $r_0 \cdot p_3$
 $q_3 \cdot p_3$

Não podemos saber qual de dois eventos concorrentes, como q_0 e p_2 , aconteceu primeiro. Contudo, como nenhum evento pode afetar o outro (não existe como um deles saber se o outro já ocorreu), não é importante qual aconteceu primeiro. É importante apenas que quaisquer processos que se importam com a ordem de dois eventos concorrentes combinem com alguma ordem.

18.1.2 Implementação

Para determinar que um evento A aconteceu antes de um evento B, precisamos de um relógio comum ou de um conjunto de relógios perfeitamente sincronizados. Como nenhum deles está disponível em um sistema distribuído, temos de definir a relação *ocorreu antes* sem o uso de relógios físicos.

Primeiro, associamos cada evento do sistema a uma **estampa de tempo (timestamp)**. Podemos, então, definir o requisito de **ordenação global**. Para cada par de eventos A e B, se $A \rightarrow B$, então a estampa de tempo de A é menor do que a estampa de tempo de B. (A seguir, veremos que a recíproca não precisa ser verdadeira.)

Como impomos o requisito de ordenação global em um ambiente distribuído? Definimos dentro de *cada* processo P_i um **relógio lógico**, RL_i . O relógio lógico pode ser implementado como um contador simples, incrementado entre dois eventos sucessivos quaisquer, executados dentro de um processo. Como o relógio lógico possui um valor aumentando **monotonicamente**, ele atribui um número exclusivo a cada evento, e se um evento A ocorre antes do evento B no processo P_i , então $RL_i(A) < RL_i(B)$. A estampa de tempo para um evento é o valor do relógio lógico para esse evento. Esse esquema garante que, para dois eventos quaisquer no mesmo processo, o requisito de ordenação global será atendido.

Infelizmente, esse esquema não garante que o requisito de ordenação global será atendido entre os processos. Para ilustrar o problema, considere dois processos P_1 e P_2 que se comunicam entre si. Suponha que P_1 envie uma mensagem para P_2 (evento A) com $RL_1(A) = 200$, e P_2 receba a mensagem (evento B) com $RL_2(B) = 195$ (como o processador para P_2 é mais lento do que o processador para P_1 , seu relógio lógico bate mais lentamente). Essa situação transgride nosso requisito, pois $A \rightarrow B$, mas a estampa de tempo de A é maior do que a estampa de tempo de B.

Para resolver essa dificuldade, exigimos que um processo avance seu relógio lógico quando receber uma mensagem cuja estampa de tempo é maior do que o valor atual de seu relógio lógico. Em particular, se o processo P_i receber uma mensagem (evento B) com estampa de tempo t e $RL_i(B) \leq t$, então ele deverá avançar seu relógio de modo que $RL_i(B) = t + 1$. Assim, em nosso exemplo, quando P_2 receber a mensagem de P_1 , ele avançará seu relógio lógico de modo que $RL_2(B) = 201$.

Finalmente, para realizar uma ordenação total, só precisamos observar que, com nosso esquema de ordenação de estampa de tempo, se as estampas de tempo de dois eventos, A e B, forem iguais, então os eventos são concorrentes. Nesse caso, podemos usar números de identidade de processo para desempatar e para criar uma ordenação total. O uso de estampas de tempo é discutido na [Seção 18.4.2](#).

18.2 Exclusão mútua

Nesta seção, apresentamos uma série de algoritmos diferentes para implementar a exclusão mútua em um ambiente distribuído. Consideramos que o sistema consiste em n processos, cada um residindo em um processador diferente. Para simplificar nossa discussão, supomos que os processos são numerados exclusivamente de 1 a n e que existe um mapeamento um para um entre processos e processadores (ou seja, cada processo possui seu próprio processador).

18.2.1 Técnica centralizada

Em uma técnica centralizada para prover exclusão mútua, um dos processos no sistema é escolhido para coordenar a entrada para a seção crítica. Cada processo que deseja invocar a exclusão mútua envia uma mensagem de *requisição* ao coordenador. Quando o processo recebe uma mensagem de *réplica* do coordenador, ele pode entrar em sua seção crítica. Depois de sair de sua seção crítica, o processo envia uma mensagem de *liberação* para o coordenador e prossegue com sua execução.

Ao receber uma mensagem de *requisição*, o coordenador verifica se algum outro processo está em sua seção crítica. Se nenhum processo estiver em sua seção crítica, o coordenador imediatamente envia de volta uma mensagem de *réplica*. Caso contrário, a requisição é enfileirada. Quando o coordenador recebe uma mensagem de *liberação*, ele remove uma das mensagens de *requisição* da fila (de acordo com algum algoritmo de escalonamento) e envia uma mensagem de *réplica* ao processo solicitante.

Deve ficar claro que esse algoritmo garante a exclusão mútua. Além disso, se a política de escalonamento dentro do coordenador for justa (como o escalonamento FCFS First-Come, First-Served), nenhum starvation poderá ocorrer. Esse esquema exige três mensagens por entrada de seção crítica: uma *requisição*, uma *réplica* e uma *liberação*.

Se o processo coordenador falhar, então um novo processo precisa tomar seu lugar. Na [Seção 18.6](#), descrevemos alguns algoritmos para eleger um novo coordenador exclusivo. Quando um novo coordenador tiver sido eleito, ele precisa consultar sequencialmente todos os processos no sistema, para reconstruir sua fila de *requisições*. Quando a fila tiver sido construída, a computação poderá retomar.

18.2.2 Técnica totalmente distribuída

Se quisermos distribuir a tomada de decisão por meio do sistema inteiro, então a solução é muito mais complicada. Uma técnica, descrita a seguir, utiliza um algoritmo baseado no esquema de ordenação de eventos descrito na [Seção 18.1](#).

Quando um processo P_i quiser entrar em sua seção crítica, ele gerará uma nova estampa de tempo, ET , e enviará a mensagem $request(P_i, ET)$ a todos os outros processos no sistema (incluindo ele mesmo). Ao receber uma mensagem de requisição, um processo pode replicar imediatamente (ou seja, enviar uma mensagem de *réplica* de volta para P_i) ou pode adiar o envio de uma réplica (porque já está em sua seção crítica, por exemplo). Um processo que recebeu uma mensagem de réplica de todos os outros processos no sistema pode entrar em sua seção crítica, enfileirando as requisições que chegam e adiando-as. Depois de terminar sua seção crítica, o processo envia mensagens de *réplica* a todas as requisições adiadas.

Decidir se o processo P_i responde imediatamente a uma mensagem $request(P_j, ET)$ ou adia sua resposta é algo baseado em três fatores:

1. Se o processo P_i estiver em sua seção crítica, então ele adia sua réplica para P_j .
2. Se o processo P_i não quiser entrar em sua seção crítica, então ele envia uma réplica imediatamente para P_j .
3. Se o processo P_i quiser entrar em sua seção crítica, mas ainda não tiver entrado nela, então ele compara sua própria estampa de tempo de *requisição* com a estampa de tempo ET da requisição que chega, feita pelo processo P_j . Se a estampa de tempo de sua própria requisição for maior do que ET , então ele envia uma réplica imediatamente para P_j (P_j pediu primeiro). Caso contrário, a réplica é adiada.

Esse algoritmo exibe o seguinte comportamento desejável:

- A exclusão mútua é obtida.
- A ausência de deadlock é garantida.
- A ausência de starvation é garantida, pois a entrada na seção crítica é escalonada de acordo com a ordenação da estampa de tempo. A ordenação da estampa de tempo garante que os processos são atendidos na ordem FCFS.
- A quantidade de mensagens por entrada na seção crítica é $2 \times (n - 1)$. Esse número representa o número mínimo de mensagens requisitadas por entrada na seção crítica quando os processos atuam independente e concorrentemente.

Para ilustrar como o algoritmo funciona, consideramos um sistema consistindo em processos P_1, P_2

e P_3 . Suponha que os processos P_1 e P_3 queiram entrar em suas seções críticas. O processo P_1 , então, envia uma mensagem *request* (P_1 , estampa de tempo = 10) aos processos P_2 e P_3 , enquanto o processo P_3 envia uma mensagem *request* (P_3 , estampa de tempo = 4) aos processos P_1 e P_2 . (As estampas de tempo 4 e 10 foram obtidas dos relógios lógicos descritos na [Seção 18.1](#).) Quando o processo P_2 recebe essas mensagens de *requisição*, ele responde imediatamente. Quando o processo P_1 recebe a *requisição* do processo P_3 , ele responde imediatamente, pois a estampa de tempo (10) em sua própria mensagem de *requisição* é maior do que a estampa de tempo (4) para o processo P_3 . Quando o processo P_3 recebe a mensagem de *requisição* do processo P_1 , ele adia sua réplica, pois a estampa de tempo (4) em sua mensagem de *requisição* é menor do que a estampa de tempo (10) para a mensagem do processo P_1 . Ao receber réplicas do processo P_1 e do processo P_2 , o processo P_3 pode entrar em sua seção crítica. Depois de sair de sua seção crítica, o processo P_3 envia uma réplica ao processo P_1 , que pode, então, entrar em sua seção crítica.

Entretanto, como esse esquema exige a participação de todos os processos no sistema, ele possui três consequências indesejáveis:

1. Os processos precisam saber a identidade de todos os processos no sistema. Quando um novo processo se junta ao grupo de processos participando do algoritmo de exclusão mútua, as ações a seguir precisam ser realizadas:
 - a. O processo precisa receber os nomes de todos os outros processos no grupo.
 - b. O nome do novo processo precisa ser distribuído a todos os outros processos no grupo.

Essa tarefa não é tão trivial quanto possa parecer, pois algumas mensagens de *requisição* e *réplica* podem estar circulando no sistema quando o novo processo ingressa no grupo. O leitor interessado deverá consultar as Notas Bibliográficas ao final do capítulo.

2. Se um processo falhar, então o esquema inteiro desmorona. Podemos resolver essa dificuldade monitorando continuamente o estado de todos os processos no sistema. Se um processo falhar, então todos os processos são notificados, de modo que não enviem mais mensagens de *requisição* para o processo que falhou. Quando um processo se recuperar, ele terá de iniciar o procedimento que permite voltar ao grupo.
3. Os processos que não entraram em sua seção crítica precisam parar frequentemente para assegurar a outros processos que eles pretendem entrar na seção crítica. Esse protocolo, portanto, é adequado para pequenos e estáveis conjuntos de processos em cooperação.

Devido às dificuldades, esse protocolo é mais adequado para conjuntos de processos cooperativos pequenos e estáveis.

18.2.3 Técnica de passagem de tokens

Outro método de prover exclusão mútua é circular um token entre os processos no sistema. Um **token** é um tipo especial de mensagem passada de um processo para outro. A posse do token dá direito de entrar na seção crítica. Como há apenas um token, somente um processo pode estar na seção crítica em determinado momento.

Consideramos que os processos no sistema estão organizados *logicamente* em uma **estrutura de anel**. A rede física de comunicação não precisa ser um anel. Desde que os processos estejam conectados um ao outro, é possível implementar um anel lógico. Para implementar a exclusão mútua, passamos o token ao redor do anel. Quando um processo recebe o token, ele pode entrar em sua seção crítica, mantendo-o. Depois de o processo sair de sua seção crítica, o token é novamente passado. Se o processo que recebe o token não quiser entrar em sua seção crítica, ele o passa para o vizinho. Esse esquema é semelhante ao algoritmo 1 no [Capítulo 6](#), mas um token é substituído por uma variável compartilhada.

Se o anel for unidirecional, podemos garantir que não haverá starvation. A quantidade de mensagens exigidas para implementar a exclusão mútua pode variar de uma mensagem por entidade, no caso de alta contenção (ou seja, cada processo deseja entrar em sua seção crítica), até uma quantidade infinita de mensagens, no caso de pouca contenção (ou seja, nenhum processo deseja entrar em sua seção crítica).

Dois tipos de falha precisam ser considerados. Primeiro, se o token for perdido, é preciso chamar uma eleição para gerar um novo token. Segundo, se um processo falhar, um novo anel lógico precisa ser estabelecido. Na [Seção 18.6](#), apresentamos um algoritmo de eleição, mas outros são possíveis. O desenvolvimento de um algoritmo para reconstruir o anel fica para você resolver no [Exercício 18.9](#).

18.3 Atomicidade

No [Capítulo 6](#), apresentamos o conceito de transação **atômica**, que é uma unidade de programa que precisa ser executada de forma indivisível, ou seja, ou todas as operações associadas a ela são executadas até o fim ou nenhuma é realizada. Quando estamos lidando com um sistema distribuído, garantir a atomicidade de uma transação se torna muito mais complicado do que em um sistema centralizado. Essa dificuldade ocorre porque várias instalações podem estar participando da execução de uma única transação. A falha de uma dessas instalações, ou a falha de um enlace de comunicação conectando essas instalações, pode resultar em computações erradas.

Garantir que a execução das transações no sistema distribuído preserve a atomicidade é a função do **coordenador de transações**. Cada instalação possui seu próprio coordenador de transações local, que é responsável por coordenar a execução de todas as transações iniciadas na instalação. Para cada uma dessas transações, o coordenador é responsável pelo seguinte:

- Iniciar a execução da transação.
- Dividir a transação em uma série de subtransações e distribuir essas subtransações para as instalações apropriadas, para execução.
- Coordenar o término da transação, que pode resultar na transação que está sendo Sconfirmada em todas as instalações ou abortada em todas as instalações.

Consideramos que cada instalação local mantém um log para fins de recuperação.

18.3.1 O protocolo de confirmação em duas fases

Para que a atomicidade seja garantida, todas as instalações em que uma transação T executou precisam combinar sobre o resultado final da execução. T precisa ser confirmada em todas as instalações ou precisa abortar em todas as instalações. Para garantir essa propriedade, o coordenador de transações de T deve executar um **protocolo de confirmação**. Entre os protocolos de confirmação mais simples e mais utilizados está o **protocolo de confirmação em duas fases (Two-Phase Commit - 2PC)**, que discutimos a seguir.

Seja T uma transação iniciada na instalação S_i e seja C_i o coordenador de transações em S_i . Quando T termina sua execução - ou seja, quando todas as instalações em que T executou informarem a C_i que T terminou -, então C_i inicia o protocolo 2PC.

■ **Fase 1.** C_i acrescenta o registro $\langle \text{prepare } T \rangle$ ao log e força o registro no armazenamento estável. Depois, ele envia uma mensagem $\text{prepare}(T)$ a todas as instalações em que T foi executada. Ao receber essa mensagem, o gerenciador de transações em cada uma dessas instalações determina se deseja confirmar sua parte de T . Se a resposta for *não*, ele acrescenta um registro $\langle \text{no } T \rangle$ ao log e depois responde enviando uma mensagem $\text{abort}(T)$ para C_i . Se a resposta for *sim*, ele acrescenta um registro $\langle \text{ready } T \rangle$ ao log e força todos os registros de log correspondentes a T para o armazenamento estável. O gerenciador de transações, então, responde com uma mensagem $\text{ready}(T)$ para C_i .

■ **Fase 2.** Quando C_i recebe respostas para a mensagem $\text{prepare}(T)$ que enviou a todas as instalações, ou quando um intervalo de tempo previamente estabelecido tiver se esgotado desde o envio da mensagem $\text{prepare}(T)$, C_i pode determinar se a transação T pode ser confirmada ou abortada. A transação T pode ser confirmada se C_i recebeu uma mensagem $\text{ready}(T)$ de todas as instalações participantes. Caso contrário, a transação T precisa ser abortada. Dependendo do veredito, um registro $\langle \text{commit } T \rangle$ ou um registro $\langle \text{abort } T \rangle$ é acrescentado ao log e forçado ao armazenamento estável. Nesse ponto, o destino da transação foi selado. Depois disso, o coordenador envia uma mensagem $\text{commit}(T)$ ou $\text{abort}(T)$ a todas as instalações participantes.

Quando uma instalação recebe essa mensagem, ela registra a mensagem no log.

Uma instalação em que T foi executada pode cancelar T incondicionalmente a qualquer momento depois do envio da mensagem $\text{ready}(T)$ ao coordenador. A mensagem $\text{ready}(T)$, com efeito, é uma promessa por uma instalação de seguir a ordem do coordenador de submeter T ou abortar T . Uma instalação pode fazer tal promessa apenas quando as informações necessárias forem mantidas no armazenamento estável. Caso contrário, se a instalação falhar após enviar $\text{ready}(T)$, ela pode não conseguir cumprir sua promessa.

Como a unanimidade é obrigatória para confirmar uma transação, o destino de T está selado assim que pelo menos uma instalação responder com $\text{abort}(T)$. Observe que a instalação coordenadora S_i pode decidir unilateralmente abortar T , visto ser uma das instalações em que T executou. O veredito final com relação a T é determinado no momento em que o coordenador escreve esse veredito (confirmar ou abortar) no log e o força para o armazenamento estável.

Em algumas implementações do protocolo 2PC, uma instalação envia uma mensagem de $\text{acknowledge } T$ ao coordenador no final da segunda fase do protocolo. Quando o coordenador recebe a mensagem $\text{acknowledge } T$ de todas as instalações, ele acrescenta o registro $\langle \text{complete } T \rangle$ ao log.

18.3.2 Tratamento de falhas no 2PC

Agora, vamos examinar com detalhes como o 2PC responde a vários tipos de falhas. Conforme veremos, uma grande desvantagem do protocolo 2PC é que a falha do coordenador pode resultar em bloqueio, e a situação em que uma decisão de submeter ou abortar T pode ter de ser adiada até o coordenador se recuperar.

18.3.2.1 Falha de uma instalação participante

Quando uma instalação participante S_k se recupera de uma falha, ela precisa examinar seu log para determinar o destino daquelas transações que estavam no meio da execução quando ocorreu a falha. Como S_k lida com T ? Consideremos cada um dos casos possíveis:

- O log contém um registro <commit T >. Nesse caso, a instalação executa redo(T).
- O log contém um registro <abort T >. Nesse caso, a instalação executa undo(T).
- O log contém um registro <ready T >. Nesse caso, a instalação precisa consultar C_i para determinar o destino de T . Se C_i estiver ativo, ele notifica S_k com relação a se T foi confirmado ou abortado. No primeiro caso, ele executa redo(T); no segundo, ele executa undo(T). Se C_i estiver inativo, S_k precisa tentar encontrar o destino de T a partir de outras instalações. Ele faz isso enviando uma mensagem *query-status* (T) a todas as instalações no sistema. Ao receber tal mensagem, uma instalação precisa consultar seu log para determinar se T foi executada lá e, se tiver sido, se T foi confirmada ou abortada. Depois, ele notifica S_k sobre esse resultado. Se nenhuma instalação tiver a informação apropriada (ou seja, se T confirmou ou abortou), então S_k não pode abortar nem confirmar T . A decisão com relação a T é adiada até que S_k possa obter a informação necessária. Assim, S_k precisa reenviar periodicamente a mensagem *query-status* (T) às outras instalações. Ela faz isso até uma instalação responder com a informação necessária. A instalação em que C_i reside sempre tem a informação necessária.
- O log não contém registros de controle (abort, commit, ready) com relação a T . A ausência de registros de controle significa que S_k falhou antes de responder à mensagem *prepare T* de C_i . Como a falha de S_k impede o envio de tal resposta, por nosso algoritmo, C_i precisa abortar T . Logo, S_k precisa executar undo(T).

18.3.2.2 Falha do coordenador

Se o coordenador falhar no meio da execução do protocolo de confirmação para a transação T , então as instalações participantes precisarão decidir sobre o destino de T . Veremos que, em certos casos, as instalações participantes não podem decidir se T será confirmada ou abortada e, por isso, essas instalações precisam esperar pela recuperação do coordenador que falhou.

- Se uma instalação ativa tiver um registro <commit T > em seu log, então T precisa ser confirmada.
- Se uma instalação ativa tiver um registro <abort T > em seu log, então T precisa ser abortada.
- Se alguma instalação *não* tiver um registro <ready T > em seu log, então o coordenador que falhou C_i não pode ter decidido confirmar T . Podemos chegar a essa conclusão porque uma instalação que não possui um registro <ready T > em seu log *não* pode ter enviado uma mensagem *ready(T)* para C_i . Contudo, o coordenador pode ter decidido abortar T . Em vez de esperar até C_i se recuperar, é preferível abortar T .
- Se nenhum dos casos anteriores acontecer, então todas as instalações ativas deverão ter um registro <ready T > em seus logs, mas nenhum registro de controle adicional (como <abort T > ou <commit T >). Como o coordenador falhou, é impossível determinar se uma decisão foi tomada - ou, se ocorreu, qual é essa decisão - até o coordenador se recuperar. Assim, as instalações ativas precisam esperar até C_i se recuperar. Enquanto o destino de T permanecer em dúvida, T pode continuar a manter recursos do sistema. Por exemplo, se o mecanismo de lock for usado, T pode manter locks de dados em instalações ativas. Essa situação é indesejável, pois horas ou dias podem passar até C_i estar novamente ativo. Durante esse tempo, outras transações podem ser forçadas a esperar por T . Como resultado, os dados não estarão disponíveis, não apenas na instalação que falhou (C_i), mas também nas instalações ativas. A quantidade de dados indisponíveis cresce à medida que o tempo de paralisação de C_i aumenta. Essa situação é denominada problema de *bloqueio*, pois T está bloqueada, dependendo da recuperação da instalação C_i .

18.3.2.3 Falha da rede

Quando um enlace falha, as mensagens no processo de roteamento pelo enlace não chegam intactas ao seu destino. Do ponto de vista das instalações conectadas por meio do enlace, as outras instalações parecem ter falhado. Assim, nossos esquemas anteriores também se aplicam aqui.

Quando diversos enlaces falham, a rede pode participar. Nesse caso, existem duas possibilidades. O coordenador e todos os seus participantes podem permanecer em uma partição; nesse caso, a

falha não tem efeito sobre o protocolo de confirmação. Como alternativa, o coordenador e seus participantes podem pertencer a várias partições; nesse caso, as mensagens entre os participantes e o coordenador são perdidas, reduzindo o caso a uma falha de enlace.

18.4 Controle de concorrência

Passamos em seguida para a questão de controle de concorrência. Nesta seção, vamos mostrar como certos esquemas de controle de concorrência discutidos no [Capítulo 6](#) podem ser modificados para que sejam usados em um ambiente distribuído.

O gerenciador de transações de um sistema de banco de dados distribuído gerencia a execução daquelas transações (ou subtransações) que acessam dados armazenados em uma instalação local. Cada uma dessas transações pode ser uma transação local (ou seja, uma transação que executa apenas nessa instalação) ou parte de uma transação global (ou seja, uma transação que executa em várias instalações). Cada gerenciador de transações é responsável por manter um log para fins de recuperação e para participar de um esquema de controle de concorrência apropriado, para coordenar a execução simultânea das transações em execução nessa instalação. Conforme veremos, os esquemas de concorrência descritos no [Capítulo 6](#) precisam ser modificados para acomodar a distribuição de transações.

18.4.1 Protocolos de lock

Os protocolos de lock em duas fases, descritos no [Capítulo 6](#), podem ser usados em um ambiente distribuído. A única mudança necessária é a maneira como o gerenciador de lock é implementado. Aqui, apresentamos vários esquemas possíveis. O primeiro lida com o caso em que nenhuma replicação de dados é permitida. Os outros esquemas se aplicam ao caso mais geral em que os dados podem ser replicados em várias instalações. Como no [Capítulo 6](#), vamos considerar a existência dos **modos de lock compartilhado e exclusivo**.

18.4.1.1 Esquema não replicado

Se nenhum dado estiver replicado no sistema, então os esquemas de lock descritos na [Seção 6.10](#) podem ser aplicados como a seguir. Cada instalação mantém um gerenciador de lock cuja função é administrar as requisições de lock e unlock para os itens de dados armazenados nessa instalação. Quando uma transação deseja efetuar o lock de um item de dados Q na instalação S_i , ela envia uma mensagem ao gerenciador de lock na instalação S_i requisitando um lock (em um modo de lock particular). Se o item de dados Q estiver em um modo incompatível, então a requisição é adiada até a requisição poder ser concedida. Quando tiver sido determinado que a requisição de lock pode ser concedida, o gerenciador de lock envia uma mensagem de volta a quem a iniciou, indicando que a requisição de lock foi concedida.

Esse esquema tem a vantagem da implementação simples. Ele requer duas transferências de mensagem para tratar de requisições de lock e uma transferência de mensagem para tratar de requisições de unlock. Contudo, o tratamento do deadlock é mais complexo. Visto que as requisições de lock e unlock não são mais feitas em uma única instalação, os diversos algoritmos de tratamento de deadlock discutidos no [Capítulo 7](#) precisam ser modificados; essas modificações serão discutidas na [Seção 18.5](#).

18.4.1.2 Técnica de coordenador único

Vários esquemas de controle de concorrência podem ser usados nos sistemas que permitem a replicação de dados. Sob a técnica de único coordenador, o sistema mantém um *único* gerenciador de lock que reside em uma *única* instalação escolhida - digamos, S_i . Todas as requisições de lock e unlock são feitas na instalação S_i . Quando uma transação precisa efetuar o lock de um item de dados, ela envia uma requisição de lock para S_i . O gerenciador de lock determina se o lock pode ser concedido imediatamente. Se puder, ele envia uma mensagem com esse efeito para a instalação em que a requisição de lock foi iniciada. Caso contrário, a requisição é adiada até poder ser concedida, quando uma mensagem será enviada à instalação em que a requisição de lock foi iniciada. A transação pode ler o item de dados de *qualquer* uma das instalações em que reside uma réplica do item de dados. No caso de um write, todas as instalações onde reside uma réplica do item de dados precisam estar envolvidas na escrita.

O esquema possui as seguintes vantagens:

- **Implementação simples.** Esse esquema exige duas mensagens para o tratamento de requisições de lock e uma mensagem para o tratamento de requisições de unlock.
- **Tratamento de deadlock simples.** Como todas as requisições de lock e unlock são feitas em uma instalação, os algoritmos de tratamento de deadlock discutidos no [Capítulo 7](#) podem ser aplicados diretamente a esse ambiente.

As desvantagens do esquema incluem as seguintes:

- **Gargalo.** A instalação S_i torna-se um gargalo, pois todas as requisições precisam ser processadas lá.
- **Vulnerabilidade.** Se a instalação S_i falhar, o controlador de concorrência estará perdido.

Qualquer processamento deverá parar ou um esquema de recuperação deverá ser usado.

Um meio-termo entre essas vantagens e desvantagens pode ser obtido por meio de uma **técnica de coordenador múltiplo**, em que a função do gerenciador de lock é distribuída por várias instalações. Cada gerenciador de lock administra as requisições de lock e unlock para um subconjunto dos itens de dados. Essa distribuição reduz o grau em que o coordenador é um gargalo, mas complica o tratamento do deadlock, pois as requisições de lock e unlock não são feitas em uma única instalação.

18.4.1.3 Protocolo da maioria

O protocolo da maioria é uma modificação do esquema de dados não replicados apresentado anteriormente. O sistema mantém um gerenciador de lock em cada instalação. Cada gerenciador controla os locks para todos os dados ou réplicas dos dados armazenados nessa instalação. Quando uma transação deseja efetuar o lock de um item de dados Q que é replicado em n instalações diferentes, ela precisa enviar uma requisição de lock a mais de metade das n instalações onde Q está armazenado. Cada gerenciador de lock determina se o lock pode ser concedido imediatamente (no que se refere a ele). Como antes, a resposta é adiada até a requisição poder ser concedida. A transação não opera sobre Q até ter obtido um lock com sucesso sobre a maioria das réplicas de Q .

Esse esquema trata dos dados replicados de uma maneira descentralizada, evitando assim as desvantagens do controle central. Todavia, ele sofre com suas próprias desvantagens:

- **Implementação.** O protocolo da maioria é mais complicado para implementar do que os esquemas anteriores. Ele exige $2(n/2 + 1)$ mensagens para o tratamento das requisições de lock e $(n/2 + 1)$ mensagens para o tratamento de requisições de unlock.
- **Tratamento de deadlock.** Como as requisições de lock e unlock não são feitas em uma instalação, os algoritmos de tratamento de deadlock precisam ser modificados ([Seção 18.5](#)). Além disso, um deadlock pode ocorrer mesmo que somente um item de dados esteja sofrendo um lock. Para ilustrar, considere um sistema com quatro instalações e replicação completa. Suponha que as transações T_1 e T_2 queiram efetuar o lock do item de dados Q no modo exclusivo. A transação T_1 pode ter sucesso no lock de Q nas instalações S_1 e S_3 , enquanto a transação T_2 pode ter sucesso no lock de Q nas instalações S_2 e S_4 . Cada um, então, precisa esperar para adquirir o terceiro lock, e por isso ocorre um deadlock.

18.4.1.4 Protocolo parcial

O protocolo parcial é semelhante ao protocolo da maioria. A diferença é que as requisições para os locks compartilhados recebem um tratamento mais favorável do que as requisições para locks exclusivos. O sistema mantém um gerenciador de locks em cada instalação. Cada gerenciador controla os locks para todos os itens de dados armazenados nessa instalação. Os locks compartilhados e exclusivos são tratados de formas diferentes.

- **Locks compartilhados.** Quando uma transação precisa efetuar o lock do item de dados Q , ela requisita um lock sobre Q do gerenciador de locks em uma instalação contendo uma réplica de Q .
- **Locks exclusivos.** Quando uma transação precisa efetuar o lock do item de dados Q , ela requisita um lock sobre Q do gerenciador de locks em todas as instalações contendo uma réplica de Q .

Como antes, a resposta à requisição é adiada até a requisição poder ser concedida.

O esquema tem a vantagem de impor menos custo adicional sobre operações de leitura do que o protocolo da maioria. A vantagem é especialmente significativa em casos comuns, em que a frequência de leituras é muito maior do que a frequência de escritas. Contudo, o custo adicional sobre as escritas é uma desvantagem. Além do mais, o protocolo parcial compartilha uma desvantagem existente no protocolo da maioria, que é a complexidade no tratamento do deadlock.

18.4.1.5 Cópia principal

Outra alternativa ainda é escolher uma das réplicas como cópia principal. Assim, para cada item de dados Q , a cópia principal de Q precisa residir em exatamente uma instalação, que denominamos *instalação principal de Q* . Quando uma transação precisa efetuar o lock em um item de dados Q , ela requisita um lock na instalação principal de Q . Novamente, a resposta à requisição é adiada até a requisição poder ser concedida.

Esse esquema nos permite tratar o controle de concorrência para os dados replicados de maneira muito semelhante à dos dados não replicados. A implementação do método é simples. Entretanto, se a instalação principal de Q falhar, Q fica inacessível, embora as outras instalações que contêm uma réplica possam estar acessíveis.

18.4.2 Estampa de tempo

A ideia principal por trás do esquema de estampa de tempo, discutido na [Seção 6.10](#), é que cada transação receba uma estampa de tempo *exclusiva*, que é usada para decidir a ordem de serialização. Nossa primeira tarefa na generalização do esquema centralizado para um esquema

distribuído é desenvolver um esquema para gerar estampas de tempo exclusivas. Nossos protocolos anteriores podem, então, ser aplicados diretamente ao ambiente não replicado.

18.4.2.1 Geração de estampas de tempo exclusivas

Dois métodos principais são usados para gerar estampas de tempo exclusivas; um é centralizado e um é distribuído. No centralizado, uma única instalação é escolhida para distribuir as estampas de tempo. A instalação pode usar um contador lógico ou seu próprio relógio local para essa finalidade.

No esquema distribuído, cada instalação gera uma estampa de tempo local exclusiva, usando um contador lógico ou o relógio local. A estampa de tempo exclusiva global é obtida pela concatenação da estampa de tempo local exclusiva com o identificador da instalação, que precisa ser exclusivo ([Figura 18.2](#)). A ordem da concatenação é importante! Usamos o identificador da instalação na posição menos significativa para garantir que as estampas de tempo globais geradas em uma instalação nem sempre sejam maiores do que aquelas geradas em outra instalação. Compare essa técnica para gerar estampas de tempo exclusivas com aquela que apresentamos na [Seção 18.1.2](#), para gerar nomes exclusivos.

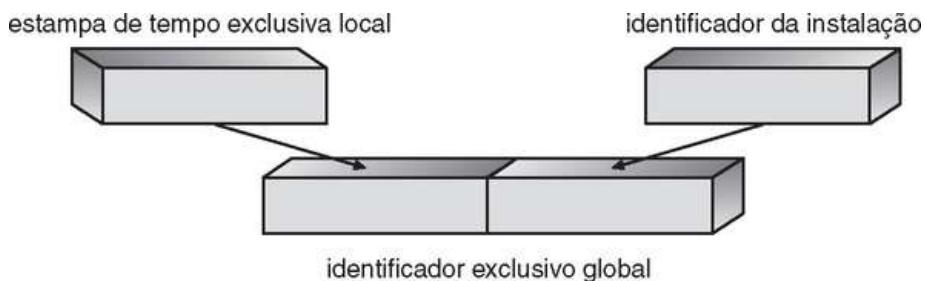


FIGURA 18.2 Geração de estampas de tempo exclusivas.

Ainda podemos ter um problema se uma instalação gerar estampas de tempo locais em uma velocidade mais rápida do que outras instalações. Nesse caso, o contador lógico da instalação rápida será maior do que o das outras instalações. Portanto, todas as estampas de tempo geradas pela instalação rápida serão maiores do que aquelas geradas por outras instalações. É preciso haver um mecanismo que garanta que as estampas de tempo locais sejam geradas de forma justa por meio do sistema. Para realizar essa geração justa de estampas de tempo, definimos dentro de cada instalação S_i um relógio lógico (RL_i), que gera a estampa de tempo exclusiva local ([Seção 18.1.2](#)). Para garantir que os diversos relógios lógicos estejam sincronizados, exigimos que uma instalação S_i avance seu relógio lógico sempre que uma transação T_i , com estampa de tempo $\langle x,y \rangle$, visitar essa instalação e x for maior do que o valor atual de RL_i . Nesse caso, a instalação S_i avança seu relógio lógico para o valor $x + 1$.

Se o relógio do sistema for utilizado para gerar as estampas de tempo, então elas são atribuídas de forma justa, desde que nenhuma instalação tenha um relógio do sistema que atrasse ou adiante. Como os relógios podem não ser perfeitamente precisos, uma técnica semelhante à usada para os relógios lógicos precisa ser usada para garantir que nenhum relógio fique muito adiante ou muito atrás de outro relógio.

18.4.2.2 Esquema de ordenação de estampa de tempo

O esquema básico de estampa de tempo, introduzido na [Seção 6.10.4.3](#), pode ser estendido de maneira direta para um sistema distribuído. Assim como no caso centralizado, a propagação de reversões (rollbacks) poderá acontecer se nenhum mecanismo for utilizado para impedir que uma transação leia um valor de item de dados que ainda não tenha sido confirmado. Para eliminar a propagação de rollbacks, podemos combinar o esquema básico de estampa de tempo da [Seção 6.10](#) com o protocolo 2PC da [Seção 18.3](#), para obter um protocolo que garanta a serialização sem propagação de rollbacks. Deixamos com você o desenvolvimento de tal algoritmo.

O esquema básico da estampa de tempo recém-descrito sofre com a propriedade indesejável de que os conflitos entre as transações são resolvidos por meio de rollbacks, em vez de esperas. Para aliviar esse problema, podemos manter em buffer várias operações read e write (ou seja, *adiá-las*) até um momento em que tenhamos certeza de que essas operações podem ocorrer sem ocasionar cancelamentos. Uma operação $read(x)$ por T_i precisa ser adiada se existir uma transação T_j que realizará uma operação $write(x)$, mas ainda não o fez e $ET(T_j) < ET(T_i)$. De modo semelhante, uma operação $write(x)$ por T_i precisa ser adiada se houver uma transação T_j que realize uma operação $read(x)$ ou $write(x)$ e $ET(T_j) < ET(T_i)$. Existem diversos métodos para garantir essa propriedade. Um deles, denominado **esquema conservador de ordenação da estampa de tempo**, exige que cada instalação mantenha uma fila de leitura e de escrita consistindo em todas as requisições read e write que devam ser executadas na instalação e que precisem ser adiadas para preservar a

propriedade acima. Não apresentaremos esse esquema aqui. Em vez disso, deixamos o desenvolvimento do algoritmo para você.

18.5 Tratamento de deadlock

Os algoritmos de prevenção de deadlock, impedimento de deadlock e detecção de deadlock, apresentados no [Capítulo 7](#), podem ser estendidos também para serem usados em um sistema distribuído. Nesta seção, descrevemos diversos desses algoritmos distribuídos.

18.5.1 Prevenção e impedimento de deadlock

Os algoritmos de prevenção de deadlock e impedimento de deadlock apresentados no [Capítulo 7](#) também podem ser usados em um sistema distribuído, desde que sejam feitas as modificações necessárias. Por exemplo, podemos usar a técnica de prevenção de deadlock por ordenação de recursos definindo uma ordem global entre os recursos do sistema, ou seja, todos os recursos no sistema inteiro recebem números exclusivos e um processo pode requisitar um recurso (em qualquer processador) com número exclusivo i somente se não estiver mantendo um recurso com um número exclusivo maior do que i . Da mesma forma, podemos usar o algoritmo do banqueiro em um sistema distribuído designando um dos processos no sistema (o banqueiro) como o processo que mantém as informações necessárias para executar o algoritmo do banqueiro. Cada requisição de recurso precisa ser canalizada pelo banqueiro.

O esquema global de prevenção de deadlock por ordenação de recursos é simples de implementar em um ambiente distribuído e exige pouco trabalho adicional. O algoritmo do banqueiro também pode ser implementado com facilidade, mas pode exigir muito mais trabalho adicional. O banqueiro pode se tornar um gargalo no sistema, pois a quantidade de mensagens de e para o banqueiro pode ser muito grande. Assim, o esquema do banqueiro parece não ser muito prático em um sistema distribuído.

Agora nos voltamos para um novo esquema de prevenção de deadlock, baseado na técnica de ordenação da estampa de tempo com preempção de recursos. Embora essa técnica possa lidar com a situação de deadlock que pode surgir em um sistema distribuído, para simplificar consideraremos apenas o caso de uma única instância de cada tipo de recurso.

Para controlar a preempção, atribuímos um número de prioridade exclusivo a cada processo. Esses números são usados para decidir se um processo P_i deve esperar por um processo P_j . Por exemplo, podemos deixar P_i esperar por P_j se P_i tiver uma prioridade maior do que a de P_j ; caso contrário, P_i sofre um rollback. Esse esquema impede deadlocks porque, para cada aresta $P_i \rightarrow P_j$ no grafo de espera, P_i possui uma prioridade maior do que P_j . Assim, um ciclo não poderá existir.

Uma dificuldade com esse esquema é a possibilidade de starvation. Alguns processos com prioridade extremamente baixa sempre poderão sofrer um rollback. Essa dificuldade pode ser evitada por meio das estampas de tempo. Cada processo no sistema recebe uma estampa de tempo exclusiva quando criado. Dois esquemas de prevenção de deadlock complementares, usando estampas de tempo, foram propostos:

1. **O esquema wait-die.** Essa técnica é baseada em uma técnica não preemptiva. Quando o processo P_i requisita um recurso mantido atualmente por P_j , P_i só tem permissão para esperar se tiver uma estampa de tempo menor do que P_j (ou seja, P_i é mais antigo que P_j). Caso contrário, P_i sofre um rollback (morre). Por exemplo, suponha que os processos P_1 , P_2 e P_3 tenham estampas de tempo 5, 10 e 15, respectivamente. Se P_1 requisitar um recurso mantido por P_2 , P_1 esperará. Se P_3 requisitar um recurso mantido por P_2 , P_3 sofrerá um rollback.
2. **O esquema wound-wait.** Essa técnica é baseada em uma técnica preemptiva e é o complemento do sistema wait-die. Quando o processo P_i requisita um recurso mantido atualmente por P_j , P_i só tem permissão para esperar se tiver uma estampa de tempo maior do que P_j (ou seja, P_i é mais recente que P_j). Caso contrário, P_j sofre um rollback (P_j é ferido por P_i). Retornando ao exemplo anterior, com os processos P_1 , P_2 e P_3 , se P_1 requisitar um recurso mantido por P_2 , então o recurso será preemptado por P_2 , e P_2 sofrerá um rollback. Se P_3 requisitar um recurso mantido por P_2 , então P_3 esperará.

Os dois esquemas podem evitar a starvation desde que, quando um processo sofrer um rollback, ele *não* receba uma nova estampa de tempo. Como as estampas de tempo sempre aumentam, um processo que tenha sofrido um rollback por fim terá a menor estampa de tempo. Assim, ele não sofrerá um rollback novamente. Contudo, existem diferenças significativas no modo como os dois esquemas operam.

- No esquema wait-die, um processo mais antigo precisa esperar até um mais novo liberar seu recurso. Assim, quanto mais antigo for o processo, mais ele costuma esperar. Por outro lado, no esquema wound-wait, um processo mais antigo nunca espera por um processo mais recente.
- No esquema wait-die, se um processo P_i morrer e sofrer um rollback porque requisitou um recurso mantido pelo processo P_j , então P_i pode emitir novamente a mesma sequência de requisições quando for reiniciado. Se o recurso ainda estiver sendo mantido por P_j , então P_i morrerá novamente. Assim, P_i pode morrer várias vezes antes de adquirir o recurso necessário.

Compare essa série com o que acontece no esquema wound-wait. O processo P_i é ferido e sofre um rollback porque P_j requisitou um recurso que mantém. Quando P_i for reiniciado e requisitar o recurso agora mantido por P_j , P_i esperará. Assim, menos reversões ocorrem no esquema wound-wait.

O maior problema com os dois esquemas é que rollbacks desnecessários poderão acontecer.

18.5.2 Detecção de deadlock

O algoritmo de prevenção de deadlock pode preemptar recursos mesmo que nenhum deadlock tenha ocorrido. Para evitar preempções desnecessárias, podemos usar um algoritmo de detecção de deadlock. Construímos um grafo de espera descrevendo o estado da alocação de recursos. Como estamos supondo haver apenas um único recurso de cada tipo, um ciclo no grafo de espera representa um deadlock.

O problema principal em um sistema distribuído é decidir como manter o grafo de espera. Ilustramos esse problema descrevendo várias técnicas comuns para lidar com esse problema. Esses esquemas exigem que cada instalação mantenha um grafo de espera *local*. Os nós do grafo correspondem a todos os processos (locais e não locais) atualmente mantendo ou requisitando qualquer um dos recursos locais a essa instalação. Por exemplo, na [Figura 18.3](#) temos um sistema consistindo em duas instalações, cada uma mantendo seu grafo de espera local. Observe que os processos P_2 e P_3 aparecem nos dois grafos, indicando que os processos requisitaram recursos nas duas instalações.

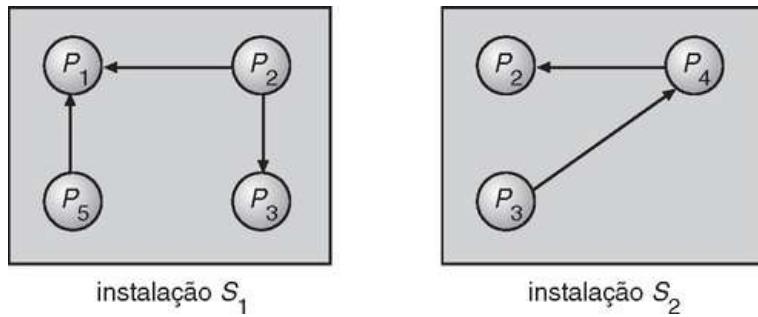


FIGURA 18.3 Dois grafos de espera locais.

Esses grafos de espera locais são construídos pelo modo normal para processos e recursos locais. Quando um processo P_i na instalação S_1 precisa de um recurso mantido pelo processo P_j na instalação S_2 , uma mensagem de requisição é enviada por P_i à instalação S_2 . A aresta $P_i \rightarrow P_j$, em seguida, é inserida no local do grafo de espera da instalação S_2 .

Nitidamente, se qualquer grafo de espera local tiver um ciclo, o deadlock ocorreu. Entretanto, o fato de não encontrarmos ciclos em qualquer um dos grafos de espera locais não significa que não existam deadlocks. Para ilustrar esse problema, considere o sistema representado na [Figura 18.3](#). Cada grafo de espera é acíclico; apesar disso, existe um deadlock no sistema. Para provar que um deadlock não ocorreu, temos de mostrar que a **união** de todos os grafos locais é acíclica. O grafo ([Figura 18.4](#)) que obtemos apanhando a união dos dois grafos de espera da [Figura 18.3](#) realmente contém um ciclo, indicando que o sistema está em estado de deadlock.

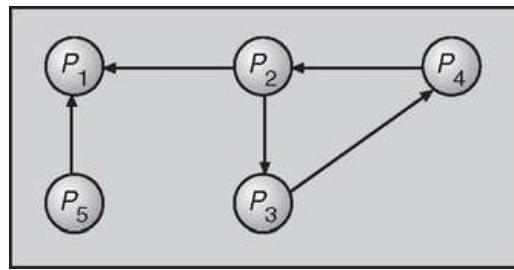


FIGURA 18.4 Grafo de espera global para a [Figura 18.3](#).

Existem vários métodos para organizar o grafo de espera em um sistema distribuído. Vamos descrever diversos esquemas comuns.

18.5.2.1 Técnica centralizada

Na técnica centralizada, um grafo de espera global é construído como a união de todos os grafos de

espera locais. Ele é mantido em um único processo: o **coordenador de detecção de deadlock**. Como existe retardo de comunicação no sistema, temos de distinguir entre dois tipos de grafos de espera. O grafo *real* descreve o estado real, porém desconhecido, do sistema em qualquer ponto no tempo, como seria visto por um observador onisciente. O grafo *construído* é uma aproximação gerada pelo coordenador durante a execução de seu algoritmo. O grafo construído precisa ser gerado de modo que, sempre que o algoritmo de detecção for envolvido, os resultados informados estejam corretos. Com *corretos*, queremos dizer o seguinte:

- Se existir um deadlock, então ele é informado corretamente.
 - Se um deadlock for informado, então o sistema está realmente em estado de deadlock. Conforme mostraremos, não é fácil construir esses algoritmos corretos.
- O grafo de espera pode ser construído em três pontos diferentes no tempo:
1. Sempre que uma nova borda é inserida ou removida de um dos grafos de espera locais.
 2. Periodicamente, quando uma série de mudanças tiver ocorrido em um grafo de espera.
 3. Sempre que o coordenador de detecção de deadlock precisar chamar o algoritmo de detecção de ciclo.

Quando o algoritmo de detecção de deadlock é chamado, o coordenador pesquisa seu grafo global. Se for encontrado um ciclo, uma *vítima* é selecionada para ser revertida. O coordenador precisa notificar a todas as instalações de que um processo em particular foi selecionado como vítima. As instalações, por sua vez, revertem o processo vítima.

Em seguida, vamos considerar cada uma das três opções de construção de gráfico listadas acima. Com a opção 1, sempre que uma aresta é inserida ou removida de um grafo local, a instalação local também precisa enviar uma mensagem ao coordenador para notificá-lo dessa modificação. Ao receber tal mensagem, o coordenador atualiza seu grafo global.

Como alternativa (opção 2), uma instalação pode enviar uma série dessas mudanças em uma única mensagem periodicamente. Retornando ao nosso exemplo anterior, o processo coordenador manterá o grafo de espera global conforme representado na [Figura 18.4](#). Quando a instalação S_2 inserir a aresta $P_3 \rightarrow P_4$ em seu grafo local, ela também enviará uma mensagem ao coordenador. De modo semelhante, quando a instalação S_1 excluir a aresta $P_5 \rightarrow P_1$ porque P_1 liberou um recurso requisitado por P_5 , uma mensagem apropriada será enviada ao coordenador.

Observe que, independentemente da opção, podem ocorrer reversões desnecessárias, como resultado de duas situações:

1. Podem existir **falsos ciclos** no grafo de espera global. Para ilustrar esse ponto, consideramos um instantâneo do sistema representado na [Figura 18.5](#). Suponha que P_2 libere o recurso que está mantendo na instalação S_1 , resultando na exclusão da borda $P_1 \rightarrow P_2$ na instalação S_1 . O processo P_2 , então, requisita um recurso mantido por P_3 na instalação S_2 , resultando no acréscimo da aresta $P_2 \rightarrow P_3$ na instalação S_2 . Se a mensagem *insert* $P_2 \rightarrow P_3$ da instalação S_2 chegar antes da mensagem *delete* $P_1 \rightarrow P_2$ da instalação S_1 , o coordenador pode descobrir o falso ciclo $P_1 \rightarrow P_2 \rightarrow P_3 \rightarrow P_1$ após o *insert* (mas antes do *delete*). A recuperação do deadlock pode ser iniciada, embora não tenha ocorrido deadlock algum.

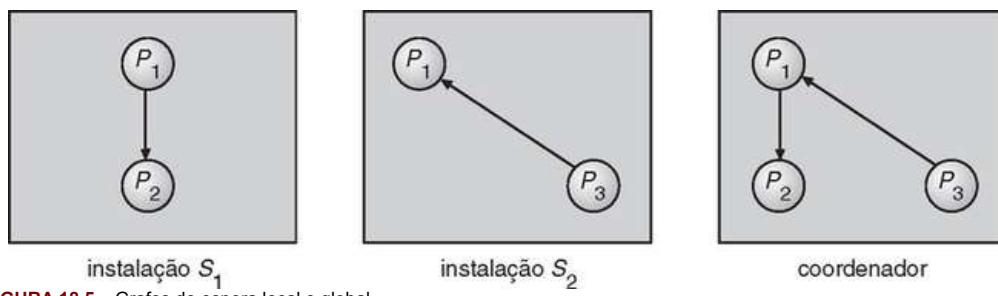


FIGURA 18.5 Grafos de espera local e global.

2. Reversões desnecessárias também podem acontecer quando um deadlock ocorreu e uma vítima foi escolhida, mas *ao mesmo tempo* um dos processos foi cancelado por motivos não relacionados com o deadlock (como quando o processo excede seu tempo limite). Por exemplo, suponha que a instalação S_1 da [Figura 18.3](#) decida cancelar P_2 . Ao mesmo tempo, o coordenador descobriu um ciclo e escolheu P_3 como vítima. Tanto P_2 quanto P_3 agora são revertidos, embora somente P_2 precisasse sofrer um rollback.

Agora, vamos apresentar um algoritmo de detecção de deadlock centralizado usando a opção 3, que detecta todos os deadlocks que ocorrem e não detecta falsos deadlocks. Para evitar o relato de falsos deadlocks, exigimos que as requisições de diferentes instalações recebam identificadores exclusivos (ou estampas de tempo). Quando o processo P_i na instalação S_1 requisitar um recurso de P_j na instalação S_2 , uma mensagem de requisição com estampa de tempo ET será enviada. A aresta

$P_i \rightarrow P_j$ com o rótulo ET é inserida no grafo de espera de S_1 . Essa aresta é inserida no grafo de espera local da instalação S_2 somente se a instalação S_2 tiver recebido a mensagem de requisição e não puder conceder imediatamente o recurso requisitado. Uma requisição de P_i para P_j na mesma instalação é tratada pelo modo normal; nenhuma estampa de tempo é associada à aresta $P_i \rightarrow P_j$.

O algoritmo de detecção é, portanto, o seguinte:

1. O controlador envia uma mensagem de início a cada instalação no sistema.
2. No recebimento dessa mensagem, uma instalação envia seu grafo de espera local ao coordenador. Cada um desses grafos de espera contém todas as informações locais que a instalação tem sobre o estado do grafo real. O grafo reflete um estado instantâneo da instalação, mas não está sincronizado em relação a qualquer outra instalação.
3. Quando o controlador tiver recebido uma réplica de cada instalação, ele construirá um grafo da seguinte forma:
 - a. O grafo construído contém um vértice para cada processo no sistema.
 - b. O grafo possui uma aresta P_i, P_j se e somente se houver uma aresta $P_i \rightarrow P_j$ em um dos grafos de espera ou uma aresta $P_i \rightarrow P_j$ com algum rótulo ET em mais de um grafo de espera.

Se o grafo construído tiver um ciclo, então o sistema está em estado de deadlock. Se o grafo construído não tiver um ciclo, então o sistema não estava em estado de deadlock quando o algoritmo de detecção foi chamado como resultado do início das mensagens enviadas pelo coordenador (na etapa 1).

18.5.2.2 Técnica totalmente distribuída

No **algoritmo de detecção de deadlock totalmente distribuído**, todos os controladores compartilham igualmente a responsabilidade de detectar o deadlock. Cada instalação constrói um grafo de espera que representa uma parte do grafo total, dependendo do comportamento dinâmico do sistema. A ideia é que, se houver um deadlock, um ciclo aparecerá em pelo menos um dos grafos parciais. Vamos apresentar um algoritmo desse tipo, que envolve a construção de grafos parciais em cada instalação.

Cada instalação mantém seu próprio grafo de espera local. Um grafo de espera local nesse esquema difere daquele descrito anteriormente, pois acrescentamos um nó adicional P_{ex} ao grafo. Um arco $P_i \rightarrow P_{ex}$ existe no grafo se P_i estiver esperando um item de dados em outra instalação, sendo mantido por *algum* processo. De modo semelhante, um arco $P_{ex} \rightarrow P_j$ existe no grafo se um processo em outra instalação estiver esperando para adquirir um recurso atualmente sendo mantido por P_j nessa instalação local.

Para ilustrar essa situação, consideremos os dois grafos de espera locais da Figura 18.3. O acréscimo do nó P_{ex} nos dois grafos resulta nos grafos de espera locais mostrados na Figura 18.6.

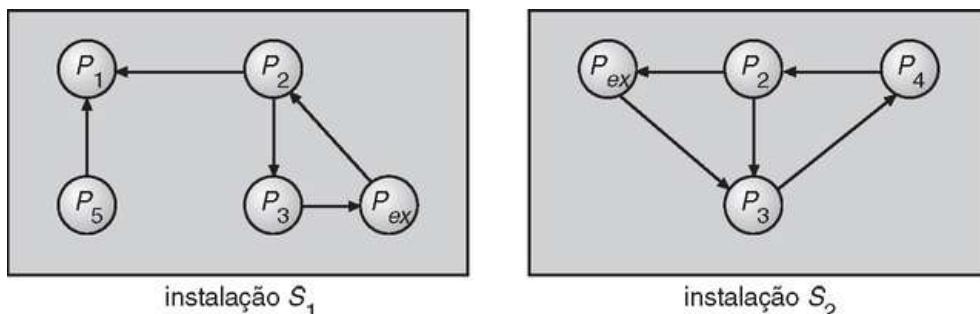


FIGURA 18.6 Grafos de espera locais aumentados da Figura 18.3.

Se um grafo de espera local tiver um ciclo que não envolva o nó P_{ex} , então o sistema está em estado de deadlock. Contudo, se um grafo local contém um ciclo envolvendo P_{ex} , isso significa a *possibilidade* de um deadlock. Para descobrir se existe um deadlock, temos de chamar um algoritmo de detecção de deadlock distribuído.

Suponha que, na instalação S_i , o grafo de espera local contenha um ciclo envolvendo o nó P_{ex} . Esse ciclo precisa estar no formato

$$P_{ex} \rightarrow P_{k1} \rightarrow P_{k2} \rightarrow \dots \rightarrow P_{kn} \rightarrow P_{ex}$$

que indica que o processo P_{kn} na instalação S_i está esperando para adquirir um item de dados localizado em alguma outra instalação - digamos, S_j . Ao descobrir esse ciclo, a instalação S_i envia à instalação S_j uma mensagem de detecção de deadlock contendo informações sobre esse ciclo.

Quando a instalação S_j recebe essa mensagem de detecção de deadlock, ela atualiza seu grafo de espera local com a nova informação. Depois, ela procura no grafo de espera recém-construído por

um ciclo não envolvendo P_{ex} . Se houver algum, um deadlock é localizado e um esquema de recuperação apropriado é chamado. Se um ciclo envolvendo P_{ex} for descoberto, então S_j transmite uma mensagem de detecção de deadlock à instalação apropriada – digamos, S_k . A instalação S_k , por sua vez, repete o procedimento. Assim, após um número finito de rodadas, um deadlock será descoberto ou a computação de detecção de deadlock termina.

Para ilustrar esse procedimento, consideremos os grafos de espera locais da [Figura 18.6](#). Suponha que a instalação S_1 descubra o ciclo

$$P_{ex} \rightarrow P_2 \rightarrow P_3 \rightarrow P_{ex}.$$

Como P_3 está esperando para adquirir um item de dados na instalação S_2 , uma mensagem de detecção de deadlock descrevendo esse ciclo é transmitida da instalação S_1 para a instalação S_2 . Quando a instalação S_2 receber essa mensagem, ela atualizará seu grafo de espera local, obtendo o grafo de espera da [Figura 18.7](#). Esse grafo contém o ciclo

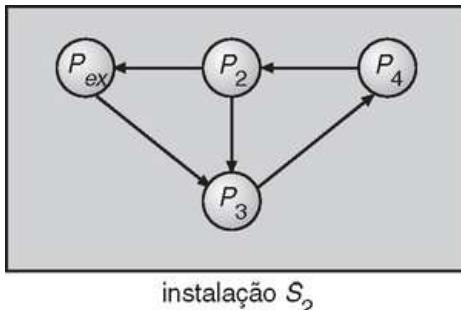


FIGURA 18.7 Grafo de espera local aumentado na instalação S_2 da [Figura 18.6](#).

$$P_2 \rightarrow P_3 \rightarrow P_4 \rightarrow P_2$$

que não inclui o nó P_{ex} . Portanto, o sistema está em estado de deadlock e um esquema de recuperação apropriado precisa ser chamado.

Observe que o resultado seria o mesmo se a instalação S_2 descobrisse o ciclo primeiro em seu grafo de espera local e enviasse a mensagem de detecção de deadlock para a instalação S_1 . No pior dos casos, as duas instalações descobrirão o ciclo praticamente ao mesmo tempo, e duas mensagens de detecção de deadlock serão enviadas: uma por S_1 para S_2 e outra por S_2 para S_1 . Essa situação resulta em transferência de mensagem e custo adicional desnecessários na atualização dos dois grafos de espera locais e procura de ciclos nos dois grafos.

Para reduzir o tráfego de mensagens, atribuímos a cada processo P_i um identificador exclusivo, que representamos com $ID(P_i)$. Quando a instalação S_k descobrir que seu grafo de espera local contém um ciclo envolvendo o nó P_{ex} no formato

$$P_{ex} \rightarrow P_{k1} \rightarrow P_{k2} \rightarrow \dots \rightarrow P_{kn} \rightarrow P_{ex}$$

ele enviará uma mensagem de detecção de deadlock para outra instalação somente se

$$ID(P_{kn}) < ID(P_{k1})$$

Caso contrário, a instalação S_k continua sua execução normal, deixando o trabalho de iniciar o algoritmo de detecção de deadlock para alguma outra instalação.

Para ilustrar esse esquema, consideremos novamente os grafos de espera mantidos nas instalações S_1 e S_2 da [Figura 18.6](#). Suponha que

$$ID(P_1) < ID(P_2) < ID(P_3) < ID(P_4)$$

Considere que as duas instalações descubram esses ciclos locais praticamente ao mesmo tempo. O ciclo na instalação S_1 tem o formato

$$P_{ex} \rightarrow P_2 \rightarrow P_3 \rightarrow P_{ex}$$

Como $ID(P_3) > ID(P_2)$, a instalação S_1 não envia uma mensagem de detecção de deadlock para a instalação S_2 .

O ciclo na instalação S_2 está no formato

$$P_{ex} \rightarrow P_3 \rightarrow P_4 \rightarrow P_2 \rightarrow P_{ex}$$

Como $ID(P_2) < ID(P_3)$, a instalação S_2 envia uma mensagem de detecção de deadlock para a instalação S_1 , que, ao receber a mensagem, atualiza seu grafo de espera local. A instalação S_1 , então, procura um ciclo no grafo e descobre que o sistema está em estado de deadlock.

18.6 Algoritmos de eleição

Conforme indicamos na [Seção 18.3](#), muitos algoritmos distribuídos empregam um processo coordenador que realiza funções necessárias a outros processos no sistema. Essas funções incluem impor a exclusão mútua, manter um grafo de espera global para detecção de deadlock, substituir um token perdido ou controlar um dispositivo de entrada ou saída no sistema. Se o processo coordenador falhar devido à falha da instalação em que reside, o sistema só pode continuar a execução reiniciando uma nova cópia do coordenador em alguma outra instalação. Os algoritmos que determinam onde uma nova cópia do coordenador deverá ser reiniciada são denominados **algoritmos de eleição**.

Os algoritmos de eleição consideram que um número de prioridade exclusivo está associado a cada processo ativo no sistema. Para facilitar a notação, consideramos que o número de prioridade do processo P_i é i . Para simplificar nossa discussão, consideramos uma correspondência biunívoca entre os processos e as instalações, e por isso nos referimos a ambos como processos. O coordenador sempre é o processo com o maior número de prioridade. Logo, quando um coordenador falha, o algoritmo precisa eleger aquele processo ativo com o maior número de prioridade. Esse número precisa ser enviado a cada processo ativo no sistema. Além disso, o algoritmo precisa prover um mecanismo para um processo recuperado identificar o coordenador ativo.

Nesta seção, apresentamos exemplos de algoritmos de eleição para duas configurações diferentes de sistemas distribuídos. O primeiro algoritmo se aplica a sistemas em que cada processo pode enviar uma mensagem a cada outro processo no sistema. O segundo algoritmo se aplica a sistemas organizados como um anel (lógica ou fisicamente). Os dois algoritmos exigem n^2 mensagens para uma eleição, onde n é o número de processos no sistema. Consideraremos que um processo que falhou sabe, na recuperação, que de fato falhou e, assim, toma as ações apropriadas para retornar ao conjunto de processos ativos.

18.6.1 O algoritmo bully

Suponha que o processo P_i envie uma requisição que não é respondida pelo coordenador dentro de um intervalo de tempo T . Nessa situação, considera-se que o coordenador falhou, e P_i tenta eleger a si mesmo como o novo coordenador. Essa tarefa é completada por meio do algoritmo a seguir.

O processo P_i envia uma mensagem de eleição a cada processo com um número de prioridade maior. O processo P_i , então, espera durante um intervalo de tempo T por uma resposta de qualquer um desses processos.

Se nenhuma resposta for recebida dentro do tempo T , P_i assume que todos os processos com números maiores do que i falharam e é eleito o novo coordenador. O processo P_i reinicia uma nova cópia do coordenador e envia uma mensagem para informar a todos os processos ativos com números de prioridade menor do que i que P_i é o novo coordenador.

Contudo, se uma resposta for recebida, P_i inicia um intervalo de tempo T' , esperando receber uma mensagem informando que um processo com um número de prioridade maior foi eleito. (Isto é, algum outro processo está se elegendo coordenador e precisa informar os resultados dentro do tempo T' .) Se nenhuma mensagem for enviada dentro de T' , então o processo com um número maior é considerado como tendo falhado, e o processo P_i deverá reiniciar o algoritmo.

Se P_i não for o coordenador, então, a qualquer momento durante a execução, P_i pode receber uma das duas mensagens do processo P_j a seguir:

1. P_j é o novo coordenador ($j > i$). O processo P_i , por sua vez, registra essa informação.
2. P_j iniciou uma eleição ($j < i$). O processo P_i envia uma resposta a P_j e inicia seu próprio algoritmo de eleição, desde que P_i ainda não tenha iniciado tal eleição.

O processo que completa seu algoritmo tem o número mais alto e é eleito como coordenador. Ele enviou seu número a todos os processos ativos com números menores. Após um processo que falhou se recuperar, ele imediatamente inicia a execução do mesmo algoritmo. Se não houver processos ativos com números mais altos, o processo recuperado força todos os processos com números menores a permitir que ele se torne o processo coordenador, mesmo que haja atualmente um coordenador ativo com um número menor. Por esse motivo, o algoritmo é denominado **algoritmo bully (valentão)**.

Vamos demonstrar a operação do algoritmo com um exemplo simples de um sistema consistindo em processos P_1 a P_4 . As operações são as seguintes:

1. Todos os processos estão ativos; P_4 é o processo coordenador.
2. P_1 e P_4 falham. P_2 determina que P_4 falhou enviando uma requisição que não é respondida dentro do tempo T . P_2 , então, inicia seu algoritmo de eleição enviando uma requisição a P_3 .
3. P_3 recebe a requisição, responde a P_2 e inicia seu próprio algoritmo enviando uma requisição de eleição a P_4 .

4. P_2 recebe a resposta de P_3 e começa a esperar por um intervalo T' .
5. P_4 não responde dentro de um intervalo T , de modo que P_3 se elege o novo coordenador e envia o número 3 a P_2 e P_1 (que P_1 não recebe, pois não está funcionando).
6. Depois, quando P_1 se recuperar, ele enviará uma requisição de eleição a P_2 , P_3 e P_4 .
7. P_2 e P_3 respondem a P_1 e iniciam seus próprios algoritmos de eleição. P_3 novamente será eleito, usando os mesmos eventos de antes.
8. Finalmente, P_4 se recupera e notifica a P_1 , P_2 e P_3 de que é o coordenador atual. (P_4 não envia requisições de eleição, pois é o processo com o número mais alto no sistema.)

18.6.2 Algoritmo de anel

O **algoritmo de anel** considera que os enlaces entre os processos são unidirecionais e que cada processo envia suas mensagens ao vizinho à sua direita. A estrutura de dados principal utilizada pelo algoritmo é a **lista ativa**, uma lista que contém os números de prioridade de todos os processos ativos no sistema quando o algoritmo termina; cada processo mantém sua própria lista ativa. O algoritmo funciona da seguinte maneira:

1. Se o processo P_i detectar uma falha no coordenador, ele cria uma nova lista ativa que está inicialmente vazia. Depois, ele envia uma mensagem $elect(i)$ ao seu vizinho da direita e acrescenta o número i à sua lista ativa.
2. Se P_i receber uma mensagem $elect(j)$ do processo à esquerda, ele terá de responder de uma dentre três maneiras:
 - a. Se essa for a primeira mensagem $elect$ que tiver visto ou enviado, P_i cria uma nova lista ativa com os números i e j . Depois, ele envia a mensagem $elect(i)$, seguida pela mensagem $elect(j)$.
 - b. Se $i \neq j$, ou seja, a mensagem recebida não contém o número de P_i , então P_i acrescenta j à sua lista ativa e encaminha a mensagem ao seu vizinho da direita.
 - c. Se $i = j$, ou seja, P_i recebe a mensagem $elect(i)$, então a lista ativa para P_i agora contém os números de todos os processos ativos no sistema. O processo P_i agora pode determinar o número maior na lista ativa, para identificar o novo processo coordenador.

Esse algoritmo não especifica como um processo em recuperação determina o número do processo coordenador atual. Uma solução exigiria que um processo em recuperação envie uma mensagem de pesquisa. Essa mensagem é encaminhada pelo anel até o coordenador atual, que, por sua vez, envia uma réplica contendo seu número.

18.7 Chegando ao acordo

Para um sistema ser confiável, precisamos de um mecanismo que permita a um conjunto de processos concordar sobre um *valor* comum. Esse acordo pode não ocorrer por vários motivos. Primeiro, o meio de comunicação pode falhar, resultando em mensagens perdidas ou ilegíveis. Segundo, os próprios processos podem falhar, resultando em comportamento imprevisível do processo. O melhor que podemos esperar, nesse caso, é que os processos falhem de uma forma limpa, interrompendo sua execução sem se desviarem do seu padrão de execução normal. No pior dos casos, os processos podem enviar mensagens ilegíveis ou incorretas a outros processos, ou até mesmo colaborar com outros processos que falharam em uma tentativa de destruir a integridade do sistema.

O **problema dos generais bizantinos** fornece uma analogia para essa situação. Várias divisões do exército bizantino, cada uma comandada por seu próprio general, cercam um acampamento inimigo. Os generais bizantinos precisam chegar a um acordo sobre atacar ou não atacar o inimigo ao amanhecer. É fundamental que todos os generais concordem, pois um ataque por somente uma das divisões resultaria em derrota. As diversas divisões estão geograficamente dispersas e os generais só podem se comunicar entre si com mensageiros que correm de um acampamento a outro. Os generais não conseguem chegar a um acordo pelo menos por dois motivos:

1. Os mensageiros podem ser apanhados pelo inimigo e, assim, podem não conseguir entregar suas mensagens. Essa situação corresponde à comunicação não confiável em um sistema computadorizado e é discutida melhor na [Seção 18.7.1](#).
2. Os generais podem ser *traidores*, tentando impedir que os generais *leais* cheguem a um acordo. Essa situação corresponde a processos defeituosos em um sistema computadorizado e é discutida melhor na [Seção 18.7.2](#).

18.7.1 Comunicações não confiáveis

Vamos supor que, se os processos falham, eles fazem isso de uma forma limpa e que o meio de comunicação não é confiável. Suponha que o processo P_i na instalação S_1 , que enviou uma mensagem ao processo P_j na instalação S_2 , precisa saber se P_j recebeu a mensagem, de modo que possa decidir como prosseguir com sua computação. Por exemplo, P_i pode decidir calcular uma função *foo* se P_j tiver recebido sua mensagem ou calcular uma função *boo* se P_j não tiver recebido a mensagem (devido a alguma falha no hardware).

Para detectar falhas, podemos usar um **esquema de time-out** semelhante ao descrito na [Seção 16.7.1](#). Quando P_i envia uma mensagem, ele também especifica um intervalo de tempo durante o qual pretende esperar por uma mensagem de confirmação de P_j . Quando P_j recebe a mensagem, ele imediatamente envia uma confirmação para P_i . Se P_i receber uma mensagem de confirmação dentro do intervalo de tempo especificado, ele pode seguramente concluir que P_j recebeu sua mensagem. Entretanto, se um tempo-límite for esgotado, então P_i precisa retransmitir sua mensagem e esperar por uma confirmação. Esse procedimento continua até que P_i receba a mensagem de confirmação, isto é, seja notificado pelo sistema de que a instalação S_2 está parada. No primeiro caso, ele calculará *foo*; no segundo, ele calculará *boo*. Observe que, se essas forem as duas únicas alternativas viáveis, P_i precisa esperar até ter sido notificado de que uma das situações ocorreu.

Agora, suponha que P_j também precise saber que P_i recebeu sua mensagem de confirmação, para poder decidir como prosseguir com sua computação. Por exemplo, P_j pode querer calcular *foo* somente se tiver certeza de que P_i recebeu sua confirmação. Em outras palavras, P_i e P_j calcularão *foo* se e somente se os dois tiverem concordado com isso. Acontece que, na presença de uma falha, não é possível realizar essa tarefa. Mais precisamente, não é possível, em um ambiente distribuído, que os processos P_i e P_j concordem completamente sobre seus respectivos estados.

Para provar essa afirmação, vamos supor que exista uma sequência mínima de transferências de mensagens de modo que, após as mensagens terem sido entregues, os dois processos concordem em calcular *foo*. Seja m' a última mensagem enviada por P_i a P_j . Como P_i não sabe se essa mensagem chegará a P_j (pois a mensagem pode se perder, devido a uma falha), P_i executará *foo* independentemente do resultado da entrega da mensagem. Assim, m' poderá ser removida da sequência sem afetar o procedimento de decisão. Logo, a sequência original não foi mínima, contradizendo nossa suposição e mostrando que não existe uma sequência. Os processos nunca podem ter certeza de que os dois calcularão *foo*.

18.7.2 Processos defeituosos

Vamos supor que o meio de comunicação seja confiável, mas que os processos possam falhar de maneira imprevisível. Considere um sistema com n processos, dos quais não mais do que m são defeituosos. Suponha que cada processo P_i tenha algum valor privado V_i . Queremos projetar um

algoritmo que permita a cada processo não defeituoso P_i construir um vetor $X_i = (A_{i,1}, A_{i,2}, \dots, A_{i,n})$, de modo que existam as seguintes condições:

1. Se P_j for um processo não defeituoso, então $A_{i,j} = V_j$.
2. Se P_i e P_j forem processos não defeituosos, então $X_i = X_j$.

Existem muitas soluções para esse problema, e elas compartilham as seguintes propriedades:

1. Um algoritmo correto pode ser criado somente se $n \geq 3 \times m + 1$.
2. O atraso no pior caso para chegar a um acordo é proporcional a $m + 1$ atrasos na troca de mensagens.
3. O número de mensagens exigido para se chegar a um acordo é grande. Nenhum processo isolado é confiável, de modo que todos os processos precisam coletar todas as informações e tomar suas próprias decisões.

Em vez de apresentar uma solução geral, que seria complicada, apresentamos um algoritmo para o caso simples onde $m = 1$ e $n = 4$. O algoritmo exige dois turnos de troca de informações:

1. Cada processo envia seu valor privado a outros três processos.
2. Cada processo envia as informações que obteve no primeiro turno para todos os outros processos.

Um processo defeituoso, obviamente, pode se recusar a enviar mensagens. Nesse caso, um processo não defeituoso pode escolher um valor qualquer e fingir que esse valor foi enviado por esse processo.

Quando esses dois turnos forem completados, um processo não defeituoso P_i pode construir seu vetor $X_i = (A_{i,1}, A_{i,2}, A_{i,3}, A_{i,4})$ da seguinte maneira:

1. $A_{i,j} = V_i$.
2. Para $j \neq i$, se pelo menos dois dos três valores informados para o processo P_j (nos dois turnos da troca) concordarem, então o valor da maioria é usado para o valor de $A_{i,j}$. Caso contrário, um valor-padrão, digamos, *nil*, será usado para definir o valor de $A_{i,j}$.

18.8 Resumo

Em um sistema distribuído sem memória comum e sem relógio comum, às vezes é impossível determinar a ordem exata em que dois eventos ocorrem. A relação *ocorreu antes* é apenas uma ordenação parcial dos eventos nos sistemas distribuídos. As estampas de tempo podem ser usadas para prover uma ordenação de evento coerente em um sistema distribuído.

A exclusão mútua em um ambiente distribuído pode ser implementada de diversas maneiras. Em um enfoque centralizado, um dos processos no sistema é escolhido para coordenar a entrada para a seção crítica. No enfoque totalmente distribuído, a tomada de decisão é distribuída pelo sistema inteiro. Um algoritmo distribuído, que se aplica a redes estruturadas em anel, é a técnica de passagem de tokens.

Para que a atomicidade seja garantida, todas as instalações em que uma transação T executou precisam combinar com o resultado final da execução. T é confirmada em todas as instalações ou cancelada em todas as instalações. Para garantir essa propriedade, o coordenador de transações de T precisa executar um protocolo de confirmação. O protocolo de confirmação mais usado é o protocolo 2PC.

Os diversos esquemas de controle de concorrência que podem ser usados em um sistema centralizado podem ser modificados para uso em um ambiente distribuído. No caso dos protocolos de lock, precisamos apenas mudar a forma como o gerenciador de lock é implementado. No caso de esquemas de estampa de tempo e validação, a única mudança necessária é o desenvolvimento de um mecanismo para gerar estampas de tempo globais exclusivas. O mecanismo pode concatenar uma estampa de tempo local com a identificação da instalação ou avançar a hora dos relógios locais sempre que uma mensagem chegar com uma estampa de tempo maior.

O método principal para lidar com deadlocks em um ambiente distribuído é a detecção de deadlock. O problema principal é decidir como manter o grafo de espera. Métodos para organizar o grafo de espera incluem um enfoque centralizado e um enfoque totalmente distribuído.

Alguns dos algoritmos distribuídos exigem o uso de um coordenador. Se o coordenador falhar por causa da falha da instalação em que reside, o sistema só pode continuar a execução reiniciando uma nova cópia do coordenador ou alguma outra instalação. Ele faz isso mantendo um coordenador de backup que esteja pronto para assumir a responsabilidade se o coordenador falhar. Outra técnica é escolher o novo coordenador após o coordenador ter falhado. Os algoritmos que determinam onde uma nova cópia do coordenador deverá ser reiniciada são chamados algoritmos de eleição. Dois algoritmos, o algoritmo bully e um algoritmo de anel, podem ser usados para eleger um novo coordenador no caso de falhas.

Exercícios

- 18.1. Discuta as vantagens e desvantagens dos dois métodos que apresentamos para gerar estampas de tempo globalmente exclusivas.
- 18.2. O esquema de estampa de tempo de relógio lógico apresentado neste capítulo fornece a seguinte garantia: se o evento A acontece antes do evento B, então a estampa de tempo de A é menor que a estampa de tempo de B. Contudo, observe que não se pode ordenar dois eventos com base apenas em suas estampas de tempo. O fato de que um evento C tem uma estampa de tempo menor que a estampa de tempo do evento D não significa necessariamente que o evento aconteceu antes do evento D; C e D poderiam ser eventos concorrentes no sistema. Discuta maneiras pelas quais o esquema de estampa de tempo de relógio lógico poderia ser estendido para distinguir eventos concorrentes dos eventos que podem ser ordenados pelo relacionamento *acontece-antes*.
- 18.3. Sua empresa está montando uma rede de computadores e você foi requisitado a escrever um algoritmo para conseguir a exclusão mútua distribuída. Que esquema você usaria? Explique sua escolha.
- 18.4. Por que a detecção de deadlock é muito mais dispendiosa em um ambiente distribuído do que em um ambiente centralizado?
- 18.5. Sua empresa está montando uma rede de computadores e você foi requisitado a desenvolver um esquema para lidar com o problema de deadlock.
- a. Você usaria um esquema de detecção de deadlock ou um esquema de prevenção de deadlock?
 - b. Se você tivesse de usar um esquema de prevenção de deadlock, qual você usaria? Explique sua escolha.
 - c. Se você tivesse de usar um esquema de detecção de deadlock, qual você usaria? Explique sua escolha.
- 18.6. Sob quais circunstâncias o esquema wait-die funciona melhor que o esquema wound-wait para conceder recursos a transações executando simultaneamente?
- 18.7. Considere as técnicas centralizada e totalmente distribuída para a detecção de deadlock. Compare os dois algoritmos em termos de complexidade de mensagem.
- 18.8. Considere um algoritmo de detecção de deadlock *hierárquico*, em que o grafo de espera global está distribuído por uma série de *controladores* diferentes, que são organizados em uma árvore. Cada controlador não folha mantém um grafo de espera que contém informações relevantes dos grafos dos controladores na subárvore abaixo dele. Em particular, considere S_A , S_B e S_C os controladores, de modo que S_C é o ancestral comum mais baixo de S_A e S_B (S_C precisa ser exclusivo, pois estamos lidando com uma árvore). Suponha que o nó T_i apareça no grafo de espera local dos controladores S_A e S_B . Então, T_i também precisa aparecer no grafo de espera local de
- Controlador S_C .
 - Cada controlador no caminho de S_C para S_A .
 - Cada controlador no caminho de S_C para S_B .
 - Além disso, se T_i e T_j aparecem no grafo de espera do controlador S_D e existe um caminho de T_i para T_j no grafo de espera de um dos filhos de S_D , então uma aresta $T_i \rightarrow T_j$ precisa estar no grafo de espera de S_D .
- Mostre que, se houver um ciclo em qualquer um dos grafos de espera, então o sistema está em deadlock.
- 18.9. Derive um algoritmo de eleição para anéis bidirecionais que seja mais eficiente do que aquele apresentado neste capítulo. Quantas mensagens são necessárias para n processos?
- 18.10. Considere uma configuração onde os processadores não estão associados a identificadores exclusivos, mas o número total de processadores é conhecido e os processadores são organizados ao longo de um anel bidirecional. É possível derivar um algoritmo de eleição para essa configuração?
- 18.11. Considere uma falha que ocorre durante o 2PC para uma transação. Para cada falha possível, explique como o 2PC garante a atomicidade da transação, apesar da falha.
- 18.12. Considere o seguinte modelo de falha para processadores com falha. Os processadores seguem o protocolo prescrito, mas poderiam falhar em pontos inesperados no tempo. Quando os processadores falham, eles deixam de funcionar e não continuam a participar do sistema distribuído. Dado esse modelo de falha, projete um algoritmo para chegar a um acordo entre um conjunto de processadores. Discuta as condições sob as quais o acordo poderia ser chegado.

Notas bibliográficas

O algoritmo distribuído para estender a relação *ocorre antes* em uma ordenação total coerente de todos os eventos no sistema foi desenvolvido por [Lamport \[1978b\]](#). Outras discussões do uso do tempo lógico para caracterizar o comportamento dos sistemas distribuídos podem ser encontradas em [Fidge \[1991\]](#), [Raynal e Singhal \[1996\]](#), [Babaoglu e Marzullo \[1993\]](#), [Schwarz e Mattern \[1994\]](#) e [Mattern \[1988\]](#).

O primeiro algoritmo geral para implementar a exclusão mútua em um ambiente distribuído foi desenvolvido também por [Lamport \[1978b\]](#). O esquema de Lamport exige $3 \times (n - 1)$ mensagens por entrada na seção crítica. Subsequentemente, [Ricart e Agrawala \[1981\]](#) propuseram um algoritmo distribuído que exige apenas $2 \times (n - 1)$ mensagens. Seu algoritmo foi apresentado na [Seção 18.2.2](#). Um algoritmo de raiz quadrada para exclusão mútua distribuída foi apresentado por [Maekawa \[1985\]](#). O algoritmo de passagem de tokens para sistemas estruturados em anel apresentado na [Seção 18.2.3](#) foi desenvolvido por [Lann \[1977\]](#). [Carvalho e Roucairol \[1983\]](#) discutiram sobre exclusão mútua em redes de computadores e [Agrawal e Abbadi \[1991\]](#) descreveram uma solução eficiente e tolerante a falhas para a exclusão mútua distribuída. Uma taxonomia simples para algoritmos de exclusão mútua distribuída foi apresentada por [Raynal \[1991\]](#).

A questão do sincronismo distribuído foi discutida por [Reed e Kanodia \[1979\]](#) (ambiente de memória compartilhada), [Lamport \[1978b\]](#), [Lamport \[1978a\]](#) e [Schneider \[1982\]](#) (processos totalmente disjuntos). Uma solução distribuída para o problema dos filósofos na mesa de jantar foi apresentada por [Chang \[1980\]](#).

O protocolo 2PC foi desenvolvido por [Lampson e Sturgis \[1976\]](#) e [Gray \[1978\]](#). Duas versões modificadas do 2PC, denominadas “presumir confirmação” e “presumir cancelamento”, reduzem o custo adicional do 2PC definindo a suposição-padrão com relação ao destino das transações ([Mohan e Lindsay \[1993\]](#)).

Os artigos lidando com os problemas de implementação do conceito de transação em um banco de dados distribuído foram apresentados por [Gray \[1981\]](#), [Traiger e outros \[1982\]](#) e [Spector e Schwarz \[1983\]](#). Discussões abrangentes abordando controle de concorrência distribuído foram oferecidas por [Bernstein e outros \[1987\]](#). [Rosenkrantz e outros \[1978\]](#) relataram o algoritmo de prevenção de deadlock distribuído com estampa de tempo. O esquema de detecção de deadlock totalmente distribuído, apresentado na [Seção 18.5.2](#), foi desenvolvido por [Obermarck \[1982\]](#). O esquema de detecção de deadlock hierárquico do [Exercício 18.4](#) apareceu em [Menasce e Muntz \[1979\]](#). [Knapp \[1987\]](#) e [Singhal \[1989\]](#) forneceram um estudo sobre detecção de deadlock nos sistemas distribuídos. Deadlocks também podem ser detectados apanhando-se snapshots globais de um sistema distribuído, conforme é discutido em [Chandy e Lamport \[1985\]](#).

O problema dos generais bizantinos foi discutido por [Lamport e outros \[1982\]](#) e [Pease e outros \[1980\]](#). O algoritmo valentão foi apresentado por [Garcia-Molina \[1982\]](#) e o algoritmo de eleição para um sistema estruturado em anel foi escrito por [Lann \[1977\]](#).

PARTE VII

SISTEMAS DE USO GERAL

ESBOÇO

Capítulo 25: Introdução a Sistemas de uso geral

Capítulo 26: Sistemas de tempo real

Capítulo 27: Sistemas multimídia

Introdução a Sistemas de uso geral

Nossa abordagem sobre sistemas operacionais tem focalizado principalmente os sistemas computadorizados de uso geral. Contudo, existem sistemas de uso especial com requisitos diferentes daqueles dos sistemas que descrevemos.

Um *sistema de tempo real* é um sistema computadorizado que exige não apenas que os recursos computadorizados sejam corretos, mas também que os resultados sejam produzidos dentro de um período especificado. Os resultados produzidos após o prazo ter se esgotado - mesmo que corretos - podem não ter qualquer valor. Para tais sistemas, muitos algoritmos de escalonamento de sistema operacional tradicional precisam ser modificados para atender os prazos mais rigorosos.

Um *sistema multiusuário* precisa ser capaz de lidar não apenas com dados convencionais, como arquivos de texto, programas e documentos de processamento de textos, mas também com dados de multimídia. Os dados de multimídia consistem em dados de mídia contínua (áudio e vídeo), além de dados convencionais. Os dados de mídia contínua - como quadros de vídeo - precisam ser entregues de acordo com certas restrições de tempo (por exemplo, 30 quadros por segundo). As demandas do tratamento dos dados de mídia contínua exigem mudanças significativas na estrutura do sistema operacional, principalmente na gerência de memória, disco e rede.

CAPÍTULO 19

Sistemas de tempo real

Nossa abordagem sobre questões de sistema operacional até aqui tem focalizado, principalmente, os sistemas operacionais de uso geral (por exemplo, sistemas de desktop e servidor). Agora, voltamos nossa atenção para os sistemas de computação de tempo real. Os requisitos dos sistemas de tempo real diferem dos de muitos dos sistemas que descrevemos, principalmente porque os sistemas de tempo real precisam produzir resultados dentro de certos limites de tempo. Neste capítulo, fornecemos uma visão geral dos sistemas computadorizados de tempo real e descrevemos como os sistemas operacionais de tempo real precisam ser construídos para atender os requisitos de tempo rigorosos desses sistemas.

OBJETIVOS DO CAPÍTULO

- Explicar os requisitos de tempo dos sistemas de tempo real.
- Distinguir entre sistemas de tempo real rígidos e flexíveis.
- Discutir as características de definição dos sistemas de tempo real.
- Descrever os algoritmos de escalonamento para os sistemas de tempo real rígidos.

19.1 Aspectos básicos

Um **sistema de tempo real** é um sistema computadorizado que exige não apenas que os resultados da computação sejam “corretos”, mas também que os resultados sejam produzidos dentro de um período especificado. Os resultados produzidos após o prazo ter passado – mesmo que corretos – podem não ter qualquer valor real. Para ilustrar, considere um robô autônomo que entrega correspondência em um complexo de escritórios. Se o seu sistema de controle de visão identificar uma parede *depois* que o robô tiver caminhado para ela, apesar de identificar a parede corretamente, o sistema não terá atendido seu requisito. Compare esse requisito de tempo com as demandas muito menos estritas de outros sistemas. Em um sistema computadorizado de desktop interativo, é desejável fornecer um tempo de resposta rápido para o usuário interativo, mas não é obrigatório fazer isso. Alguns sistemas – como um sistema de processamento em batch – podem não ter requisito de temporização algum.

Os sistemas de tempo real executando no hardware de computador tradicional são usados em inúmeras aplicações. Além disso, muitos sistemas de tempo real estão embutidos em “dispositivos especializados”, como aparelhos domésticos comuns (por exemplo, fornos de micro-ondas e lavadoras de louças), dispositivos digitais do consumidor (por exemplo, câmeras fotográficas e MP3 players) e dispositivos de comunicação (por exemplo, telefones celulares e dispositivos portáteis BlackBerry). Eles também estão presentes em entidades maiores, como automóveis e aviões. Um **sistema embutido** é um dispositivo de computação que faz parte de um sistema maior, em que a presença de um dispositivo de computação normalmente não é óbvia para o usuário.

Para ilustrar, considere um sistema embutido para controlar uma lavadora de louças doméstica. O sistema embutido pode permitir várias opções para programar a operação da lavadora – a temperatura da água, o tipo de limpeza (leve ou pesada), até mesmo um timer indicando quando a lavadora deverá começar. Provavelmente, o usuário da lavadora não sabe que existe realmente um computador embutido no aparelho. Como outro exemplo, imagine um sistema embutido controlando os freios ABS em um automóvel. Cada roda no automóvel tem um sensor detectando quanto deslizamento e tração estão ocorrendo, e cada sensor envia continuamente seus dados para o controlador do sistema. Apanhando os resultados desses sensores, o controlador diz ao mecanismo de freio em cada roda quanta pressão de freio deve ser aplicada. Novamente, para o usuário (neste caso, o motorista do automóvel), a presença de um sistema computadorizado embutido pode não ser aparente. Porém, é importante observar que nem todos os sistemas embutidos são de tempo real. Por exemplo, um sistema embutido controlando um forno doméstico pode não ter qualquer requisito de tempo real.

Alguns sistemas de tempo real são identificados como **sistemas de segurança crítica**. Em um sistema de segurança crítica, a operação incorreta – normalmente devido a um prazo não atendido – resulta em algum tipo de “catástrofe”. Alguns exemplos de sistemas de segurança crítica incluem sistemas de armas, sistemas de freio ABS, sistemas de tráfego aéreo e sistemas embutidos relacionados com a saúde, como marca-passos. Nesses cenários, o sistema de tempo real *precisa* responder aos eventos pelos prazos especificados; caso contrário, um dano sério – ou pior – pode acontecer. Porém, uma maioria significativa dos sistemas embutidos não se qualifica como de segurança crítica, incluindo máquinas de FAX, fornos de micro-ondas, relógios de pulso e dispositivos de rede como switches e roteadores. Para esses dispositivos, requisitos de prazo não cumpridos resultam em, talvez, nada mais do que um usuário insatisfeito.

A computação de tempo real pode ser de dois tipos: rígida e flexível. Um **sistema de tempo real rígido** tem os requisitos mais rigorosos, garantindo que as tarefas de tempo real críticas sejam completadas dentro de seus prazos. Os sistemas de segurança crítica normalmente são sistemas de tempo real rígido. Um **sistema de tempo real flexível** é menos restritivo, simplesmente fazendo uma tarefa de tempo real crítica receber prioridade sobre outras tarefas e que ela retenha essa prioridade até que termine. Muitos sistemas operacionais comerciais – bem como o Linux – fornecem suporte para tempo real flexível.

19.2 Características do sistema

Nesta seção, exploramos as características dos sistemas de tempo real e discutimos sobre problemas relacionados com o projeto de sistemas operacionais de tempo real flexíveis e rígidos.

As características a seguir são típicas de muitos sistemas de tempo real:

- Finalidade única.
- Pequeno tamanho.
- Produzidos em massa com pouco custo.
- Requisitos de tempo específicos.

A seguir, examinamos cada uma dessas características.

Diferente dos PCs, que possuem muitos usos, um sistema de tempo real normalmente atende apenas a uma única finalidade, como controlar freios ABS ou entregar música em um player MP3. É pouco provável que um sistema de tempo real controlando o sistema de navegação de uma companhia aérea também leia DVDs! O projeto de um sistema operacional de tempo real reflete sua natureza de única finalidade e normalmente é muito simples.

Muitos sistemas de tempo real existem em ambientes onde o espaço físico é restrito. Considere a quantidade de espaço disponível em um relógio ou em um forno de micro-ondas - ele é consideravelmente menor que o que existe em um computador de desktop. Como resultado das restrições de espaço, a maioria dos sistemas de tempo real não possui o poder de processamento da CPU nem a quantidade de memória disponível nos PCs comuns de desktop. Embora a maioria dos sistemas contemporâneos de desktop e servidor utilize processadores de 32 ou 64 bits, muitos sistemas de tempo real utilizam processadores de 8 ou 16 bits. De modo semelhante, um PC desktop poderia ter vários gigabytes de memória física, enquanto um sistema de tempo real poderia ter menos de um megabyte. Vamos nos referir à **pegada** de um sistema como a quantidade de memória exigida para executar o sistema operacional e suas aplicações. Como a quantidade de memória é limitada, a maioria dos sistemas operacionais de tempo real precisa ter pegadas pequenas.

Em seguida, considere onde muitos sistemas de tempo real são implementados. Eles normalmente são encontrados em aparelhos domésticos e dispositivos de consumidor. Dispositivos como câmeras digitais, fornos de micro-ondas e termostatos são produzidos em massa em ambientes bastante conscientes do custo. Assim, os microprocessadores para sistemas de tempo real também precisam ser produzidos em massa a um baixo custo.

Uma maneira de reduzir o custo de um controlador embutido é usar uma técnica alternativa para organizar os componentes do sistema computadorizado. Em vez de organizar o computador em torno da estrutura mostrada na [Figura 19.1](#), onde os barramentos fornecem o mecanismo de interconexão para componentes individuais, muitos controladores de sistemas embutidos utilizam uma estratégia conhecida como **sistema no chip** (**System-On-Chip - SOC**). Aqui, a CPU, a memória (incluindo cache), a unidade de gerência de memória (Memory Management Unit - MMU) e quaisquer portas periféricas conectadas, como portas USB, estão contidas em um único circuito integrado. A estratégia SOC normalmente é menos dispendiosa do que a organização orientada a barramento da [Figura 19.1](#).

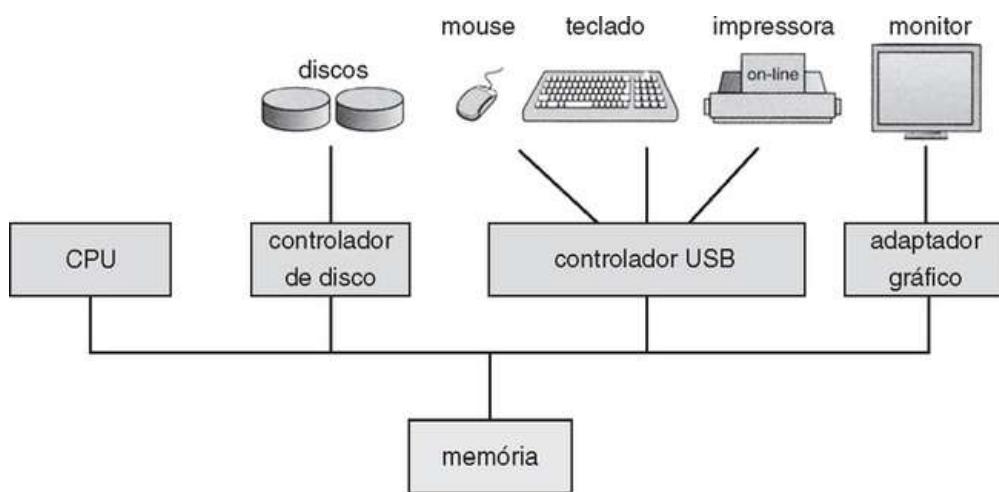


FIGURA 19.1 Organização orientada a barramento.

Agora passamos para a característica final, já identificada para os sistemas de tempo real: requisitos de tempo específicos. De fato, essa é a característica de definição de tais sistemas. Consequentemente, a principal tarefa dos sistemas operacionais de tempo real rígidos e flexíveis é o suporte aos requisitos de tempo usando algoritmos de escalonamento que dão aos processos de

tempo real as mais altas prioridades de escalonamento. Além do mais, os escalonadores precisam garantir que a prioridade de uma tarefa em tempo real não degrada com o tempo. Outra técnica relacionada para resolver os requisitos de temporização é minimizar o tempo de resposta para eventos como interrupções.

19.3 Recursos dos kernels de tempo real

Nesta seção, discutimos os recursos necessários para projetar um sistema operacional que admite processos em tempo real. Antes de começarmos, porém, vamos considerar o que normalmente *não* é necessário para um sistema de tempo real. Começamos examinando vários recursos fornecidos em muitos dos sistemas operacionais discutidos até aqui neste texto, incluindo Linux, UNIX e as diversas versões do Windows. Esses sistemas normalmente fornecem suporte para o seguinte:

- Uma série de dispositivos periféricos, como telas gráficas, unidades de CD e unidades de DVD.
- Mecanismos de proteção e segurança.
- Múltiplos usuários.

O suporte para esses recursos normalmente resulta em um kernel sofisticado e grande. Por exemplo, o Windows XP possui mais de 40 milhões de linhas de código-fonte. Por outro lado, um sistema operacional de tempo real típico possui um projeto muito simples, normalmente escrito em milhares em vez de milhões de linhas de código-fonte. Não poderíamos esperar que esses sistemas simples incluíssem os recursos listados acima.

Mas por que os sistemas de tempo real não fornecem esses recursos, que são cruciais para os sistemas comuns de desktop e servidor? Existem vários motivos, mas três são mais importantes. Primeiro, como a maioria dos sistemas de tempo real serve a uma única finalidade, eles não exigem muitos dos recursos encontrados em um PC de desktop. Pense em um relógio de pulso digital: ele obviamente não precisa de suporte para uma unidade de disco ou DVD, muito menos memória virtual. Além do mais, um sistema de tempo real típico não inclui a noção de usuário: o sistema admite um pequeno número de tarefas, que normalmente aguardam a entrada dos dispositivos de hardware (sensores, identificação de visão, e assim por diante). Segundo, os recursos admitidos pelos sistemas operacionais de desktop comuns não podem ser fornecidos sem processadores velozes e grande quantidade de memória. Ambos estão indisponíveis nos sistemas de tempo real, devido às restrições de espaço, conforme já explicamos. Além disso, muitos sistemas de tempo real não possuem espaço suficiente para dar suporte a unidades de disco periféricas ou telas gráficas, embora alguns sistemas possam ter suporte para sistemas de arquivos usando memória não volátil (NVRAM). Terceiro, o suporte a recursos comuns nos ambientes de computação de desktop aumentaria bastante o custo dos sistemas de tempo real, o que poderia torná-los economicamente inviáveis.

Outras considerações se aplicam quando consideramos a memória virtual em um sistema de tempo real. Para fornecer recursos de memória virtual, conforme descrevemos no [Capítulo 9](#), é preciso que o sistema inclua uma unidade de gerência de memória (MMU) para traduzir endereços lógicos para físicos. Contudo, as MMUs normalmente aumentam o custo e o consumo de energia do sistema. Além disso, o tempo necessário para traduzir endereços lógicos para endereços físicos – especialmente no caso de uma perda do buffer TLB – pode ser proibitivo em um ambiente de tempo real rígido. Na discussão a seguir, examinamos várias técnicas para a tradução de endereços nos sistemas de tempo real.

A [Figura 19.2](#) ilustra três estratégias diferentes para gerenciar a tradução de endereços disponível aos projetistas de sistemas operacionais de tempo real. Nesse cenário, a CPU gera o endereço lógico L , que precisa ser mapeado para o endereço físico P . A primeira técnica é evitar os endereços lógicos e fazer a CPU gerar endereços físicos diretamente. Essa técnica – conhecida como **modo de endereçamento real** – não emprega técnicas de memória virtual e está efetivamente afirmando que P é igual a L . Um problema com o modo de endereçamento é a ausência de proteção de memória entre os processos. O modo de endereçamento real também pode exigir que os programadores especifiquem o local físico onde seus programas são carregados na memória. Contudo, o benefício dessa técnica é que o sistema é muito rápido, e nenhum tempo é gasto na tradução de endereços. O modo de endereçamento real é muito comum nos sistemas embutidos com restrições de tempo real rígidas. Na verdade, alguns sistemas operacionais de tempo real rodando em microprocessadores contendo uma MMU realmente desativam a MMU para ganhar o benefício do desempenho ao referenciar os endereços físicos diretamente.

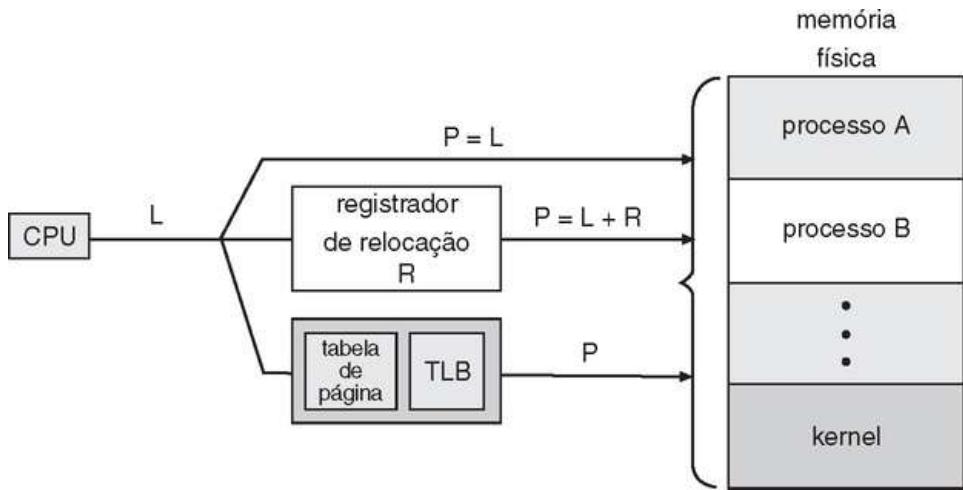


FIGURA 19.2 Tradução de endereços nos sistemas de tempo real.

Uma segunda estratégia para traduzir endereços é usar uma técnica semelhante ao registrador de relocação dinâmico, mostrado na [Figura 8.4](#). Nesse cenário, um registrador de relocação R é definido para o local de memória onde um programa é carregado. O endereço físico P é gerado somando-se o conteúdo do registrador de relocação R a L . Alguns sistemas de tempo real configuraram a MMU para funcionar dessa maneira. O benefício óbvio dessa estratégia é que a MMU pode facilmente traduzir endereços lógicos para endereços físicos usando $P = L + R$. Contudo, esse sistema ainda sofre de falta de proteção de memória entre os processos.

A última técnica é quando o sistema de tempo real fornece funcionalidade de memória virtual total, conforme descrito no [Capítulo 9](#). Nesse caso, a tradução de endereço ocorre por meio de tabelas de página e um buffer look-aside de tradução, ou TLB. Além de permitir que um programa seja carregado em qualquer local da memória, essa estratégia também fornece proteção de memória entre os processos. Para sistemas sem unidades de disco conectadas, a paginação e o swapping por demanda podem não ser possíveis. Contudo, os sistemas podem fornecer tais recursos usando memória flash NVRAM. Os sistemas Lynx-OS e OnCore são exemplos de sistemas operacionais de tempo real fornecendo suporte completo para memória virtual.

19.4 Implementando sistemas operacionais de tempo real

Tendo em mente as muitas variações possíveis, agora identificamos os recursos necessários para implementar um sistema operacional de tempo real. Essa lista de forma alguma é absoluta; alguns sistemas fornecem mais recursos do que listamos a seguir, enquanto outros sistemas fornecem menos.

- Escalonamento preemptivo, baseado em prioridade.
- Kernel preemptivo.
- Latência reduzida.

Um recurso notável que omitimos da lista é o suporte para redes. Contudo, decidir dar suporte para protocolos de rede como TCP/IP é simples: se o sistema de tempo real tiver que ser conectado a uma rede, o sistema operacional precisa fornecer capacidades de rede. Por exemplo, um sistema que colhe dados de tempo real e os transmite a um servidor obviamente precisa incluir recursos de rede. Como alternativa, um sistema embutido autocontido não exigindo interação com outros sistemas computadorizados não possui um requisito de rede óbvio.

No restante desta seção, examinamos os requisitos básicos listados anteriormente e identificamos como eles podem ser implementados em um sistema operacional de tempo real.

19.4.1 Escalonamento baseado em prioridade

O recurso mais importante de um sistema operacional de tempo real é responder imediatamente a um processo de tempo real assim que o processo requisitar a CPU. Como resultado, o escalonador para um sistema operacional de tempo real precisa dar suporte a um algoritmo baseado em prioridade com preempção. Lembre-se de que os algoritmos de escalonamento baseados em prioridade atribuem uma prioridade a cada processo, com base em sua importância; as tarefas mais importantes recebem prioridades mais altas do que aquelas julgadas menos importantes. Se o escalonador também admitir preempção, um processo atualmente em execução na CPU será apropriado se um processo de prioridade mais alta se tornar disponível para execução.

Algoritmos de escalonamento preemptivos, baseados em prioridade, são discutidos com detalhes no [Capítulo 5](#), onde também apresentamos exemplos dos recursos de escalonamento de tempo real flexível dos sistemas operacionais Solaris, Windows XP e Linux. Cada um desses sistemas atribui aos processos de tempo real a prioridade de escalonamento mais alta. Por exemplo, o Windows XP possui 32 níveis de prioridade diferentes; os níveis mais altos - valores de prioridade de 16 a 31 - são reservados para processos de tempo real. Solaris e Linux possuem esquemas de priorização semelhantes.

Contudo, observe que fornecer um escalonador preemptivo, baseado em prioridade, só garante funcionalidade de tempo real flexível. Os sistemas de tempo real rígidos precisam garantir ainda que as tarefas de tempo real serão atendidas de acordo com seus requisitos de prazo, e fazer essas garantias pode exigir recursos de escalonamento adicionais. Na [Seção 19.5](#) explicamos os algoritmos de escalonamento apropriados para os sistemas de tempo real rígidos.

19.4.2 Kernels preemptivos

Kernels não preemptivos não permitem a preempção de um processo executando no modo kernel; um processo no modo kernel será executado até que saia do modo kernel, até que seja bloqueado ou passe o controle da CPU voluntariamente. Por outro lado, um kernel preemptivo permite a preempção de uma tarefa rodando no modo kernel. O projeto de kernels preemptivos pode ser bastante difícil, e as aplicações tradicionais orientadas a usuário, como planilhas, processadores de textos e navegadores Web, normalmente não exigem esses tempos de resposta rápidos. Como resultado, alguns sistemas operacionais de desktop comerciais - como o Windows XP - não são preemptivos.

Contudo, para atender os requisitos de temporização dos sistemas de tempo real - em particular, os sistemas de tempo real rígidos -, os kernels preemptivos são obrigatórios. Caso contrário, uma tarefa de tempo real poderia ter que esperar por um período arbitrariamente longo enquanto outra tarefa estivesse ativa no kernel.

Existem várias estratégias para tornar um kernel preemptível. Uma delas é inserir **pontos de preempção** em chamadas de sistema de longa duração. Um ponto de preempção verifica se um processo de alta prioridade precisa ser executado. Nesse caso, ocorre uma troca de contexto. Depois, quando o processo de alta prioridade termina, o processo interrompido continua com a chamada de sistema. Os pontos de preempção só podem ser colocados em locais *seguros* no kernel, ou seja, somente onde as estruturas de dados do kernel não estão sendo modificadas. Uma segunda estratégia para tornar um kernel preemptível é com o uso de mecanismos de sincronização, que discutimos no [Capítulo 6](#). Com esse método, o kernel sempre poderá ser preemptível, pois quaisquer dados do kernel sendo atualizados são protegidos contra modificação pelo processo de alta prioridade.

19.4.3 Reduzindo a latência

Considere a natureza controlada por evento de um sistema de tempo real: o sistema normalmente está esperando que ocorra um evento em tempo real. Os eventos podem surgir no software (como quando um temporizador expira) ou no hardware (como quando um veículo de controle remoto detecta que está se aproximando de um obstáculo). Quando ocorre um evento, o sistema precisa responder e atendê-lo o mais rapidamente possível. Vamos nos referir à **latência de evento** como a quantidade de tempo que se passa desde que um evento ocorre até que ele seja atendido (Figura 19.3).

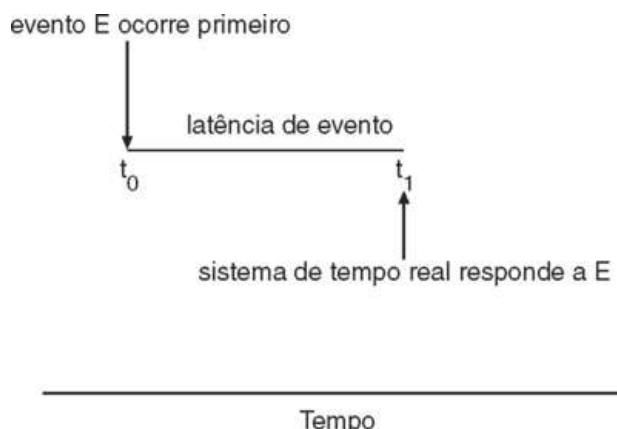


FIGURA 19.3 Latência de evento.

Normalmente, diferentes eventos possuem diferentes requisitos de latência. Por exemplo, o requisito de latência para um sistema de freios ABS poderia ser de três a cinco milissegundos, significando que, desde a hora em que uma roda detecta inicialmente que está deslizando, o sistema controlando os freios tem de três a cinco milissegundos para responder e controlar a situação. Qualquer resposta que leve mais tempo poderia resultar na mudança de direção descontrolada do automóvel. Por outro lado, um sistema embutido controlando o radar em uma companhia aérea poderia tolerar um período de latência de vários segundos.

Dois tipos de latências afetam o desempenho dos sistemas de tempo real:

1. Latência de interrupção.
2. Latência de despacho.

Latência de interrupção refere-se ao período desde a chegada de uma interrupção na CPU até o início da rotina que atende a interrupção. Quando ocorre uma interrupção, o sistema operacional precisa primeiro completar a instrução que está executando e determinar o tipo de interrupção que aconteceu. Depois, ele precisa salvar o estado do processo atual antes de atender a interrupção usando a rotina de serviço de interrupção (ISR) específica. O tempo total exigido para realizar essas tarefas é a latência de interrupção (Figura 19.4). Obviamente, é crucial para os sistemas operacionais de tempo real reduzir a latência de interrupção para garantir que as tarefas de tempo real recebam atenção imediata.

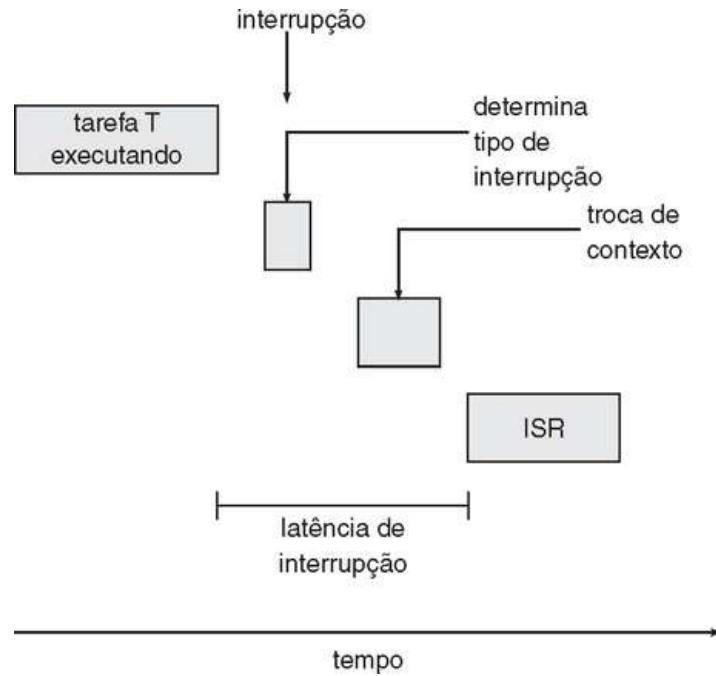


FIGURA 19.4 Latência de interrupção.

Um fator importante que contribui para a latência de interrupção é a quantidade de tempo em que as interrupções podem ser desativadas enquanto as estruturas de dados do kernel estão sendo atualizadas. Os sistemas operacionais de tempo real exigem que as interrupções sejam desativadas por períodos muito curtos. Porém, para sistemas de tempo real rígidos, a latência de interrupção não apenas deve ser reduzida, mas de fato deve ser limitada para garantir o comportamento determinístico exigido dos kernels de tempo real rígido.

A quantidade de tempo exigida para o despachante de escalonamento parar um processo e iniciar outro é conhecida como **latência de despacho**. Fornecer tarefas de tempo real com acesso imediato à CPU exige que os sistemas operacionais de tempo real reduzam essa latência. A técnica mais eficaz para manter a latência de despacho baixa é fornecer kernels preemptivos.

Na [Figura 19.5](#), representamos a composição da latência de despacho. A **fase de conflito** da latência de despacho tem dois componentes:

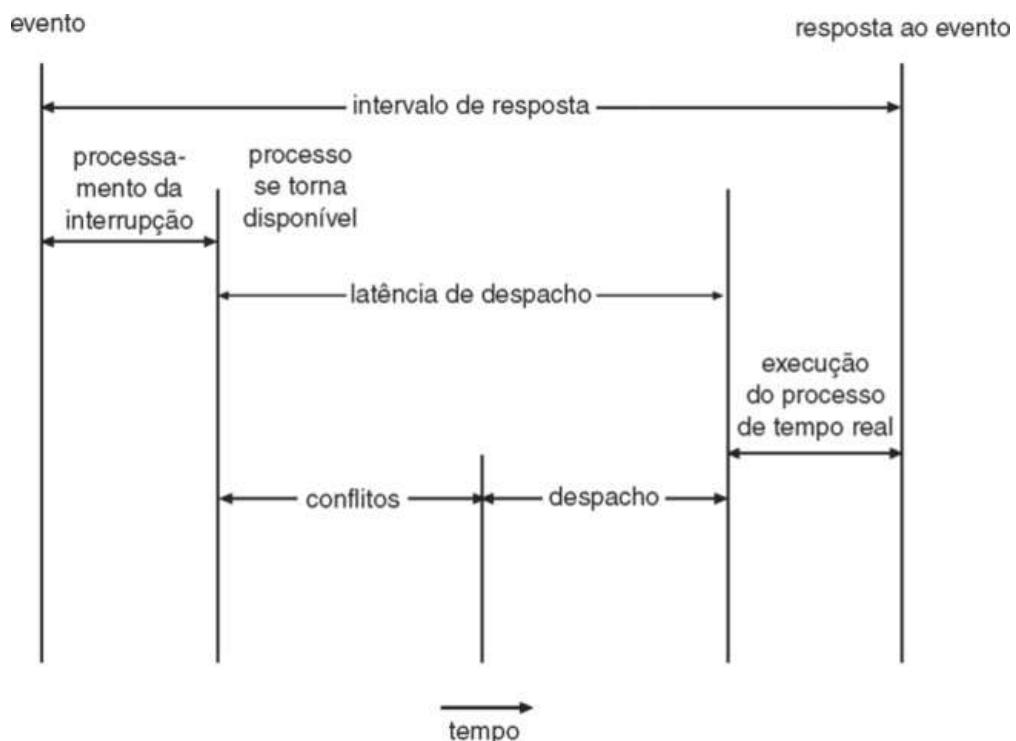


FIGURA 19.5 Latência de despacho.

1. Preempção de qualquer processo executando no kernel.
2. Liberação pelos processos de baixa prioridade dos recursos necessários por um processo de alta prioridade.

Como exemplo, no Solaris, a latência de despacho com preempção desativada é de mais de 100 milissegundos. Com a preempção ativada, ela é reduzida para menos de um milissegundo.

Um problema que pode afetar a latência de despacho surge quando um processo de prioridade mais alta precisa ler ou modificar dados do kernel que estão atualmente sendo acessados por um processo de prioridade inferior ou uma cadeia de processos de prioridade inferior. Como os dados do kernel normalmente são protegidos com um lock, o processo de prioridade mais alta terá que esperar até que um com prioridade inferior termine com o recurso. A situação se torna mais complicada se o processo de prioridade inferior for tomado em favor de outro processo com prioridade mais alta. Como exemplo, suponha que tenhamos três processos, L , M e H , cujas prioridades sigam a ordem $L < M < H$. Suponha também que o processo H exija o recurso R , que atualmente está sendo acessado pelo processo L . Normalmente, o processo H esperaria que L termine de usar o recurso R . Contudo, agora suponha que o processo M se torne executável, tomando a vez do processo L . Indiretamente, um processo com uma prioridade mais baixa - o processo M - afetou o tempo que o processo H precisa esperar por L para abrir mão do recurso R .

Esse problema, conhecido como **inversão de prioridade**, pode ser resolvido pelo uso do **protocolo de herança de prioridade**. De acordo com esse protocolo, todos os processos que estão acessando recursos necessários por um processo de prioridade mais alta herdam a prioridade mais alta até que terminem de usar os recursos em questão. Quando terminam, suas prioridades revertem aos valores originais. No exemplo anterior, um protocolo de herança de prioridade permite que o processo L herde temporariamente a prioridade do processo H , evitando assim que o processo M se aproprie de sua execução. Quando o processo L tiver acabado de usar o recurso R , ele abrirá mão de sua prioridade herdada de H e assumirá sua prioridade original. Como o recurso R agora está disponível, o processo H - e não o M - será executado em seguida.

19.5 Escalonamento de CPU em tempo real

Nossa abordagem sobre escalonamento até aqui tem focalizado principalmente os sistemas de tempo real flexíveis. Contudo, como já dissemos, o escalonamento para tais sistemas não fornece garantias sobre quando um processo crítico será escalonado; ele garante apenas que o processo receberá preferência em relação a processos não críticos. Sistemas de tempo real rígidos possuem requisitos mais rigorosos. Uma tarefa precisa ser atendida dentro do seu prazo; atender depois que o prazo tiver expirado é o mesmo que não atender de nenhuma forma.

Agora, considere o escalonamento para sistemas de tempo real rígidos. Entretanto, antes de prosseguirmos com os detalhes dos escalonadores individuais, temos que definir certas características dos processos que devem ser escalonados. Primeiro, os processos são considerados **periódicos**. Ou seja, eles exigem a CPU em intervalos constantes (períodos). Cada processo periódico possui um tempo de processamento fixo t , uma vez adquirindo a CPU, um prazo d quando deve ser atendido pela CPU, e um período p . O relacionamento do tempo de processamento, prazo e período pode ser expresso como $0 \leq t \leq d \leq p$. A **taxa** de uma tarefa periódica é $1/p$. A [Figura 19.6](#) ilustra a execução de um processo periódico com o tempo. Os escalonadores podem tirar proveito desse relacionamento e atribuir prioridades de acordo com os requisitos de prazo ou taxa de um processo periódico.

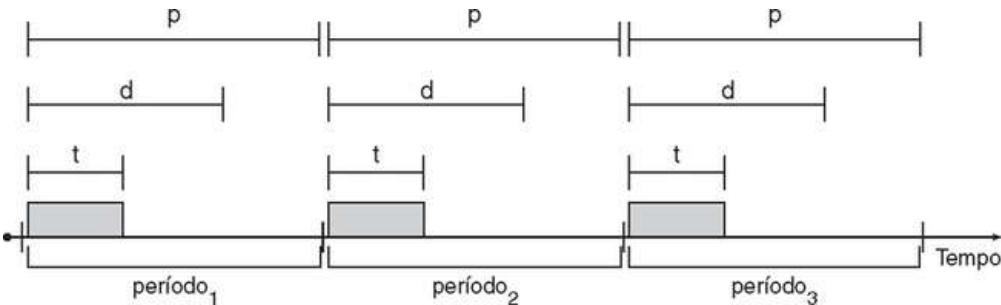


FIGURA 19.6 Tarefa periódica.

O incomum a respeito dessa forma de escalonamento é que um processo pode ter que anunciar seus requisitos de prazo para o escalonador. Depois, usando uma técnica conhecida como algoritmo de **controle de admissão**, o escalonador ou admite o processo, garantindo que o processo completará a tempo, ou rejeita a requisição como impossível, se não puder garantir que a tarefa seja atendida no seu prazo.

Nas próximas seções, exploramos os algoritmos de escalonamento que consideram os requisitos de prazo dos sistemas de tempo real rígidos.

19.5.1 Escalonamento de taxa monotônica

O algoritmo de escalonamento de **taxa monotônica** escalona tarefas periódicas usando uma política de prioridade estática com preempção. Se um processo de prioridade inferior estiver executando e um processo de prioridade mais alta se tornar disponível para execução, ele apropriará o processo com prioridade menor. Ao entrar no sistema, cada tarefa periódica recebe uma prioridade inversamente baseada em seu período: quanto menor o período, maior a prioridade; quanto maior o período, menor a prioridade. O raciocínio por trás dessa política é atribuir uma prioridade mais alta a tarefas que exigem a CPU com mais frequência. Além do mais, o escalonamento de taxa monotônica considera que o tempo de processamento de um processo periódico é o mesmo para cada rajada de CPU. Ou seja, toda vez que um processo adquire a CPU, a duração de sua rajada de CPU é igual.

Vamos considerar um exemplo. Temos dois processos P_1 e P_2 . Os períodos para P_1 e P_2 são 50 e 100, respectivamente, ou seja, $p_1 = 50$ e $p_2 = 100$. Os tempos de processamento são $t_1 = 20$ para P_1 e $t_2 = 35$ para P_2 . O prazo para cada processo exige que ele termine sua rajada de CPU no início de seu próprio período.

Primeiro, temos que nos perguntar se é possível escalar essas tarefas de modo que cada uma atenda seus prazos. Se medirmos a utilização de CPU de um processo P_i como a razão entre sua rajada e seu período - t_i/p_i -, a utilização de CPU de P_1 será $20/50 = 0,40$, e a de P_2 será $35/100 = 0,35$, para uma utilização de CPU total de 75%. Portanto, parece que podemos escalar essas tarefas de modo que ambas atendam seus prazos e ainda deixem a CPU com ciclos disponíveis.

Primeiro, suponha que atribuímos a P_2 uma prioridade mais alta que a P_1 . A execução de P_1 e P_2 aparece na [Figura 19.7](#). Como podemos ver, P_2 inicia a execução primeiro e termina no tempo 35.

Nesse ponto, P_1 começa; ele termina sua rajada de CPU no tempo 55. Porém, o primeiro prazo para P_1 foi no tempo 50, de modo que o escalonador fez P_1 perder seu prazo.



FIGURA 19.7 Escalonamento de tarefas quando P_2 tem uma prioridade mais alta que P_1 .

Agora, suponha que usemos o escalonamento de taxa monotônica, em que atribuímos a P_1 uma prioridade mais alta que P_2 , pois o período de P_1 é mais curto que o de P_2 . A execução desses processos aparece na [Figura 19.8](#). P_1 começa primeiro e termina sua rajada de CPU no tempo 20, atendendo, assim, seu primeiro prazo. P_2 começa a executar nesse ponto e executa até o tempo 50. Nesse momento, ele é tomado por P_1 , embora ainda tenha 5 milissegundos restantes em sua rajada de CPU. P_1 termina sua rajada de CPU no tempo 70, quando o escalonador retorna a P_2 . P_2 completa sua rajada de CPU no tempo 75, também atendendo seu primeiro prazo. O sistema está ocioso até o tempo 100, quando P_1 é escalonado novamente.

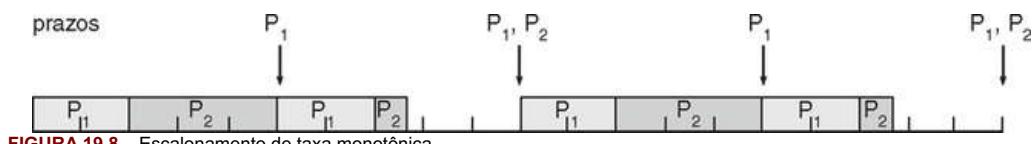


FIGURA 19.8 Escalonamento de taxa monetônica.

O escalonamento de taxa monetônica é considerado ideal no sentido de que, se um conjunto de processos não puder ser escalonado por esse algoritmo, ele não poderá ser escalonado por qualquer outro algoritmo que atribua prioridades estáticas. Em seguida, vamos examinar um conjunto de processos que não pode ser escalonado usando o algoritmo de taxa monetônica. Considere que o processo P_1 tenha um período de $p_1 = 50$ e uma rajada de CPU de $t_1 = 25$. Para P_2 , os valores correspondentes são $p_2 = 80$ e $t_2 = 35$. O escalonamento de taxa monetônica atribuiria ao processo P_1 uma prioridade mais alta, pois tem o período mais curto. A utilização total de CPU dos dois processos é $(25/50) + (35/80) = 0,94$, e, portanto, parece lógico que os dois processos possam ser escalonados e ainda deixem a CPU com 6% de tempo disponível. A [Figura 19.9](#) mostra o escalonamento de processos P_1 e P_2 . Inicialmente, P_1 é executado até que complete sua rajada de CPU no tempo 25. O processo P_2 , então, começa a executar e funciona até o tempo 50, quando é tomado por P_1 . Nesse ponto, P_2 ainda tem 10 milissegundos restantes em sua rajada de CPU. O processo P_1 é executado até o tempo 75; consequentemente, P_2 perde o prazo para terminar sua rajada de CPU no tempo 80.

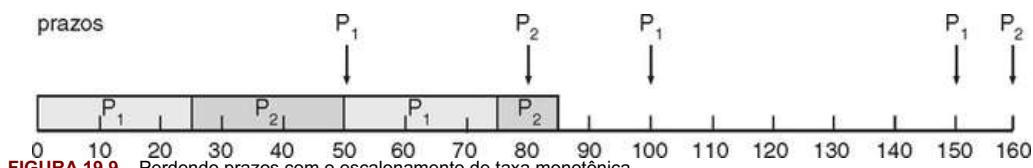


FIGURA 19.9 Perdendo prazos com o escalonamento de taxa monetônica.

Apesar de ser o ideal, portanto, o escalonamento de taxa monetônica tem uma limitação: a utilização de CPU é limitada e nem sempre é possível maximizar totalmente os recursos da CPU. A utilização de CPU no pior caso para o escalonamento de N processos é

$$2\left(2^{1/n} - 1\right).$$

Com um processo no sistema, a utilização de CPU é de 100%, mas ela cai para aproximadamente 69% quando o número de processos se aproxima do infinito. Com dois processos, a utilização de

CPU está limitada a cerca de 83%. A utilização de CPU combinada para os dois processos escalonados nas Figuras 19.7 e 19.8 é de 75%; portanto, o algoritmo de escalonamento de taxa monotônica tem garantia de escaloná-los de modo que possam atender seus prazos. Para os dois processos escalonados na [Figura 19.9](#), a utilização de CPU combinada é de aproximadamente 94%; portanto, o escalonamento de taxa monotônica não pode garantir que eles possam ser escalonados de modo que atendam seus prazos.

19.5.2 Escalonamento do prazo mais antigo primeiro

O escalonamento do prazo mais antigo primeiro (Earliest Deadline First – EDF) atribui prioridades de acordo com o prazo. Quanto mais antigo o prazo, maior a prioridade; quanto mais recente o prazo, menor a prioridade. Sob a política de EDF, quando um processo se torna executável, ele precisa anunciar seus requisitos de prazo ao sistema. As prioridades podem ter que ser ajustadas para refletirem o prazo do processo que se tornou executável recentemente. Observe como isso difere do escalonamento de taxa monotônica, onde as prioridades são fixas.

Para ilustrar o escalonamento EDF, novamente agendamos os processos mostrados na [Figura 19.9](#), que deixaram de cumprir os requisitos de prazo sob o escalonamento de taxa monotônica. Lembre-se de que P_1 tem valores de $p_1 = 50$ e $t_1 = 25$ e que P_2 tem valores de $p_2 = 80$ e $t_2 = 35$. O escalonamento EDF desses processos pode ser visto na [Figura 19.10](#). O processo P_1 tem o prazo mais antigo, de modo que sua prioridade inicial é mais alta que a do processo P_2 . O processo P_2 começa a executar no final da rajada de CPU para P_1 . Contudo, enquanto o escalonamento de taxa monotônica permite que P_1 se aproprie de P_2 no início do seu próprio período no tempo 50, o escalonamento EDF permite que o processo P_2 continue executando. P_2 agora tem uma prioridade mais alta que P_1 , pois seu próprio prazo (no tempo 80) é anterior ao de P_1 (no tempo 100). Assim, tanto P_1 quanto P_2 atenderam seus primeiros prazos. O processo P_1 novamente começa a executar no tempo 60 e completa sua segunda rajada de CPU no tempo 85, também atendendo seu segundo prazo no tempo 100. P_2 começa a executar nesse ponto, somente para ser tomado por P_1 no início do seu próprio período de tempo 100. P_2 é interrompido porque P_1 tem o prazo mais antigo (tempo 150) do que P_2 (tempo 160). No tempo 125, P_1 completa sua rajada de CPU e P_2 continua sua execução, terminando no tempo 145 e também atendendo seu prazo. O sistema fica ocioso até o tempo 150, quando P_1 é escalonado para executar mais uma vez.

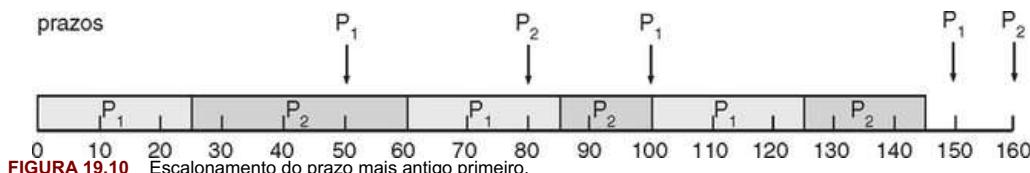


FIGURA 19.10 Escalonamento do prazo mais antigo primeiro.

Ao contrário do algoritmo de taxa monotônica, o escalonamento EDF não exige que os processos sejam periódicos, e um processo não precisa de uma quantidade constante de tempo de CPU por rajada. O único requisito é que um processo anuncie seu prazo ao escalonador quando se tornar executável. O apelo do escalonamento EDF é que ele é teoricamente ideal: teoricamente, ele pode escalonar processos de modo que cada um possa atender seus requisitos de prazo, e a utilização de CPU será de 100%. Contudo, na prática, é impossível alcançar esse nível de utilização de CPU, devido ao custo da troca de contexto entre os processos e o tratamento da interrupção.

19.5.3 Escalonamento de quotas proporcionais

Os escalonadores de quotas proporcionais operam alocando T quotas entre todas as aplicações. Uma aplicação pode receber N quotas de tempo, garantindo, assim, que a aplicação terá N/T do tempo total do processador. Como exemplo, suponha que haja um total de $T = 100$ quotas para serem divididas entre três processos, A , B e C . A recebe 50 quotas, B recebe 15 quotas e C recebe 20 quotas. Esse esquema garante que A terá 50% do tempo total do processador, B terá 15% e C terá 20%.

Os escalonadores de quotas proporcionais precisam trabalhar em conjunto com uma política de controle de admissão para garantir que uma aplicação receba suas quotas de tempo alocadas. Uma política de controle de admissão admitirá um cliente requisitando determinado número de quotas somente se houver quotas suficientes à disposição. Em nosso exemplo atual, alocamos $50 + 15 + 20 = 85$ quotas do total de 100 quotas. Se um novo processo D requisitasse 30 quotas, o controlador de admissão negaria a entrada de D no sistema.

19.5.4 Escalonamento Pthread

O padrão POSIX também fornece extensões para a computação de tempo real - POSIX.1b. Nesta seção, abordamos parte da API Pthread do POSIX relacionada com o escalonamento de threads de tempo real. Pthreads definem duas classes de escalonamento para threads de tempo real:

- **SCHED_FIFO**.
- **SCHED_RR**.

SCHED_FIFO escalona threads de acordo com uma política de primeiro a chegar, primeiro a ser atendido, usando uma fila FIFO conforme esboçado na [Seção 5.3.1](#). Contudo, não existe divisão de tempo entre as threads de mesma prioridade. Portanto, a thread de tempo real de mais alta prioridade na frente da fila FIFO receberá a CPU até que termine ou seja bloqueada. SCHED_RR (de round-robin) é semelhante a SCHED_FIFO, exceto que fornece divisão de tempo entre as threads de mesma prioridade. Pthreads fornecem uma classe de escalonamento adicional - SCHED_OTHER -, Smas sua implementação é indefinida e específica do sistema; ela pode se comportar diferente em sistemas diferentes.

A API Pthread especifica as duas funções a seguir para apanhar e definir a política de escalonamento:

- `pthread_attr_getsched_policy(pthread_attr_t *attr, int *policy)`.
- `pthread_attr_setsched_policy(pthread_attr_t *attr, int policy)`.

O primeiro parâmetro das duas funções é um ponteiro para o conjunto de atributos para a thread. O segundo parâmetro é (1) um ponteiro para um inteiro que é definido como a política de escalonamento atual (para `pthread_attr_getsched_policy()`) ou (2) um valor inteiro (SCHED_FIFO, SCHED_RR ou SCHED_OTHER) para a função `pthread_attr_setsched_policy()`. As duas funções retornam valores diferentes de zero se houver um erro.

Na [Figura 19.11](#), ilustramos um programa Pthread usando essa API. Esse programa primeiro determina a política de escalonamento atual, e depois define o algoritmo de escalonamento para SCHED_OTHER.

```

#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[ ])
{
    int i, policy;
    pthread_t tid[NUM_THREADS];
    pthread_attr_t attr;

    /* apanha os atributos default */
    pthread_attr_init(&attr);

    /* apanha a política de escalonamento atual */
    if (pthread_attr_getschedpolicy(&attr, &policy) != 0)
        fprintf(stderr, "Não foi possível obter a política.\n");
    else {
        if (policy == SCHED_OTHER)
            printf("SCHED_OTHER\n");
        else if (policy == SCHED_RR)
            printf("SCHED_RR\n");
        else if (policy == SCHED_FIFO)
            printf("SCHED_FIFO\n");
    }

    /* define a política de escalonamento - FIFO, RR ou OTHER */
    if (pthread_attr_setschedpolicy(&attr, SCHED_OTHER) != 0)
        fprintf(stderr, "Não foi possível definir a política.\n");

    /* cria as threads */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_create(&tid[i], &attr, runner, NULL);

    /* agora une em cada thread */
    for (i = 0; i < NUM_THREADS; i++)
        pthread_join(tid[i], NULL);
}

/* Cada thread iniciará o controle nesta função */
void *runner(void *param)
{
    /* faz algum trabalho... */

    pthread_exit(0);
}

```

FIGURA 19.11 API de escalonamento Pthread.

19.6 Um exemplo: VxWorks 5.x

Nesta seção, descrevemos o VxWorks, um sistema operacional de tempo real popular, fornecendo suporte de tempo real rígido. O VxWorks, desenvolvido comercialmente pela Wind River Systems, é bastante usado em automóveis, dispositivos de consumidor e industriais, e equipamentos de rede, como switches e roteadores. O VxWorks também é usado para controlar os dois veículos - *Spirit* e *Opportunity* - que iniciaram a exploração do planeta Marte em 2004.

A organização do VxWorks pode ser vista na [Figura 19.12](#). O VxWorks gira em torno do microkernel *Wind*. Lembre-se, pela nossa discussão na [Seção 2.7.3](#), que os microkernels são criados de modo que o kernel do sistema operacional forneça um mínimo de recursos; utilitários adicionais, como redes, sistemas de arquivos e gráficos, são fornecidos em bibliotecas fora do kernel. Essa técnica fornece muitos benefícios, incluindo a redução do tamanho do kernel - um recurso desejável para um sistema embutido que exige uma pegada pequena.

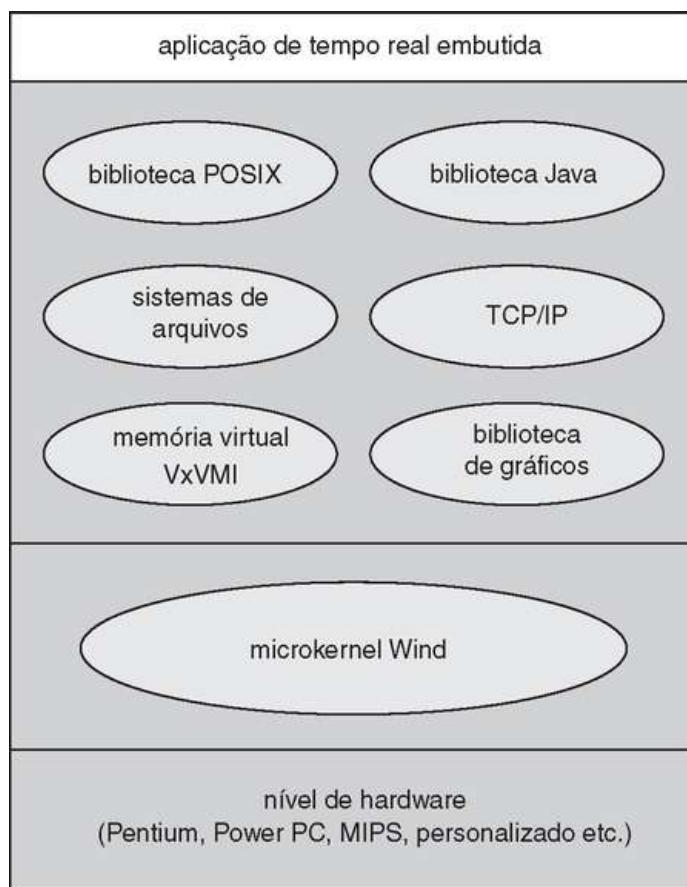


FIGURA 19.12 A organização do VxWorks.

O microkernel *Wind* admite os seguintes recursos básicos:

- **Processos.** O microkernel *Wind* fornece suporte para processos individuais e threads (usando a API Pthread). Contudo, semelhante ao Linux, o VxWorks não distingue entre processos e threads, e refere-se a ambos como **tarefas**.
- **Escalonamento.** O microkernel *Wind* fornece dois modelos de escalonamento separados: escalonamento preemptivo e round-robin não preemptivo, com 256 níveis de prioridade diferentes. O escalonador também admite a API POSIX para threads de tempo real, conforme explicamos na [Seção 19.5.4](#).
- **Interrupções.** O microkernel *Wind* também gerencia as interrupções. Para dar suporte aos requisitos de tempo real rígidos, os tempos de latência de interrupção e despacho são limitados.
- **Comunicação entre processos.** O microkernel *Wind* fornece memória compartilhada e passagem de mensagens como mecanismos para a comunicação entre tarefas separadas. Ele também permite que as tarefas se comuniquem usando uma técnica conhecida como **pipes** - um mecanismo que se comporta da mesma maneira que uma fila FIFO, mas permite que as tarefas se comuniquem escrevendo em um arquivo especial, o pipe. Para proteger os dados compartilhados por tarefas separadas, o VxWorks fornece semáforos e locks mutex com um protocolo de herança de prioridade para impedir a inversão de prioridades.

Fora do microkernel, o VxWorks inclui diversas bibliotecas de componentes que fornecem suporte para POSIX, Java, redes TCP/IP e coisas desse tipo. Todos os componentes são opcionais, permitindo que o projetista de um sistema embutido personalize o sistema de acordo com suas necessidades específicas. Por exemplo, se o uso de redes não for necessário, a biblioteca TCP/IP pode ser excluída da imagem do sistema operacional. Essa estratégia permite que o projetista do sistema operacional inclua apenas os recursos necessários, minimizando assim o tamanho - ou pegada - do sistema operacional.

O VxWorks usa uma técnica interessante para a gerência de memória, admitindo dois níveis de memória virtual. O primeiro nível, que é muito simples, permite o controle do cache com base em cada página. Essa política possibilita que uma aplicação especifique certas páginas como não passíveis de cache. Quando os dados estão sendo compartilhados por tarefas separadas executando em uma arquitetura de multiprocessador, é possível que os dados compartilhados possam residir em caches separados, locais aos processadores individuais. A menos que uma arquitetura admita uma política de coerência de cache para garantir que os mesmos dados residindo em dois caches não sejam diferentes, esses dados compartilhados não deverão ser mantidos em cache e, em vez disso, residem apenas na memória principal, de modo que todas as tarefas mantêm uma visão consistente dos dados.

O segundo nível de memória virtual exige o componente opcional de memória virtual VxVMI ([Figura 19.12](#)), junto com o suporte do processador para uma unidade de gerência de memória (MMU). Quando esse componente opcional é carregado em um sistema com MMU, o VxWorks permite que uma tarefa marque certas áreas de dados como *privadas*. Uma área de dados marcada como privada só pode ser acessada pela tarefa a que pertence. Além do mais, o VxWorks permite que páginas contendo código do kernel junto com o vetor de interrupção sejam declaradas como somente de leitura. Isso é útil, pois o VxWorks não distingue entre modos usuário e kernel; todas as aplicações são executadas no modo kernel, dando a uma aplicação acesso a todo o espaço de endereços do sistema.

19.7 Resumo

Um sistema de tempo real é um sistema computadorizado que exige que os resultados cheguem dentro de um prazo; os resultados chegando após o prazo ter sido passado são inúteis. Muitos sistemas de tempo real são embutidos em dispositivos de consumidor e industriais. Existem dois tipos de sistemas de tempo real: sistemas de tempo real flexíveis e rígidos. Os sistemas de tempo real flexíveis são menos restritivos, atribuindo às tarefas de tempo real uma prioridade de escalonamento mais alta que outras tarefas. Os sistemas de tempo real rígidos precisam garantir que as tarefas de tempo real sejam atendidas dentro de seu prazo. Além dos requisitos de tempo rigorosos, os sistemas de tempo real podem ainda ser caracterizados como tendo apenas uma única finalidade e executando em pequenos dispositivos, pouco dispendiosos.

LINUX EM TEMPO REAL

O sistema operacional Linux está sendo usado cada vez mais em ambientes de tempo real. Já explicamos seus recursos de escalonamento de tempo real flexível ([Seção 5.6.3](#)), pelos quais as tarefas de tempo real recebem a prioridade mais alta no sistema. Recursos adicionais na versão 2.6 do kernel tornam o Linux cada vez mais adequado para sistemas embutidos. Esses recursos incluem um kernel totalmente preemptivo e um algoritmo de escalonamento mais eficiente, que roda em tempo $O(1)$ independente do número de tarefas ativas no sistema. A versão 2.6 também facilita o transporte do Linux para diferentes arquiteturas de hardware, dividindo o kernel em componentes modulares.

Outra estratégia para integrar o Linux aos ambientes de tempo real envolve a combinação do sistema operacional Linux com um pequeno kernel de tempo real, fornecendo assim um sistema que atua tanto como sistema de uso geral quanto de tempo real. Essa é a técnica tomada pelo sistema operacional RTLinux. No RTLinux, o kernel padrão do Linux é executado como uma tarefa em um pequeno sistema operacional de tempo real. O kernel de tempo real trata de todas as interrupções - direcionando cada interrupção para um tratador no kernel-padrão ou para um tratador de interrupção no kernel de tempo real. Além do mais, o RTLinux impede que o kernel-padrão do Unix desative interrupções, garantindo que não poderá aumentar a latência do sistema de tempo real. O RTLinux também fornece diferentes políticas de escalonamento, incluindo escalonamento de taxa monotônica ([Seção 19.5.1](#)) e escalonamento do prazo mais antigo primeiro ([Seção 19.5.2](#)).

Para atender os requisitos de tempo, os sistemas operacionais de tempo real precisam empregar diversas técnicas. O escalonador para um sistema operacional de tempo real precisa ter suporte para um algoritmo baseado em prioridade com preempção. Além do mais, o sistema operacional precisa permitir que as tarefas executando no kernel sejam apropriadas em favor de tarefas de tempo real de prioridade mais alta. Os sistemas operacionais de tempo real também resolvem questões de tempo minimizando a latência de interrupção e despacho.

Os algoritmos de escalonamento de tempo real incluem escalonamento de taxa monotônica e prazo mais antigo primeiro. O escalonamento de taxa monotônica atribui a tarefas que exigem a CPU com mais frequência uma prioridade mais alta do que as tarefas que exigem a CPU com menos frequência. O escalonamento pelo prazo mais antigo primeiro atribui a prioridade de acordo com os prazos que aparecem - quanto mais antigo o prazo, maior a prioridade. O escalonamento por quotas proporcionais utiliza uma técnica de dividir o tempo do processador em quotas e atribuir a cada processo uma quantidade de quotas, garantindo, assim, que cada processo tenha sua quota proporcional de tempo de CPU. A API Pthread também fornece diversos recursos para escalonar os threads de tempo real.

Exercícios

19.1. Identifique se o escalonamento de tempo real rígido ou flexível é mais apropriado nos ambientes a seguir:

- a. Termostato em uma casa
- b. Sistema de controle para uma usina nuclear
- c. Sistema de economia de combustível em um automóvel
- d. Sistema de aterrissagem em um avião

19.2. Discuta maneiras como o problema da inversão de prioridade poderia ser resolvido em um sistema de tempo real. Além disso, discuta se as soluções poderiam ser implementadas dentro do contexto de um escalonador de quotas proporcionais.

19.3. O kernel do Linux 2.6 pode ser montado sem um sistema de memória virtual. Explique por que esse recurso pode agradar os projetistas de sistemas de tempo real.

19.4. Sob quais circunstâncias o escalonamento de taxa monotônica é inferior ao escalonamento do prazo mais antigo primeiro para atender os prazos associados aos processos?

19.5. Considere dois processos, P_1 e P_2 , onde $p_1 = 50$, $t_1 = 25$, $p_2 = 75$ e $t_2 = 30$.

a. Esses dois processos podem ser escalonados com o uso do escalonamento de taxa monotônica? Ilustre sua resposta usando um diagrama de Gantt, como aqueles nas Figuras 19.7 a 19.10.

b. Ilustre o escalonamento desses dois processos usando o escalonamento do prazo mais antigo primeiro (EDF).

19.6. Quais são os diversos componentes da latência de interrupção e despacho?

19.7. Explique por que os tempos de latência de interrupção e despacho precisam ser limitados em um sistema de tempo real rígido.

Notas bibliográficas

Os algoritmos de escalonamento para sistemas de tempo real rígidos, como o escalonamento de taxa monotônica e o escalonamento do prazo mais antigo primeiro, são apresentados em [Liu e Layland \[1973\]](#). Outros algoritmos de escalonamento e extensões aos algoritmos anteriores são apresentados em [Jensen e outros \[1985\]](#), [Lehoczky e outros \[1989\]](#), [Audsley e outros \[1991\]](#), [Mok \[1983\]](#) e [Stoica e outros \[1996\]](#). [Mok \[1983\]](#) descreve um algoritmo dinâmico de atribuição de prioridade chamado escalonamento de menor frouxidão primeiro. [Stoica e outros \[1996\]](#) analisam o algoritmo de quotas proporcionais. Informações úteis com relação a diversos sistemas operacionais populares usados nos sistemas embutidos podem ser obtidas em <http://rtlinux.org>, <http://windriver.com> e <http://qnx.com>. Tendências futuras e questões de pesquisa importantes no campo de sistemas embutidos são discutidas em um artigo de pesquisa por [Stankovic \[1996\]](#).

CAPÍTULO 20

Sistemas multimídia

Nos capítulos anteriores, geralmente nos preocupamos com a forma como os sistemas operacionais tratam de dados convencionais, como arquivos de texto, programas, binários, documentos de processamento de textos e planilhas. Contudo, os sistemas operacionais também podem ter que tratar de outros tipos de dados. Uma tendência relativamente recente na tecnologia é a incorporação **de dados de multimídia** nos sistemas computadorizados. Os dados de multimídia consistem em dados de mídia contínua (áudio e vídeo) e também arquivos convencionais. Os dados de mídia contínua diferem dos dados convencionais porque os dados de mídia contínua - como quadros de vídeo - precisam ser entregues (por stream) de acordo com certas restrições de tempo (por exemplo, 30 quadros por segundo). Neste capítulo, exploramos as demandas dos dados de mídia contínua. Também discutimos com mais detalhes como esses dados diferem dos dados convencionais e como essas diferenças afetam o projeto dos sistemas operacionais que admitem os requisitos dos sistemas de multimídia.

OBJETIVOS DO CAPÍTULO

- Identificar as características dos dados de multimídia.
- Examinar diversos algoritmos usados para compactar os dados de multimídia.
- Explorar os requisitos do sistema operacional dos dados de multimídia, incluindo escalonamento de CPU e disco e gerenciamento de rede.

20.1 O que é multimídia?

O termo *multimídia* descreve uma grande variedade de aplicações que estão em uso popular hoje em dia. Estas incluem arquivos de áudio e vídeo, como arquivos de áudio MP3, filmes de DVD e prévias de clipes de vídeo curtos ou notícias baixadas pela Internet. As aplicações de multimídia também incluem webcasts ao vivo (broadcast pela World Wide Web) de falas ou eventos esportivos e até mesmo webcams ao vivo, permitindo que um espectador em Manhattan observe clientes em um café em Paris. As aplicações de multimídia não precisam ser de áudio ou vídeo; na verdade, uma aplicação de multimídia normalmente inclui uma combinação de ambos. Por exemplo, um filme pode consistir em trilhas de áudio e vídeo separadas. Nem as aplicações de multimídia precisam ser entregues apenas a computadores pessoais de desktop. Cada vez mais, elas estão sendo direcionadas para dispositivos menores, como PDAs (assistentes digitais pessoais) e telefones celulares. Por exemplo, um corretor de ações pode ter valores de ações entregues em tempo real em seu PDA.

Nesta seção, exploramos diversas características dos sistemas de multimídia e examinamos como os arquivos de multimídia podem ser entregues de um servidor para um sistema cliente. Também examinamos os padrões comuns para representar arquivos de vídeo e áudio de multimídia.

20.1.1 Entrega de mídia

Os dados de multimídia são armazenados no sistema de arquivos exatamente como quaisquer outros dados. A maior diferença entre um arquivo comum e um arquivo de multimídia é que o arquivo de multimídia precisa ser acessado em uma taxa específica, enquanto o acesso ao arquivo comum não exige um tempo especial. Vamos usar o vídeo como exemplo do que significa "taxa". O vídeo é representado por uma série de imagens, formalmente conhecidas como **quadros**, que são exibidos em sucessão rápida. Quanto mais rápidos os quadros forem exibidos, mais suave o vídeo aparece. Em geral, uma taxa de 24 a 30 quadros por segundo é necessária para que o vídeo apareça suave ao olho humano. (O olho retém a imagem de cada quadro por um curto período depois que tiver sido apresentada, uma característica conhecida como **persistência de visão**. Uma taxa de 24 a 30 quadros por segundo é rápida o suficiente para parecer contínua.) Uma taxa inferior a 24 quadros por segundo resultará em uma apresentação de aparência picada. O arquivo de vídeo precisa ser acessado pelo sistema de arquivos em uma taxa coerente com a taxa em que o vídeo está sendo exibido. Vamos nos referir aos dados com os requisitos de taxa associados como **dados de mídia contínua**.

Os dados de multimídia podem ser entregues a um cliente pelo sistema de arquivos local ou por um servidor remoto. Quando os dados são entregues pelo sistema de arquivos local, nos referimos à entrega como **reprodução local**. Alguns exemplos incluem observar um DVD em um computador laptop ou escutar um arquivo de áudio MP3 em um player MP3 portátil. Nesses casos, os dados compreendem um arquivo regular que está armazenado no sistema de arquivos local e é reproduzido (ou seja, visto ou escutado) a partir desse sistema.

Os arquivos de multimídia também podem ser armazenados em um servidor remoto e entregues a um cliente por uma rede, usando uma técnica conhecida como **streaming**. Um cliente pode ser um computador pessoal ou um dispositivo menor, como um computador portátil, PDA ou telefone celular. Os dados de mídia contínua ao vivo - como webcams ao vivo - também são enviados por streaming de um servidor aos clientes.

Existem dois tipos de técnicas de streaming: download progressivo e streaming de tempo real. Com um **download progressivo**, um arquivo de mídia contendo áudio ou vídeo é baixado e armazenado no sistema de arquivos local do cliente. À medida que o arquivo está sendo baixado, o cliente é capaz de reproduzir o arquivo de mídia sem ter que esperar que o arquivo seja totalmente baixado. Como o arquivo de mídia por fim é armazenado no sistema cliente, o download progressivo é mais útil para arquivos de mídia relativamente pequenos, como pequenos clipes de vídeo.

O **streaming em tempo real** difere do download progressivo porque o arquivo de mídia é enviado ao cliente, mas só é reproduzido - e não armazenado - pelo cliente. Como o arquivo de mídia não é armazenado no sistema cliente, o streaming em tempo real é preferível ao download progressivo para arquivos de mídia que poderiam ser muito grandes para armazenamento no sistema, como longos vídeos e transmissões de rádio e televisão pela Internet.

O download progressivo e o streaming de tempo real podem permitir que um cliente passe para diferentes pontos no stream, assim como você pode usar os botões de avanço e retrocesso rápido em um controlador de DVD para passar a diferentes pontos no disco de DVD. Por exemplo, poderíamos passar para o final de um streaming de vídeo de 5 minutos ou reproduzir determinada seção de um clipe de filme. A capacidade de se movimentar dentro do stream de mídia é conhecida como **acesso aleatório**.

Dois tipos de streaming de tempo real estão disponíveis: streaming ao vivo e streaming por demanda. O **streaming ao vivo** é usado para entregar um evento, como um concerto ou uma palestra, ao vivo enquanto está realmente ocorrendo. Uma transmissão de programa de rádio pela

Internet é um exemplo de um stream de tempo real ao vivo. De fato, um dos autores deste texto escuta regularmente uma estação de rádio favorita de Vermont enquanto está em sua casa em Utah, pois é transmitida ao vivo pela Internet. O streaming em tempo real ao vivo também é usado para aplicações como webcams ao vivo e videoconferência. Devido à sua entrega ao vivo, esse tipo de streaming de tempo real não permite que os clientes tenham acesso aleatório a diferentes pontos no stream de mídia. Além disso, a entrega ao vivo significa que um cliente que deseja ver (ou escutar) um stream ao vivo em particular já em andamento se “juntará” à sessão “tarde”, perdendo assim as partes anteriores do stream. A mesma coisa acontece com uma transmissão de televisão ou rádio ao vivo. Se você começar a ver o noticiário das 19 horas às 19:10, terá perdido os primeiros 10 minutos da transmissão.

O **streaming por demanda** é usado para entregar streams de mídia como filmes com extensão total e palestras arquivadas. A diferença entre streaming ao vivo e por demanda é que o streaming por demanda não ocorre enquanto o evento está ocorrendo. Assim, por exemplo, enquanto ver um streaming ao vivo é como ver uma transmissão de noticiário na televisão, ver um stream por demanda é como ver um filme em um aparelho de DVD em alguma hora mais conveniente – não existe a noção de chegar tarde. Dependendo do tipo de streaming por demanda, um cliente pode ou não ter acesso aleatório ao stream.

Exemplos de produtos de mídia de streaming bem conhecidos incluem RealPlayer, Apple QuickTime e Windows Media Player. Esses produtos incluem servidores que enviam a mídia e players de mídia do cliente, que são usados para a reprodução.

20.1.2 Características dos sistemas de multimídia

As demandas dos sistemas de multimídia são diferentes das demandas das aplicações tradicionais. Em geral, os sistemas de multimídia podem ter as seguintes características:

1. Os arquivos de multimídia podem ser muito grandes. Por exemplo, um arquivo de vídeo MPEG-1 de 100 minutos exige aproximadamente 1,125 GB de espaço de armazenamento; 100 minutos de televisão de alta definição (HDTV) exigem aproximadamente 15 GB de armazenamento. Um servidor armazenando centenas ou milhares de arquivos de vídeo digital pode, assim, exigir vários terabytes de armazenamento.
2. A mídia contínua pode exigir taxas de dados muito altas. Considere o vídeo digital, em que um quadro de vídeo colorido é exibido em uma resolução de 800×600 . Se usarmos 24 bits para representar a cor de cada pixel (que nos permite ter 2^{24} ou aproximadamente 16 milhões de cores diferentes), um único quadro exigirá $800 \times 600 \times 24 = 11.520.000$ bits de dados. Se os quadros são exibidos a uma taxa de 30 por segundo, é preciso uma largura de banda com mais de 345 Mbps.
3. As aplicações de multimídia são sensíveis a atrasos de tempo durante a reprodução. Quando um arquivo de mídia contínua é entregue a um cliente, a entrega precisa continuar a uma determinada taxa durante a reprodução da mídia; caso contrário, o ouvinte ou espectador estará sujeito a pausas durante a apresentação.

20.1.3 Aspectos do sistema operacional

Para um sistema de computador fornecer dados de mídia contínua, ele precisa garantir a taxa específica e os requisitos de tempo – algo também conhecido como requisitos de **qualidade de serviço** ou QoS – da mídia contínua.

Fornecer essas garantias de QoS afeta diversos componentes em um sistema computadorizado e influencia aspectos do sistema operacional, como escalonamento de CPU, escalonamento de disco e gerenciamento de rede. Exemplos específicos incluem o seguinte:

1. Compactação e decodificação podem exigir processamento significativo da CPU.
2. Tarefas de multimídia precisam ser escalonadas com certas prioridades, para garantir os requisitos de prazo da mídia contínua.
3. De modo semelhante, os sistemas de arquivos precisam ser eficientes para atender os requisitos de taxa da mídia contínua.
4. Os protocolos de rede precisam ter suporte para os requisitos de largura de banda enquanto minimizam o atraso e a oscilação (que discutiremos mais adiante, ainda neste capítulo).

Mais adiante, exploraremos estes e vários outros aspectos relacionados com QoS. Primeiro, porém, fornecemos uma visão geral de várias técnicas para compactar dados de multimídia. Conforme sugerimos anteriormente, a compactação faz demandas significativas à CPU.

20.2 Compactação

Devido aos requisitos de tamanho e taxa dos sistemas de multimídia, os arquivos de multimídia normalmente são compactados da sua forma original para uma forma muito menor. Quando um arquivo tiver sido compactado, ele ocupará menos espaço para armazenamento e poderá ser entregue a um cliente mais rapidamente. A compactação é particularmente importante quando o conteúdo está sendo enviado por uma conexão da rede. Ao discutir sobre a compactação de arquivos, normalmente nos referimos à **razão de compactação**, que é a razão entre o tamanho do arquivo original e o tamanho do arquivo compactado. Por exemplo, um arquivo de 800 KB que é compactado para 100 KB possui uma razão de compactação de 8:1.

Quando um arquivo tiver sido compactado (**codificado**), ele precisará ser descompactado (**decodificado**) antes que possa ser acessado. Um recurso do algoritmo utilizado para compactar o arquivo afeta a descompactação posterior. Os algoritmos de compactação são classificados como **com perda** ou **sem perda**. Com a compactação com perda, alguns dos dados originais são perdidos quando o arquivo é decodificado, enquanto a compactação sem perda garante que o arquivo compactado sempre possa ser restaurado de volta à sua forma original. Em geral, as técnicas sem perdas fornecem razões de compactação muito mais altas. Obviamente, porém, somente certos tipos de dados podem tolerar a compactação com perda - a saber, imagens, áudio e vídeo. Os algoritmos de compactação com perda normalmente funcionam eliminando determinados dados, como frequências muito altas ou muito baixas, que o ouvido humano não pode detectar. Alguns algoritmos de compactação com perda usados com vídeo operam armazenando apenas as diferenças entre quadros sucessivos. Os algoritmos sem perda são usados para compactar arquivos de texto, como programas de computador (por exemplo, arquivos **zipados**), pois queremos restaurar esses arquivos compactados ao seu estado original.

Diversos esquemas diferentes de compactação com perda para dados de mídia contínua estão disponíveis comercialmente. Nesta seção, abordamos um utilizado pelo Moving Picture Experts Group, mais conhecido como MPEG.

O MPEG refere-se a um conjunto de formatos de arquivo e padrões de compactação para vídeo digital. Como o vídeo digital normalmente também possui uma parte de áudio, cada um dos padrões é dividido em três camadas. As camadas 3 e 2 se aplicam às partes de áudio e vídeo do arquivo de mídia. A camada 1, conhecida como **camada de sistemas**, contém informações de temporização para permitir que o player MPEG multiplexe as partes de áudio e vídeo, de modo que sejam sincronizadas durante a reprodução. Existem três padrões MPEG principais: MPEG-1, MPEG-2 e MPEG-4.

O MPEG-1 é usado para vídeo digital e seu stream de áudio associado. A resolução do MPEG-1 é de 352×240 a 30 quadros por segundo com uma taxa de bits de até 1,5 Mbps. Isso fornece uma qualidade ligeiramente inferior à dos vídeos convencionais de videocassete. Os arquivos de áudio MP3 (um meio popular para armazenar música) utilizam a camada de áudio (camada 3) do MPEG-1. Para o vídeo, o MPEG-1 pode conseguir uma razão de compactação de até 200:1, embora na prática as razões de compactação sejam muito inferiores. Como o MPEG-1 não exige altas taxas de dados, ele normalmente é usado para baixar clipes de vídeo curtos pela Internet.

O MPEG-2 fornece melhor qualidade do que o MPEG-1 e é usado para compactar filmes de DVD e televisão digital (incluindo televisão de alta definição, ou HDTV). O MPEG-2 identifica diversos **níveis** e **perfis** de compactação de vídeo. O nível refere-se à resolução do vídeo; o perfil caracteriza a qualidade do vídeo. Em geral, quanto maior o nível de resolução e melhor a qualidade do vídeo, mais alta é a taxa de dados exigida. As taxas de bits típicas para arquivos codificados por MPEG-2 são de 1,5 Mbps a 15 Mbps. Como o MPEG-2 exige taxas mais altas, ele normalmente é inadequado para entrega de vídeo por uma rede, e geralmente é usado para reprodução local.

O MPEG-4 é o mais recente dos padrões, e é usado para transmitir áudio, vídeo e gráficos, incluindo camadas de animação bidimensionais e tridimensionais. A animação torna possível para os usuários finais interagirem com o arquivo durante a reprodução. Por exemplo, um comprador de casa em potencial pode baixar um arquivo MPEG-4 e fazer um passeio virtual por uma casa que esteja pensando em comprar, passando de um cômodo para outro enquanto escolhe. Outro recurso atraente do MPEG-4 é que ele fornece um nível escalável de qualidade, permitindo a remessa por conexões de rede relativamente lentas, como modems de 56 Kbps ou por redes locais de alta velocidade, com taxas de vários megabits por segundo. Além do mais, fornecendo um nível de qualidade escalável, os arquivos de áudio e vídeo MPEG-4 podem ser enviados a dispositivos sem fio, incluindo computadores portáteis, PDAs e telefones celulares.

Todos os três padrões MPEG discutidos aqui realizam a compactação com perda para conseguir altas taxas de compactação. A ideia fundamental por trás da compactação MPEG é armazenar as diferenças entre os quadros sucessivos. Não abordamos outros detalhes de como o MPEG realiza a compactação, mas encorajamos o leitor interessado a consultar as Notas Bibliográficas ao final deste capítulo.

20.3 Requisitos dos kernels de multimídia

Como resultado das características descritas na [Seção 20.1.2](#), as aplicações de multimídia normalmente exigem níveis de serviço do sistema operacional que diferem dos requisitos das aplicações tradicionais, como processadores de textos, compiladores e planilhas. Os requisitos de tempo e taxa talvez sejam os aspectos que mais preocupam, pois a reprodução de dados de áudio e vídeo exige que os dados sejam entregues dentro de determinado prazo e a uma taxa contínua, fixa. As aplicações tradicionais normalmente não têm essas restrições de tempo e taxa.

As tarefas que exigem dados em intervalos - ou **periódicos** - constantes são conhecidas como **processos periódicos**. Por exemplo, um vídeo MPEG-1 poderia exigir uma taxa de 30 quadros por segundo durante a reprodução. Para manter essa taxa, é preciso que um quadro seja entregue aproximadamente a cada $1/30$ de segundo ou 3,34 centésimos de segundo. Para colocar isso no contexto dos prazos, vamos supor que o quadro Q_j venha após o quadro Q_i na reprodução do vídeo e que o quadro Q_i seja exibido no tempo T_0 . O prazo para exibir o quadro Q_j é de 3,34 centésimos de segundo após o tempo T_0 . Se o sistema operacional for incapaz de exibir o quadro nesse prazo, o quadro será omitido do stream.

Como já dissemos, os requisitos e prazos de taxa são conhecidos como requisitos de qualidade de serviço (QoS). Existem três níveis de QoS:

1. **Serviço de melhor esforço.** O sistema faz a tentativa do melhor esforço para satisfazer os requisitos, porém nenhuma garantia é feita.
2. **QoS flexível.** Esse nível trata diferentes tipos de tráfego de diferentes maneiras, dando a certo tráfego streams de prioridade maior do que outros streams. Contudo, assim como o serviço de melhor esforço, nenhuma garantia é feita.
3. **QoS rígida.** Os requisitos de qualidade de serviço são garantidos.

Os sistemas operacionais tradicionais - os sistemas que discutimos no texto até aqui - normalmente fornecem apenas o serviço do melhor esforço e contam com **provisão adicional**, ou seja, eles consideram que a quantidade total de recursos disponíveis tenderá a ser maior que uma carga de trabalho do pior caso exigiria. Se a demanda ultrapassar a capacidade do recurso, deverá haver intervenção manual, e um processo (ou vários processos) precisa ser removido do sistema. Contudo, os sistemas de multimídia da próxima geração não podem fazer tais suposições. Esses sistemas precisam fornecer aplicações de mídia contínua, com as garantias se tornando possíveis pela QoS rígida. Portanto, no restante desta discussão, quando nos referirmos a QoS, queremos dizer QoS rígida. Em seguida, exploraremos diversas técnicas que permitem que os sistemas multimídia forneçam tais garantias de nível de serviço.

Existem diversos parâmetros definindo QoS para aplicações de multimídia, incluindo os seguintes:

- **Throughput.** Throughput é a quantidade total de trabalho feito durante determinado intervalo. Para aplicações de multimídia, o throughput é a taxa de dados exigida.
- **Retardo.** O retardo refere-se ao tempo decorrido desde que uma requisição é submetida inicialmente até que o resultado desejado é produzido. Por exemplo, o tempo a partir do momento que um cliente requisita um stream de mídia até que o stream é entregue é o retardo.
- **Jitter.** O jitter está relacionado com o retardo; porém, enquanto o retardo se refere ao tempo que um cliente precisa esperar para receber um stream, o jitter refere-se aos retardos que ocorrem durante a reprodução do stream. Certas aplicações de multimídia, como o streaming de tempo real por demanda, podem tolerar esse tipo de retardo. No entanto, o jitter geralmente é considerado inaceitável para aplicações de mídia contínua, pois pode significar longas pausas - ou quadros perdidos - durante a reprodução. Os clientes normalmente podem compensar o jitter colocando uma quantidade de dados em buffer - digamos, o correspondente a 5 segundos - antes de iniciar a reprodução.
- **Confiabilidade.** Confiabilidade refere-se ao modo como os erros são tratados durante a transmissão e o processamento de mídia contínua. Os erros podem ocorrer devido a pacotes perdidos na rede ou atrasos de processamento pela CPU. Nestes e em outros cenários, os erros não podem ser corrigidos, pois os pacotes normalmente chegam muito tarde para serem úteis.

A qualidade de serviço pode ser **negociada** entre o cliente e o servidor. Por exemplo, os dados de mídia contínua podem ser compactados em diferentes níveis de qualidade: quanto maior a qualidade, mais alta a taxa de dados exigida. Um cliente pode negociar uma taxa de dados específica com um servidor, combinando assim com um nível de qualidade durante a reprodução. Além do mais, muitos players de mídia permitem que o cliente configure o player de acordo com a velocidade da conexão do cliente com a rede. Isso permite que um cliente receba um serviço de streaming a uma taxa de dados específica de uma conexão em particular. Assim, o cliente está negociando a qualidade do serviço com o provedor de conteúdo.

Para fornecer garantias de QoS, os sistemas operacionais normalmente utilizam **controle de admissão**, que é a prática de admitir uma requisição de serviço apenas se o servidor tiver recursos suficientes para satisfazer a requisição. Vemos o controle de admissão com muita frequência em nossa vida diária. Por exemplo, um cinema só pode admitir clientes enquanto houver assentos

disponíveis. (Há também muitas situações na vida diária em que o controle de admissão não é prático, mas seria desejável!) Se nenhuma política de controle de admissão fosse usada em um ambiente de multimídia, as demandas sobre o sistema poderiam se tornar tão grandes que o sistema se tornaria incapaz de atender suas garantias de QoS.

No [Capítulo 6](#), discutimos o uso de semáforos como um método de implementação de uma política de controle de admissão simples. Nesse cenário, existe um número finito de recursos não compartilháveis. Quando um recurso for requisitado, atenderemos a requisição somente se houver recursos suficientes; caso contrário, o processo requisitante deverá esperar até que haja um recurso disponível. Podemos usar semáforos para implementar uma política de controle de admissão primeiro inicializando um semáforo para o número de recursos disponíveis. Cada requisição de recurso é feita por meio de uma operação `wait()` no semáforo; um recurso é liberado com uma chamada a `signal()` no semáforo. Quando todos os recursos estiverem em uso, chamadas subsequentes a `wait()` serão bloqueadas até que haja um `signal()` correspondente.

Uma técnica comum para implementar o controle de admissão é usar as **reservas de recursos**. Por exemplo, os recursos em um servidor de arquivos podem incluir CPU, memória, sistema de arquivos, dispositivos e rede ([Figura 20.1](#)). Observe que os recursos podem ser exclusivos ou compartilhados e que pode haver uma instância única ou múltiplas instâncias de cada tipo de recurso. Para usar um recurso, um cliente precisa fazer uma requisição de reserva para o recurso com antecedência. Se a requisição não puder ser atendida, a reserva é negada. Um esquema de controle de admissão atribui um **gerenciador de recursos** a cada tipo de recurso. As requisições de recursos possuem requisitos de QoS associados – por exemplo, taxas de dados exigidas. Quando chega uma requisição de recurso, o gerenciador de recursos determina se o recurso pode atender as demandas de QoS da requisição. Se não, a requisição pode ser rejeitada ou um nível de QoS inferior pode ser negociado entre o cliente e o servidor. Se a requisição for aceita, o gerenciador de recursos reserva os recursos para o cliente requisitante, garantindo assim ao cliente que os requisitos de QoS desejados serão atendidos. Na [Seção 20.7.2](#), examinamos o algoritmo de controle de admissão usados para assegurar as garantias de QoS no servidor de armazenamento de multimídia CineBlitz.

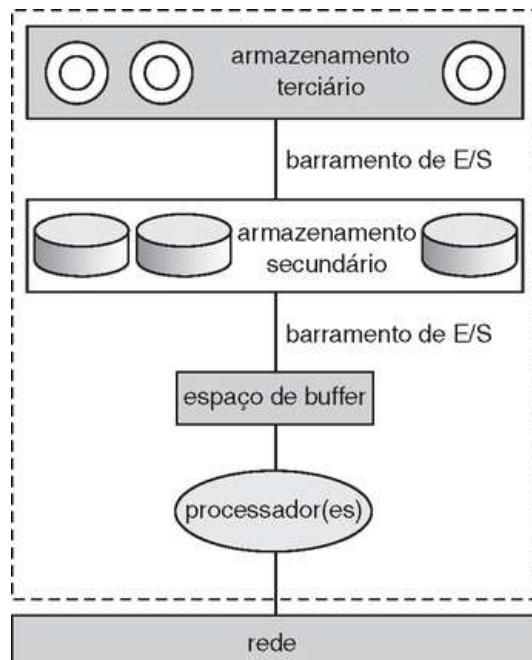


FIGURA 20.1 Recursos em um servidor de arquivos.

20.4 Escalonamento de CPU

No [Capítulo 19](#), que explicou os sistemas de tempo real, distinguimos entre **sistemas de tempo real flexíveis** e **sistemas de tempo real rígidos**. Os sistemas de tempo real flexíveis dão prioridade de escalonamento aos processos críticos. Um sistema de tempo real flexível garante que um processo crítico receberá preferência em relação ao processo não crítico, mas não fornece garantias com relação a quando o processo crítico será escalonado. Contudo, um requisito típico de mídia contínua é que os dados devem ser entregues a um cliente em determinado prazo; os dados que não chegam no prazo são inúteis. Os sistemas de multimídia, assim, exigem um escalonamento de tempo real rígido para garantir que uma tarefa crítica será atendida dentro de um período de tempo garantido.

Outra questão de escalonamento se refere à utilização, por um algoritmo de escalonamento, de **prioridade estática** ou **prioridade dinâmica** - uma distinção que discutimos inicialmente no [Capítulo 5](#). A diferença entre as duas é que a prioridade de um processo permanecerá inalterada se o escalonador lhe atribuir uma prioridade estática. Os algoritmos de escalonamento que atribuem prioridades dinâmicas permitem que as prioridades mudem com o tempo. A maioria dos sistemas operacionais utiliza prioridades dinâmicas quando escalonam tarefas que não sejam de tempo real com a intenção de dar prioridade mais alta aos processos interativos. Contudo, ao escalaronar tarefas de tempo real, a maioria dos sistemas atribui prioridades estáticas, pois o projeto do escalonador é menos complexo.

Várias estratégias de escalonamento de tempo real discutidas na [Seção 19.5](#) podem ser usadas para atender os requisitos de QoS de taxa e prazo das aplicações de mídia contínua.

20.5 Escalonamento de disco

Discutimos sobre o escalonamento de disco inicialmente no [Capítulo 12](#). Lá, focalizamos principalmente os sistemas que tratam de dados convencionais; para esses sistemas, os objetivos do escalonamento são imparcialidade e throughput. Como resultado, a maioria dos escalonadores de disco tradicionais emprega alguma forma do algoritmo SCAN ([Seção 12.4.3](#)) ou C-SCAN ([Seção 12.4.4](#)).

No entanto, os arquivos de mídia contínua possuem duas restrições que os arquivos de dados convencionais geralmente não têm: prazos de tempo e requisitos de taxa. Essas duas restrições precisam ser satisfeitas para preservar as garantias de QoS, e os algoritmos de escalonamento de disco precisam ser otimizados para as restrições. Infelizmente, essas duas restrições normalmente estão em conflito. Os arquivos de mídia contínua normalmente exigem taxas muito altas de largura de banda de disco para satisfazer seus requisitos de taxa de dados. Como os discos possuem taxas de transferência relativamente baixas e taxas de latência relativamente altas, os escalonadores de disco precisam reduzir os tempos de latência para assegurar uma largura de banda alta. Contudo, reduzir os tempos de latência pode resultar em uma política de escalonamento que não prioriza de acordo com os prazos. Nesta seção, exploramos dois algoritmos de escalonamento de disco que cumprem os requisitos de QoS para sistemas de mídia contínua.

20.5.1 Escalonamento pelo prazo mais antigo primeiro

Apresentamos o algoritmo do prazo mais antigo primeiro (EDF) na [Seção 19.5.2](#), como um exemplo de um algoritmo de escalonamento de CPU que atribui prioridades de acordo com os prazos. O EDF também pode ser usado como um algoritmo de escalonamento de disco; nesse contexto, o EDF utiliza uma fila para ordenar requisições de acordo com o tempo em que cada requisição precisa ser completada (seu prazo). O EDF é semelhante ao algoritmo do menor tempo de busca primeiro (SSTF), que foi discutido na [Seção 12.4.2](#), exceto que, em vez de atender a requisição mais próxima do cilindro atual, atendemos as requisições de acordo com o prazo - a requisição com o prazo mais próximo é atendida primeiro.

O problema com essa técnica é que atender as requisições estritamente de acordo com o prazo pode resultar em tempos de busca maiores, pois as cabeças do disco podem se mover aleatoriamente por todo o disco sem qualquer consideração com sua posição atual. Por exemplo, suponha que uma cabeça de disco esteja atualmente no cilindro 75 e a fila de cilindros (ordenada de acordo com os prazos) seja 98, 183, 105. Sob o escalonamento EDF estrito, a cabeça do disco se moverá de 75 para 98, depois para 183 e depois de volta para 105. Observe que a cabeça passa pelo cilindro 105 enquanto trafega de 98 para 183. É possível que o escalonador de disco pudesse ter atendido a requisição para o cilindro 105 a caminho do cilindro 183 e ainda ter preservado o requisito de prazo para o cilindro 183.

20.5.2 Escalonamento SCAN-EDF

O problema fundamental com o escalonamento EDF estrito é que ele ignora a posição das cabeças de leitura e escrita do disco; é possível que as cabeças se movimentem loucamente para frente e para trás no disco, ocasionando tempos de busca inaceitáveis, que afetam negativamente o throughput do disco. Lembre-se de que esse é o mesmo problema enfrentado com o escalonamento FCFS ([Seção 12.4.1](#)). No contexto do escalonamento de CPU, podemos resolver esse problema adotando o escalonamento SCAN, onde o braço do disco se move em uma direção, atendendo as requisições de acordo com sua proximidade com o cilindro atual. Quando o braço do disco atinge o final do disco, ele começa a se mover na direção contrária. Essa estratégia otimiza os tempos de busca.

O SCAN-EDF é um algoritmo híbrido, que combina o EDF com o escalonamento SCAN. O SCAN-EDF começa com a ordenação EDF, mas atende as requisições com o mesmo prazo usando a ordem SCAN. E se várias requisições tiverem prazos diferentes e relativamente próximos? Nesse caso, o SCAN-EDF pode agrupar requisições, usando a ordenação SCAN para atender as requisições no mesmo lote. Existem muitas técnicas de agrupamento de requisições com prazos semelhantes; o único requisito é que a reordenação das requisições dentro de um lote não deve impedir que uma requisição seja atendida em seu prazo. Se os prazos forem distribuídos uniformemente, os lotes podem ser organizados em grupos de determinado tamanho - digamos, 10 requisições por lote.

Outra técnica é agrupar as requisições cujos prazos caem dentro de determinado patamar de tempo - digamos, 100 milissegundos. Vamos considerar um exemplo em que agrupamos as requisições dessa maneira. Suponha que tenhamos as seguintes requisições, cada uma com um prazo especificado (em milissegundos) e um cilindro requisitado:

requisição	prazo	cilindro
A	150	25
B	201	112
C	399	95
D	94	31
E	295	185
F	78	85
G	165	150
H	125	101
I	300	85
J	210	90

Suponha que estejamos no *tempo*, o cilindro sendo atendido atualmente seja 50 e a cabeça de disco esteja se movendo para o cilindro 51. De acordo com nosso esquema de lotes, as requisições D e F estarão no primeiro lote; A, G e H no lote 2; B, E e J no lote 3; e C e I no último lote. As requisições dentro de cada lote serão ordenadas de acordo com a ordem SCAN. Assim, no lote 1, primeiro atenderemos a requisição F e depois a requisição D. Observe que estamos movendo para baixo nos números de cilindro, de 85 para 31. No lote 2, primeiro atendemos a requisição A; depois as cabeças começam a se mover para cima nos cilindros, atendendo as requisições H e depois G. O lote 3 é atendido na ordem E, B, J. As requisições I e C são atendidas no lote final.

20.6 Gerenciamento de rede

Talvez a principal questão de QoS com os sistemas de multimídia trate da preservação dos requisitos de taxa. Por exemplo, se um cliente quiser ver um vídeo compactado com MPEG-1, a qualidade de serviço dependerá bastante da capacidade do sistema de remeter os quadros na taxa requisitada.

Nossa explicação sobre questões como algoritmos de escalonamento de CPU e disco focalizou como essas técnicas podem ser usadas para atender melhor os requisitos de qualidade de serviço das aplicações de multimídia. Contudo, se o arquivo de mídia estiver sendo enviado por uma rede – talvez a Internet –, questões relacionadas com a maneira como a rede remete os dados de multimídia também podem afetar significativamente o modo como as demandas de QoS são atendidas. Nesta seção, exploramos diversas questões de rede relacionadas com as demandas exclusivas da mídia contínua.

Antes de prosseguirmos, vale a pena observar que as redes de computador em geral – e a Internet em particular – atualmente não fornecem protocolos de rede que possam garantir a remessa de dados com requisitos de tempo. (Existem alguns protocolos próprios – principalmente aqueles trabalhando em roteadores Cisco – que permitem que certo tráfego de rede seja priorizado para atender os requisitos de QoS. Esses protocolos próprios não são generalizados para uso pela Internet e, portanto, não são discutidos aqui.)

Quando os dados são roteados por uma rede, é provável que a transmissão encontre congestionamentos, atrasos e outras questões de tráfego de rede – questões que estão além do controle do criador dos dados. Para dados de multimídia com requisitos de tempo, quaisquer questões de tempo precisam ser sincronizadas entre os hosts nas pontas: o servidor remetendo o conteúdo e o cliente que o reproduz.

Um protocolo que resolve questões de tempo é o **protocolo de transporte de tempo real** (Real-time Transport Protocol – RTP). O RTP é um padrão da Internet para a remessa de dados de tempo real, incluindo áudio e vídeo. Ele pode ser usado para transportar formatos de mídia como arquivos de áudio MP3 e arquivos de vídeo compactados usando MPEG. O RTP não fornece quaisquer garantias de QoS; em vez disso, fornece recursos que permitem que um receptor remova o jitter introduzido pelos retardos e congestionamentos na rede.

Nas próximas seções, consideraremos duas outras técnicas para lidar com os requisitos exclusivos da mídia contínua.

20.6.1 Unicasting e multicasting

Em geral, existem três métodos para entregar o conteúdo de um servidor para um cliente por uma rede:

- **Unicasting.** O servidor remete o conteúdo para um único cliente. Se o conteúdo estiver sendo entregue a mais de um cliente, o servidor precisará estabelecer um unicast separado para cada cliente.
- **Broadcasting.** O servidor remete o conteúdo a todos os clientes, independentemente de eles quererem receber o conteúdo ou não.
- **Multicasting.** O servidor remete o conteúdo a um grupo de receptores que indicam que desejam receber esse conteúdo; esse método fica em algum ponto entre o unicasting e o broadcasting.

Um problema com a remessa por unicasting é que o servidor precisa estabelecer uma sessão unicast separada para cada cliente. Isso parece um desperdício especialmente para o streaming de tempo real ao vivo, onde o servidor precisa fazer várias cópias do mesmo conteúdo, uma para cada cliente. Obviamente, o broadcasting nem sempre é apropriado, pois nem todos os clientes podem querer receber o stream. (Basta dizer que o broadcasting normalmente só é usado por redes locais e não é possível pela Internet pública.)

O multicasting parece ser um meio-termo razoável, pois permite que o servidor remeta uma única cópia do conteúdo a todos os clientes, indicando que desejam recebê-la. A dificuldade com o multicasting de um ponto de vista prático é que os clientes precisam estar fisicamente próximos do servidor ou de roteadores intermediários, que repassam o conteúdo do servidor de origem. Se a rota do servidor até o cliente tiver que atravessar roteadores intermediários, os roteadores também precisarão admitir o multicasting. Se essas condições não forem atendidas, os atrasos que ocorrem durante o roteamento podem resultar em violação dos requisitos de tempo da mídia contínua. No pior dos casos, se um cliente estiver conectado a um roteador intermediário que não admite o multicasting, o cliente será incapaz de receber qualquer stream de multicast!

Atualmente, a maior parte da mídia de streaming é remetida por canais unicast; contudo, o multicasting é usado em diversas áreas onde a organização do servidor e dos clientes é conhecida com antecedência. Por exemplo, uma empresa com vários sites em um país pode ser capaz de garantir que todos os sites sejam conectados a roteadores de multicasting e estejam dentro de uma proximidade física razoável dos roteadores. A organização, então, será capaz de fornecer uma apresentação do presidente usando multicasting.

20.6.2 Protocolo de streaming em tempo real

Na [Seção 20.1.1](#), descrevemos algumas características da mídia streaming. Conforme observamos lá, os usuários podem ser capazes de acessar aleatoriamente um stream de mídia, talvez voltando ou pausando, como fariam com um aparelho de DVD. Como isso é possível?

Para responder a essa pergunta, vamos considerar como a mídia streaming é remetida aos clientes. Uma técnica é enviar a mídia por um servidor Web padrão usando o protocolo de transporte de hipertexto ou HTTP - o protocolo usado para remeter documentos de um servidor Web. Frequentemente, os clientes usam um **player de mídia**, como QuickTime, RealPlayer ou Windows Media Player, para reproduzir a mídia remetida de um servidor Web padrão. Normalmente, o cliente primeiro requisita um **metafile**, que contém o local (possivelmente identificado por um URL - Uniform Resource Locator) do arquivo de mídia streaming. Esse metafile é remetido ao navegador Web do cliente, e o navegador então inicia o player de mídia apropriado, de acordo com o tipo de mídia especificado pelo metafile. Por exemplo, um stream Real Audio exigiria o RealPlayer, enquanto o Windows Media Player seria usado para reproduzir mídia de streaming do Windows. O player de mídia, então, entra em contato com o servidor Web e requisita a mídia streaming. O stream é remetido do servidor Web para o player de mídia usando requisições HTTP padrão. Esse processo é esboçado na [Figura 20.2](#).

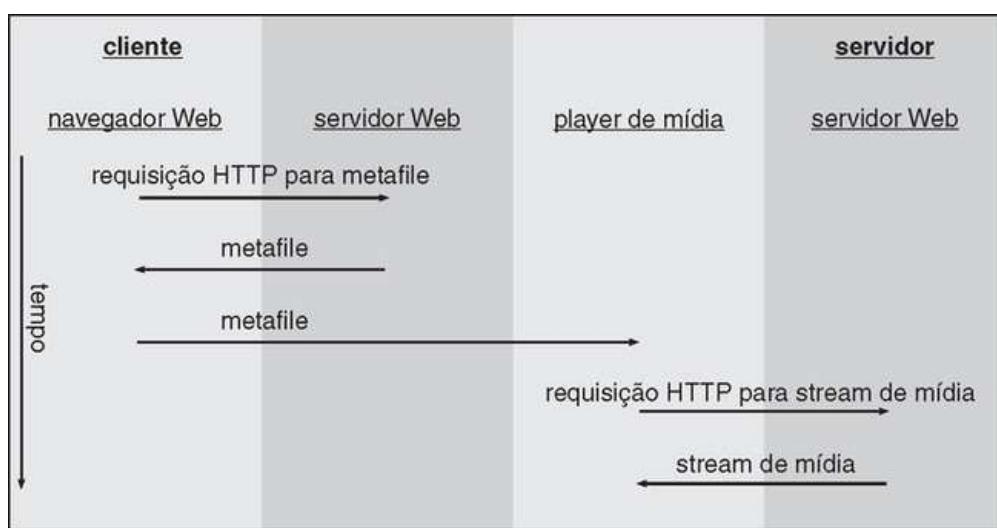


FIGURA 20.2 Mídia streaming de um servidor Web convencional.

O problema com a remessa de mídia streaming de um servidor Web padrão é que o HTTP é considerado um protocolo **sem estado**; assim, um servidor Web não mantém o estado (ou status) de sua conexão com um cliente. Como resultado, é difícil que um cliente pare durante a remessa de conteúdo de mídia streaming, pois a pausa exigiria que o servidor Web soubesse onde deveria iniciar no stream quando o cliente quisesse retomar a reprodução.

Uma estratégia alternativa é usar um servidor streaming especializado, que é criado especificamente para mídia streaming. Um protocolo criado para a comunicação entre os servidores streaming e players de mídia é conhecido como **protocolo de streaming em tempo real**, ou **RTSP**. A vantagem significativa que o RTSP fornece em relação ao HTTP é uma conexão com estado entre o cliente e o servidor, que permite que o cliente interrompa ou busque posições aleatórias no stream durante a reprodução. A remessa de mídia streaming usando RTSP é semelhante à remessa usando HTTP ([Figura 20.2](#)), pois o metafile é entregue usando um servidor Web convencional. Contudo, em vez de um servidor Web, a mídia streaming é entregue a partir de um servidor streaming usando o protocolo RTSP. A operação do RTSP é mostrada na [Figura 20.3](#).

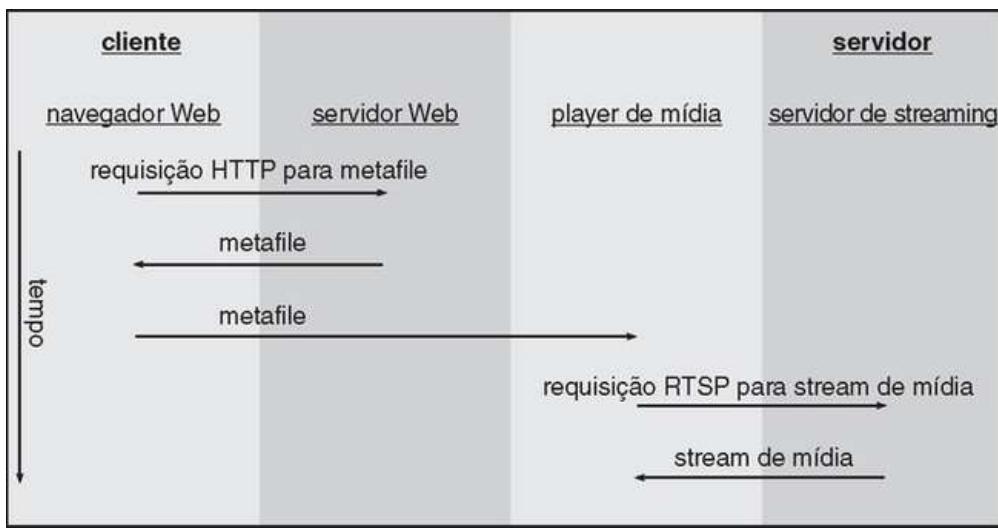


FIGURA 20.3 Protocolo de streaming em tempo real (RTSP).

O RTSP define vários comandos como parte do seu protocolo; esses comandos são enviados de um cliente para um servidor streaming RTSP. Os comandos incluem os seguintes:

- **SETUP.** O servidor aloca recursos para uma sessão do cliente.
- **PLAY.** O servidor remete um stream para uma sessão do cliente estabelecida a partir de um comando SETUP.
- **PAUSE.** O servidor suspende a remessa de um stream, mas mantém os recursos para a sessão.
- **TEARDOWN.** O servidor interrompe a conexão e libera os recursos alocados para a sessão.

Os comandos podem ser ilustrados com uma máquina de estado para o servidor, como mostra a [Figura 20.4](#). Como você pode ver na figura, o servidor RTSP pode estar em um dentre três estados: **init**, **ready** e **playing**. As transições entre esses três estados são disparadas quando o servidor recebe um dos comandos RTSP do cliente.

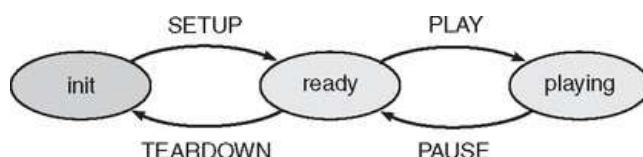


FIGURA 20.4 Máquina de estado finito representando o RTSP.

Usar RTSP em vez de HTTP para mídia streaming fornece várias outras vantagens, mas elas estão relacionadas principalmente com aspectos de rede e, portanto, estão além do escopo deste texto. Encorajamos os leitores interessados a consultarem as Notas Bibliográficas ao final deste capítulo, em busca de fontes de mais informações.

20.7 Um exemplo: CineBlitz

O servidor de armazenamento multimídia CineBlitz é um servidor de mídia de alto desempenho que admite mídia contínua com requisitos de taxa (como vídeo e áudio) e dados convencionais sem requisitos de taxa associados (como texto e imagens). O CineBlitz refere-se aos clientes com requisitos de taxa como **clientes de tempo real**, enquanto os **clientes não de tempo real** não possuem restrições de taxa. O CineBlitz garante atender os requisitos de taxa dos clientes de tempo real implementando um controlador de admissão, admitindo um cliente apenas se houver recursos suficientes para permitir a recuperação de dados na taxa exigida. Nesta seção, exploramos os algoritmos de escalonamento de disco e controle de admissão do CineBlitz.

20.7.1 Escalonamento de disco

O escalonador de disco do CineBlitz atende as requisições em **ciclos**. No início de cada ciclo de serviço, as requisições são colocadas na ordem C-SCAN ([Seção 12.4.4](#)). Lembre-se, pelas discussões anteriores sobre C-SCAN, que as cabeças de disco se movem de uma ponta do disco para a outra. Porém, em vez de reverter a direção quando atingem o final do disco, como no escalonamento de disco SCAN puro ([Seção 12.4.3](#)), as cabeças de disco se movem de volta ao início do disco.

20.7.2 Controle de admissão

O algoritmo de controle de admissão no CineBlitz precisa monitorar as requisições dos clientes de tempo real e não de tempo real, garantindo que as duas classes de clientes recebam atendimento. Além do mais, o controlador de admissão precisa fornecer as garantias de taxa exigidas pelos clientes de tempo real. Para garantir a imparcialidade, somente uma fração p de tempo é reservada para os clientes de tempo real, enquanto o restante, $1 - p$, é reservado para clientes não de tempo real. Aqui, exploramos o controlador de admissão para clientes de tempo real apenas; assim, o termo *cliente* refere-se a um cliente de tempo real.

O controlador de admissão no CineBlitz monitora diversos recursos do sistema, como largura de banda de disco e latência de disco, enquanto acompanha o espaço de buffer disponível. O controlador de admissão CineBlitz admite um cliente somente se houver largura de banda de disco e espaço de buffer suficientes para apanhar dados para o cliente em sua taxa exigida.

O CineBlitz enfileira as requisições para arquivos de mídia contínua, onde $R_1, R_2, R_3, \dots, R_n$ são as requisições e r_i é a taxa de dados exigida para determinada requisição R_i . As requisições na fila são atendidas em ordem cíclica usando uma técnica conhecida como **buffering duplo**, onde um buffer é alocado para cada requisição R_i de tamanho $2 \times T \times r_i$.

Durante cada ciclo I , o servidor precisa, para cada requisição R_j :

1. Apanhar os dados do disco para o buffer ($I \bmod 2$).
2. Transferir dados de buffer ($(I + 1) \bmod 2$) para o cliente.

Esse processo é ilustrado na [Figura 20.5](#). Para N clientes, o espaço de buffer total B exigido é

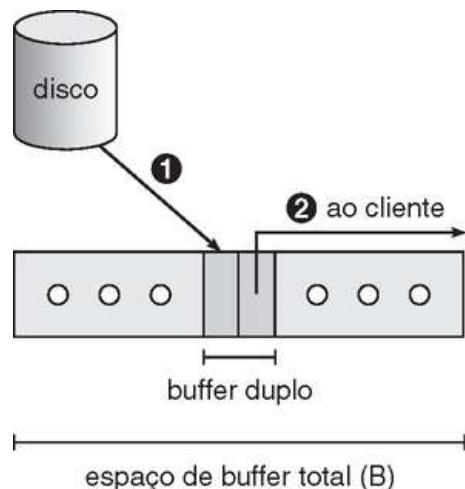


FIGURA 20.5 Buffering duplo no CineBlitz.

$$\sum_{i=1}^N 2 \times T \times r_i \leq B$$

A ideia fundamental por trás do controlador de admissão no CineBlitz é ligar as **(20.1)** requisições para entrada à fila, de acordo com os seguintes critérios:

1. O tempo de serviço para cada requisição é estimado primeiro.
2. Uma requisição é admitida somente se a soma dos tempos de serviço estimados para todas as requisições admitidas não ultrapassar a duração do ciclo de serviço T .

Suponha que $T \times r_i$ bits sejam apanhados durante um ciclo para cada cliente de tempo real R_i com taxa r_i . Se R_1, R_2, \dots, R_n são os clientes atualmente ativos no sistema, então o controlador de admissão precisa garantir que os tempos totais para apanhar $T \times r_1, T \times r_2, \dots, T \times r_n$ bits para os clientes de tempo real correspondentes não ultrapassam T . Exploramos os detalhes dessa política de admissão no restante desta seção.

Se b é o tamanho de um bloco de disco, então o número máximo de blocos de disco que podem ser apanhados para a requisição R_k durante cada ciclo é $\lceil (T \times r_k)/b \rceil + 1$. O 1 nessa fórmula vem do fato de que, se $T \times r_k$ for menor que b , então é possível que $T \times r_k$ bits se espalhem pela última parte de um bloco de disco e o início de outro, fazendo dois blocos serem apanhados. Sabemos que a recuperação de um bloco de disco envolve (a) uma busca para a trilha contendo o bloco; e (b) o retardo rotacional à medida que os dados na trilha desejada chegam sob a cabeça do disco. Conforme descrevemos, o CineBlitz utiliza um algoritmo de escalonamento de disco C-SCAN, de modo que os blocos de disco são apanhados na ordem classificada de suas posições no disco.

Se t_{busca} e t_{rot} se referem aos tempos de busca e atraso rotacional no pior caso, a latência máxima contraída para atender a N requisições é

$$2 \times t_{busca} + \sum_{i=1}^N \left(\left\lceil \frac{T \times r_i}{b} \right\rceil \right) \times t_{rot}$$

Nessa equação, o componente $2 \times t_{busca}$ refere-se à latência de busca de disco máxima **(20.2)** contraída em um ciclo. O segundo componente reflete a soma das recuperações dos blocos de disco multiplicada pelo atraso rotacional no pior caso.

Se a taxa de transferência do disco é r_{disco} , então o tempo para transferir $T \times r_k$ bits de dados para a requisição R_k é $(T \times r_k)/r_{disco}$. Como resultado, o tempo total para apanhar $T \times r_1, T \times r_2, \dots, T \times r_n$ bits para as requisições R_1, R_2, \dots, R_n é a soma da equação 20.2 e

$$\sum_{i=1}^N \frac{T \times r_i}{r_{disco}}$$

Portanto, o controlador de admissão no CineBlitz só admite um novo cliente R_i se pelo **(20.3)** menos $2 \times T \times r_i$ bits de espaço livre no buffer estiverem disponíveis para o cliente e a seguinte equação for satisfeita:

$$2 \times t_{busca} + \sum_{i=1}^N \left(\left\lceil \frac{T \times r_i}{b} \right\rceil + 1 \right) \times t_{rot} + \sum_{i=1}^N \frac{T \times r_i}{r_{disco}} \leq T \quad (20.4)$$

20.8 Resumo

As aplicações de multimídia estão em uso comum nos sistemas de computador modernos. Os arquivos de multimídia incluem arquivos de vídeo e áudio, que podem ser remetidos aos sistemas como computadores de desktop, assistentes digitais pessoais e telefones celulares. A principal distinção entre dados de multimídia e dados convencionais é que os dados de multimídia possuem requisitos específicos de taxa e prazo. Como os arquivos de multimídia possuem requisitos de tempo específicos, os dados precisam ser compactados com frequência antes da remessa para reprodução por um cliente. Os dados de multimídia podem ser remetidos do sistema de arquivos local ou de um servidor de multimídia por uma conexão de rede, usando uma técnica conhecida como streaming.

Os requisitos de temporização dos dados de multimídia são conhecidos como requisitos de qualidade de serviço, e os sistemas operacionais convencionais normalmente não podem fazer garantias de qualidade de serviço. Para fornecer qualidade de serviço, os sistemas de multimídia precisam fornecer uma forma de controle de admissão, pela qual um sistema aceita uma requisição somente se puder atender o nível de qualidade de serviço especificado pela requisição. Fornecer garantias de qualidade de serviço exige a avaliação de como um sistema operacional realiza escalonamento de CPU, escalonamento de disco e gerenciamento de rede. O escalonamento de CPU e disco normalmente utiliza os requisitos de prazo de uma tarefa de mídia contínua como um critério de escalonamento. O gerenciamento de rede exige o uso dos protocolos que lidam com atraso e jitter causados pela rede, além de permitir que um cliente interrompa ou passe para diferentes posições no stream durante a reprodução.

Exercícios

- 20.1. Forneça exemplos de aplicações de multimídia que são remetidas pela Internet.
- 20.2. Faça a distinção entre download progressivo e streaming de tempo real.
- 20.3. Quais dos seguintes tipos de aplicações streaming de tempo real podem tolerar atraso? Quais podem tolerar jitter?
- Streaming de tempo real ao vivo
 - Streaming de tempo real por demanda
- 20.4. Discuta quais técnicas poderiam ser usadas para atender os requisitos de qualidade de serviço para aplicações de multimídia nos seguintes componentes de um sistema:
- Escalonador de processo
 - Escalonador de disco
 - Gerenciador de memória
- 20.5. Explique por que os protocolos tradicionais da Internet para transmitir dados não são suficientes para fornecer garantias de qualidade de serviço exigidas para um sistema multimídia. Discuta quais mudanças são exigidas para fornecer garantias de QoS.
- 20.6. Suponha que um arquivo de vídeo digital esteja sendo exibido a uma taxa de 30 quadros por segundo; a resolução de cada quadro é de 640×480 , e 24 bits estão sendo usados para representar cada cor. Supondo que nenhuma compactação esteja sendo usada, qual é a largura de banda necessária para remeter esse arquivo? Em seguida, supondo que o arquivo esteja sendo compactado a uma razão de 200:1, qual é a largura de banda necessária para remeter o arquivo compactado?
- 20.7. Uma aplicação de multimídia consiste em um conjunto contendo 100 imagens, 10 minutos de vídeo e 10 minutos de áudio. Os tamanhos compactados das imagens, vídeo e áudio são 500 MB, 550 MB e 8 MB, respectivamente. As imagens foram compactadas a uma razão de 15:1, e o vídeo e o áudio foram compactados a 200:1 e 10:1, respectivamente. Quais eram os tamanhos das imagens, vídeo e áudio antes da compactação?
- 20.8. Suponha que queremos compactar um arquivo de vídeo digital usando a tecnologia MPEG-1. A taxa de bit de destino é de 1,5 Mbps. Se o vídeo for exibido em uma resolução de 352×240 a 30 quadros por segundo usando 24 bits para representar cada cor, qual é a razão de compactação necessária para conseguir a taxa de bits desejada?
- 20.9. Considere dois processos, P_1 e P_2 , onde $p_1 = 50$, $t_1 = 25$, $p_2 = 75$ e $t_2 = 30$.
- Esses dois processos podem ser escalonados usando o escalonamento de taxa monotônica? Ilustre sua resposta usando um diagrama de Gantt.
 - Ilustre o escalonamento desses dois processos usando o escalonamento do prazo mais antigo primeiro (EDF).
- 20.10. A tabela a seguir contém diversas requisições com seus prazos e cilindros associados. As requisições com prazos ocorrendo dentro de 100 milissegundos um do outro serão agrupadas em lotes. A cabeça de disco está atualmente no cilindro 94 e está movendo para o cilindro 95. Se o escalonamento de disco SCAN-EDF for usado, como essas requisições podem ser agrupadas em lote e qual é a ordem das requisições dentro de cada lote?

requisição	prazo	cilindro
R1	57	77
R2	300	95
R3	250	25
R4	88	28
R5	85	100
R6	110	90
R7	299	50
R8	300	77
R9	120	12
R10	212	2

- 20.11. Repita o [Exercício 20.10](#), mas desta vez com as requisições de lote que possuem prazos ocorrendo dentro de 75 milissegundos uma da outra.
- 20.12. Compare unicasting, multicasting e broadcasting como técnicas para remeter o conteúdo por uma rede de computadores.
- 20.13. Descreva por que o HTTP normalmente é insuficiente para remeter mídia streaming.
- 20.14. Que princípio de operação é usado pelo sistema CineBlitz na realização do controle de admissão para requisições de arquivos de mídia?

Notas bibliográficas

Fuhrt [1994] fornece uma visão geral dos sistemas de multimídia. Os tópicos relacionados com a remessa de multimídia pelas redes podem ser encontrados em Kurose e Ross [2005]. O suporte do sistema operacional para multimídia é discutido em Steinmetz [1995] e Leslie e outros [1996]. O gerenciamento de recursos para recursos como capacidade de processamento e buffers de memória são discutidos em Mercer e outros [1994] e Druschel e Peterson [1993]. Reddy e Wyllie [1994] dão uma boa visão geral de questões relacionadas com o uso de E/S em um sistema de multimídia. As discussões relacionadas com o modelo de programação apropriado para o desenvolvimento de aplicações de multimídia são apresentadas em Regehr e outros [2000]. Um sistema de controle de admissão para um escalonador de taxa monotônica é considerado em Lauzac e outros [2003]. Bolosky e outros [1997] apresentam um sistema para servir dados de vídeo e discutem as questões de gerenciamento de escalonamento que surgem em tal sistema. Os detalhes de um protocolo de streaming em tempo real podem ser encontrados em <http://www.rtsp.org>. Tudor [1995] fornece um tutorial sobre MPEG-2. Um tutorial sobre técnicas de compactação de vídeo pode ser encontrado em <http://www.wave-report.com/tutorials/VC.htm>.

PARTE VIII

ESTUDOS DE CASO

ESBOÇO

- [Capítulo 28: Introdução a Estudos de caso](#)
- [Capítulo 29: O sistema Linux](#)
- [Capítulo 30: Windows XP](#)
- [Capítulo 31: Sistemas operacionais marcantes](#)

Introdução a Estudos de caso

Agora, podemos integrar os conceitos descritos neste livro descrevendo sistemas operacionais reais. Dois desses sistemas são explicados com muitos detalhes - Linux e Windows XP. Escolhemos o Linux por diversos motivos: ele é popular, está disponível gratuitamente e representa um sistema UNIX com todos os recursos. Isso dá ao aluno de sistemas operacionais uma oportunidade de ler - e modificar - código-fonte de um sistema operacional *real*.

Também incluímos o Windows XP com mais detalhes. Este sistema operacional relativamente recente, da Microsoft, ganhou muita popularidade, tanto no mercado de máquinas isoladas quanto no mercado de servidores de grupo de trabalho. Escolhemos o Windows XP porque ele oferece uma oportunidade para estudarmos um sistema operacional moderno, que possui um projeto e uma implementação muito diferentes do UNIX.

Além destes, discutimos rapidamente outros sistemas operacionais altamente influentes. Escolhemos a ordem de apresentação para realçar as semelhanças e diferenças entre os sistemas; ela não é estritamente cronológica e não reflete a importância relativa dos sistemas.

Também oferecemos uma explicação on-line de três outros sistemas. O sistema FreeBSD é outro sistema UNIX. Porém, enquanto o Linux combina recursos de vários sistemas UNIX, FreeBSD é baseado no modelo BSD do UNIX. O código-fonte do FreeBSD, assim como o código-fonte do Linux, está disponível gratuitamente. O sistema operacional Mach é um sistema operacional moderno, que oferece compatibilidade com o BSD UNIX. Windows 2000 é outro sistema operacional moderno da Microsoft, para o Intel Pentium e microprocessadores mais recentes; ele é compatível com aplicações MS-DOS e Microsoft Windows.

CAPÍTULO 21

O sistema Linux

Este capítulo apresenta um exame profundo do sistema operacional Linux. Examinando um sistema completo, real, podemos ver como os conceitos que discutimos se relacionam tanto entre si quanto com a prática.

O Linux é uma versão do UNIX que obteve muita popularidade nos últimos anos. Neste capítulo, veremos a história e o desenvolvimento do Linux, abordando as interfaces do usuário e do programador que o Linux apresenta - interfaces que devem muito à tradição do UNIX. Também discutimos sobre os métodos internos pelos quais o Linux implementa essas interfaces. O Linux é um sistema operacional em rápida evolução. Este capítulo descreve especificamente o kernel do Linux 2.6, que foi lançado no final de 2003.

OBJETIVOS DO CAPÍTULO

- Explorar a história do sistema operacional UNIX, do qual o Linux é derivado, e os princípios sobre os quais o Linux foi projetado.
- Examinar o modelo de processo do Linux e ilustrar como o Linux escalona os processos e fornece comunicação entre processos.
- Examinar a gerência de memória no Linux.
- Explorar como o Linux implementa os sistemas de arquivos e gerencia os dispositivos de E/S.

21.1 História do Linux

O Linux possui aparência e estilo muito semelhantes aos de outros sistemas UNIX; na realidade, a compatibilidade com o UNIX tem sido um objetivo de projeto importante no projeto do Linux. Contudo, o Linux é muito mais recente do que a maioria dos sistemas UNIX. Seu desenvolvimento começou em 1991, quando um estudante finlandês, Linus Torvalds, escreveu e batizou o **Linux**, um kernel pequeno, porém autocontido, para o processador 80386, o primeiro verdadeiro processador de 32 bits na série de CPUs da Intel compatíveis com o PC.

Desde o início do seu desenvolvimento, o código-fonte do Linux tem estado disponível gratuitamente na Internet. Como resultado, a história do Linux tem sido de colaboração por muitos usuários do mundo inteiro, o que corresponde quase exclusivamente à Internet. Desde um kernel inicial que implementava parcialmente um pequeno subconjunto dos serviços do sistema UNIX, o sistema Linux cresceu para incluir muita funcionalidade do UNIX.

Em seus primeiros dias, o desenvolvimento do Linux girava em torno do kernel do sistema operacional - o executivo privilegiado que controla todos os recursos do sistema e que interage diretamente com o hardware do computador. Precisamos de muito mais do que esse kernel para produzir um sistema operacional completo. É importante fazermos a distinção entre o kernel do Linux e um sistema Linux. O **kernel do Linux** é um pedaço de software inteiramente original, desenvolvido do zero pela comunidade Linux. O **sistema Linux**, como o conhecemos hoje, inclui uma variedade de componentes, alguns escritos do zero, outros emprestados de outros projetos de desenvolvimento, e outros criados em colaboração com outras equipes.

O sistema Linux básico é um ambiente-padrão para aplicações e programação do usuário, mas não impõe qualquer meio padrão de controlar a funcionalidade disponível como um todo. Enquanto o Linux amadurecia, houve a necessidade de outra camada de funcionalidade em cima do sistema Linux. Isso precisa ser atendido por diversas distribuições Linux. Uma **distribuição** Linux inclui todos os componentes-padrão do sistema Linux e mais um conjunto de ferramentas administrativas para simplificar a instalação inicial e a subsequente atualização do Linux, e para controlar a instalação e remoção de outros pacotes no sistema. Uma distribuição moderna normalmente também inclui ferramentas para o gerenciamento de sistemas de arquivos, criação e gerenciamento de contas do usuário, administração de redes, navegadores Web, processadores de textos e assim por diante.

21.1.1 O kernel do Linux

O primeiro kernel do Linux lançado ao público foi a versão 0.01, datada de 14 de maio de 1991. Ele não usava rede, executava apenas em processadores Intel compatíveis com o 80386 e hardware do PC, e tinha suporte para driver de dispositivo extremamente limitado. O subsistema de memória virtual também era muito básico e não incluía suporte para arquivos mapeados na memória; porém, até mesmo essa primeira personalização admitia páginas compartilhadas com cópia na escrita. O único sistema de arquivos aceito era o sistema de arquivos Minix - os primeiros kernels do Linux foram desenvolvidos sobre uma plataforma Minix. Todavia, o kernel implementava processos UNIX apropriados, com espaços de endereços protegidos.

A próxima versão importante, o Linux 1.0, foi lançada em 14 de março de 1994. Essa versão culminou três anos de rápido desenvolvimento do kernel do Linux. Talvez o único grande recurso novo tenha sido o uso de redes: o Linux 1.0 incluía suporte para protocolos de rede TCP/IP padrão do UNIX, bem como a interface de socket compatível com BSD para a programação de redes. O suporte para driver de dispositivo foi acrescentado para a execução de IP sobre uma rede Ethernet ou (usando os protocolos PPP ou SLIP) sobre linhas seriais ou modems.

O kernel do Linux 1.0 também incluía um sistema de arquivos novo e bastante melhorado, sem as limitações do sistema de arquivos Minix original, e admitia uma série de controladores SCSI para o acesso em alto desempenho aos discos. Os desenvolvedores estenderam o subsistema de memória virtual para dar suporte à paginação para arquivos de troca e mapeamento na memória de arquivos arbitrários (mas somente o mapeamento na memória para leitura foi implementado na versão 1.0).

Um grupo extra de suporte de hardware foi acrescentado nessa versão. Embora ainda restrito à plataforma de PC da Intel, o suporte do hardware cresceu para incluir dispositivos de disquete e CD-ROM, além de placas de som, mouse e teclados internacionais. A simulação de ponto flutuante também foi acrescentada ao kernel para usuários do 80386 que não tinham o coprocessador matemático 80387, e foi implementado o mecanismo de **Interprocess Communication - IPC (comunicação entre processos)** no estilo do System V UNIX, incluindo memória compartilhada, semáforos e filas de mensagens. O suporte simples para módulos do kernel carregáveis e descarregáveis dinamicamente foi fornecido também.

Nesse ponto, o desenvolvimento foi iniciado no esforço de trabalho do kernel do 1.1, mas diversos patches para reparo de bugs foram lançados subsequentemente à versão 1.0. Esse padrão foi adotado como convenção de numeração-padrão para os kernels do Linux: kernels com um número de versão secundária ímpar, como 1.1, 1.3 ou 2.1, são **kernels de desenvolvimento**; números de

versão secundária pares são **kernels de produção** estáveis. As atualizações dos kernels estáveis são realizadas apenas para remediar algo, enquanto os kernels de desenvolvimento podem incluir funcionalidades mais novas e relativamente pouco testadas.

Em março de 1995, foi lançado o kernel da versão 1.2. Essa versão não oferecia a mesma melhoria de funcionalidade da versão 1.0, mas incluía suporte para uma faixa muito mais ampla de hardware, incluindo a nova arquitetura de barramento de hardware PCI. Os desenvolvedores acrescentaram outro recurso específico do PC - suporte para o modo 8086 virtual da CPU 80386 -, para permitir a simulação do sistema operacional DOS para computadores PC. Eles também atualizaram a pilha de rede para fornecer suporte para o protocolo IPX e tornaram a implementação do IP mais completa, incluindo contabilidade e a funcionalidade de firewall.

O kernel da versão 1.2 também foi o último kernel do Linux apenas para PC. A distribuição de fonte para o Linux 1.2 incluiu suporte parcialmente implementado para CPUs SPARC, Alpha e MIPS, mas a integração total dessas outras arquiteturas não foi iniciada antes do lançamento do kernel estável da versão 1.2.

O Linux versão 1.2 se concentrou em suporte mais amplo para o hardware e implementações mais completas da funcionalidade existente. Na época, havia muita funcionalidade nova em desenvolvimento, mas a integração do novo código ao código-fonte principal do kernel foi adiada para depois do lançamento do kernel estável da versão 1.2. Como resultado, o esforço de desenvolvimento da versão 1.3 viu muita funcionalidade nova sendo acrescentada ao kernel.

Esse trabalho foi lançado como o Linux 2.0 em junho de 1996. Essa versão recebeu um incremento de número de versão principal em consideração a duas capacidades novas: suporte para arquiteturas múltiplas, incluindo uma versão Alpha nativa em 64 bits, e suporte para arquiteturas multiprocessadas. As distribuições Linux baseadas na versão 2.0 também estão disponíveis para os processadores Motorola da série 68000 e para os sistemas SPARC da Sun. Uma versão derivada do Linux, executando em cima do Mach Microkernel, também é executada em sistemas PC e PowerMac.

As mudanças na versão 2.0 não pararam aí. O código de gerência de memória foi substancialmente melhorado, para fornecer um cache unificado para os dados do sistema de arquivos independente do caching dos dispositivos de bloco. Como resultado dessa mudança, o kernel forneceu um desempenho muito melhor para o sistema de arquivos e a memória virtual. Pela primeira vez, o caching do sistema de arquivos foi estendido para os sistemas de arquivos em rede, e as regiões mapeadas na memória também puderam ser escritas, em vez de somente lidas.

O kernel da versão 2.0 também incluiu uma melhoria no desempenho do TCP/IP, e uma série de novos protocolos de rede foi acrescentada, incluindo AppleTalk, redes de radioamador AX.25 e suporte para ISDN. Foi acrescentada a capacidade de montar volumes de rede Netware e SMB (Microsoft LanManager) remotos.

Outras melhorias importantes na versão 2.0 foram o suporte para threads internas do kernel, para tratamento de dependências entre módulos carregáveis e para carregamento automático de módulos por demanda. A configuração dinâmica do kernel no momento da execução foi muito melhorada, por meio de uma interface de configuração nova e padronizada. Recursos adicionais incluíram cotas do sistema de arquivos e classes de escalonamento de processos em tempo real compatíveis com POSIX.

Melhorias continuaram com o lançamento do Linux 2.2, em janeiro de 1999. Uma porta para sistemas UltraSPARC foi acrescentada. O suporte para redes foi melhorado, com firewalls mais flexíveis, melhor gerenciamento de roteamento e tráfego, e suporte para a janela grande do TCP e confirmações (acks) seletivas. Discos Acorn, Apple e NT puderam ser lidos e o NFS foi melhorado, sendo acrescentado um daemon NFS no modo kernel. O tratamento de sinais, interrupções e algumas operações de E/S teve locks em um nível mais minucioso do que antes, para melhorar o desempenho do multiprocessador simétrico (Symmetric Multiprocessor - SMP).

Os avanços nas versões 2.4 e 2.6 do kernel incluem melhor suporte para sistemas SMP, sistemas de arquivo journaling e melhorias no sistema de gerência de memória. O escalonador de processos foi modificado na versão 2.6, fornecendo um algoritmo de escalonamento $O(1)$ eficiente. Além disso, o kernel do Linux 2.6 agora é preemptivo, permitindo que um processo seja interrompido enquanto executa no modo kernel.

21.1.2 O sistema Linux

De muitas maneiras, o kernel do Linux forma o centro do projeto do Linux, mas outros componentes compõem o sistema operacional Linux completo. Enquanto o kernel do Linux é composto de código escrito do zero especificamente para o projeto Linux, grande parte do software de suporte, que compõe o sistema Linux, não é exclusiva do Linux, mas é comum a diversos sistemas operacionais tipo UNIX. Em particular, o Linux utiliza muitas ferramentas desenvolvidas como parte do sistema operacional BSD da Berkeley, X Window System do MIT e o projeto GNU da Free Software Foundation.

Esse compartilhamento de ferramentas tem funcionado nas duas direções. As bibliotecas principais do sistema Linux foram originadas pelo projeto GNU, mas a comunidade Linux melhorou

bastante as bibliotecas, resolvendo omissões, ineficiências e bugs. Outros componentes, como o **compilador C GNU (gcc)**, já tinham qualidade suficientemente alta para serem usados diretamente no Linux. As ferramentas de administração de rede sob o Linux foram derivadas do código desenvolvido inicialmente para o 4.3 BSD, mas derivativas de BSD mais recentes, como FreeBSD, possuem código emprestado do Linux. Exemplos incluem a biblioteca matemática de simulação de ponto flutuante da Intel e os drivers de dispositivos para hardware de som do PC.

O sistema Linux como um todo é mantido por uma rede livre de desenvolvedores colaborando pela Internet, com pequenos grupos de indivíduos tendo a responsabilidade de manter a integridade de componentes específicos. Um pequeno número de sites públicos de armazenamento por FTP (File Transfer Protocol) na Internet atua como repositórios-padrão para esses componentes. O documento **File System Hierarchy Standard (padrão de hierarquia do sistema de arquivos)** também é preservado pela comunidade Linux como uma maneira de manter a compatibilidade por meio dos diversos componentes do sistema. Esse padrão especifica o esquema geral de um sistema de arquivos Linux; ele determina sob quais nomes de diretório deverão ser armazenados arquivos de configuração, bibliotecas, binários do sistema e arquivos de dados em tempo de execução.

21.1.3 Distribuições do Linux

Teoricamente, qualquer um pode instalar um sistema Linux, apanhando as revisões mais recentes dos componentes do sistema necessários nos sites FTP e compilando-os. Nos primeiros dias do Linux, essa operação era aquela que um usuário Linux tinha de executar. Entretanto, com o amadurecimento do Linux, diversos indivíduos e grupos tentaram tornar essa tarefa menos dolorosa, fornecendo um conjunto de pacotes-padrão, previamente compilado, para facilitar a instalação.

Essas coleções, ou distribuições, incluem muito mais do que apenas o sistema Linux básico. Elas incluem utilitários extras de instalação e gerenciamento de sistemas, bem como pacotes pré-compilados e prontos para instalar com muitas das ferramentas UNIX mais comuns, como servidores de notícias, navegadores Web, ferramentas de processamento e edição de textos, e até mesmo jogos.

As primeiras distribuições controlavam esses pacotes fornecendo um meio de desempacotar todos os arquivos nos lugares apropriados. Todavia, uma das contribuições importantes das distribuições modernas é o gerenciamento avançado de pacotes. As distribuições Linux de hoje incluem um banco de dados de rastreamento de pacotes para permitir que os pacotes sejam instalados, atualizados ou removidos sem dificuldade.

A distribuição SLS, nos primeiros dias do Linux, foi a primeira coleção de pacotes Linux a ser reconhecida como uma distribuição completa. Embora ela pudesse ser instalada como uma única entidade, a SLS não possuía ferramentas de gerenciamento de pacotes, que agora são esperadas em qualquer distribuição Linux. A distribuição **Slackware** representou uma grande melhoria na qualidade geral, mesmo se também com um gerenciamento de pacotes fraco; de fato, ela ainda é uma das distribuições mais instaladas na comunidade Linux.

Desde o lançamento do Slackware, diversas distribuições Linux comerciais e não comerciais foram colocadas à disposição. **Red Hat** e **Debian** são distribuições populares, a primeira de uma empresa comercial de suporte do Linux e a segunda da comunidade Linux de software gratuito, respectivamente. Outras versões com suporte comercial do Linux incluem distribuições da **Caldera**, **Craftworks** e **WorkGroup Solutions**. Um grande segmento Linux na Alemanha resultou em várias distribuições dedicadas na linguagem alemã, incluindo versões da **SuSE** e **Unifix**. Existem muitas distribuições Linux em circulação para podermos listar todas aqui. No entanto, a variedade de distribuições não proíbe a compatibilidade entre as distribuições Linux. O formato de arquivo de pacote RPM é utilizado ou, pelo menos, entendido pela maioria das distribuições, e as aplicações comerciais distribuídas nesse formato podem ser instaladas e executadas em qualquer distribuição que possa aceitar arquivos RPM.

21.1.4 Licenciamento do Linux

O kernel do Linux é distribuído sob a GNU General Public License (GPL), cujos termos são estabelecidos pela Free Software Foundation. O Linux não é um software de domínio público: **domínio público** significa que os autores renunciaram ao direito autoral sobre o software, mas o direito autoral sobre o código do Linux ainda é mantido pelos diversos autores do código. Contudo, o Linux é um software *gratuito*, no sentido de que as pessoas podem copiá-lo, modificá-lo, usá-lo da maneira que quiserem e distribuir suas próprias cópias, sem quaisquer restrições.

As principais implicações dos termos de licenciamento do Linux são que ninguém usando Linux ou criando sua própria versão do Linux (um exercício legítimo) pode tornar o produto derivado proprietário. O software lançado sob a GPL não pode ser redistribuído como um produto apenas binário. Se você lançar um software que inclua quaisquer componentes cobertos pela GPL, então, sob a GPL, você precisa deixar o código-fonte disponível junto com quaisquer distribuições binárias. (Essa restrição não proíbe de criar – ou mesmo vender – distribuições de software apenas em binário, desde que alguém que receba os binários também tenha a oportunidade de obter o código-

fonte, por um custo de distribuição razoável.)

21.2 Princípios de projeto

Em seu projeto geral, o Linux é semelhante a qualquer outra implementação UNIX tradicional, não de microkernel. Ele é um sistema multusuário e multitarefa, com um conjunto completo de ferramentas compatíveis com o UNIX. O sistema de arquivos do Linux adere à semântica tradicional do UNIX, e o modelo de rede padrão do UNIX é implementado. Os detalhes internos do projeto do Linux foram bastante influenciados pela história do desenvolvimento desse sistema operacional.

Embora o Linux seja executado em diversas plataformas, ele foi desenvolvido exclusivamente na arquitetura PC. Muito desse desenvolvimento inicial foi executado por entusiastas individuais, em vez de instalações de desenvolvimento ou pesquisa com muita verba, de modo que, desde o início, o Linux tentou espremer o máximo de funcionalidade possível dos recursos limitados. Hoje, o Linux pode executar com confiança em uma máquina multiprocessada, com centenas de megabytes de memória principal e muitos gigabytes de espaço em disco, mas ainda é capaz de operar de forma útil com menos de 4 MB de RAM.

À medida que os PCs se tornaram mais poderosos e a memória e os discos rígidos se tornaram mais baratos, os kernels do Linux originais, mínimos, cresceram para implementar mais e mais funcionalidade do UNIX. Velocidade e eficiência ainda são objetivos de projeto importantes, mas o trabalho recente no Linux tem se concentrado em um terceiro objetivo de projeto: padronização. Um dos preços pagos pela diversidade das implementações UNIX atualmente disponíveis é que o código-fonte escrito para uma variedade não pode ser necessariamente compilado ou executado de forma correta em outra. Mesmo quando as mesmas chamadas de sistema estão presentes em dois sistemas UNIX diferentes, elas não precisam se comportar da mesma maneira. Os padrões POSIX compreendem um conjunto de especificações de diferentes aspectos do comportamento do sistema operacional. Existem documentos POSIX para a funcionalidade comum do sistema operacional e para extensões como threads de processo e operações de tempo real. O Linux foi projetado para ser compatível com documentos POSIX relevantes; pelo menos duas distribuições Linux conseguiram a certificação POSIX oficial.

Por oferecer interfaces-padrão para o programador e para o usuário, o Linux apresenta poucas surpresas para alguém acostumado com o UNIX. Não vamos detalhar essas interfaces sob o Linux. Contudo, como padrão, a interface de programação do Linux adere à semântica do UNIX SVR4, em vez do comportamento do BSD. Um conjunto separado de bibliotecas está disponível para implementar a semântica do BSD nos lugares em que os dois comportamentos são bem diferentes.

Existem muitos outros padrões no mundo do UNIX, mas a certificação total do Linux com relação a esses padrões às vezes é lenta porque normalmente a certificação só está disponível com o pagamento de uma taxa, e o custo envolvido na certificação da compatibilidade de um sistema operacional com a maioria dos padrões é substancial. No entanto, o suporte a uma grande base de aplicações é importante para qualquer sistema operacional, de modo que a implementação de padrões é um objetivo importante para o desenvolvimento do Linux, mesmo que a implementação não esteja formalmente certificada. Além do padrão POSIX básico, o Linux atualmente admite as extensões de threads POSIX - Pthreads - e um subconjunto das extensões POSIX para controle de processos em tempo real.

21.2.1 Componentes do sistema Linux

O sistema Linux é composto de três corpos principais de código, alinhados com a maioria das implementações UNIX tradicionais:

1. **Kernel.** O kernel é responsável por manter todas as abstrações importantes do sistema operacional, incluindo coisas como memória virtual e processos.
2. **Bibliotecas do sistema.** As bibliotecas do sistema definem um conjunto-padrão de funções por meio das quais as aplicações podem interagir com o kernel. Essas funções implementam grande parte da funcionalidade do sistema operacional que não precisa dos privilégios totais do código do kernel.
3. **Utilitários do sistema.** Os utilitários do sistema são programas que realizam tarefas de gerenciamento de disco individuais e especializadas. Alguns utilitários do sistema podem ser invocados apenas uma vez para inicializar e configurar algum aspecto do sistema; outros – conhecidos como *daemons* na terminologia UNIX – podem ser executados permanentemente, tratando de tarefas como responder as conexões de rede que chegam, aceitar requisições de logon dos terminais ou atualizar arquivos de log.

A Figura 21.1 ilustra os diversos componentes que compõem um sistema Linux completo. A distinção mais importante aqui é entre o kernel e tudo o mais. Todo o código do kernel é executado no modo privilegiado do processador, com acesso total a todos os recursos físicos do computador. O Linux se refere a esse modo privilegiado como o **modo kernel**. Sob o Linux, nenhum código no modo usuário está embutido no kernel. Qualquer código de suporte do sistema operacional que não precise ser executado no modo kernel é colocado nas bibliotecas do sistema.

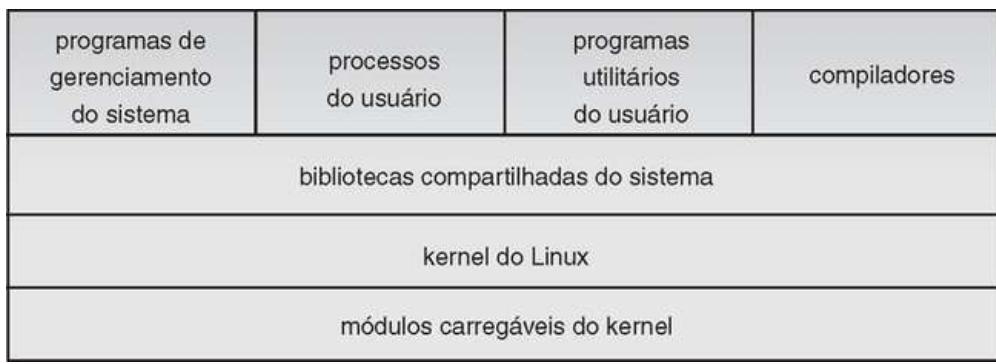


FIGURA 21.1 Componentes do sistema Linux.

Embora diversos sistemas operacionais modernos tenham adotado uma arquitetura de passagem de mensagens para o interior do seu kernel, o Linux retém o modelo histórico do UNIX: o kernel é criado como um único binário monolítico. O motivo principal é para melhorar o desempenho: como todo o código do kernel e as estruturas de dados são mantidos em um único espaço de endereços, nenhuma troca de contexto é necessária quando um processo chama uma função do sistema operacional ou quando uma interrupção de hardware é entregue. Não é apenas o código básico de escalonamento e memória virtual que ocupa esse espaço de endereços; *tudo* o código do kernel, incluindo todos os drivers de dispositivos, sistemas de arquivos e código de rede, está presente no mesmo espaço de endereços único.

Ainda com o kernel compartilhando esse mesmo espaço, há espaço para modularidade. Da mesma maneira como as aplicações do usuário podem carregar bibliotecas compartilhadas durante a execução, para puxar um código necessário, também o kernel do Linux pode carregar (e descarregar) módulos dinamicamente durante a execução. O kernel não precisa saber com antecedência quais módulos podem ser carregados – são componentes carregáveis independentes.

O kernel do Linux forma a base do sistema operacional Linux. Ele fornece toda a funcionalidade necessária para executar processos e proporciona serviços do sistema para gerar acesso arbitrado e protegido aos recursos do hardware. O kernel implementa todos os recursos exigidos para se qualificar como um sistema operacional. Contudo, por si próprio, o sistema operacional fornecido pelo kernel do Linux não se parece nada com um sistema UNIX. Ele não possui muitos dos recursos extras do UNIX, e os recursos que fornece não estão necessariamente no formato em que uma aplicação UNIX espera que apareçam. A interface do sistema operacional visível às aplicações em execução não é mantida pelo kernel. Em vez disso, as aplicações fazem chamadas às bibliotecas do sistema, que por sua vez chamam os serviços do sistema operacional conforme a necessidade.

As bibliotecas do sistema fornecem muitos tipos de funcionalidade. No nível mais simples, elas permitem que as aplicações façam requisições de serviço do sistema do kernel. Fazer uma chamada de sistema envolve a transferência do controle do modo de usuário não privilegiado para o modo do kernel privilegiado; os detalhes dessa transferência variam de uma arquitetura para outra. As bibliotecas cuidam da coleta dos argumentos da chamada de sistema e, se for necessário, da arrumação desses argumentos no formato especial necessário para fazer a chamada de sistema.

As bibliotecas também podem fornecer versões mais complexas das chamadas básicas do sistema. Por exemplo, as funções de tratamento de arquivos em buffer da linguagem C são todas implementadas nas bibliotecas do sistema, fornecendo controle mais avançado da E/S de arquivo do que as chamadas de sistema básicas do kernel. As bibliotecas também fornecem rotinas que não correspondem de forma alguma às chamadas de sistema, como algoritmos de classificação, funções matemáticas e rotinas de manipulação de strings. Todas as funções necessárias para dar suporte à execução de aplicações UNIX ou POSIX são implementadas aqui nas bibliotecas do sistema.

O sistema Linux inclui uma grande variedade de programas no modo usuário – tanto utilitários do sistema quanto utilitários do usuário. Os utilitários do sistema incluem todos os programas necessários para inicializar o sistema, como aqueles que configuram dispositivos de rede ou carregam módulos do kernel. Programas de servidor em execução contínua também contam como utilitários do sistema; esses programas tratam de requisições de login do usuário, conexões de rede que chegam e as filas de impressora.

Nem todos os utilitários-padrão servem para as principais funções de administração do sistema. O ambiente de usuário do UNIX contém grande quantidade de utilitários-padrão para realizar tarefas simples do dia a dia, como listar diretórios, mover e excluir arquivos ou exibir o conteúdo de um arquivo. Utilitários mais complexos podem realizar funções de processamento de textos, como classificar os dados textuais ou realizar pesquisas de combinação de padrão sobre o texto inserido. Juntos, esses utilitários formam um conjunto de ferramentas-padrão, que os usuários podem esperar em qualquer sistema UNIX; embora não realizando qualquer função do sistema operacional, eles são uma parte importante do sistema Linux básico.

21.3 Módulos do kernel

O kernel do Linux possui a capacidade de carregar e descarregar seções quaisquer do código do kernel por demanda. Esses módulos carregáveis do kernel são executados em um modo de kernel privilegiado e, como consequência, possuem acesso total a todas as capacidades de hardware da máquina em que são executados. Em teoria, não há restrição sobre o que um módulo do kernel tem permissão para fazer; normalmente, um módulo pode implementar um driver de dispositivo, um sistema de arquivos ou um protocolo de rede.

Os módulos do kernel são convenientes por vários motivos. O código-fonte do Linux é gratuito, de modo que qualquer um querendo escrever código do kernel é capaz de compilar um kernel modificado e reiniciar para carregar essa nova funcionalidade; porém, compilar, linkar e carregar o kernel inteiro novamente é um ciclo desajeitado para ser realizado quando se está desenvolvendo um novo driver. Se você utilizar módulos do kernel, não terá de criar um novo kernel para testar um novo driver - o driver pode ser compilado isoladamente e carregado no kernel já em execução. Naturalmente, quando um novo driver é escrito, ele pode ser distribuído como um módulo, para que outros usuários possam se beneficiar dele sem ter de recriar seus kernels.

Esse último ponto possui outra implicação. Por estar coberto pela licença GPL, o kernel do Linux não pode ser lançado com componentes proprietários incluídos nele, a menos que esses novos componentes também sejam lançados sob a GPL e o código-fonte esteja disponível por demanda. A interface do módulo do kernel permite que terceiros escrevam e distribuam, em seus próprios termos, drivers de dispositivos ou sistemas de arquivos que não poderiam ser distribuídos sob a GPL.

Os módulos do kernel permitem que um sistema Linux seja configurado com um kernel-padrão, mínimo, sem quaisquer drivers de dispositivos extras embutidos. Quaisquer drivers de dispositivos que o usuário necessite podem ser carregados explicitamente pelo sistema no boot ou carregados automaticamente pelo sistema por demanda e descarregados quando não estiverem em uso. Por exemplo, um driver de CD-ROM poderia ser carregado quando um CD fosse montado e descarregado da memória quando o CD fosse desmontado do sistema de arquivos.

O suporte do módulo sob o Linux possui três componentes:

1. O **gerenciamento de módulo** permite aos módulos serem carregados na memória e falarem com o restante do kernel.
2. O **registro do driver** permite que os módulos digam ao resto do kernel que um novo driver está disponível.
3. Um **mecanismo de solução de conflito** permite que diferentes drivers de dispositivos reservem recursos de hardware e protejam esses recursos contra uso acidental por outro driver.

21.3.1 Gerenciamento de módulos

O carregamento de um módulo exige mais do que apenas carregar seu conteúdo binário na memória do kernel. O sistema também precisa ter certeza de que quaisquer referências que o módulo faça aos símbolos ou aos pontos de entrada do kernel serão atualizadas para que apontem para os locais corretos no espaço de endereços do kernel. O Linux trata dessa atualização de referência dividindo a tarefa de carregamento de módulo em duas seções separadas: o gerenciamento das seções de código de módulo na memória do kernel e o tratamento dos símbolos que os módulos têm permissão para referenciar.

O Linux mantém uma tabela de símbolos interna no kernel. Essa tabela de símbolos não contém o conjunto completo de símbolos definidos no kernel durante sua compilação: em vez disso, um símbolo precisa ser exportado explicitamente pelo kernel. O conjunto de símbolos exportados constitui uma interface bem definida, pela qual um módulo pode interagir com o kernel.

Embora a exportação de símbolos de uma função do kernel exija uma requisição explícita pelo programador, nenhum esforço especial é necessário para importar esses símbolos para um módulo. Um escritor de módulo usa o link externo padrão da linguagem C: quaisquer símbolos externos referenciados pelo módulo, mas não declarados por ele, são marcados como não resolvidos no binário do módulo final, produzido pelo compilador. Quando um módulo tiver de ser carregado para o kernel, um utilitário do sistema primeiro verifica o módulo em busca dessas referências não resolvidas. Todos os símbolos que ainda precisam ser resolvidos são pesquisados na tabela de símbolos do kernel, e os endereços corretos desses símbolos no kernel em execução são substituídos pelo código do módulo. Somente depois disso é que o módulo é passado ao kernel para o carregamento. Se o utilitário do sistema não puder resolver quaisquer referências no módulo consultando-as na tabela de símbolos do kernel, então o módulo é rejeitado.

O carregamento do módulo é realizado em dois estágios. Primeiro, o utilitário carregador do módulo pede ao kernel para reservar uma área contínua de memória virtual do kernel para o módulo. O kernel retorna o endereço da memória alocada, e o utilitário carregador pode usar esse endereço para relocar o código de máquina do módulo para o endereço de carregamento correto. Uma segunda chamada de sistema, então, passa o módulo, mais qualquer tabela de símbolo que o

novo módulo deseja exportar, para o kernel. O próprio módulo agora é copiado literalmente para o espaço alocado, e a tabela de símbolos do kernel é atualizada com os novos símbolos, para possível uso por outros módulos ainda não carregados.

O componente final de gerenciamento de módulo é o requisitante de módulo. O kernel define uma interface de comunicação à qual um programa de gerenciamento de módulo pode se conectar. Com essa conexão estabelecida, o kernel informará ao processo de gerenciamento sempre que um processo requisitar um driver de dispositivo, sistema de arquivos ou serviço de rede que não estejam carregados e dará ao gerenciador a oportunidade de carregar esse serviço. A requisição de serviço original completará depois de o módulo ser carregado. O processo gerenciador consulta o kernel para ver se um módulo carregado dinamicamente ainda está em uso e descarrega esse módulo quando não for mais necessário de forma ativa.

21.3.2 Registro de driver

Quando um módulo é carregado, ele fica não mais do que em uma região isolada da memória, a menos que permita que o restante do kernel conheça qual nova funcionalidade ele fornece. O kernel mantém tabelas dinâmicas de todos os drivers conhecidos e fornece um conjunto de rotinas para permitir que os drivers sejam acrescentados ou removidos dessas tabelas a qualquer momento. O kernel providencia para chamar a rotina de partida de um módulo quando esse módulo for carregado, e chama a rotina de limpeza do módulo antes de esse módulo ser descarregado: essas rotinas são responsáveis por registrar a funcionalidade do módulo.

Um módulo pode registrar muitos tipos de drivers e pode registrar mais de um driver, se desejar. Por exemplo, um driver de dispositivo poderia querer registrar dois mecanismos separados para acessar o dispositivo. As tabelas de registro incluem os seguintes itens:

- **Drivers de dispositivos.** Esse drivers incluem dispositivos de caractere (como impressora, terminal ou mouse), dispositivos de bloco (incluindo todas as unidades de disco) e dispositivos de interface de rede.
- **Sistemas de arquivo.** O sistema de arquivos pode ser tudo que implemente as rotinas de chamada de sistema de arquivos virtual do Linux. Ele pode implementar um formato para armazenar arquivos em um disco, mas pode também ser um sistema de arquivos de rede, como o NFS, ou um sistema de arquivos virtual, cujo conteúdo seja gerado por demanda, como o sistema de arquivos /proc do Linux.
- **Protocolos de rede.** Um módulo pode implementar um protocolo de rede inteiro, como IPX, ou um novo conjunto de regras de filtragem de pacote para um firewall de rede.
- **Formato binário.** Esse formato especifica um modo de reconhecer e carregar um novo tipo de arquivo executável.

Além disso, um módulo pode registrar um novo conjunto de entradas nas tabelas `sysctl` e `/proc`, para permitir que o módulo seja configurado dinamicamente ([Seção 21.7.4](#)).

21.3.3 Resolução de conflitos

As implementações UNIX comerciais normalmente são vendidas para executar no hardware próprio de um fornecedor. Uma vantagem de uma solução de único fornecedor é que o vendedor do software tem uma boa ideia sobre as configurações de hardware que são possíveis. O hardware do PC, no entanto, vem em diversas configurações, com a possibilidade de inúmeros drivers para dispositivos como placas de rede, controladores SCSI e adaptadores de vídeo. O problema do gerenciamento da configuração de hardware se torna mais severo quando drivers de dispositivos modulares são aceitos, pois o conjunto de dispositivos atualmente ativos se torna dinamicamente variável.

O Linux fornece um mecanismo central para resolução de conflito, a fim de ajudar a arbitrar o acesso a certos recursos de hardware. Seus objetivos são os seguintes:

- Impedir que os módulos se choquem no acesso aos recursos de hardware.
- Impedir que **sondagens automáticas** - sondas de driver de dispositivo que detectam a configuração do dispositivo automaticamente - interfiram com os drivers de dispositivo existentes.
- Resolver conflitos entre vários drivers tentando acessar o mesmo hardware; por exemplo, quando tanto o driver de impressora paralela como o driver de rede IP da linha paralela (PLIP) tentam acessar a porta da impressora paralela.

Para essas finalidades, o kernel mantém listas de recursos de hardware alocados. O PC possui uma quantidade limitada de portas de E/S possíveis (endereços em seu espaço de endereços de E/S de hardware), linhas de interrupção e canais de DMA; quando qualquer driver de dispositivo deseja acessar tal recurso, primeiro ele deve reservar o recurso com o banco de dados do kernel. A propósito, esse requisito permite que o administrador do sistema determine exatamente quais recursos foram alocados por qual driver a qualquer momento.

Um módulo deve usar esse mecanismo para resolver com antecedência quaisquer recursos de hardware que espera utilizar. Se a reserva for rejeitada porque o recurso não está presente ou porque já está em uso, então fica a critério do módulo decidir como prosseguir. Ele poderá recusar

sua inicialização e requisitar que seja descarregado, se não puder continuar, ou então poderá prosseguir, usando recursos de hardware alternativos.

21.4 Gerência de processos

Um processo é o contexto básico dentro do qual toda a atividade requisitada pelo usuário é atendida dentro do sistema operacional. Para ser compatível com outros sistemas UNIX, o Linux precisa usar um modelo de processo semelhante aos de outras versões do UNIX. Contudo, o Linux opera de forma diferente do UNIX em alguns lugares-chave. Nesta seção, vamos rever o modelo de processo tradicional do UNIX e introduzir o modelo de threads do Linux.

21.4.1 O modelo de processo fork() e exec()

O princípio básico da gerência de processos do UNIX é separar duas operações: a criação de processos e a execução de um novo programa. Um novo processo é criado pela chamada de sistema `fork()`, e um novo programa é executado após uma chamada a `exec()`. Essas são duas funções distintamente separadas. Um novo processo pode ser criado com `fork()` sem que um novo programa seja executado - o novo subprocesso continua a executar exatamente o mesmo programa que o primeiro processo pai estava executando. Da mesma forma, a execução de um novo programa não exige que um novo processo seja criado primeiro: qualquer processo pode chamar `exec()` a qualquer momento. O programa atualmente em execução é imediatamente terminado, e o novo programa inicia sua execução no contexto do processo existente.

Esse modelo possui a vantagem de grande simplicidade. Não é preciso especificar cada detalhe do ambiente de um novo programa na chamada de sistema que executa esse programa; o novo programa é executado em seu ambiente existente. Se um processo pai deseja modificar o ambiente em que um novo programa deve ser executado, ele pode usar `fork()` e, depois, ainda executando o programa original em um processo filho, fazer quaisquer chamadas de sistema necessárias para modificar esse processo filho antes de executar o novo programa.

Sob o UNIX, então, um processo comprehende todas as informações que o sistema operacional precisa manter para rastrear o contexto de uma única execução de um único programa. No Linux, podemos separar esse contexto em uma série de seções específicas. Em geral, as propriedades do processo estão em três grupos: identidade, ambiente e contexto do processo.

21.4.1.1 Identidade do processo

A identidade de um processo consiste principalmente nos seguintes itens:

- **ID de processo (PID).** Cada processo possui um identificador exclusivo. O PID é usado para especificar o processo ao sistema operacional quando uma aplicação faz uma chamada de sistema para sinalizar, modificar ou esperar pelo processo. Identificadores adicionais associam o processo a um grupo de processos (normalmente, uma árvore de processos criados pelo comando de um único usuário) e sessão de login.
- **Credenciais.** Cada processo precisa ter um ID de usuário associado e um ou mais IDs de grupo (grupos de usuários são discutidos na [Seção 10.6.2](#)) que determinam os direitos de um processo para acessar recursos e arquivos do sistema.
- **Personalidade.** As personalidades do processo não são encontradas tradicionalmente nos sistemas UNIX, mas, sob o Linux, cada processo possui um identificador de personalidade associado, que pode modificar a semântica de certas chamadas de sistema. As personalidades são usadas por bibliotecas de simulação para requisitar que as chamadas de sistema sejam compatíveis com certas variantes do UNIX.

A maioria desses identificadores está sob o controle limitado do próprio processo. O grupo do processo e os identificadores de sessão podem ser mudados se o processo quiser iniciar um novo grupo ou sessão. Suas credenciais podem ser alteradas, sujeitas a verificações de segurança apropriadas. Entretanto, o PID principal de um processo é inalterável e identifica exclusivamente esse processo até o término.

21.4.1.2 Ambiente do processo

O ambiente de um processo é herdado do seu pai e é composto de dois vetores terminados em nulo: o vetor de argumento e o vetor de ambiente. O **vetor de argumento** lista os argumentos da linha de comandos usados para invocar o programa em execução, e, por convenção, começa com o nome do próprio programa. O **vetor de ambiente** é uma lista de pares "NOME=VALOR", que associa variáveis de ambiente nomeadas a valores de texto quaisquer. O ambiente não é mantido na memória do kernel, mas é armazenado no próprio espaço de endereços no modo usuário do processo, como o primeiro dado no topo da pilha do processo.

Os vetores de argumento e ambiente não são alterados quando um novo processo é criado. O novo processo filho herdará o ambiente que seu pai possui. Todavia, um ambiente novo é configurado quando um novo programa é invocado. Ao chamar `exec()`, um processo precisa fornecer o ambiente para o novo programa. O kernel passa essas variáveis de ambiente ao programa seguinte, substituindo o ambiente atual do processo. O kernel deixa outros vetores do ambiente e da linha de

comandos intactos – sua interpretação é deixada para bibliotecas e aplicações no modo usuário.

A passagem de variáveis de ambiente de um processo para o seguinte e a herança dessas variáveis pelos filhos de um processo fornecem maneiras flexíveis de passar informações aos componentes do software de sistema no modo usuário. Diversas variáveis de ambiente importantes possuem significados convencionais para partes relacionadas do software do sistema. Por exemplo, a variável `TERM` é configurada para nomear o tipo de terminal conectado à sessão de login de um usuário; muitos programas utilizam essa variável para determinar como realizar operações na tela do usuário, como mover o cursor ou rolar uma região de texto. Os programas com suporte para múltiplos idiomas utilizam a variável `LANG` para determinar em que idioma serão exibidas as mensagens do sistema para os programas que incluem suporte para múltiplos idiomas.

O mecanismo de variável de ambiente ajusta o sistema operacional com base em cada processo, e não no sistema como um todo. Os usuários podem escolher suas próprias linguagens ou selecionar seus próprios editores independentemente um do outro.

21.4.1.3 Contexto do processo

As propriedades de identidade e ambiente são configuradas quando um processo é criado e não são alteradas até esse processo terminar. Um processo pode escolher mudar alguns aspectos de sua identidade se precisar fazer isso ou, então, pode alterar seu ambiente. Ao contrário, o contexto do processo é o estado do programa em execução a qualquer momento; ele muda constantemente. O contexto do processo inclui as seguintes partes:

- **Contexto de escalonamento:** A parte mais importante do contexto do processo é o seu contexto de escalonamento: as informações que o escalonador precisa para suspender e reiniciar o processo. Essas informações incluem as cópias salvas de todos os registradores do processo. Registradores de ponto flutuante são armazenados separadamente e restaurados apenas quando necessário, para os processos que não usam aritmética de ponto flutuante não contraírem o custo adicional de salvar esse estado. O contexto de escalonamento também inclui informações sobre prioridade de escalonamento e sobre quaisquer sinais pendentes esperando para serem entregues ao processo. Uma parte importante do contexto de escalonamento é a pilha do kernel do processo: uma área separada da memória do kernel reservada para uso exclusivo do código no modo kernel. Tanto as chamadas de sistema quanto as interrupções que ocorrem enquanto o processo está sendo executado usarão essa pilha.
- **Contabilidade.** O kernel mantém informações de contabilidade sobre os recursos consumidos em cada processo e os recursos totais consumidos pelo processo em seu tempo de vida inteiro até o momento.
- **Tabela de arquivos.** A tabela de arquivos é um array de ponteiros para estruturas de arquivo do kernel. Ao fazer chamadas de sistema de E/S de arquivo, os processos se referem aos arquivos por seu índice para essa tabela.
- **Contexto do sistema de arquivos.** Enquanto a tabela de arquivos lista os arquivos abertos existentes, o contexto do sistema de arquivos se aplica a requisições para abrir novos arquivos. A raiz atual e os diretórios default a serem usados para novas buscas de arquivos são armazenados aqui.
- **Tabela do tratador de sinais.** Os sistemas UNIX podem entregar sinais assíncronos a um processo em resposta a diversos eventos externos. A tabela do tratador de sinais define a rotina no espaço de endereços do processo a ser chamada quando chega um sinal específico.
- **Contexto da memória virtual.** O contexto da memória virtual descreve o conteúdo completo do espaço de endereços privado de um processo; discutimos isso na [Seção 21.6](#).

21.4.2 Processos e threads

O Linux fornece a chamada de sistema `fork()` com a funcionalidade tradicional de duplicar um processo. O Linux também fornece a capacidade de criar threads usando a chamada de sistema `clone()`. Contudo, o Linux não distingue entre processos e threads. Na verdade, o Linux geralmente usa o termo *tarefa* – em vez de *processo* ou *thread* – quando se refere a um fluxo de controle dentro de um programa. Quando `clone()` é invocado, ele recebe um conjunto de flags que determinam quanto compartilhamento deve ocorrer entre as tarefas pai e filha. Alguns desses flags são:

flag	significado
<code>CLONE_FS</code>	A informação do sistema de arquivos é compartilhada.
<code>CLONE_VM</code>	O mesmo espaço de memória é compartilhado.
<code>CLONE_SIGHAND</code>	Os tratadores de sinal são compartilhados.
<code>CLONE_FILES</code>	O conjunto de arquivos abertos é compartilhado.

Assim, se `clone()` receber os flags `CLONE_FS`, `CLONE_VM`, `CLONE_SIGHAND` e `CLONE_FILES`, as tarefas pai e filha compartilharão a mesma informação do sistema de arquivos (como o diretório de trabalho atual), o mesmo espaço de memória, os mesmos tratadores de sinal e o mesmo conjunto de arquivos abertos. Usar `clone()` dessa maneira é equivalente a criar uma thread em outros sistemas, pois a tarefa pai compartilha a maioria dos seus recursos com sua tarefa filha. Contudo, se nenhum desses

flags for definido quando `clone()` for invocado, não haverá compartilhamento, resultando em uma funcionalidade semelhante à chamada de sistema `fork()`.

A falta de distinção entre processos e threads é possível porque o Linux não mantém o contexto inteiro de um processo dentro da estrutura de dados do processo principal; em vez disso, ele mantém o contexto dentro de subcontextos independentes. Assim, o contexto do sistema de arquivos de um processo, a tabela de descriptor de arquivo, a tabela do tratador de sinais e o contexto da memória virtual são mantidos em estruturas de dados separadas, de modo que qualquer quantidade de processos pode facilmente compartilhar um subcontexto apontando para o mesmo subcontexto.

Os argumentos da chamada de sistema `clone()` informam quais subcontextos copiar e quais compartilhar, quando criar um novo processo. O novo processo sempre recebe uma nova identidade de um novo contexto de escalonamento, de acordo com os argumentos passados; porém, ele pode criar novas estruturas de dados do subcontexto inicializadas para serem cópias das estruturas do pai ou configurar um novo processo para usar as mesmas estruturas de dados do subcontexto então usadas pelo pai. A chamada de sistema `fork()` é nada mais do que um caso especial de `clone()` que copia todos os subcontextos, sem compartilhar nenhum.

21.5 Escalonamento

O escalonamento é a tarefa de alocar tempo de CPU a diferentes tarefas dentro de um sistema operacional. Normalmente, pensamos no escalonamento como sendo a execução e a interrupção de processos, mas outro aspecto do escalonamento também é importante no Linux: a execução das diversas tarefas do kernel. As tarefas do kernel compreendem tanto tarefas requisitadas por um processo em execução quanto tarefas executadas internamente em favor de um driver de dispositivo.

21.5.1 Escalonamento de processos

O Linux possui dois algoritmos separados de escalonamento de processos. Um é o algoritmo de tempo compartilhado para o escalonamento preemptivo imparcial entre os diversos processos; o outro foi criado para tarefas de tempo real, nas quais prioridades absolutas são mais importantes do que a imparcialidade.

O algoritmo de escalonamento usado para as tarefas de tempo compartilhado de rotina receberam uma grande melhoria com a versão 2.5 do kernel. Versões anteriores do kernel do Linux executavam uma variação do algoritmo de escalonamento tradicional do UNIX, que não fornece suporte adequado para sistemas SMP e não funciona bem quando o número de tarefas no sistema cresce. A melhoria do escalonador com a versão 2.5 do kernel agora fornece um algoritmo de escalonamento que executa em tempo constante - conhecido como $O(1)$ -, independentemente do número de tarefas no sistema. O novo escalonador também fornece maior suporte para SMP, incluindo afinidade de processador e balanceamento de carga, além de manter imparcialidade e suporte para tarefas interativas.

O escalonador do Linux é um algoritmo preemptivo, baseado em prioridade, com duas faixas de prioridade separadas: uma faixa de **tempo real** de 0 a 99 e um valor **nice** variando de 100 a 140. Essas duas faixas são mapeadas para um esquema de prioridade global pelo qual valores numericamente inferiores indicam prioridades mais altas.

Ao contrário dos escalonadores para muitos outros sistemas, o escalonador do Linux atribui a tarefas de maior prioridade quotas de tempo mais longas, e a tarefas de menor prioridade quotas de tempo mais curtas. Devido à natureza exclusiva do escalonador, isso é apropriado para o Linux, como veremos em breve. O relacionamento entre as prioridades e o tamanho da fatia de tempo pode ser visto na [Figura 21.2](#).



FIGURA 21.2 Relacionamento entre prioridades e tamanho da fatia de tempo.

Uma tarefa executável é considerada elegível para execução na CPU enquanto tiver tempo restante em sua fatia de tempo. Quando uma tarefa tiver esgotado sua fatia de tempo, ela será considerada **expirada**, e não é elegível para execução novamente até que todas as outras tarefas também tenham esgotado sua quota de tempo. O kernel mantém uma lista de todas as tarefas executáveis em uma estrutura de dados **runqueue**. Devido ao seu suporte para SMP, cada processador mantém sua própria runqueue e é escalonado independentemente. Cada runqueue contém dois arrays de prioridade - **ativo** e **expirado**. O array ativo contém todas as tarefas com tempo restante em suas fatias de tempo, e o array expirado contém todas as tarefas expiradas. Cada um desses arrays de prioridade inclui uma lista de tarefas indexadas de acordo com a prioridade ([Figura 21.3](#)). O escalonador escolhe a tarefa com a prioridade mais alta pelo array ativo, para execução na CPU. Em máquinas de multiprocessamento, isso significa que cada processador está escalonando a tarefa de prioridade mais alta a partir de sua própria estrutura runqueue. Quando todas as tarefas tiverem esgotado suas fatias de tempo (ou seja, quando o array ativo estiver vazio),

os dois arrays de prioridade serão trocados enquanto o array expirado se tornará o array ativo, e vice-versa.

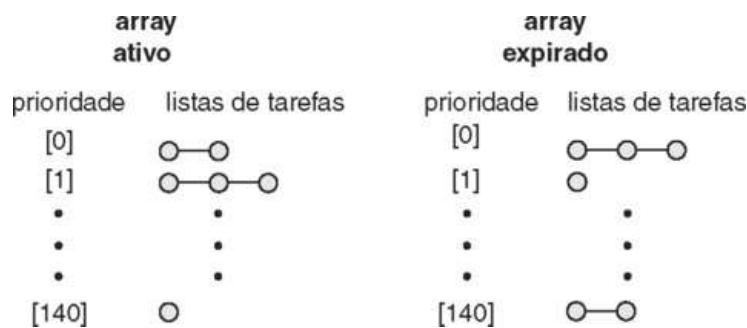


FIGURA 21.3 Listas de tarefas indexadas de acordo com a prioridade.

As tarefas recebem prioridades dinâmicas que são baseadas no valor *nice* mais ou menos um valor até 5. Se um valor é somado ou subtraído do valor *nice* de uma tarefa, isso depende da interatividade da tarefa. A interatividade de uma tarefa é determinada pelo tempo em que ela esteve dormindo enquanto esperava pela E/S. As tarefas que são mais interativas normalmente possuem tempos de dormir maiores e, portanto, são mais prováveis de terem ajustes mais próximos de -5, pois o escalonador favorece tais tarefas interativas. Por outro lado, as tarefas com tempos de dormir menores normalmente utilizam mais CPU e, portanto, têm suas prioridades reduzidas.

A prioridade dinâmica de uma tarefa é recalculada quando a tarefa tiver esgotado sua quota de tempo e estiver para ser movida para o array expirado. Assim, quando os dois arrays são trocados, todas as tarefas no novo array ativo recebem novas prioridades e fatias de tempo correspondentes.

O escalonamento de tempo real do Linux é ainda mais simples. O Linux implementa as duas classes de escalonamento de tempo real exigidas pelo POSIX.1b: First Come First Served (FCFS) e Round Robin ([Seções 5.3.1 e 5.3.4](#), respectivamente). Nos dois casos, cada processo possui uma prioridade além da sua classe de escalonamento. Os processos com diferentes prioridades ainda podem competir entre si até certo ponto no escalonamento de tempo real; no escalonamento de tempo real, o escalonador sempre executa o processo com a prioridade mais alta. Entre os processos com mesma prioridade, ele executa o processo que estava esperando por mais tempo. A única diferença entre o escalonamento FCFS e o Round Robin é que os processos FCFS continuam a executar até terminarem ou serem bloqueados, enquanto um processo em Round Robin será preemptado após um tempo e movido para o final da fila de escalonamento, de modo que os processos por revezamento com mesma prioridade automaticamente compartilharão o tempo entre eles. Ao contrário das tarefas de tempo compartilhado, as tarefas de tempo real recebem prioridades estáticas.

O escalonamento de tempo real do Linux é de tempo real flexível – em vez de rígido. O escalonador fornece garantias estritas sobre as prioridades relativas dos processos de tempo real, mas o kernel não fornece quaisquer garantias sobre a rapidez com que um processo de tempo real será escalonado depois de esse processo se tornar executável.

21.5.2 Sincronismo do kernel

O modo como o kernel escalona suas próprias operações é bem diferente do modo como ele escalona os processos. Uma requisição para a execução no modo kernel pode ocorrer de duas maneiras. Um programa em execução pode requisitar um serviço do sistema operacional, seja explicitamente por uma chamada de sistema ou implicitamente – por exemplo, quando ocorre uma page fault. Como alternativa, um controlador de dispositivo pode fornecer uma interrupção de hardware que faz a CPU começar a executar um tratador definido pelo kernel para essa interrupção.

O problema imposto ao kernel é que todas essas tarefas podem tentar acessar as mesmas estruturas de dados internas. Se uma tarefa do kernel estiver no meio do acesso a algumas estruturas de dados quando uma rotina de serviço de interrupção for executada, então essa rotina de serviço não poderá acessar ou modificar os mesmos dados sem correr o risco de adulteração de dados. Esse fato está relacionado com a ideia de seções críticas: partes de código que acessam dados compartilhados e que não podem ter permissão para executar concorrentemente. Como resultado, o sincronismo do kernel envolve muito mais do que apenas escalonamento de processos. É preciso haver uma estrutura que permita que as tarefas do kernel sejam executadas sem violar a integridade dos dados compartilhados.

Antes da versão 2.6, o Linux era um kernel não preemptivo, significando que um processo rodando no modo kernel não poderia ser preemptado – mesmo que um processo de maior prioridade se tornasse disponível para execução. Com a versão 2.6, o kernel do Linux se tornou totalmente preemptivo; assim, uma tarefa agora pode ser preemptada quando estiver rodando no kernel.

O kernel do Linux fornece spinlocks e semáforos (além de versões leitor-escritor desses dois locks) para bloquear no kernel. Em máquinas SMP, o mecanismo de bloqueio fundamental é um spinlock; o kernel é projetado de modo que o spinlock seja mantido apenas por curtas durações. Em máquinas de processador único, spinlocks são impróprios para uso e são substituídos pela ativação e desativação da preempção do kernel. Ou seja, em máquinas de único processador, em vez de manter um spinlock, a tarefa desativa a preempção do kernel. Esse padrão é resumido a seguir:

processador único	processadores múltiplos
Desativa preempção do kernel.	Adquire spinlock.
Ativa preempção do kernel.	Libera spinlock.

O Linux usa uma técnica interessante para desativar e ativar a preempção do kernel. Ele fornece duas chamadas de sistema simples - `preempt_disable()` e `preempt_enable()` - para desativar e ativar a preempção do kernel. Contudo, além disso, o kernel não é preemptível se uma tarefa no modo kernel estiver mantendo um lock. Para impor essa regra, cada tarefa no sistema tem uma estrutura `thread_info` que inclui o campo `preempt_count`, um contador indicando o número de locks que estão sendo mantidos pela tarefa. Quando um lock é adquirido, `preempt_count` é incrementado. De modo semelhante, ele é decrementado quando um lock é liberado. Se o valor de `preempt_count` para a tarefa atualmente em execução for maior que zero, não é seguro interromper o kernel, pois essa tarefa atualmente mantém um lock. Se o contador for zero, o kernel pode ser seguramente interrompido, supondo que não haja chamadas pendentes para `preempt_disable()`.

Spinlocks - junto com a ativação e desativação da preempção do kernel - são usados no kernel apenas quando o lock é mantido por curtas durações. Quando um lock tiver que ser mantido por períodos maiores, serão usados semáforos.

A segunda técnica de proteção que o Linux utiliza se aplica a seções críticas que ocorrem nas rotinas de atendimento de interrupção. A ferramenta básica é o hardware de controle de interrupção do processador. Desativando as interrupções (ou usando spinlocks) durante uma seção crítica, o kernel garante que poderá prosseguir sem o risco de acesso concorrente a estruturas de dados compartilhadas.

No entanto, existe uma penalidade para desativar interrupções. Na maioria das arquiteturas de hardware, as instruções que ativam e desativam interrupções são dispendiosas. Além do mais, desde que as interrupções permaneçam desativadas, toda a E/S é suspensa, e qualquer dispositivo esperando pelo atendimento terá de esperar até as interrupções serem reativadas, de modo que o desempenho cai. O kernel do Linux utiliza uma arquitetura de sincronismo que permite que seções críticas longas sejam executadas por toda a sua duração sem que as interrupções sejam desativadas. Essa capacidade é útil especialmente no código de rede: uma interrupção em um driver de dispositivo de rede pode sinalizar a chegada de um pacote de rede inteiro, que pode resultar em muito código sendo executado para desmontar, rotear e encaminhar esse pacote dentro da rotina de serviço de interrupção.

O Linux implementa essa arquitetura separando rotinas de serviço de interrupção em duas seções: a metade superior e a metade inferior. A **metade superior** é uma rotina de serviço de interrupção normal, e é executada com as interrupções recursivas desativadas; as interrupções de uma prioridade mais alta podem interromper a rotina, mas as interrupções de prioridade igual ou inferior são desativadas. A **metade inferior** de uma rotina de serviço é executada, com todas as interrupções ativadas, por um escalonador em miniatura que garante que as metades inferiores nunca serão interrompidas novamente. O escalonador da metade inferior é chamado automaticamente sempre que existir uma rotina de serviço de interrupção.

Essa separação significa que o kernel pode completar qualquer processamento complexo que precise ser feito em resposta a uma interrupção sem se preocupar em ser interrompido. Se outra interrupção ocorrer enquanto a metade inferior estiver executando, então essa interrupção pode requisitar que a mesma metade inferior seja executada, mas a execução será adiada até a execução atual ser completada. Cada execução da metade inferior pode ser interrompida por uma metade superior, mas nunca pode ser interrompida por uma metade inferior semelhante.

A arquitetura de metade inferior e metade superior é completada por um mecanismo para desativar as metades inferiores selecionadas enquanto executam o código normal do kernel, em primeiro plano. O kernel pode codificar seções críticas facilmente usando esse sistema: os tratadores de interrupção podem codificar suas seções críticas como metades inferiores e, quando o kernel em primeiro plano quiser entrar em uma seção crítica, ele pode desativar quaisquer metades inferiores relevantes para impedir que quaisquer outras seções críticas o interrompam. Ao final da seção crítica, o kernel pode reativar as metades inferiores e executar quaisquer tarefas da metade inferior que tenham sido enfileiradas pelas rotinas de serviço de interrupção da metade superior durante a seção crítica.

A Figura 21.4 resume os diversos níveis de proteção de interrupção dentro do kernel. Cada nível pode ser interrompido pelo código executando em um nível mais alto, mas nunca será interrompido pelo código executando em um nível igual ou inferior; exceto pelo código do modo usuário, os processos do usuário sempre podem ser apropriados por outro processo quando ocorrer uma interrupção por escalonamento em tempo compartilhado.

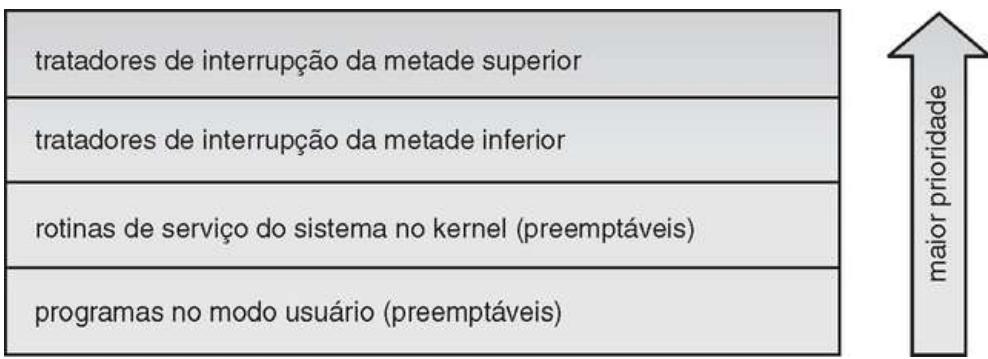


FIGURA 21.4 Níveis de proteção de interrupção.

21.5.3 Multiprocessamento simétrico

O kernel do Linux 2.0 foi o primeiro kernel do Linux estável a admitir o hardware de **multiprocessador simétrico (Symmetric Multiprocessor - SMP)**, permitindo que processos separados executem em paralelo, em processadores separados. Originalmente, a implementação do SMP impõe a restrição de que somente um processador de cada vez poderia executar o código no modo do kernel.

Na versão 2.2 do kernel, um único spinlock do kernel (às vezes chamado de **BKL**, de “big kernel lock”) era criado para permitir que múltiplos processos (rodando em diferentes processadores) fossem ativos no kernel ao mesmo tempo. Contudo, o BKL fornecia muito pouca granularidade de bloqueio. Versões posteriores do kernel tornaram a implementação SMP mais escalável, dividindo esse único spinlock do kernel em vários locks, cada um protegendo apenas um pequeno subconjunto das estruturas de dados do kernel. Esses spinlocks são descritos na [Seção 21.5.2](#). O kernel 2.6 forneceu melhorias adicionais para SMP, incluindo afinidade de processador e algoritmos de balanceamento de carga.

21.6 Gerência de memória

A gerência de memória sob o Linux possui dois componentes. O primeiro trata da alocação e liberação da memória física: páginas, grupos de páginas e pequenos blocos de memória. O segundo trata da memória virtual, que é a memória mapeada no espaço de endereços dos processos em execução. Nesta seção descrevemos esses dois componentes e depois examinamos os mecanismos pelos quais os componentes carregáveis de um novo programa são trazidos para a memória virtual de um processo em resposta a uma chamada de sistema `exec()`.

21.6.1 Gerência da memória física

Devido às características específicas do hardware, o Linux separa a memória física em três **zonas** ou regiões diferentes:

- `ZONE_DMA`.
- `ZONE_NORMAL`.
- `ZONE_HIGHMEM`.

Essas zonas são específicas da arquitetura. Por exemplo, na arquitetura Intel 80×86, certos dispositivos ISA (Industry Standard Architecture) só podem acessar os 16 MB inferiores da memória física usando o DMA. Nesses sistemas, os primeiros 16 MB de memória física compreendem `ZONE_DMA`. `ZONE_NORMAL` identifica a memória física que é mapeada no espaço de endereços da CPU. Essa zona é usada para a maioria das requisições de memória de rotina. Para arquiteturas que não limitam o que o DMA pode acessar, `ZONE_DMA` não está presente e `ZONE_NORMAL` é utilizada. Finalmente, `ZONE_HIGHMEM` (de high memory, memória alta) refere-se à memória física que não é mapeada no espaço de endereços do kernel. Por exemplo, na arquitetura Intel de 32 bits (onde 2^{32} fornece um espaço de endereços de 4 GB), o kernel é mapeado para os primeiros 896 MB do espaço de endereços; a memória restante é conhecida como **memória alta** e é alocada a partir de `ZONE_HIGHMEM`. O relacionamento entre as zonas e os endereços físicos na arquitetura Intel 80×86 aparece na [Figura 21.5](#). O kernel mantém uma lista dessas páginas livres para cada zona. Quando chega uma requisição de memória física, o kernel satisfaz a requisição usando a zona apropriada.

zona	memória física
<code>ZONE_DMA</code>	< 16 MB
<code>ZONE_NORMAL</code>	16 .. 896 MB
<code>ZONE_HIGHMEM</code>	> 896 MB

FIGURA 21.5 Relacionamento de zonas e endereços físicos no Intel 80×86.

O gerenciador de memória física principal no kernel do Linux é o **alocador de página**. Cada zona tem seu próprio alocador, que é responsável por alocar e liberar todas as páginas físicas para a zona e é capaz de alocar intervalos de páginas fisicamente contíguas por requisição. O alocador utiliza um **sistema buddy** ([Seção 9.8.1](#)) para registrar as páginas físicas disponíveis. Nesse esquema, as unidades de memória alocável adjacentes estão emparelhadas. Cada região de memória alocável possui um parceiro adjacente (ou buddy). Sempre que duas regiões parceiras alocadas forem liberadas, elas serão combinadas para formar uma região maior - um *buddy-heap*. Essa região maior também possui um parceiro, com o qual pode combinar para formar uma região livre ainda maior. Por outro lado, se uma pequena requisição de memória não puder ser satisfeita pela alocação de uma pequena região livre existente, então uma região livre maior será subdividida em dois parceiros para satisfazer a requisição. Listas interligadas separadas são usadas para registrar as regiões de memória livres de cada tamanho permitido; sob o Linux, o menor tamanho alocável sob esse mecanismo é uma única página física. A [Figura 21.6](#) mostra um exemplo de alocação buddy-heap. Uma região de 4 KB está sendo alocada, mas a menor região disponível é de 16 KB. A região é repartida recursivamente até uma parte com o tamanho desejado estar disponível.

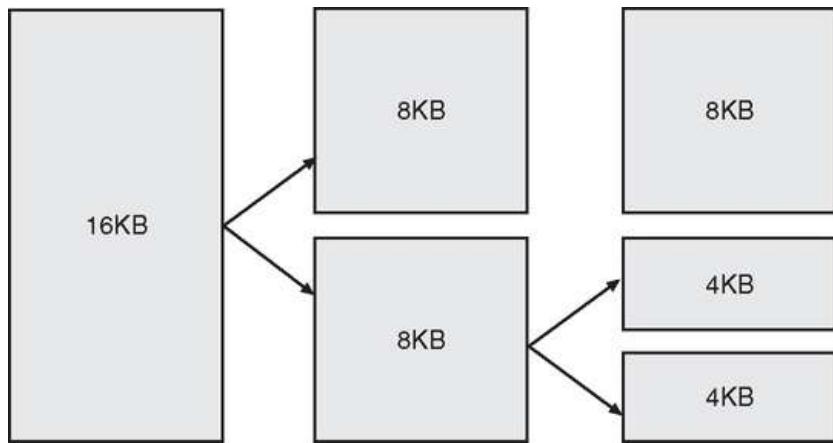


FIGURA 21.6 Dividindo a memória em um sistema buddy.

Por fim, todas as alocações de memória no kernel do Linux são realizadas estaticamente, por drivers que reservam uma área contígua da memória durante o momento da partida do sistema, ou dinamicamente, pelo alocador de páginas. Contudo, as funções do kernel não precisam usar o alocador básico para reservar memória. Os subsistemas de gerência de memória especializados utilizam o alocador de página básico para gerenciar seu próprio pool de memória. Os mais importantes são: o sistema de memória virtual, descrito na [Seção 21.6.2](#); o alocador de tamanho variável `kmalloc()`; o alocador de slab, usado para alocar memória para estruturas de dados do kernel; e o cache de página, usado para colocar páginas pertencentes aos arquivos em cache.

Muitos componentes do sistema operacional Linux precisam alocar páginas inteiras à requisição, mas, constantemente, blocos de memória menores são necessários. O kernel fornece um alocador adicional para requisições de qualquer tamanho, sendo que o tamanho de uma requisição não é conhecido antecipadamente e só pode ter alguns poucos bytes. Semelhante à função `malloc()` da linguagem C, esse serviço `kmalloc()` aloca páginas inteiras por demanda, mas depois as divide em partes menores. O kernel mantém listas de páginas em uso pelo serviço `kmalloc()`. A alocação de memória envolve determinar a lista apropriada e apanhar a primeira parte livre disponível na lista ou alocar uma nova página e a repartir. As regiões da memória reivindicadas pelo sistema `kmalloc()` são alocadas permanentemente, até serem liberadas de forma explícita; o sistema `kmalloc()` não pode relocar ou reivindicar essas regiões em resposta a faltas de memória.

Outra estratégia adotada pelo Linux para alocar memória do kernel é conhecida como alocação de slab. Um **slab** é utilizado para alocar memória para as estruturas de dados do kernel e é composto de uma ou mais páginas fisicamente contíguas. Um **cache** consiste em um ou mais slabs. Existe um único cache para cada estrutura de dados exclusiva do kernel - um cache para a estrutura de dados representando os descritores de processos, um cache para os objetos de arquivo, um cache para semáforos e assim por diante. Cada cache é preenchido com **objetos** que são instâncias da estrutura de dados do kernel que o cache representa. Por exemplo, o cache representando semáforos armazena instâncias de objetos de semáforo, e o cache representando descritores de processos armazena instâncias de objetos descritores de processos. O relacionamento entre slabs, caches e objetos aparece na [Figura 21.7](#). A figura mostra dois objetos do kernel com 3 KB de tamanho e três objetos de 7 KB de tamanho. Esses objetos são armazenados nos respectivos caches para objetos de 3 KB e 7 KB.

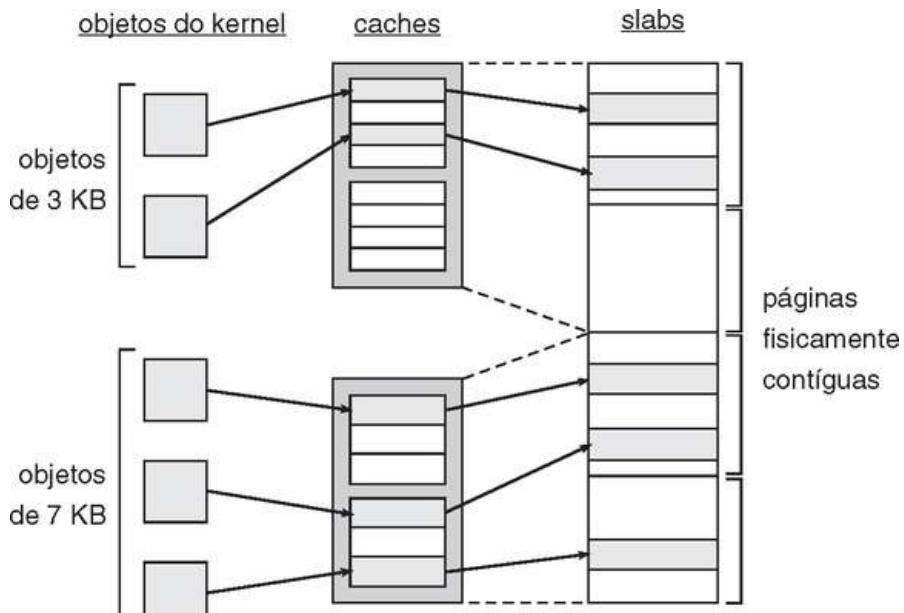


FIGURA 21.7 Alocador de slabs no Linux.

O algoritmo de alocação de slab utiliza caches para armazenar objetos do kernel. Quando um cache é criado, diversos objetos são alocados ao cache. O número de objetos no cache depende do tamanho do slab associado. Por exemplo, um slab de 12 KB (composto de três páginas consecutivas de 4 KB) poderia armazenar seis objetos de 2 KB. Inicialmente, todos os objetos no cache são marcados como livres. Quando um novo objeto para uma estrutura de dados do kernel é necessário, o alocador pode atribuir qualquer objeto livre do cache para satisfazer a requisição. O objeto atribuído a partir do cache é marcado como **usado**.

Vamos considerar um cenário em que o kernel requisita memória do alocador de slab para um objeto representando um descritor de processo. Nos sistemas Linux, um descritor de processo é do tipo `struct task_struct`, que exige aproximadamente 1,7 KB de memória. Quando o kernel do Linux cria uma nova tarefa, ele requisita a memória necessária para o objeto `struct task_struct` a partir do seu cache. O cache atenderá a requisição usando um objeto `struct task_struct` que já foi alocado em um slab e está marcado como livre.

No Linux, um slab pode estar em um dos três estados possíveis:

1. **Full**. Todos os objetos no slab estão marcados como usados.
2. **Empty**. Todos os objetos no slab estão marcados como livres.
3. **Partial**. O slab consiste em objetos usados e livres.

O alocador de slab primeiro tenta satisfazer a requisição com um objeto livre em um slab parcial. Se não houver um, um objeto livre é atribuído a partir do slab vazio. Se nenhum slab vazio estiver disponível, um novo slab é alocado a partir das páginas físicas contíguas e atribuído a um cache; a memória para o objeto é alocada a partir desse slab.

Dois outros subsistemas principais no Linux que realizam seu próprio gerenciamento de páginas físicas: o cache de página e o sistema de memória virtual. Esses sistemas estão bastante relacionados. O **cache de página** é o cache principal do kernel para dispositivos orientados em bloco ([Seção 21.8.1](#)) e arquivos mapeados na memória, sendo o principal mecanismo por meio do qual é realizada a E/S para esses dispositivos. Tanto os sistemas de arquivos baseados em disco nativos do Linux quanto o sistema de arquivos em rede NFS utilizam o cache de página. O cache de página coloca páginas inteiras de conteúdo de arquivo em cache e não está limitado a dispositivos em bloco; ele também pode manter dados de rede em cache. O sistema de memória virtual gerencia o conteúdo do espaço de endereço virtual de cada processo. Esses dois sistemas interagem de perto, pois a leitura de uma página de dados para o cache de página exige o mapeamento de páginas no cache de página usando o sistema de memória virtual. Nas próximas seções, veremos o sistema de memória virtual com mais detalhes.

21.6.2 Memória virtual

O sistema de memória virtual do Linux é responsável por manter o espaço de endereços visível a cada processo. Ele cria páginas de memória virtual por demanda e gerencia o carregamento dessas páginas do disco ou sua troca para o disco conforme a necessidade. Sob o Linux, o gerenciador de memória virtual mantém duas visões separadas do espaço de endereços de um processo: como um conjunto de regiões separadas e como um conjunto de páginas.

A primeira visão de um espaço de endereços é a visão lógica, descrevendo instruções que o sistema de memória virtual recebeu com relação ao esquema do espaço de endereços. Nessa visão,

o espaço de endereços consiste em um conjunto de regiões não sobrepostas, com cada região representando um subconjunto contínuo, alinhado por página, do espaço de endereços. Cada região é descrita internamente por uma única estrutura `vm_area_struct`, que define as propriedades da região, incluindo as permissões de leitura, escrita e execução do processo na região, e informações sobre quaisquer arquivos associados à região. As regiões de cada espaço de endereços estão ligadas a uma árvore binária balanceada para permitir a pesquisa rápida da região correspondente a qualquer endereço virtual.

O kernel também mantém uma segunda visão física de cada espaço de endereços. Essa visão é armazenada nas tabelas de página de hardware para o processo. As entradas da tabela de página identificam o local atual exato de cada página da memória virtual, esteja ela no disco ou na memória física. A visão física é gerenciada por um conjunto de rotinas, que são invocadas pelos tratadores de interrupção de software do kernel sempre que um processo tenta acessar uma página que não esteja atualmente presente nas tabelas de página. Cada `vm_area_struct` na descrição do espaço de endereços contém um campo que aponta para uma tabela de funções que implementam as principais funções de gerenciamento de página para determinada região da memória virtual. Todas as requisições para ler ou escrever uma página não disponível, por fim, são despachadas para o tratador apropriado na tabela de função da estrutura para `vm_area_struct`, de modo que as rotinas centrais de gerência de memória não tenham de saber os detalhes da gerência de cada tipo possível de região da memória.

21.6.2.1 Regiões da memória virtual

O Linux implementa vários tipos de regiões da memória virtual. Uma propriedade que caracteriza a memória virtual é o armazenamento de apoio para a região, que descreve de onde vieram as páginas para uma região. A maioria das regiões de memória possui o apoio de um arquivo ou de nada. Uma região sem apoio algum é o tipo mais simples de região de memória virtual. Tal região representa a **memória de demanda zero**: quando um processo tenta ler uma página nessa região, ele recebe uma página da memória preenchida com zeros.

Uma região apoiada por um arquivo atua como uma porta de visão para uma seção desse arquivo: sempre que o processo tenta acessar uma página dentro dessa região, a tabela de página é preenchida com o endereço de uma página dentro do cache de página do kernel correspondente ao deslocamento apropriado no arquivo. A mesma página da memória física é usada tanto pelo cache de página quanto pelas tabelas de página do processo, de modo que quaisquer mudanças feitas no arquivo pelo sistema de arquivos são imediatamente visíveis a quaisquer processos que tenham mapeado esse arquivo em seu espaço de endereços. Qualquer quantidade de processos pode mapear a mesma região do mesmo arquivo, e todos eles acabarão usando a mesma página de memória física para a finalidade.

A região da memória virtual também é definida por sua reação a escritas. O mapeamento de uma região no espaço de endereços do processo pode ser *privado* ou *compartilhado*. Se um processo escreve em uma região mapeada privadamente, então o paginador detecta que uma cópia na escrita é necessária, para manter as mudanças locais ao processo. Por outro lado, as escritas em uma região compartilhada resultam na atualização do objeto sendo mapeado nessa região, de modo que a mudança será visível imediatamente a qualquer outro processo que esteja mapeando esse objeto.

21.6.2.2 Tempo de vida de um espaço de endereços virtuais

O kernel criará um espaço de endereços virtuais em duas situações: quando um processo executar um novo programa com a chamada de sistema `exec()` e na criação de um novo processo pela chamada de sistema `fork()`. O primeiro caso é fácil: quando um novo programa é executado, o processo recebe um novo espaço de endereços virtuais, completamente vazio. As rotinas que carregam o programa são encarregadas de preencher o espaço de endereços com as regiões da memória virtual.

O segundo caso, a criação de um novo processo com `fork()`, envolve a criação de uma cópia completa do espaço de endereços virtuais do processo existente. O kernel copia os descriptores `vm_area_struct` do processo pai, depois cria um novo conjunto de tabelas de página para o filho. As tabelas de página do pai são copiadas diretamente para as do filho, e a contagem de referência de cada página coberta é incrementada; assim, depois de fork, o pai e o filho compartilham as mesmas páginas físicas da memória em seus espaços de endereço.

Um caso especial ocorre quando a operação de cópia alcança uma região da memória virtual que é mapeada de forma privada. Quaisquer páginas às quais o processo pai tenha escrito dentro de tal região são privadas, e as mudanças subsequentes a essas páginas pelo pai ou pelo filho não podem atualizar a página no espaço de endereços do outro processo. Quando as entradas da tabela de página para tais regiões são copiadas, elas são definidas como somente leitura e marcadas para cópia na escrita. Desde que nenhum processo modifique essas páginas, os dois processos compartilham a mesma página de memória física. Entretanto, se algum processo tentar modificar uma página de cópia na escrita, a contagem de referência sobre a página será verificada. Se a página ainda estiver compartilhada, então o processo copia o conteúdo da página para uma página

nova da memória física e usa sua cópia. Esse mecanismo garante que as páginas de dados privadas sejam compartilhadas entre os processos sempre que possível; as cópias são feitas apenas quando for absolutamente necessário.

21.6.2.3 Swapping e paginação

Uma tarefa importante para um sistema de memória virtual é relocar páginas da memória física para o disco quando essa memória for necessária. Os primeiros sistemas UNIX realizavam essa relocação trocando o conteúdo dos processos inteiros ao mesmo tempo, mas as versões modernas do UNIX contam mais com a paginação - o movimento de páginas individuais da memória virtual entre a memória física e o disco. O Linux não implementa a troca de processo inteiro; ele usa exclusivamente o mecanismo de página mais recente.

O sistema de paginação pode ser dividido em duas seções. Primeiro, o **algoritmo de política** decide quais páginas escrever no disco e quando escrevê-las. Segundo, o **mecanismo de paginação** executa a transferência e pagina os dados de volta para a memória física quando forem necessários novamente.

A **política pageout** do Linux utiliza uma versão modificada do algoritmo do relógio (ou segunda chance) descrito na [Seção 9.4.5.2](#). No Linux, o relógio de múltiplas passagens é utilizado, e cada página possui uma *idade* ajustada a cada passada do relógio. A idade é, mais precisamente, uma medida da história recente da página ou quanta atividade a página viu recentemente. As páginas acessadas com frequência obterão um valor de idade mais alto, mas a idade das páginas acessadas com pouca frequência cairá a cada passada, até chegar a zero. Esse valor de idade permite ao paginador selecionar páginas para paginar com base em uma política de Least Frequently Used (LFU).

O mecanismo de paginação admite a paginação tanto para dispositivos de troca dedicados quanto para partições e para arquivos normais, embora a troca de um arquivo seja significativamente mais lenta, devido ao custo adicional ocasionado pelo sistema de arquivos. Os blocos são alocados dos dispositivos de troca de acordo com um mapa de bits dos blocos usados, que é mantido na memória física o tempo inteiro. O alocador utiliza um algoritmo do próximo ajuste para tentar escrever páginas em trechos contínuos dos blocos de disco, a fim de melhorar o desempenho. O alocador registra o fato de que uma página foi paginada para o disco usando um recurso das tabelas de página nos processadores modernos: o bit de página ausente, na entrada da tabela de página, é marcado, permitindo que o restante da entrada da tabela de página seja preenchido com um índice, identificando onde a página foi escrita.

21.6.2.4 Memória virtual do kernel

O Linux reserva, para seu uso interno, uma região constante, dependente da arquitetura, do espaço de endereço virtual de cada processo. As entradas da tabela de página que são mapeadas para essas páginas do kernel são marcadas como protegidas, de modo que as páginas não sejam visíveis ou modificáveis quando o processador estiver executando no modo usuário. Essa área de memória virtual do kernel contém duas regiões. A primeira seção é uma área estática, que contém as referências da tabela de página a cada página de memória física disponível no sistema, de modo que ocorrerá uma simples tradução de endereço físico para virtual quando o código do kernel for executado. O centro do kernel como também todas as páginas alocadas pelo alocador de página normal residem nessa região.

O restante da seção reservada do espaço de endereços do kernel não é reservado para qualquer finalidade específica. As entradas da tabela de página nessa faixa de endereços podem ser modificadas pelo kernel para apontar para quaisquer outras áreas da memória. O kernel fornece um par de facilidades que permite aos processos usarem essa memória virtual. A função `vmalloc()` aloca um número arbitrário de páginas físicas da memória que pode ser fisicamente contígua para uma única região da quase-memória virtual do kernel. A função `vremap()` mapeia uma sequência de endereços virtuais para apontar para uma área da memória usada por um driver de dispositivo para a E/S mapeada na memória.

21.6.3 Execução e carregamento de programas do usuário

A execução de programas do usuário pelo kernel do Linux é disparada por uma chamada à chamada de sistema `exec()`. Essa chamada `exec()` manda que o kernel execute um novo programa dentro do processo atual, substituindo completamente o contexto de execução atual pelo contexto inicial do novo programa. A primeira tarefa desse serviço do sistema é verificar se o processo de chamada tem direitos de permissão para o arquivo que está sendo executado. Quando essa questão tiver sido verificada, o kernel chama a rotina do carregador, a fim de iniciar a execução do programa. O carregador não carrega necessariamente o conteúdo do arquivo de programa na memória física, mas ele, pelo menos, configura o mapeamento do programa na memória virtual.

Não existe uma rotina isolada no Linux para carregar um novo programa. Em vez disso, o Linux mantém uma tabela de possíveis funções do carregador e dá a cada função a oportunidade de tentar

carregar determinado arquivo quando uma chamada de sistema `exec()` é executada. O motivo inicial para essa tabela do carregador foi que, entre as versões 1.0 e 1.2 dos kernels, o formato-padrão dos arquivos binários do Linux foi alterado. Os kernels do Linux mais antigos entendiam um formato `a.out` para os arquivos binários - um formato relativamente simples, comum nos sistemas UNIX mais antigos. Os sistemas Linux mais recentes utilizam o formato ELF mais moderno, agora aceito pela maioria das implementações atuais do UNIX. O **ELF** possui diversas vantagens em relação ao `a.out`, incluindo flexibilidade e extensibilidade: novas seções podem ser acrescentadas a um binário ELF (por exemplo, para acrescentar informações de depuração extras), sem fazer as rotinas do carregador se tornarem confusas. Permitindo o registro de várias rotinas do carregador, o Linux pode admitir os formatos binários ELF e `a.out` em um único sistema em execução.

Nas [Seções 21.6.3.1](#) e [21.6.3.2](#), concentrarmo-nos exclusivamente no carregamento e na execução dos binários no formato ELF. O procedimento para carregar binários `a.out` é mais simples, mas semelhante em operação.

21.6.3.1 Mapeamento de programas na memória

Sob o Linux, o carregador de binários não carrega um arquivo binário na memória física. Em vez disso, as páginas do arquivo binário são mapeadas para regiões da memória virtual. Somente quando o programa tenta acessar determinada página, um page fault resultará no carregamento dessa página na memória física usando paginação sob demanda.

É responsabilidade do carregador de binários do kernel configurar o mapeamento inicial da memória. Um arquivo binário no formato ELF consiste em um cabeçalho seguido por várias seções alinhadas na página. O carregador ELF trabalha lendo o cabeçalho e mapeando as seções do arquivo para regiões separadas da memória virtual.

A [Figura 21.8](#) mostra o esquema típico das regiões da memória configuradas pelo carregador ELF. Em uma região reservada em uma extremidade do espaço de endereços fica o kernel, em sua própria região privilegiada da memória virtual, inacessível aos programas normais no modo usuário. O restante da memória virtual está disponível às aplicações, que podem usar as funções de mapeamento de memória do kernel para criar regiões que mapeiam uma parte de um arquivo ou que estão disponíveis para os dados da aplicação.

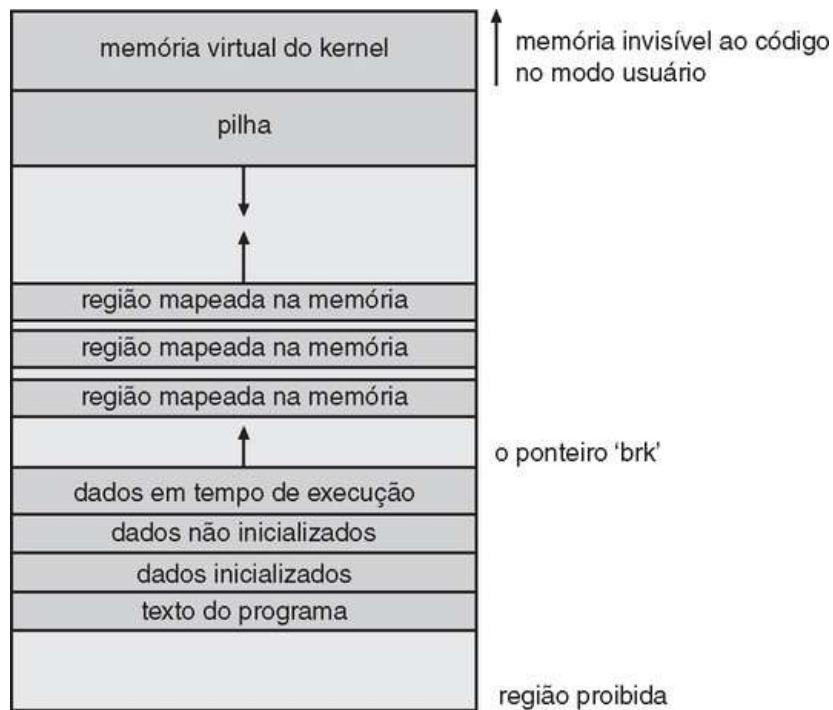


FIGURA 21.8 Layout da memória para os programas ELF.

A tarefa do carregador é configurar o mapeamento de memória inicial para permitir o início da execução do programa. As regiões que precisam ser inicializadas incluem a pilha e as regiões de texto e dados do programa.

A pilha é criada no tipo da memória virtual no modo usuário; ela cresce para baixo, em direção aos endereços de número mais baixo. Ela inclui cópias dos argumentos e variáveis de ambiente dados ao programa na chamada de sistema `exec()`. As outras regiões são criadas próximo à extremidade inferior da memória virtual. As seções do arquivo binário que contêm texto do programa ou dados somente de leitura são mapeadas para a memória como uma região protegida contra escrita. Os

dados inicializados que podem ser escritos são mapeados em seguida; depois, quaisquer dados não inicializados são mapeados como uma região privada com demanda zero.

Diretamente após essas regiões de tamanho fixo está uma região de tamanho variável, que os programas podem expandir conforme for necessário para manter os dados alocados durante a execução. Cada processo possui um ponteiro, `brk`, que aponta para a extensão atual dessa região de dados, e os processos podem estender ou contrair sua região `brk` com uma única chamada de sistema - `sbrk()`.

Quando esses mapeamentos tiverem sido configurados, o carregador inicializará o registrador do contador de programa do processo com o ponto inicial registrado no cabeçalho ELF, e o processo poderá ser escalonado.

21.6.3.2 Link estático e dinâmico

Quando o programa tiver sido carregado e for iniciada sua execução, todo o conteúdo necessário do arquivo binário terá sido carregado no espaço de endereços virtual do processo. Entretanto, a maioria dos programas também precisa executar funções das bibliotecas do sistema, e essas funções da biblioteca também precisam ser carregadas. No caso mais simples, as funções de biblioteca necessárias são embutidas no arquivo binário executável do programa. Esse programa está linkado estaticamente às suas bibliotecas, e executáveis linkados estaticamente podem iniciar sua execução assim que são carregados.

A principal desvantagem do link estático é que cada programa gerado precisa conter cópias exatas das mesmas funções comuns à biblioteca do sistema. É muito mais eficiente, em termos de memória física e uso do espaço em disco, carregar as bibliotecas do sistema para a memória somente uma vez. O link dinâmico permite esse tipo de carregamento.

O Linux implementa o link dinâmico no modo usuário por meio de uma biblioteca especial do linker. Cada programa linkado dinamicamente contém uma pequena função vinculada estaticamente, que é chamada quando o programa é iniciado. Essa função estática mapeia a biblioteca de links à memória e executa o código que a função contém. A biblioteca de links lê uma lista de bibliotecas dinâmicas exigidas pelo programa e os nomes das variáveis e funções necessárias nessas bibliotecas, lendo as informações contidas nas seções do binário ELF. Depois, ela mapeia as bibliotecas no meio da memória virtual e resolve as referências aos símbolos contidos nessas bibliotecas. Não importa onde na memória essas bibliotecas compartilhadas são mapeadas. Elas são compiladas para um **código independente de posição (Position-Independent Code - PIC)**, que pode ser executado em qualquer endereço na memória.

21.7 Sistemas de arquivos

O Linux retém o modelo de sistema de arquivo-padrão do UNIX. No UNIX, um arquivo não precisa ser um objeto armazenado no disco ou apanhado por uma rede a partir de um servidor de arquivos remoto. Em vez disso, os arquivos do UNIX podem ser qualquer coisa capaz de lidar com a entrada ou saída de um fluxo de dados. Os drivers de dispositivo podem aparecer como arquivos, e os canais de comunicação entre processos ou conexões de rede também se parecem com arquivos para o usuário.

O kernel do Linux trata de todos esses tipos de arquivo escondendo os detalhes da implementação de qualquer tipo de arquivo isolado atrás de uma camada de software, o sistema de arquivos virtual (VFS - Virtual File System). Aqui, explicamos primeiro o sistema de arquivos virtual e depois discutimos o sistema de arquivos-padrão do Linus - ext2fs.

21.7.1 O sistema de arquivos virtual

O VFS do Linux foi projetado em torno dos princípios orientados a objeto. Ele possui dois componentes: um conjunto de definições, que especifica com qual objeto de sistema de arquivo pode se parecer, e uma camada de software, para manipular esses objetos. O VFS define quatro tipos de objetos principais:

- Um **objeto inode** representa um arquivo individual.
- Um **objeto arquivo** representa um arquivo aberto.
- Um **objeto superbloco** representa um sistema de arquivos inteiro.
- Um **objeto dentry** representa uma entrada de diretório individual.

Para cada um desses três tipos de objeto, o VFS define um conjunto de operações. Cada objeto de um desses tipos contém um ponteiro para uma tabela de função. A tabela de função lista os endereços das funções reais que implementam as operações definidas para esse objeto. Por exemplo, uma API abreviada para algumas das operações do objeto de arquivo inclui:

- `int open(...)` - Abre um arquivo.
- `ssize_t read(...)` - Lê de um arquivo.
- `ssize_t write(...)` - Grava em um arquivo.
- `int mmap(...)` - Mapeia um arquivo na memória.

A definição completa do objeto arquivo é especificada na struct `file_operations`, que está localizada no arquivo `/usr/include/linux/fs.h`. É preciso haver uma implementação do objeto arquivo (para um tipo de arquivo específico) para implementar cada função especificada na definição do objeto arquivo.

A camada de software do VFS pode realizar uma operação sobre um desses objetos chamando a função apropriada a partir da tabela de função desse objeto, sem ter que saber, com antecedência, exatamente com que tipo de objeto está lidando. O VFS não sabe (ou não se importa em saber) se um inode representa um arquivo em rede, um arquivo em disco, um socket de rede ou um arquivo de diretório. A função apropriada para a operação `read()` desse arquivo sempre estará no mesmo lugar em sua tabela de função, e a camada de software do VFS chamará essa função sem se preocupar em como os dados serão lidos.

Os objetos inode e arquivo são os mecanismos usados para o acesso aos arquivos. Um objeto inode é uma estrutura de dados contendo ponteiros para os blocos de disco que contêm o conteúdo real do arquivo, e um objeto arquivo representa um ponto de acesso para os dados em um arquivo aberto. Um processo não pode acessar o conteúdo de um inode sem primeiro obter um objeto arquivo apontando para o inode. O objeto arquivo registra onde no arquivo o processo está atualmente lendo ou escrevendo, para registrar a E/S sequencial do arquivo. Ele também memoriza se o processo pediu permissões de escrita quando o arquivo foi aberto e registra a atividade do processo, se necessário, para realizar a leitura antecipada adaptativa, levando dados do arquivo para a memória antes de o processo solicitá-los, para melhorar o desempenho.

Os objetos de arquivo normalmente pertencem a um único processo, mas os objetos inode não pertencem. Mesmo quando um arquivo não está mais sendo usado por quaisquer processos, seu objeto inode ainda pode ser colocado em cache pelo VFS, para melhorar o desempenho, se o arquivo for usado novamente no futuro próximo. Todos os dados de arquivo em cache são linkados a uma lista no objeto inode do arquivo. O inode também mantém informações-padrão sobre cada arquivo, como proprietário, tamanho e horário da modificação mais recente.

Os arquivos de diretório são tratados de forma ligeiramente diferente dos outros arquivos. A interface de programação do UNIX define diversas operações sobre diretórios, como criar, excluir e renomear um arquivo em um diretório. As chamadas de sistema para essas operações no diretório não exigem que o usuário abra os arquivos envolvidos, diferentemente do caso da leitura e escrita de dados. O VFS, portanto, define essas operações de diretório no objeto inode, em vez do objeto arquivo.

O objeto superbloco representa um conjunto de arquivos conectados, que formam um sistema de arquivos autocontido. O kernel do sistema operacional mantém um único objeto superbloco para

cada dispositivo de disco montado como um sistema de arquivos e para cada sistema de arquivos em rede atualmente conectado. A responsabilidade principal do objeto superbloco é fornecer acesso aos inodes. O VFS identifica cada inode por um par exclusivo sistema de arquivos/número de inode e encontra o inode correspondente a determinado número de inode pedindo ao objeto superbloco para retornar o inode com esse número.

Finalmente, um objeto dentry representa uma entrada de diretório que pode incluir o nome de um diretório no nome do caminho de um arquivo (como `/usr`) ou o arquivo real (como `stdio.h`). Por exemplo, o arquivo `/usr/include/stdio.h` contém as entradas de diretório (1) `/`, (2) `usr`, (3) `include` e (4) `stdio.h`. Cada um desses valores é representado por um objeto dentry separado.

Como exemplo de como os objetos dentry são usados, considere a situação em que um processo deseja abrir o arquivo com o nome de caminho `/usr/include/stdio.h` usando um editor. Como o Linux trata os nomes de diretório como arquivos, a tradução desse caminho exige primeiro obter o inode para a raiz, `/`. O sistema operacional precisa, então, ler esse arquivo para obter o inode para o arquivo `include`. Ele precisa continuar esse processo até obter o inode para o arquivo `stdio.h`. Como a tradução de nome de caminho pode ser uma tarefa demorada, o Linux mantém um cache de objetos dentry, que é consultado durante a tradução de nome de caminho. A obtenção do inode a partir do cache dentry é consideravelmente mais rápida do que ler o arquivo no disco.

21.7.2 O sistema de arquivos ext2fs do Linux

O sistema de arquivos de disco-padrão usado pelo Linux é denominado **ext2fs**, por motivos históricos. O Linux foi programado originalmente com um sistema de arquivos compatível com o Minix, para facilitar a troca de dados com o sistema de desenvolvimento Minix, mas esse sistema de arquivos ficou bastante restringido pelos limites de 14 caracteres para os nomes de arquivo e o tamanho máximo do sistema de arquivos de 64 MB. O sistema de arquivos Minix foi substituído por um novo sistema de arquivos, denominado **sistema de arquivos estendido (extended file system - extfs)**. Uma modificação de projeto posterior sobre esse sistema de arquivos, para melhorar o desempenho e a escalabilidade e acrescentar alguns recursos que faltavam, levou ao **segundo sistema de arquivos estendido (ext2fs)**.

O ext2fs do Linux possui muito em comum com o Fast File System (FFS) do BSD. Ele utiliza um mecanismo semelhante para localizar os blocos de dados pertencentes a um arquivo específico, armazenando ponteiros do bloco de dados em blocos indiretos por meio do sistema de arquivos com até três níveis de indireção. Assim como no FFS, os arquivos de diretório são armazenados em disco da mesma forma que os arquivos normais, embora seu conteúdo seja interpretado de forma diferente. Cada bloco em um arquivo de diretório consiste em uma lista interligada de entradas; cada entrada contém a extensão da entrada, o nome de um arquivo e o número do inode ao qual essa entrada se refere.

As principais diferenças entre o ext2fs e o FFS estão nas políticas de alocação de disco. No FFS, o disco é alocado a arquivos em blocos de 8 KB. Esses blocos são subdivididos em fragmentos de 1 KB para armazenamento de pequenos arquivos ou blocos parcialmente preenchidos no final de um arquivo. Por outro lado, o ext2fs não utiliza fragmento algum, mas realiza todas as suas alocações em unidades menores. O tamanho default do bloco no ext2fs é de 1 KB, embora blocos com 2 KB e 4 KB também sejam aceitos.

Para manter um alto desempenho, o sistema operacional precisa tentar realizar operações de E/S em grandes pedaços sempre que possível, agrupando as requisições de E/S fisicamente adjacentes. O agrupamento reduz o custo adicional por requisição advindo dos drivers de dispositivos, discos e hardware do controle de disco. Uma requisição de E/S para 1 KB é muito pequena para manter um bom desempenho, de modo que o ext2fs utiliza políticas de alocação projetadas para colocar os blocos logicamente adjacentes de um arquivo em blocos fisicamente adjacentes no disco, para poder submeter uma requisição de E/S para vários blocos de disco como uma única operação.

A política de alocação ext2fs vem em duas partes. Como no FFS, um sistema de arquivos ext2fs é particionado em vários **grupos de blocos**. O FFS utiliza um conceito semelhante, o de **grupos de cilindros**, no qual cada grupo corresponde a um único cilindro de um disco físico. Entretanto, a moderna tecnologia de unidade de disco empacota setores no disco em diferentes densidades e, portanto, com diferentes tamanhos de cilindro, dependendo da distância que a cabeça de disco está do centro do disco. Portanto, os grupos de cilindros de tamanho fixo não correspondem necessariamente à geometria do disco.

Ao alocar um arquivo, o ext2fs precisa primeiro selecionar o grupo de blocos para esse arquivo. Para blocos de dados, ele tenta escolher o mesmo grupo de blocos que o inode do arquivo alocou. Para alocações de inode, ele seleciona o grupo de blocos no qual o diretório pai do arquivo reside, para arquivos não de diretório. Os arquivos de diretório não são mantidos juntos, eles ficam dispersos por todos os grupos de blocos disponíveis. Essas políticas são projetadas não só para manter informações relacionadas dentro do mesmo grupo de blocos, mas também para distribuir a carga do disco entre os grupos de blocos do disco, reduzindo a fragmentação de qualquer área do disco.

Dentro de um grupo de blocos, o ext2fs tenta manter as alocações fisicamente contíguas, se

possível, reduzindo a fragmentação, se puder. Ele mantém um mapa de bits de todos os blocos livres em um grupo de blocos. Ao alocar os primeiros blocos para um arquivo, ele começa procurando um bloco livre a partir do início do grupo de blocos; ao estender um arquivo, ele continua a procurar desde o bloco alocado mais recentemente ao arquivo. A pesquisa é realizada em dois estágios. Primeiro, o ext2fs procura um byte livre inteiro no mapa de bits; se não puder encontrar um, ele procura qualquer bit livre. A procura de bytes livres visa alocar o espaço do disco em pedaços de pelo menos oito blocos, onde for possível.

Quando um bloco livre tiver sido identificado, a pesquisa será estendida para trás até um bloco alocado ser encontrado. Quando um byte livre for encontrado no mapa de bits, essa extensão para trás impedirá que o ext2fs deixe um buraco entre o bloco alocado mais recentemente no byte anterior diferente de zero e o byte zero encontrado. Quando o bloco seguinte a ser localizado tiver sido encontrado pela pesquisa de bit ou byte, o ext2fs estenderá a alocação para frente por até oito blocos e **pré-alocará** esses blocos extras ao arquivo. Essa pré-alocação ajuda a reduzir a fragmentação durante as escritas intercaladas em arquivos separados e também reduz o custo da CPU da alocação de disco, alocando múltiplos blocos simultaneamente. Os blocos pré-alocados são retornados ao mapa de bits do espaço livre quando o arquivo for fechado.

A **Figura 21.9** ilustra as políticas de alocação. Cada linha representa uma sequência de bits marcados e desmarcados em um mapa de bits de alocação, indicando blocos usados e livres no disco. No primeiro caso, se pudermos encontrar quaisquer blocos livres suficientemente próximos ao início da pesquisa, nós os alocamos, não importa como possam estar fragmentados. A fragmentação é compensada parcialmente pelo fato de que os blocos são próximos e provavelmente podem ser lidos sem quaisquer buscas de disco, e a alocação de todos eles a um arquivo é melhor em longo prazo do que alocar blocos isolados para separar arquivos quando grandes áreas livres se tornarem escassas no disco. No segundo caso, não encontramos imediatamente um bloco livre nas proximidades e, por isso, buscamos à frente um byte livre inteiro no mapa de bits. Se alocássemos esse byte como um todo, acabaríamos criando uma área fragmentada de espaço livre entre ele e a alocação anterior; portanto, antes de alocar, recuamos para fazer essa alocação esvaziar com a alocação anterior a ela, e depois alocamos para frente para satisfazer a alocação default de oito blocos.

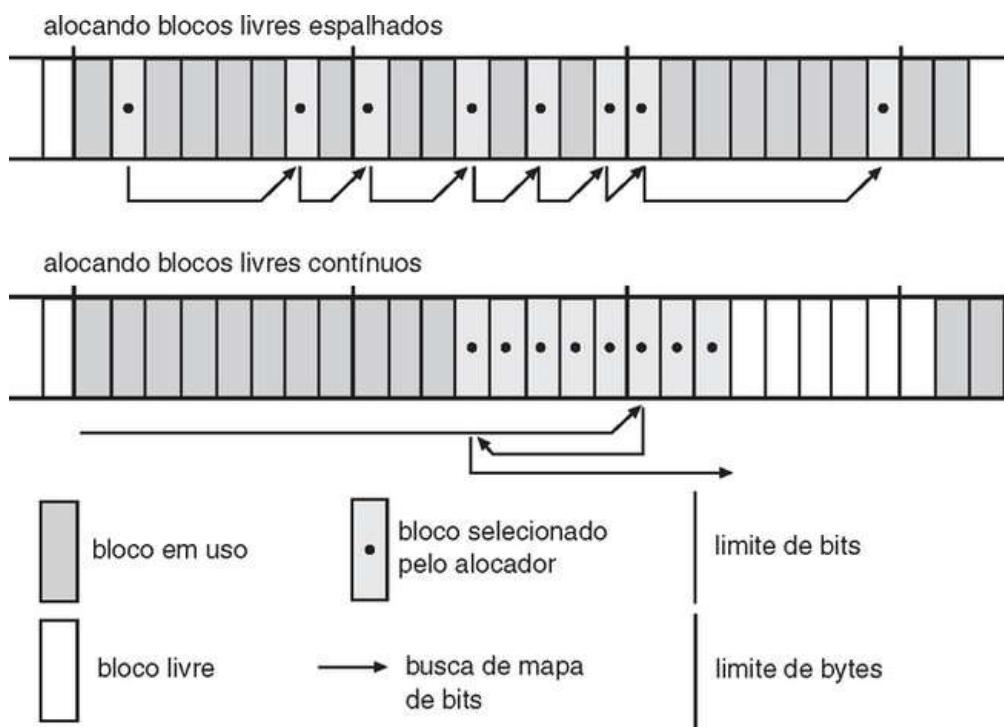


FIGURA 21.9 Políticas de alocação de bloco do ext2fs.

21.7.3 Diário

Um recurso popular em um sistema de arquivos é o **diário**, pelo qual as modificações no sistema de arquivos são escritas sequencialmente em um diário. Um conjunto de operações que realiza uma tarefa específica é uma **transação**. Quando uma transação é escrita no diário, ela é considerada confirmada, e a chamada de sistema modificando o sistema de arquivos (ou seja, `write()`) pode retornar ao processo do usuário, permitindo que ele continue sua execução. Nesse meio-tempo, as

entradas de diário relacionadas com a transação são reproduzidas pelas estruturas reais do sistema de arquivos. À medida que são feitas mudanças, um ponteiro é atualizado para indicar quais ações foram completadas e quais ainda estão incompletas. Quando uma transação confirmada inteira é completada, ela é removida do diário. O diário, que na realidade é um buffer circular, pode estar em uma seção separada do sistema de arquivos ou pode até mesmo estar em um disco fisicamente separado. É mais eficiente, porém mais complexo, tê-lo sob cabeças de leitura-escrita separadas, diminuindo, assim, a disputa pela cabeça e os tempos de busca.

Se o sistema de arquivos falhar, algumas transações podem permanecer no diário. Essas transações nunca foram completadas no sistema de arquivos, embora fossem confirmadas pelo sistema operacional, de modo que precisam ser completadas quando o sistema se recuperar. As transações podem ser executadas a partir do ponteiro até que o trabalho termine, e as estruturas do sistema de arquivos permanecem consistentes. O único problema ocorre quando uma transação tiver sido abortada. Ou seja, ela não foi confirmada antes que o sistema falhasse. Quaisquer mudanças a partir dessas transações que foram aplicadas ao sistema de arquivos precisam ser desfeitas, novamente preservando a consistência do sistema de arquivos. Essa recuperação é tudo o que é preciso após uma falha, eliminando todos os problemas com a verificação de consistência.

Os sistemas de arquivos com diário normalmente também são mais rápidos do que os sistemas sem diário, pois as atualizações prosseguem muito mais rapidamente quando são aplicadas ao diário na memória, em vez de diretamente nas estruturas de dados em disco. O motivo para essa melhoria é encontrado na vantagem de desempenho da E/S sequencial em relação à E/S aleatória. As escritas aleatórias síncronas e dispendiosas ao sistema de arquivos são transformadas em escritas sequenciais síncronas muito menos dispendiosas, feitas ao diário do sistema de arquivos. Essas mudanças, por sua vez, são reproduzidas assincronamente por meio de escritas aleatórias nas estruturas apropriadas. O resultado geral é um ganho significativo no desempenho das operações orientadas a metadados do sistema de arquivos, como criação e exclusão de arquivo.

O diário não é fornecido no ext2fs. Contudo, ele é fornecido em outro sistema de arquivos comum disponível para os sistemas Linux, o **ext3**, que é baseado no ext2fs.

21.7.4 O sistema de arquivos de processo do Linux

A flexibilidade do VFS do Linux nos permite implementar um sistema de arquivos que não armazena dados de maneira persistente, mas fornece uma interface a alguma outra funcionalidade. O **sistema de arquivos de processo** do Linux, conhecido como sistema de arquivos /proc, é um exemplo de um sistema de arquivos cujo conteúdo não é armazenado em lugar algum, mas calculado por demanda, de acordo com as requisições de E/S de arquivo do usuário.

Um sistema de arquivos /proc não é exclusivo do Linux. O UNIX SVR4 introduziu um sistema de arquivos /proc como uma interface eficiente para o suporte de depuração de processos do kernel: cada subdiretório do sistema de arquivos corresponde não a um diretório em qualquer disco, mas a um processo ativo no sistema atual. Uma listagem do sistema de arquivos revela um diretório por processo, com o nome do diretório sendo a representação decimal ASCII do processo id (PID) exclusivo do processo.

O Linux implementa tal sistema de arquivos /proc, mas o estende bastante, acrescentando uma série de diretórios extras e arquivos de texto sob o diretório raiz do sistema de arquivos. Essas novas entradas correspondem a diversas estatísticas sobre o kernel e os drivers carregados associados. O sistema de arquivos /proc fornece um meio para os programas acessarem essas informações como arquivos de texto puro; o ambiente de usuário UNIX padrão fornece ferramentas poderosas para processar esses arquivos. Por exemplo, no passado, o comando `ps` tradicional do UNIX, para listar os estados de todos os processos em execução, foi implementado como um processo privilegiado que lê o estado do processo diretamente da memória virtual do kernel. No Linux, esse comando é implementado como um programa não privilegiado que analisa e formata as informações do /proc.

O sistema de arquivos /proc precisa implementar duas coisas: uma estrutura de diretórios e o conteúdo de arquivo em seu interior. Como um sistema de arquivos UNIX é definido como um conjunto de modos de arquivo e diretório identificado por seus números de inode, o sistema de arquivos /proc precisa definir um número de inode exclusivo e persistente para cada diretório e os arquivos associados. Quando houver esse mapeamento, o sistema de arquivos poderá usar esse número de inode para identificar que operação será exigida quando um usuário tentar ler do inode de determinado arquivo ou realizar uma pesquisa no inode de um diretório em particular. Quando forem lidos os dados de um desses arquivos, o sistema de arquivos /proc coletará as informações apropriadas, as deixará em um formato textual e as colocará no buffer de leitura do processo requisitante.

O mapeamento de número de inode para tipo de informação divide o número de inode em dois campos. No Linux, um PID possui 16 bits, mas um número de inode possui 32 bits. Os 16 bits superiores do número de inode são interpretados como um PID, e os bits restantes definem que tipo de informação está sendo requisitado sobre esse processo.

Um PID zero não é válido, de modo que um campo de PID zero no número de inode indica que

esse inode contém informações globais, e não específicas ao processo. Existem arquivos globais separados no `/proc` para prestar informações, como a versão do kernel, memória livre, estatísticas de desempenho e drivers atualmente em execução.

Nem todos os números de inode nesse intervalo são reservados: o kernel pode alocar novos mapeamentos de inode do `/proc` dinamicamente, mantendo um mapa de bits de números de inode alocados. Ele também mantém uma estrutura de dados em árvore das entradas globais registradas do sistema de arquivos `/proc`: cada entrada contém o número de inode do arquivo, nome do arquivo e permissões de acesso, juntamente com as funções especiais usadas para gerar o conteúdo do arquivo. Os drivers podem registrar e remover o registro de entradas nessa árvore a qualquer momento, e uma seção especial da árvore – aparecendo sob o diretório `/proc/sys` – é reservada para as variáveis do kernel. Os arquivos sob essa árvore são tratados por um conjunto de tratadores comuns, permitindo a leitura e a escrita dessas variáveis; desse modo, um administrador de sistemas pode ajustar o valor dos parâmetros do kernel escrevendo os novos valores desejados no arquivo apropriado, em representação decimal ASCII.

Para permitir o acesso eficiente a essas variáveis de dentro das aplicações, a subárvore `/proc/sys` fica disponível por meio de uma chamada de sistema especial, `sysctl()`, que lê e escreve as mesmas variáveis em binário, em vez de texto, sem o overhead do sistema de arquivos. A `sysctl()` não é uma facilidade extra; ela lê a árvore de entrada dinâmica do `/proc` para identificar as variáveis às quais a aplicação está se referindo.

21.8 Entrada e saída

Para o usuário, o sistema de E/S no Linux se parece muito com o de qualquer UNIX, ou seja, até certo ponto, todos os drivers de dispositivos se parecem com arquivos normais. Os usuários podem abrir um canal de acesso para um dispositivo da mesma maneira como eles podem abrir qualquer outro arquivo - os dispositivos podem aparecer como objetos dentro do sistema de arquivos. O administrador do sistema pode criar arquivos especiais dentro de um sistema de arquivos que contêm referências a um driver de dispositivo específico, e um usuário abrindo tal arquivo poderá ler e escrever no dispositivo referenciado. Usando o sistema normal de proteção de arquivo, que determina quem pode acessar qual arquivo, o administrador pode definir permissões de acesso para cada dispositivo.

O Linux divide todos os dispositivos em três classes: dispositivos de bloco, dispositivos de caractere e dispositivos de rede. A [Figura 21.10](#) ilustra a estrutura geral do sistema de driver de dispositivo.

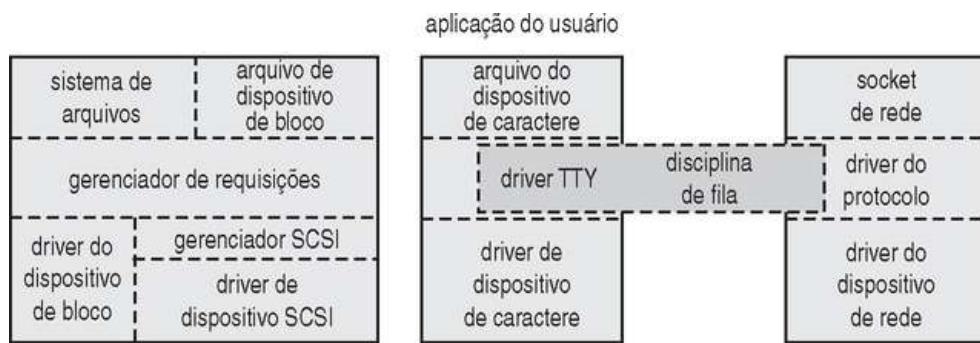


FIGURA 21.10 Estrutura em bloco do driver de dispositivo.

Dispositivos de bloco (block device) incluem todos os dispositivos que permitem acesso aleatório a blocos de dados de tamanho fixo e completamente independentes, incluindo discos rígidos, disquetes, CD-ROMs e memória flash. Os dispositivos de bloco normalmente são usados para armazenar sistemas de arquivos, mas o acesso direto a um dispositivo de bloco também é permitido, de modo que os programas podem criar e reparar o sistema de arquivos que o dispositivo contém. As aplicações também podem acessar esses dispositivos de bloco diretamente, se desejarem; por exemplo, uma aplicação de banco de dados pode preferir realizar seu próprio layout ajustado dos dados no disco, em vez de usar o sistema de arquivos de uso geral.

Dispositivos de caractere incluem a maioria dos outros dispositivos, como mouse e teclado. A diferença fundamental entre dispositivos de bloco e caractere é o acesso aleatório - os dispositivos de bloco podem ser acessados aleatoriamente, enquanto os dispositivos de caractere são acessados apenas em série. Por exemplo, buscar determinada posição em um arquivo poderia ser aceito para um DVD, mas não faz sentido algum em um dispositivo de apontamento, como um mouse.

Dispositivos de rede são tratados de forma diferente dos dispositivos de bloco e de caractere. Os usuários não podem transferir dados diretamente para os dispositivos de rede; em vez disso, eles precisam se comunicar indiretamente abrindo uma conexão ao subsistema de rede do kernel. Discutimos a interface para os dispositivos de rede separadamente, na [Seção 21.10](#).

21.8.1 Dispositivos de bloco

Os dispositivos de bloco fornecem a interface principal para todos os dispositivos de disco em um sistema. O desempenho é particularmente importante para discos, e o sistema de dispositivo em bloco precisa fornecer funcionalidade para garantir que o acesso ao disco será o mais rápido possível. Essa funcionalidade é alcançada por meio do escalonamento de operações de E/S.

No contexto dos dispositivos de bloco, um **bloco** representa a unidade com a qual o kernel realiza a E/S. Quando um bloco é lido para a memória, ele é armazenado em um **buffer**. O **gerenciador de requisições** é a camada de software que gerencia a leitura e a escrita do conteúdo do buffer de e para um driver de dispositivo de bloco.

Uma lista separada de requisições é mantida para cada driver de dispositivo de bloco. Tradicionalmente, essas requisições foram escalonadas de acordo com um algoritmo elevador unidirecional (C-SCAN), que explora a ordem em que as requisições são inseridas e removidas das listas. As listas de requisição são mantidas em ordem crescente por número de setor inicial. Quando uma requisição é aceita para processamento por um driver de dispositivo de bloco, ela é removida da lista. Ela é removida apenas depois que a E/S é terminada, quando o driver continua com a próxima requisição na lista, mesmo que novas requisições tenham sido inseridas na lista antes da

requisição ativa. À medida que novas requisições de E/S são feitas, o gerenciador de requisições tenta mesclar as requisições nas listas.

O escalonamento das operações de E/S mudou um pouco com a versão 2.6 do kernel. O problema fundamental com o algoritmo elevador é que as operações de E/S concentradas em uma região específica do disco podem resultar em starvation das requisições que precisam ocorrer em outras regiões do disco. O **escalonador de prazo de E/S** usado na versão 2.6 funciona de modo semelhante ao algoritmo elevador, exceto que também associa um prazo a cada requisição, resolvendo assim o problema de starvation. Como padrão, o prazo para requisições de leitura é de 0,5 segundo, e para requisições de escrita é de 5 segundos. O escalonador de prazo mantém uma **fila classificada** de operações de E/S pendentes, classificada por número de setor. Contudo, ele também mantém duas outras filas - uma **fila de leitura** para operações de leitura e uma **fila de escrita** para operações de escrita. Essas duas filas são ordenadas de acordo com o prazo. Cada requisição de E/S é colocada na fila classificada e na fila de leitura ou de escrita, conforme o caso. Normalmente, as operações de E/S ocorrem a partir da fila classificada. Contudo, se um prazo expirar para uma requisição na fila de leitura ou de escrita, as operações de E/S serão escalonadas a partir da fila que contém a requisição expirada. Essa política garante que uma operação de E/S não esperará mais do que seu tempo de expiração.

21.8.2 Dispositivos de caractere

Um driver de dispositivo de caractere pode ser praticamente qualquer driver de dispositivo que não ofereça acesso aleatório a blocos de dados fixos. Quaisquer drivers de dispositivo de caractere registrados no kernel do Linux também precisam registrar um conjunto de funções que implementa as operações de E/S de arquivo que o driver pode tratar. O kernel não realiza qualquer pré-processamento de uma requisição de leitura ou escrita em arquivo para um dispositivo de caractere, mas passa a requisição ao dispositivo em questão e permite que o dispositivo trate da requisição.

A principal exceção a essa regra é o subconjunto especial de drivers de dispositivos de caractere que implementam dispositivos terminais. O kernel mantém uma interface padrão para esses drivers, por meio de um conjunto de estruturas `tty_struct`. Cada uma dessas estruturas fornece uso de buffers e controle de fluxo sobre o fluxo de dados do dispositivo terminal e alimenta esses dados em uma disciplina de fila.

Uma **disciplina de fila** é um interpretador para as informações do dispositivo terminal. A disciplina de fila mais comum é a disciplina `tty`, que cola o fluxo de dados do terminal nos fluxos de entrada e saída padrão dos processos em execução do usuário, permitindo que esses processos se comuniquem diretamente com o terminal do usuário. Essa tarefa é complicada pelo fato de que mais de um processo desse tipo pode estar executando concorrentemente, e a disciplina de fila `tty` é responsável por unir e separar a entrada e saída do terminal dos vários processos conectados a ela enquanto esses processos estão suspensos ou são despertados pelo usuário.

Outras disciplinas de fila também são implementadas, e não têm nada a ver com a E/S para um processo do usuário. Os protocolos de rede PPP e SLIP são maneiras de codificar uma conexão de rede por um dispositivo terminal, como uma linha serial. Esses protocolos são implementados sob o Linux como drivers que, em uma extremidade, aparecem ao sistema terminal como disciplinas de fila e, na outra extremidade, aparecem ao sistema de rede como um driver de dispositivo de rede. Depois de uma dessas disciplinas de fila ter sido ativada em um dispositivo terminal, quaisquer dados que apareçam nesse terminal serão roteados diretamente para o driver de dispositivo de rede apropriado.

21.9 Comunicação entre processos

O Linux fornece um ambiente rico para permitir que os processos se comuniquem uns com os outros. A comunicação pode ser uma questão de permitir que outro processo saiba que algum evento ocorreu ou então pode envolver a transferência de dados de um processo para outro.

21.9.1 Síncronismo e sinais

O mecanismo-padrão do Linux para informar a um processo que um evento ocorreu é o **sinal**. Os sinais podem ser enviados de qualquer processo para qualquer outro processo, com restrições sobre os sinais enviados aos processos pertencentes a outro usuário. Porém existe uma quantidade limitada de sinais à disposição, e eles não podem transportar informações: só está disponível a um processo o fato de que ocorreu um sinal. Os sinais não precisam ser gerados por outro processo. O kernel também gera sinais internamente; por exemplo, ele pode enviar um sinal a um processo servidor quando os dados chegam em um canal da rede, para um processo pai quando um filho termina ou para um processo em espera quando um temporizador expira.

Internamente, o kernel do Linux não usa sinais para se comunicar com processos executando no modo kernel: se um processo no modo kernel estiver esperando até que ocorra um evento, ele normalmente não usará sinais para receber notificação desse evento. Em vez disso, a comunicação sobre eventos assíncronos que chegam dentro do kernel é realizada por meio de estados de escalonamento e estruturas `wait_queue` (fila de espera). Esses mecanismos permitem que os processos no modo kernel informem uns aos outros sobre eventos relevantes e também permitem que os eventos sejam gerados por drivers de dispositivos ou pelo sistema de rede. Sempre que um processo deseja esperar até que algum evento seja completado, ele se coloca em uma **fila de espera** associada a esse evento e informa ao escalonador que não é mais elegível para execução. Quando o evento tiver terminado, ele despertará cada processo na fila de espera. Esse procedimento permite que vários processos esperem por um único evento. Por exemplo, se vários processos estiverem tentando ler um arquivo de um disco, então todos eles serão despertados quando os dados tiverem sido lidos para a memória com sucesso.

Embora os sinais sempre tenham sido o mecanismo principal para comunicação de eventos assíncronos entre os processos, o Linux também implementa o mecanismo de semáforo do UNIX System V. Um processo pode esperar por um semáforo tão facilmente quanto pode esperar por um sinal, mas os semáforos possuem duas vantagens: grande quantidade de semáforos pode ser compartilhada entre vários processos independentes, e as operações sobre os semáforos podem ser realizadas de forma atômica. Internamente, o mecanismo de fila de espera padrão do Linux sincroniza os processos que estão se comunicando com semáforos.

21.9.2 Passagem de dados entre processos

O Linux fornece diversos mecanismos para passar dados entre os processos. O mecanismo de **pipe** padrão do UNIX permite que um processo filho herde um canal de comunicação do seu pai; os dados escritos em uma extremidade do pipe podem ser lidos no outro. Sob o Linux, os pipes aparecem como outro tipo de inode para o software do sistema de arquivos virtual, e cada pipe possui um par de filas de espera para sincronizar o leitor e o escritor. O UNIX também define um conjunto de facilidades de rede que pode enviar fluxos de dados para processos locais e remotos. As redes são abordadas na [Seção 21.10](#).

Outro método de comunicações entre os processos, a memória compartilhada, oferece um modo extremamente rápido para a comunicação de uma quantidade grande ou pequena de dados. Quaisquer dados escritos por um processo na região da memória compartilhada podem ser lidos por outro processo que tenha mapeado essa região para o espaço de endereços. A principal desvantagem da memória compartilhada é que, por si só, ela não fornece sincronismo. Um processo não pode perguntar ao sistema operacional se um pedaço da memória compartilhada foi escrito nem suspender a execução até tal escrita ocorrer. A memória compartilhada se torna poderosa quando usada em conjunto com outro mecanismo de comunicação entre processos que ofereça o sincronismo que falta.

Uma região da memória compartilhada no Linux é um objeto persistente que pode ser criado ou excluído pelos processos. Tal objeto é tratado como se fosse um pequeno espaço de endereços independente: os algoritmos de paginação do Linux podem decidir enviar as páginas de memória compartilhada para o disco, assim como podem enviar as páginas de dados de um processo. O objeto de memória compartilhada atua como um armazenamento de apoio para as regiões da memória compartilhada, assim como um arquivo pode atuar como um armazenamento de apoio para a região mapeada na memória. Quando um arquivo é mapeado na região do espaço de endereços virtuais, então quaisquer falhas de página que ocorrem fazem a página apropriada do arquivo ser mapeada na memória virtual. De modo semelhante, os mapeamentos de memória compartilhada instruem as falhas de página para mapear em páginas de um objeto persistente da memória compartilhada.

Também, como acontece para os arquivos, os objetos da memória compartilhada se lembram de seu conteúdo mesmo que nenhum processo os esteja mapeando para a memória virtual.

21.10 Estrutura de rede

As redes são uma área de funcionalidade importante para o Linux. O Linux não apenas admite os protocolos-padrão da Internet, usados para a maioria das comunicações de UNIX para UNIX, mas também implementa uma série de protocolos nativos a outros sistemas operacionais, fora o UNIX. Em particular, como o Linux foi implementado originalmente, sobretudo em PCs, em vez de grandes estações de trabalho ou em sistema da classe servidor, ele admite muitos dos protocolos utilizados em redes PC, como AppleTalk e IPX.

Internamente, as redes no kernel do Linux são implementadas por três camadas de software:

1. A interface de socket.
2. Drivers de protocolo.
3. Drivers de dispositivos de rede.

As aplicações do usuário realizam todas as requisições de rede por meio da interface de socket. Essa interface foi projetada para se parecer com a camada de socket do 4.3 BSD, de modo que quaisquer programas projetados para utilizar os sockets do Berkeley funcionarão no Linux sem quaisquer mudanças no código-fonte. A interface de socket do BSD é suficientemente genérica para representar os endereços de rede para uma grande gama de protocolos de rede. Essa interface única é usada no Linux para acessar não apenas os protocolos implementados nos sistemas BSD padrão, mas todos os protocolos admitidos pelo sistema.

A camada de software seguinte é a pilha de protocolos, que é semelhante em organização à própria estrutura do BSD. Sempre que quaisquer dados de rede chegam nessa camada, tenham vindo de um socket de aplicação ou de um driver de dispositivos de rede, espera-se que os dados tenham sido marcados com um identificador que especifica qual protocolo de rede eles contêm. Os protocolos podem se comunicar entre si, se desejarem; por exemplo, dentro do conjunto de protocolos da Internet, protocolos separados gerenciam roteamento, informe de erros e retransmissão confiável de dados perdidos.

A camada de protocolo pode reescrever pacotes, criar novos pacotes, separar ou montar pacotes em fragmentos, ou descartar os dados que chegam. Por fim, quando a camada de protocolo tiver terminado de processar um conjunto de pacotes, ela os passa adiante, para a interface de socket, se os dados forem destinados para uma conexão local, ou para um driver de dispositivos, se os dados precisarem ser transmitidos remotamente. A camada de protocolo decide para qual socket ou dispositivo enviará o pacote.

Toda a comunicação entre as camadas da pilha de rede é realizada pela passagem de estruturas skbuff (socket buffer) isoladas. Cada uma dessas estruturas contém um conjunto de ponteiros para uma única área contígua da memória, representando um buffer dentro do qual os pacotes de rede podem ser construídos. Os dados válidos em um skbuff não precisam começar no início do buffer do skbuff e não precisam seguir até o final. O código de rede pode acrescentar dados ou remover dados de qualquer extremidade do pacote, desde que o resultado ainda caiba no skbuff. Essa capacidade é especialmente importante nos microprocessadores modernos, nos quais as melhorias na velocidade de CPU ultrapassam muito o desempenho da memória principal: a arquitetura skbuff permite flexibilidade na manipulação de cabeçalhos de pacote e somas de verificação, enquanto evita qualquer cópia de dados desnecessária.

O conjunto de protocolos mais importante no sistema de rede do Linux é o conjunto do protocolo da Internet (IP). Esse conjunto é composto de diversos protocolos separados. O protocolo IP implementa o roteamento entre diferentes hosts em qualquer lugar na rede. Em cima do protocolo de roteamento estão montados os protocolos UDP, TCP e ICMP. O protocolo UDP transporta datagramas individuais entre os hosts. O protocolo TCP implementa conexões confiáveis entre os hosts com remessa garantida de pacotes em ordem e retransmissão automática de dados perdidos. O protocolo ICMP é usado para transportar diversas mensagens de erro e status entre os hosts.

Cada pacote (skbuff) que chega no software de protocolo da pilha de rede já deverá estar marcado com um identificador interno, indicando para qual protocolo o pacote é relevante. Diferentes drivers de dispositivos de rede codificam o tipo de protocolo de diferentes maneiras; assim, o protocolo para os dados que chegam precisa ser identificado no driver de dispositivos. O driver de dispositivos utiliza uma tabela de hash de identificadores de protocolo de rede conhecidos, para pesquisar o protocolo apropriado, e passa o pacote para esse protocolo. Novos protocolos podem ser acrescentados à tabela de hash como módulos carregáveis no kernel.

Os pacotes IP que chegam são entregues ao driver IP. A tarefa dessa camada é realizar o roteamento. Após decidir para onde o pacote deve ser enviado, o driver IP encaminha o pacote para o driver de protocolo interno apropriado para ser entregue localmente ou o injeta de volta para uma fila de drivers de dispositivo de rede apropriada, para ser encaminhado a outro host. Ele realiza a decisão de roteamento usando duas tabelas: a Forwarding Information Base - FIB (base de informações de encaminhamento) persistente e um cache de decisões de encaminhamento recentes. A FIB mantém informações de configuração de roteamento e pode especificar rotas com base em um endereço de destino específico ou em um curinga representando vários destinos. A FIB é organizada como um conjunto de tabelas de hash indexadas por endereço de destino; as tabelas representando

as rotas mais específicas sempre são pesquisadas primeiro. Pesquisas bem-sucedidas dessa tabela são acrescentadas à tabela de caching de rota, que abriga as rotas somente por destino específico; nenhum curinga é armazenado no cache, de modo que as pesquisas podem ser feitas rapidamente. Uma entrada no cache de rotas expira após um período fixo sem acertos.

Em vários estágios, o software IP passa os pacotes para uma seção de código separada para **gerenciamento de firewall** - a filtragem seletiva de pacotes de acordo com critérios arbitrários, normalmente para fins de segurança. O gerenciador de firewall mantém diversas **cadeias de firewall** separadas e permite que um skbuff seja associado a qualquer cadeia. As cadeias são reservadas para finalidades separadas: uma é usada para pacotes encaminhados, uma para pacotes entrando nesse host e outra para dados gerados nesse host. Cada cadeia é mantida como uma lista ordenada de regras, onde uma regra específica uma de uma série de funções de decisão de firewall possíveis e mais alguns dados arbitrários para associação.

Duas outras funções realizadas pelo driver IP são desmontagem e remontagem de pacotes grandes. Se um pacote que está sendo enviado for muito grande para ser enfileirado em um dispositivo, ele é dividido em **fragmentos** menores, enfileirados para o driver. No host receptor, esses fragmentos precisam ser remontados. O driver IP mantém um objeto ipfrag para cada fragmento aguardando a remontagem, e um ipq para cada datagrama sendo montado. Os fragmentos que chegam são associados a cada ipq conhecido. Se for encontrada uma combinação, o fragmento é acrescentado a ele; caso contrário, um novo ipq é criado. Quando o fragmento final de um ipq tiver chegado, um skbuff completamente novo será construído para manter o novo pacote, e esse pacote será passado de volta ao driver IP.

Os pacotes associados pelo IP conforme destinados a esse host são passados adiante para um dos outros drivers de protocolo. Os protocolos UDP e TCP compartilham um meio de associar pacotes aos sockets de origem e destino: cada par de sockets conectado é identificado exclusivamente por seus endereços de origem e destino e pelos números de porta de origem e destino. As listas de socket são vinculadas às tabelas de hash com chaves contendo esses quatro valores de endereço e porta para pesquisa de socket nos pacotes que chegam. O protocolo TCP precisa lidar com conexões não confiáveis, de modo que mantém listas ordenadas de pacotes que saem não confirmados, para retransmitir após determinado tempo limite, e de pacotes que chegam fora de ordem, para serem apresentados ao socket quando os dados que faltam tiverem chegado.

21.11 Segurança

O modelo de segurança do Linux está bastante relacionado com mecanismos de segurança típicos do UNIX. Os problemas de segurança podem ser classificados em dois grupos:

1. **Autenticação.** Certificar-se de que ninguém possa acessar o sistema sem primeiro provar que possui direitos de entrada.
2. **Controle de acesso.** Fornecer um mecanismo para verificar se um usuário tem o direito de acessar um certo objeto e impedir o acesso aos objetos conforme a necessidade.

21.11.1 Autenticação

A autenticação no UNIX tem sido feita por meio de um arquivo de senhas publicamente legíveis. A senha de um usuário é combinada com um valor de "sal" aleatório, e o resultado é codificado com uma função de transformação de mão única e armazenado no arquivo de senhas. O uso da função de mão única significa que a senha original não pode ser deduzida do arquivo de senhas, exceto por tentativa e erro. Quando um usuário apresenta uma senha ao sistema, a senha é recombina com o valor de sal armazenado no arquivo de senha e passada pela mesma transformação de mão única. Se o resultado combinar com o conteúdo do arquivo de senha, então a senha é aceita.

Historicamente, as implementações desse mecanismo no UNIX têm vários problemas. As senhas eram limitadas a oito caracteres, e o número de valores de sal possíveis era tão baixo que um invasor poderia combinar com facilidade um dicionário de senhas normalmente utilizado a cada valor de sal possível e ter uma boa chance de combinar com uma ou mais senhas no arquivo de senhas, obtendo acesso não autorizado a quaisquer contas comprometidas como resultado disso. Extensões ao mecanismo de senha foram introduzidas para manter a senha criptografada secreta em um arquivo que não está legível publicamente, que permitam senhas maiores ou que utilizem métodos mais seguros de codificação da senha. Outros mecanismos de autenticação foram introduzidos para limitar os horários durante os quais um usuário tem permissão para se conectar ao sistema. Além disso, existem mecanismos para distribuir informações de autenticação a todos os sistemas relacionados em uma rede.

Um novo mecanismo de segurança foi desenvolvido por fornecedores de UNIX para resolver essas questões. O sistema de **Pluggable Authentication Modules - PAM (módulos de autenticação conectáveis)** é baseado em uma biblioteca compartilhada que pode ser usada por qualquer componente do sistema que precise autenticar usuários. Uma implementação desse sistema está disponível sob o Linux. O PAM permite que módulos de autenticação sejam carregados por demanda conforme especificado em um arquivo de configuração no nível do sistema. Se um novo mecanismo de autenticação for acrescentado mais tarde, ele poderá ser acrescentado ao arquivo de configuração, e todos os componentes do sistema serão capazes de tirar proveito dele. Um módulo PAM pode especificar métodos de autenticação, restrições de conta, funções de configuração de sessão ou funções de troca de senha (para que, quando os usuários mudarem suas senhas, todos os mecanismos de autenticação necessários possam ser atualizados de uma só vez).

21.11.2 Controle de acesso

O controle de acesso sob sistemas UNIX, incluindo o Linux, é realizado por meio de identificadores numéricos exclusivos. Um user id (UID - identificador de usuário) identifica um único usuário ou um único conjunto de direitos de acesso. Um group id (GID - identificador de grupo) é um identificador extra que pode ser usado para identificar direitos pertencentes a mais de um usuário.

O controle de acesso é aplicado a diversos objetos no sistema. Cada arquivo disponível no sistema é protegido pelo mecanismo de controle de acesso-padrão. Além disso, outros objetos compartilhados, como seções da memória compartilhada e semáforos, empregam o mesmo sistema de acesso.

Cada objeto em um sistema UNIX sob o controle de acesso do usuário e grupo possui um único UID e um único GID associado a ele. Os processos do usuário também possuem um único UID, mas podem ter mais de um GID. Se o GID de um processo combinar com o UID de um objeto, então o processo possui **direitos de usuário** ou **direitos de proprietário** para esse objeto. Se os UIDs não combinarem, mas qualquer um dos GIDs do processo combinar com o GID do objeto, então os **direitos de grupo** serão conferidos; caso contrário, o processo tem **direitos globais** ao objeto.

O Linux realiza controle de acesso atribuindo aos objetos uma **máscara de proteção**, que especifica quais modos de acesso - leitura, escrita ou execução - devem ser concedidos aos processos com acesso de proprietário, grupo ou mundo. Assim, o proprietário de um objeto poderia ter acesso total para leitura, escrita e execução de um arquivo; outros usuários em determinado grupo poderiam receber acesso de leitura, mas não acesso de escrita; e todos os outros poderiam não receber acesso algum.

A única exceção é o UID **root** privilegiado. Um processo com esse UID especial recebe acesso automático a qualquer objeto no sistema, contornando as verificações de acesso normais. Esses

processos também recebem permissão para realizar operações privilegiadas, como ler qualquer memória física ou abrir sockets de rede reservados. Esse mecanismo permite que o kernel impeça que usuários normais acessem esses recursos: a maioria dos principais recursos internos do kernel pertence implicitamente a esse UID root.

O Linux implementa o mecanismo `setuid` padrão do UNIX; esse mecanismo permite que um programa seja executado com privilégios diferentes daqueles do usuário que executa o programa: por exemplo, o programa `lpr` (que submete uma tarefa a uma fila de impressão) tem acesso às filas de impressão do sistema mesmo que o usuário que executa esse programa não o tenha. A implementação de `setuid` no UNIX distingue entre o UID *real* e *efetivo* de um processo. O UID real é aquele do usuário que executa o programa; o UID efetivo é aquele do proprietário do arquivo.

No Linux, esse mecanismo é aumentado de duas maneiras. Primeiro, o Linux implementa o mecanismo de `user-id` salvo da especificação POSIX, que permite que um processo perca e readquira seu UID efetivo repetidamente. Por motivos de segurança, um programa pode querer realizar a maioria de suas operações em um modo seguro, abrindo mão dos privilégios concedidos por seu status `setuid`, mas pode querer realizar operações selecionadas com todos os seus privilégios. As implementações-padrão do UNIX só conseguem essa capacidade trocando os UIDs real e efetivo; o UID efetivo anterior é lembrado, mas o UID real do programa nem sempre corresponde ao UID do usuário que executa o programa. UIDs salvos permitem que um processo defina seu UID efetivo como o seu UID real e depois volte ao valor anterior do seu UID efetivo, sem ter de modificar o UID real em momento algum.

A segunda melhoria oferecida pelo Linux é o acréscimo de uma característica do processo que concede apenas um subconjunto dos direitos do UID efetivo. As propriedades de processo **fsuid** e **fsgid** são usadas quando os direitos de acesso são concedidos aos arquivos. A propriedade apropriada é definida toda vez que o UID ou o GID efetivo é definido. Contudo, o `fsuid` e o `fsgid` podem ser definidos independentemente dos ids efetivos, permitindo que um processo acesse arquivos em favor de outro usuário sem assumir a identidade desse outro usuário de qualquer outra maneira. Especificamente, os processos servidores podem usar esse mecanismo para enviar arquivos a determinado usuário sem que o processo se torne vulnerável a encerramento ou suspensão por parte desse usuário.

Por último, o Linux fornece um mecanismo para a passagem flexível de direitos de um programa para outro que tem se tornado comum nas versões modernas do UNIX. Quando um socket de rede local tiver sido configurado entre dois processos quaisquer no sistema, qualquer um desses processos pode enviar ao outro processo um descritor de arquivo para um de seus arquivos abertos; o outro processo recebe uma duplicata do descritor para o mesmo arquivo. Esse mecanismo permite que um cliente passe o acesso a um único arquivo seletivamente para algum processo servidor, sem conceder a esse processo quaisquer outros privilégios. Por exemplo, não é mais necessário que um servidor de impressão possa ler todos os arquivos de um usuário que submete uma nova tarefa de impressão; o cliente da impressão pode passar ao servidor os descritores de arquivo para quaisquer arquivos a serem impressos, negando o acesso do servidor a qualquer um dos outros arquivos do usuário.

21.12 Resumo

O Linux é um sistema operacional moderno e gratuito, baseado nos padrões do UNIX. Ele tem sido projetado para execução eficiente e confiável no hardware comum do PC, sendo também executado em diversas outras plataformas. Ele fornece uma interface de programação e uma interface de usuário compatíveis com os sistemas UNIX padrão, e pode executar uma grande quantidade de aplicações UNIX, incluindo uma quantidade cada vez maior de aplicações com suporte comercial.

O Linux não evoluiu no vácuo. Um sistema Linux completo inclui diversos componentes desenvolvidos de forma independente do próprio Linux. O kernel centrado do sistema operacional Linux é original, mas permite a execução de muito software UNIX gratuito já existente, resultando em um sistema operacional compatível com o UNIX, sem código proprietário.

O kernel do Linux é implementado como um kernel monolítico tradicional por motivos de desempenho, mas seu projeto é modular o bastante para permitir que a maioria dos drivers seja carregada e descarregada dinamicamente durante a execução.

O Linux é um sistema multusuário, fornecendo proteção entre os processos e execução de múltiplos processos de acordo com um escalonador de tempo compartilhado. Os processos recém-criados podem compartilhar partes seletivas de seu ambiente de execução com seus processos pai, permitindo a programação multithreads. A comunicação entre processos tem o suporte dos mecanismos do System V - filas de mensagem, semáforos e memória compartilhada - e da interface de socket do BSD. Diversos protocolos de rede podem ser acessados, simultaneamente, por meio da interface de socket.

O sistema de gerência de memória utiliza o compartilhamento de página e a cópia na escrita para minimizar a duplicação de dados compartilhados por diferentes processos. As páginas são carregadas por demanda, quando são referenciadas inicialmente, e são paginadas de volta ao armazenamento de apoio de acordo com um algoritmo LFU, se a memória física precisar ser reivindicada.

Para o usuário, o sistema de arquivos se parece com uma árvore de diretórios hierárquica, que obedece à semântica do UNIX. Internamente, o Linux utiliza uma camada de abstração para gerenciar vários sistemas de arquivos. Sistemas de arquivos orientados a dispositivo, em rede e virtuais são admitidos. Os sistemas de arquivos orientados a dispositivo acessam o armazenamento em disco por meio de um cache de página, que é unificado com o sistema de memória virtual.

Exercícios práticos

- 21.1. Módulos do kernel carregáveis dinamicamente oferecem flexibilidade quando são acrescentados drivers a um sistema, mas eles também possuem desvantagens? Sob quais circunstâncias um kernel seria compilado em um único arquivo binário, e quando seria melhor mantê-lo separado em módulos? Explique sua resposta.
- 21.2. O multithreading é uma técnica de programação bastante utilizada. Descreva três maneiras diferentes de implementar threads e compare esses três métodos com o mecanismo clone() do Linux. Quando seria melhor ou pior usar cada um desses mecanismos alternativos do que usar clones?
- 21.3. O kernel do Linux não permite a paginação para fora da memória do kernel. Que efeito tem essa restrição sobre o projeto do kernel? Cite duas vantagens e duas desvantagens dessa decisão de projeto.
- 21.4. Discuta três vantagens do link dinâmico (compartilhado) das bibliotecas em comparação com o link estático. Descreva dois casos em que o link estático é preferível.
- 21.5. Compare o uso de sockets em rede com o uso da memória compartilhada como mecanismo para comunicação de dados entre processos em um único computador. Quais são as vantagens de cada método? Quando cada um poderia ser preferido?
- 21.6. Houve uma época em que os sistemas UNIX usavam otimizações de layout de disco com base na posição de rotação dos dados do disco, mas as implementações modernas, incluindo o Linux, otimizam para o acesso sequencial aos dados. Por que eles fazem isso? De quais características do hardware o acesso sequêncial tira proveito? Por que a otimização rotacional não é mais tão útil?

Exercícios

- 21.7. Quais são as vantagens e desvantagens de escrever um sistema operacional em uma linguagem de alto nível, como C?
- 21.8. Em quais circunstâncias a sequência de chamadas de sistema `fork()` `exec()` é mais apropriada? Quando é preferível usar `vfork()`?
- 21.9. Que tipo de socket deve ser usado para implementar um programa de transferência de arquivos entre computadores? Que tipo deve ser usado para um programa que testa periodicamente para ver se outro computador está ativo na rede? Explique sua resposta.
- 21.10. O Linux pode ser executado em diversas plataformas de hardware. Que medidas os desenvolvedores do Linux precisam tomar para garantir que o sistema seja portável para diferentes processadores e arquiteturas de gerência de memória e também para reduzir a quantidade de código do kernel específica da arquitetura?
- 21.11. Quais são as vantagens e desvantagens de tornar apenas alguns dos símbolos definidos dentro de um kernel acessíveis a um módulo carregável do kernel?
- 21.12. Quais são os principais objetivos do mecanismo de resolução de conflito usado pelo kernel do Linux para a carga de módulos do kernel?
- 21.13. Discuta como a operação `clone()` admitida pelo Linux é usada para dar suporte a processos e threads.
- 21.14. As threads do Linux devem ser classificadas como threads em nível de usuário ou threads em nível de kernel? Corrobore sua resposta com os argumentos apropriados.
- 21.15. Que custos extras ocorrem pela criação e escalonamento de um processo, comparado com o custo de uma thread clonada?
- 21.16. O escalonador do Linux implementa o escalonamento de tempo real *flexível*. Que recursos necessários para certas tarefas de programação de tempo real estão faltando? Como eles poderiam ser acrescentados ao kernel?
- 21.17. Sob quais circunstâncias um processo do usuário requisitaria uma operação que resulta na alocação de uma região da memória de demanda zero?
- 21.18. Que cenários fariam uma página de memória ser mapeada para o espaço de endereços de um programa do usuário com o atributo de cópia na escrita ativado?
- 21.19. No Linux, as bibliotecas compartilhadas realizam muitas operações centrais ao sistema operacional. Qual é a vantagem de manter essa funcionalidade a partir do kernel? Existem desvantagens? Explique sua resposta.
- 21.20. A estrutura de diretórios de um sistema operacional Linux poderia ser composta de arquivos correspondentes a vários sistemas de arquivos diferentes, incluindo o sistema de arquivos /proc do Linux. Como a necessidade de dar suporte a diferentes tipos de sistemas de arquivos poderia afetar a estrutura do kernel do Linux?
- 21.21. Quais são as diferenças entre o recurso setuid do Linux e o recurso setuid do Unix padrão?
- 21.22. O código-fonte do Linux está disponível ampla e gratuitamente pela Internet ou por vendedores de CD-ROM. Quais são três implicações dessa disponibilidade para a segurança do sistema Linux?

Notas bibliográficas

O sistema Linux é um produto da Internet; como resultado, a maioria da documentação disponível sobre o Linux está disponível de alguma forma na Internet. Os sites a seguir referenciam a maioria das informações úteis disponíveis:

- As Linux Cross-Reference Pages, em <http://lxr.linux.no/>, mantêm as listagens atuais do kernel do Linux navegáveis pela Web e com referência cruzada total.
- Linux-HQ, em <http://www.linuxhq.com/>, hospeda uma grande quantidade de informações relativas aos kernels do Linux 2.x. Esse site também inclui links para as páginas iniciais da maioria das distribuições Linux, bem como arquivamentos das principais listas de correspondência.
- O Linux Documentation Project, em <http://sunsite.unc.edu/linux/>, lista muitos livros sobre Linux disponíveis no formato-fonte como parte do Linux Documentation Project. O projeto também hospeda os guias *How-To* do Linux, que contêm uma série de sugestões e dicas relativas aos aspectos do Linux.
- O *Kernel Hackers' Guide* é um guia disponível na Internet para os detalhes internos do kernel em geral. Esse site em constante expansão está localizado em <http://www.redhat.com:8080/HyperNews/get/khg.html>.
- O site Linux Kernel Newbies (<http://www.kernelnewbies.org/>) é um recurso para apresentar o Kernel do Linux aos iniciantes.

Existem muitas listas de discussão dedicadas ao Linux. As mais importantes estão mantidas por um gerenciador de lista de discussão que pode ser contatado no endereço majordomo@vger.rutgers.edu. Envie uma mensagem de correio eletrônico para esse endereço com a única linha "help" no corpo da mensagem, para obter informações sobre como acessar o servidor de lista e se inscrever em qualquer uma das listas.

Finalmente, o próprio sistema Linux pode ser obtido pela Internet. Distribuições Linux completas podem ser obtidas a partir dos sites das empresas interessadas, e a comunidade Linux também mantém arquivamentos dos componentes atuais do sistema em vários lugares na Internet. Os mais importantes são estes:

- <ftp://tsx-11.mit.edu/pub/linux/>
- <ftp://sunsite.unc.edu/pub/Linux/>
- <ftp://linux.kernel.org/pub/linux/>

Além de investigar recursos da Internet, você poderá ler a respeito dos detalhes do kernel do Linux em [Bovet e Cesati \[2002\]](#) e [Love \[2004\]](#).

CAPÍTULO 22

Windows XP

O sistema operacional Microsoft Windows XP é um sistema operacional multitarefa preemptivo de 32/64 bits para AMD K6/K7, Intel IA32/IA64 e microprocessadores mais recentes. Sucessor do Windows NT/2000, o Windows XP também tem como finalidade substituir o sistema operacional Windows 95/98. Neste capítulo, discutimos os principais objetivos, a arquitetura em camadas do sistema, que facilita seu uso, o sistema de arquivos, redes e a interface de programação.

OBJETIVOS DO CAPÍTULO

- Explorar os princípios sobre os quais o Windows XP foi projetado e os componentes específicos do sistema.
- Entender como o Windows XP pode executar programas projetados para outros sistemas operacionais.
- Fornecer uma explicação detalhada do sistema de arquivos do Windows XP.
- Ilustrar os protocolos de rede suportados no Windows XP.
- Descrever as interfaces disponíveis aos programadores de sistemas e aplicações.

22.1 História

Em meados da década de 1980, a Microsoft e a IBM se uniram para desenvolver o sistema operacional OS/2, escrito em linguagem assembly para sistemas Intel 80286 de monoprocessadores. Em 1988, a Microsoft decidiu começar do zero e desenvolver um sistema operacional portável de “nova tecnologia” (ou NT), que admitisse as interfaces de programação de aplicação (APIs) do OS/2 e POSIX. Em outubro de 1988, Dave Cutler, o arquiteto do sistema operacional VAX/VMS da DEC, foi contratado e recebeu a licença para a construção desse novo sistema operacional.

Originalmente, a equipe planejou usar o OS/2 como ambiente nativo do NT, mas, durante o desenvolvimento, o NT foi alterado para usar a API Windows de 32 bits (ou API Win32), refletindo a popularidade do Windows 3.0. As primeiras versões do NT foram o Windows NT 3.1 e o Windows NT 3.1 Advanced Server. (Nessa época, o Windows de 16 bits estava na versão 3.1.) O Windows NT versão 4.0 adotou a interface com o usuário do Windows 95 e incorporou software de servidor Web e navegador Web da Internet. Além disso, as rotinas da interface com o usuário e todo o código gráfico foram passados para o kernel, para melhorar o desempenho, com o efeito colateral de menor confiabilidade do sistema. Embora as versões anteriores do NT tivessem sido portadas para outras arquiteturas de microprocessador, a versão Windows 2000, lançada em fevereiro de 2000, deu suporte apenas para processadores Intel (e compatíveis), devido a fatores de mercado. O Windows 2000 incorporava mudanças significativas. Ele acrescentava o Active Directory (um serviço de diretório baseado no padrão X.500), melhor suporte para rede e laptop, suporte para dispositivos plug-and-play, um sistema de arquivos distribuído e suporte para mais processadores e mais memória.

Em outubro de 2001, o Windows XP foi lançado como uma atualização ao sistema operacional desktop Windows 2000 e um substituto para o Windows 95/98. Em 2002, apareceram as versões de servidor do Windows XP (chamadas Windows .Net Server). O Windows XP atualiza a interface gráfica do usuário com um projeto visual que aproveita os avanços de hardware mais recentes e muitos recursos novos e **fáceis de usar**. Diversos recursos foram acrescentados para reparar automaticamente os problemas nas aplicações e no próprio sistema operacional. Como resultado dessas mudanças, o Windows XP fornece melhor experiência de rede e dispositivos, incluindo configuração zero de dispositivos sem fio (wireless), mensagens instantâneas, mídia streaming, fotografia/vídeo digital, melhorias de desempenho incríveis, tanto para desktop quanto para grandes sistemas multiprocessados, e melhor confiabilidade e segurança até mesmo em relação a sistemas operacionais Windows anteriores.

O Windows XP usa uma arquitetura cliente-servidor (como o Mach) para implementar várias personalidades do sistema operacional, como Win32 e POSIX, com processos no nível do usuário denominados subsistemas. A arquitetura de subsistema permite que melhorias sejam feitas a uma personalidade do sistema operacional sem afetar a compatibilidade de aplicação de nenhuma outra personalidade.

O Windows XP é um sistema operacional multiusuário, admitindo acesso simultâneo por meio de serviços distribuídos ou de várias instâncias da interface gráfica com o usuário, por meio do servidor de terminal do Windows. As versões de servidor do Windows XP admitem sessões de servidor de terminal simultâneas a partir dos sistemas desktop do Windows. As versões desktop do servidor de terminal multiplexam o teclado, mouse e monitor entre as sessões virtuais do terminal para cada usuário conectado. Esse recurso, chamado troca rápida do usuário, permite aos usuários passarem de um para outro no console de um PC sem ter de se desconectar e conectar ao sistema.

O Windows XP é a primeira versão do Windows de 64 bits. O NT File System (NTFS) nativo e muitas das APIs Win32 sempre usaram inteiros de 64 bits quando apropriado, de modo que a principal extensão para 64 bits no Windows XP é o suporte para endereços grandes.

Existem duas versões desktop do Windows XP. O Windows XP Professional é o sistema desktop para usuários especialistas, no trabalho e em casa. O Windows XP Personal fornece a confiabilidade e facilidade de uso do Windows XP, mas faltam os recursos mais avançados necessários para trabalhar de forma transparente com o Active Directory ou executar aplicações POSIX.

Os membros da família Windows .Net Server utilizam os mesmos componentes básicos das versões desktop, mas acrescentam uma gama de recursos necessários, como grupos de servidores Web, servidores de impressão e arquivos, sistemas agrupados e grandes máquinas de centro de dados. As grandes máquinas de centro de dados podem ter até 64 GB de memória e 32 processadores, em sistemas IA32, e 128 GB e 64 processadores nos sistemas IA64.

22.2 Princípios de projeto

Os objetivos de projeto da Microsoft para o Windows XP incluem segurança, confiabilidade, compatibilidade com aplicações Windows e POSIX, alto desempenho, extensibilidade, portabilidade e suporte internacional. Discutimos cada um desses objetivos nas próximas seções.

22.2.1 Segurança

Os objetivos de **segurança** do Windows XP exigiram mais do que apenas a adesão aos padrões de projeto que permitiram ao Windows NT 4.0 receber a classificação de segurança C2 do governo dos Estados Unidos (o que significa um nível de proteção moderado contra software defeituoso e ataques maliciosos). Uma revisão e um teste completo do código foram combinados com sofisticadas ferramentas de análise automática para identificar e investigar defeitos em potencial que pudessem representar vulnerabilidades de segurança.

22.2.2 Confiabilidade

O Windows 2000 foi o sistema operacional mais confiável e estável que a Microsoft já havia lançado até aquele ponto. Grande parte dessa confiabilidade vinha da maturidade do código-fonte, teste de estresse profundo do sistema e detecção automática de muitos erros sérios nos drivers. Os requisitos de **confiabilidade** para o Windows XP foram ainda mais rigorosos. A Microsoft usou revisões manuais e automáticas completas no código, para identificar mais de 63.000 linhas nos arquivos-fonte que pudessem conter problemas não detectados pelo teste e começou a rever cada área para verificar se o código estava correto.

O Windows XP estendeu a verificação de driver para apanhar bugs mais sutis, melhorar as facilidades para encontrar erros de programação no código em nível de usuário e sujeitar aplicações de terceiros, drivers e dispositivos a um processo de certificação rigoroso. Além do mais, o Windows XP acrescenta novas facilidades para monitorar a saúde do PC, incluindo o download de reparos para problemas antes de eles serem encontrados pelos usuários. A confiabilidade percebida do Windows XP também foi melhorada, tornando a interface gráfica com o usuário mais fácil de usar, por meio de um projeto visual melhor, menus mais simples e melhorias na facilidade com a qual os usuários podem descobrir como realizar tarefas comuns.

22.2.3 Compatibilidade com aplicações Windows e Posix

Como dissemos, o Windows XP não é apenas uma atualização do Windows 2000, mas também um substituto para o Windows 95/98. O Windows 2000 focalizou principalmente a compatibilidade para aplicações comerciais. Os requisitos para Windows XP incluem uma compatibilidade muito mais alta com aplicações de consumidor executadas no Windows 95/98. A **compatibilidade das aplicações** é difícil de ser alcançada, porque cada aplicação verifica uma versão em particular do Windows, pode depender de alguma maneira de detalhes de alguma implementação das APIs e pode ter bugs latentes, disfarçados no sistema anterior, e assim por diante.

O Windows XP introduz uma camada de compatibilidade encontrada entre as aplicações e as APIs Win32. Essa camada faz o Windows XP parecer compatível (quase) bug a bug com as versões anteriores do Windows. Ele, como as versões NT anteriores, mantém o suporte para a execução de muitas aplicações de 16 bits usando uma camada de *thunking*, que traduz as chamadas da API de 16 bits para as chamadas equivalentes de 32 bits. De modo semelhante, a versão de 64 bits do Windows XP fornece uma camada de *thunking* que traduz as chamadas da API de 32 bits para chamadas nativas de 64 bits. Além disso, o suporte para POSIX no Windows XP foi bastante melhorado por um novo subsistema POSIX, denominado Interix. A maioria dos softwares compatíveis com UNIX que existem é compilada e executada sob o Interix sem modificação alguma.

22.2.4 Alto desempenho

O Windows XP foi projetado para fornecer **alto desempenho** em sistemas desktop (em grande parte restritos pelo desempenho da E/S), sistemas servidores (nos quais a CPU é o gargalo) e grandes ambientes multiprocessadores e threads (onde o lock e o gerenciamento da fila do cache são fundamentais para a escalabilidade). Para satisfazer os requisitos de desempenho, o NT utiliza diversas técnicas, como E/S assíncrona, protocolos otimizados para redes (por exemplo, lock otimista de dados distribuídos, requisições batch), gráficos baseados no kernel e caching sofisticado de dados do sistema de arquivos. Os algoritmos de gerência de memória e sincronismo são projetados com uma consciência sobre as considerações de desempenho relacionadas com filas de cache e multiprocessadores.

O Windows XP melhorou ainda mais o desempenho, reduzindo a extensão do caminho do código nas funções críticas, usando melhores algoritmos e estruturas de dados por processador, usando

coloração de memória para arquiteturas NUMA (Non-Uniform Memory Access) e implementando protocolos de lock mais escaláveis, como spinlocks enfileirados. Os novos protocolos de lock ajudam a reduzir os ciclos do barramento do sistema e incluem listas e filas sem lock, uso de operações de leitura/modificação/escrita atômicas (como o incremento interbloqueado) e outras técnicas de lock avançadas.

Os subsistemas que constituem o Windows XP se comunicam entre si de forma eficiente por uma chamada de procedimento local (Local Procedure Call - LPC) que fornece troca de mensagens com alto desempenho. Exceto enquanto são executadas no despachante do kernel, as threads nos subsistemas do Windows XP podem ser preemptadas por threads de prioridade mais alta. Assim, o sistema responde rapidamente aos eventos externos. Além disso, o Windows XP foi projetado para multiprocessamento simétrico; em um computador multiprocessado, várias threads podem ser executadas ao mesmo tempo.

22.2.5 Extensibilidade

Extensibilidade refere-se à capacidade de um sistema operacional acompanhar os avanços na tecnologia de computação. Para facilitar as mudanças com o tempo, os desenvolvedores implementaram o Windows XP usando uma arquitetura em camadas. O executivo do Windows XP roda no kernel ou no modo protegido e fornece os serviços básicos do sistema. Em cima do executivo, vários subsistemas de servidor operam no modo usuário. Entre eles, estão os **subsistemas de ambiente** que simulam diferentes sistemas operacionais. Assim, os programas escritos para MS-DOS, Microsoft Windows e POSIX executam no Windows XP no ambiente apropriado. (Ver mais informações sobre subsistemas de ambiente na [Seção 22.4](#).) Devido à sua estrutura modular, outros subsistemas de ambiente podem ser acrescentados ao Windows XP sem afetar o executivo. Além disso, o Windows XP utiliza drivers carregáveis no sistema de E/S, de modo que novos sistemas de arquivo, novos tipos de dispositivos de E/S e novos tipos de rede podem ser acrescentados enquanto o sistema está sendo executado. O Windows XP utiliza um modelo cliente-servidor como o sistema operacional Mach e admite o processamento distribuído por remote procedure calls (RPCs), conforme definido pela Open Software Foundation.

22.2.6 Portabilidade

Um sistema operacional é **portável** se puder ser movido de uma arquitetura de hardware para outra com poucas mudanças. O Windows XP foi projetado para ser portável. Como acontece no sistema operacional UNIX, o Windows XP é escrito em C e C++. A maior parte do código dependente do processador é isolada em uma biblioteca de vínculo dinâmico (Dynamic Link Library - DLL) denominada **camada de abstração do hardware (Hardware-Abstraction Layer - HAL)**. Uma DLL é um arquivo mapeado no espaço de endereços de um processo, de modo que quaisquer funções na DLL pareçam fazer parte do processo. As camadas superiores do kernel do Windows XP dependem das interfaces da HAL em vez do hardware básico, aprimorando a portabilidade do Windows XP. A HAL manipula o hardware diretamente, isolando o restante do Windows XP das diferenças de hardware entre as plataformas em que é executado.

Embora, por motivos de mercado, o Windows 2000 só tenha vindo em plataformas compatíveis com a Intel IA32, ele também foi testado em plataformas IA32 e Alpha, da DEC, até o seu lançamento, para garantir a portabilidade. O Windows XP funciona em processadores compatíveis com IA32 e IA64. A Microsoft reconhece a importância do desenvolvimento e teste em múltiplas plataformas, visto que, na prática, manter a portabilidade é uma questão de *usar ou largar*.

22.2.7 Suporte internacional

O Windows XP também foi projetado para uso **internacional e multilíngue**. Ele fornece suporte para diferentes locais por meio da API de **suporte à linguagem nacional (National-Language-Support - NLS)**. A API NLS fornece rotinas especializadas para formatar datas, horas e moedas de acordo com costumes nacionais. As comparações de strings são especializadas, para levar em conta conjuntos de caracteres variados. O UNICODE é o código de caracteres nativo do Windows XP. O Windows XP aceita caracteres ANSI convertendo-os para caracteres UNICODE antes de manipulá-los (conversão de 8 bits para 16 bits). As strings de texto do sistema são mantidas em arquivos de recursos que podem ser trocados para adaptar o sistema a diferentes idiomas. Vários locais podem ser usados simultaneamente, o que é importante para indivíduos e empresas multilíngues.

22.3 Componentes do sistema

A arquitetura do Windows XP é um sistema de módulos em camadas, como mostra a Figura 22.1. As principais camadas são a HAL, o kernel e o executivo, todas executando no modo protegido, e uma coleção de subsistemas e serviços que executam no modo usuário. Os subsistemas do modo usuário estão em duas categorias. Os subsistemas de ambiente simulam diferentes sistemas operacionais; os **subsistemas de proteção** fornecem funções de segurança. Uma das principais vantagens desse tipo de arquitetura é que as interações entre os módulos são mantidas simples. O restante desta seção descreve essas camadas e subsistemas.

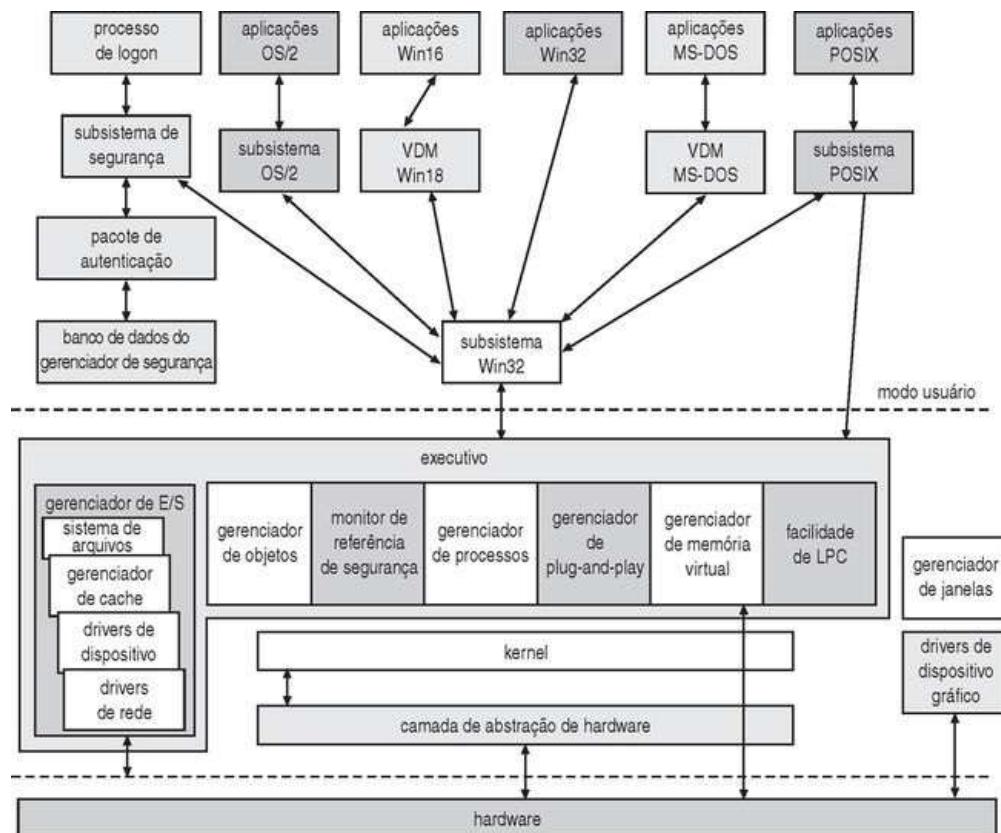


FIGURA 22.1 Diagrama em blocos do Windows XP.

22.3.1 Camada de abstração do hardware

A HAL é a camada de software que esconde as diferenças de hardware dos níveis superiores do sistema operacional, para ajudar a tornar o Windows XP portável. A HAL exporta uma interface de máquina virtual usada pelo despachante do kernel, pelo executivo e pelos drivers de dispositivo. Uma vantagem dessa técnica é que somente uma única versão de cada driver de dispositivo é necessária - ela é executada em todas as plataformas de hardware sem portar o código do driver. A HAL também fornece suporte para o multiprocessamento simétrico. Os drivers de dispositivos mapeiam dispositivos e os acessam diretamente, mas os detalhes administrativos do mapeamento de memória, configuração de barramentos de E/S, configuração de DMA e tratamento de facilidades específicas da placa-mãe são fornecidos pelas interfaces da HAL.

22.3.2 Kernel

O kernel do Windows XP fornece quatro responsabilidades principais: escalonamento de threads, tratamento de interrupção e exceção, sincronismo de baixo nível do processador e recuperação após uma falta de energia. O kernel é orientado a objeto. Um *tipo de objeto* no Windows 2000 é um tipo de dados definido pelo sistema que possui um conjunto de atributos (valores de dados) e um conjunto de métodos (por exemplo, funções ou operações). Um *objeto* é uma instância de um tipo de objeto. O kernel realiza seu trabalho usando um conjunto de objetos do kernel cujos atributos armazenam os dados do kernel e cujos métodos realizam as atividades do kernel.

22.3.2.1 Despachante do kernel

O despachante (dispatcher) do kernel fornece o alicerce para o executivo e os subsistemas. A maior parte do despachante nunca é paginada para fora da memória, e sua execução nunca é preemptada. Suas principais responsabilidades são escalonamento de threads, implementação de primitivas de sincronismo, gerenciador de temporizador, interrupções de software (chamadas de procedimento assíncronas e adiadas) e despacho de exceção.

22.3.2.2 Threads e escalonamento

Como muitos sistemas operacionais modernos, o Windows XP utiliza processos e threads para o código executável. Cada processo possui uma ou mais threads, e cada thread possui seu próprio estado de escalonamento, incluindo prioridade real, afinidade do processador e informações de uso da CPU.

Os seis estados de thread possíveis são pronto (ready), standby, executando (running), esperando (waiting), transição (transition) e terminado (terminated). **Pronto** indica esperando para executar. A thread pronta com prioridade mais alta é movida para o estado **standby**, que significa que é a próxima thread a ser executada. Em um sistema multiprocessado, cada processo mantém uma thread no estado standby. Uma thread está **executando** quando está sendo executada em um processador. Ela é executada até ser preemptada por uma thread de prioridade mais alta, terminar sua execução, seu tempo de execução alocado (**quantum**) terminar ou ser bloqueada sobre um objeto despachante, como um evento sinalizando o término da E/S. Uma thread está no estado **esperando** quando estiver esperando até um objeto despachante sinalizar. Uma nova thread está no estado de **transição** enquanto espera pelos recursos necessários para execução. Uma thread entra no estado **terminado** quando termina sua execução.

O despachante usa um esquema de prioridade de 32 níveis para determinar a ordem de execução da thread. As prioridades são divididas em duas classes: classe variável e classe de tempo real. A classe variável contém threads com prioridades de 0 a 15, e a classe de tempo real contém threads com prioridades variando de 16 a 31. O despachante usa uma fila para cada prioridade de escalonamento e atravessa o conjunto de filas da prioridade mais alta à mais baixa, até encontrar uma fila que esteja pronta para executar. Se uma thread tiver uma afinidade de processador em particular, mas esse processador não estiver disponível, o despachante segue em frente e continua procurando uma thread pronta que queira executar no processador disponível. Se nenhuma thread pronta for encontrada, o despachante executa uma thread especial, chamada thread idle.

Quando o quantum de tempo de uma thread se esgotar, a interrupção do relógio enfileira uma chamada de procedimento adiada (deferred procedure call - DPC) de fim de quantum para o processador, a fim de reescalonar o processador. Se a thread preemptada estiver na classe de prioridade variável, sua prioridade será reduzida. A prioridade nunca é reduzida para menos do que a prioridade de base. A redução da prioridade da thread costuma limitar o consumo de CPU das threads voltadas para computação. Quando uma thread de prioridade variável é liberada de uma operação de espera, o despachante aumenta sua prioridade. O valor do aumento depende do dispositivo pelo qual a thread estava esperando; por exemplo, uma thread que estava esperando pela E/S do teclado obteria um grande aumento de prioridade, enquanto uma thread esperando por uma operação de disco receberia um aumento moderado. Essa estratégia costuma dar bons tempos de resposta às threads interativas que utilizam mouse e janelas. Ela também permite que as threads I/O-bound mantenham os dispositivos de E/S ocupados, enquanto permite que as threads CPU-bound usem ciclos de CPU de reserva em segundo plano. Essa estratégia é usada por vários sistemas operacionais de tempo compartilhado, incluindo UNIX. Além disso, a thread associada à janela GUI ativa do usuário recebe um aumento de prioridade para melhorar seu tempo de resposta.

O escalonamento ocorre quando uma thread entra no estado pronto ou esperando, quando uma thread termina ou quando uma aplicação muda a prioridade ou a afinidade do processador de uma thread. Se uma thread de tempo real com maior prioridade ficar pronta enquanto uma thread de menor prioridade estiver sendo executada, a thread de menor prioridade é preemptada. Essa preempção dá a uma thread de tempo real o acesso preferencial à CPU quando a thread precisa de tal acesso. O Windows XP não é um sistema operacional de tempo real rígido, pois não garante que uma thread de tempo real começará a executar dentro de um limite de tempo em particular.

22.3.2.3 Implementação das primitivas de sincronismo

As estruturas de dados fundamentais do sistema operacional são gerenciadas como objetos usando facilidades comuns para alocação, contagem de referência e segurança. **Objetos despachantes** controlam o despacho e o sincronismo no sistema.

Alguns exemplos desses objetos são os seguintes:

- O **objeto de evento** é usado para registrar uma ocorrência de evento e sincronizá-la com alguma ação. Os eventos de notificação sinalizam todas as threads esperando, e os eventos de sincronismo sinalizam uma única thread esperando.
- O **mutante** fornece exclusão mútua no modo kernel ou usuário com a noção de propriedade.
- O **mutex**, disponível apenas no modo kernel, fornece exclusão mútua livre de deadlock.

- Um **objeto semáforo** atua como contador ou porta para controlar o número de threads que acessam um recurso.
- O **objeto thread** é a entidade escalonada pelo despachante do kernel e associada a um **objeto de processo** que encapsula um espaço de endereço virtual.
- Os **objetos de temporizador** são usados para registrar a hora e sinalizar tempos-limite quando as operações levarem muito tempo e precisarem ser interrompidas ou uma atividade periódica tiver de ser escalonada.

Muitos dos objetos despachantes são acessados no modo usuário por meio de uma operação de abertura que retorna um descritor. O código no modo usuário consulta ou espera que os descritores se sincronizem com outras threads, bem como o sistema operacional (ver [Seção 22.7.1](#)).

22.3.2.4 Interrupção de software: chamadas de procedimento assíncronas e chamadas de procedimento adiadas

O despachante implementa dois tipos de interrupções de software: chamadas de procedimento assíncronas e chamadas de procedimento adiadas. Uma chamada de procedimento assíncrona (Asynchronous Procedure Call - APC) interrompe uma thread em execução e chama um procedimento. As APCs são usadas para iniciar a execução de uma nova thread, terminar processos e fornecer notificação de que uma E/S assíncrona foi concluída. As APCs são enfileiradas para threads específicas e permitem ao sistema executar o código do sistema e do usuário dentro do contexto de um processo.

As chamadas de procedimento adiadas (Deferred Procedure Calls - DPCs) são usadas para postergar o processamento da interrupção. Depois de tratar de todos os processos de interrupção de dispositivo em bloco, a rotina de serviço de interrupção (Interrupt Service Routine - ISR) escalona o processamento restante enfileirando uma DPC. O despachante escalona as interrupções de software em uma prioridade mais baixa do que as interrupções de dispositivo, de modo que as DPCs não bloqueiam outras ISRs. Além de adiar o processamento de interrupção do dispositivo, o despachante utiliza DPCs para processar os términos de prazo de temporizador e preemir a thread no final do quantum de escalonamento.

A execução de DPCs impede que as threads sejam escalonadas no processador atual e também evita que as APCs sinalizem o término da E/S. Isso é feito para que o término das rotinas de DPC não utilize um tempo estendido. Como alternativa, o despachante mantém um banco de threads trabalhadoras. ISRs e DPCs enfileiram itens de trabalho para as threads trabalhadoras. As rotinas de DPC são restritas de modo que não possam causar falhas de página (paginadas para fora da memória), chamar serviços do sistema ou tomar qualquer outra ação que possivelmente possa resultar em uma tentativa de bloquear a execução sobre um objeto despachante. Ao contrário das APCs, as rotinas de DPC não fazem suposições sobre qual contexto de processo o processador está executando.

22.3.2.5 Exceções e interrupções

O despachante do kernel também fornece tratamento de interceptação (trap handler) para exceções e interrupções geradas por hardware ou software. O Windows XP define várias exceções independentes da arquitetura, incluindo:

- Violão de acesso à memória.
- Overflow de inteiros.
- Overflow e underflow de ponto flutuante.
- Divisão de inteiro por zero.
- Divisão de ponto flutuante por zero.
- Instrução ilegal.
- Falta de alinhamento de dados.
- Instrução privilegiada.
- Erro de leitura de página.
- Violão de acesso.
- Cota de arquivo de página ultrapassada.
- Ponto de interrupção do depurador.
- Execução passo a passo do depurador.

Os tratadores de interceptação tratam de exceções simples. O tratamento elaborado de exceções é realizado pelo despachante de exceções do kernel. O **despachante de exceções** cria um registro de exceção contendo o motivo para a exceção e encontra um tratador de exceção para lidar com ela.

Quando ocorre uma exceção no modo kernel, o despachante da exceção chama uma rotina para localizar o tratador da exceção. Se nenhum tratador for encontrado, um erro fatal do sistema ocorrerá e o usuário receberá a infame “tela azul da morte”, que significa falha do sistema.

O tratamento de exceção é mais complexo para os processos no modo usuário, pois um subsistema de ambiente (como o sistema POSIX) configura uma porta de depurador e uma porta de exceção para cada processo criado. Se uma porta de depurador for registrada, o tratador de exceção envia a exceção para a porta. Se a porta de depurador não for encontrada ou não tratar dessa exceção, o

despachante tenta encontrar um tratador de exceção apropriado. Se nenhum tratador for encontrado, o depurador é chamado novamente para apanhar o erro para depuração. Se nenhum depurador estiver sendo executado, uma mensagem é enviada à porta de exceção do processo para dar ao subsistema de ambiente uma chance de traduzir a exceção. Por exemplo, o ambiente POSIX traduz as mensagens de exceção do Windows XP para sinais POSIX antes de enviá-los à thread que causou a exceção. Finalmente, se nada mais funcionar, o kernel termina o processo que contém a thread que causou a exceção.

O despachante de interrupção no kernel trata das interrupções chamando uma rotina de serviço de interrupção (ISR) fornecida por um driver de dispositivo ou uma rotina de tratador de interrupção (trap) do kernel. A interrupção é representada por um objeto de interrupção que contém todas as informações necessárias para tratar da interrupção. O uso de um objeto de interrupção facilita a associação entre rotinas de serviço de interrupção e uma interrupção, sem ter de acessar o hardware de interrupção diretamente.

Diferentes arquiteturas de processador, como Intel ou DEC Alpha, possuem diferentes tipos e quantidades de interrupções. Para a portabilidade, o despachante de interrupção mapeia as interrupções de hardware em um conjunto-padrão. As interrupções são priorizadas e atendidas em ordem de prioridade. Existem 32 níveis de requisição de interrupção (**Interrupt Request Levels - IRQLs**) no Windows XP. Oito são reservadas para uso do kernel; os 24 restantes representam interrupções de hardware por meio da HAL (embora a maioria dos sistemas IA32 utilize apenas 16). As interrupções do Windows XP são definidas na [Figura 22.2](#).

níveis de interrupção	tipos de interrupções
31	verificação de máquina ou erro de barramento
30	falta de energia
29	notificação entre processadores (solicita que outro processador atue, por exemplo, para despachar um processo ou atualizar a TLB)
28	relógio (usado para rastrear o tempo)
27	perfil
3-26	interrupções de hardware com IRQ tradicionais do PC
2	despacho e chamada de procedimento adiada (DPC) (kernel)
1	chamada de procedimento assíncrona (APC)
0	passiva

FIGURA 22.2 Níveis de interrupção no Windows XP.

O kernel utiliza uma **tabela de despacho de interrupção** para vincular cada nível de interrupção a uma rotina de serviço. Em um computador multiprocessado, o Windows XP mantém uma tabela de despacho de interrupção separada para cada processador, e a IRQL de cada processador pode ser definida de maneira independente para mascarar as interrupções. Todas as interrupções ocorridas em um nível igual ou menor do que a IRQL de um processador são bloqueadas até a IRQL ser atendida por uma thread no nível do kernel ou por uma ISR retornando do processamento da interrupção. O Windows XP aproveita essa propriedade e utiliza interrupções de software para distribuir APCs e DPCs, realizar funções do sistema como sincronismo de threads com término de E/S, iniciar despachos de thread e tratar de temporizadores.

22.3.3 Executivo

O executivo do Windows XP fornece um conjunto de serviços utilizados por todos os subsistemas de ambiente. Os serviços são agrupados da seguinte maneira: gerenciador de objetos, gerenciador de memória virtual, gerenciador de processos, facilidade de chamada de procedimento local, gerenciador de E/S, monitor de referência de segurança, gerenciadores de plug-and-play e energia, registry e inicialização.

22.3.3.1 Gerenciador de objetos

O Windows XP utiliza um conjunto genérico de interfaces para gerenciar as entidades do modo kernel que são manipuladas pelos programas no modo usuário. O Windows XP chama essas entidades de *objetos*, e o componente do executivo que as manipula é o **gerenciador de objetos**. Cada processo possui uma tabela de objetos contendo entradas que rastreiam os objetos usados pelo processo. O código no modo usuário acessa esses objetos usando um valor opaco, chamado

descriptor, que é retornado por muitas APIs. Os descritores de objeto também podem ser criados duplicando-se um descriptor existente, seja pelo mesmo processo ou por um processo diferente. Alguns exemplos de objetos são semáforos, mutex, eventos, processos e threads. São todos *objetos despachantes*. As threads podem ser bloqueadas no despachante do kernel esperando que algum desses objetos seja sinalizado. As APIs de processo, thread e memória virtual utilizam descritores de processo e thread para identificar o processo ou a thread sobre o qual operarão. Outros exemplos de objetos incluem arquivos, seções, portas e diversos objetos internos de E/S. Os objetos de arquivo são usados para manter o estado aberto de arquivos e dispositivos. As seções são usadas para mapear arquivos. Arquivos abertos são descritos em termos de objetos de arquivo. As extremidades de comunicação locais são implementadas como objetos de porta.

O gerenciador de objetos mantém o espaço de nomes interno do Windows XP. Diferentemente do UNIX, que define a raiz no espaço de nomes do sistema no sistema de arquivos, o Windows XP utiliza um espaço de nomes abstrato e conecta os sistemas de arquivos como dispositivos.

Ele fornece interfaces para definir tipos de objeto e instâncias de objeto, traduzir nomes para objetos, manter o espaço de nomes abstrato (por meio de diretórios internos e links simbólicos) e gerenciar a criação e a exclusão de objeto. Os objetos são gerenciados por meio de contadores de referência no código do modo protegido e descritores no modo usuário. Contudo, alguns componentes do modo kernel utilizam as mesmas APIs que o código do modo usuário e, sendo assim, utilizam descritores para manipular objetos. Se um descriptor precisar existir além do tempo de vida do processo atual, ele é marcado como um descriptor do kernel e armazenado na tabela de objetos para o processo do sistema. O espaço de nomes abstrato não persiste entre os boots do sistema, mas é montado a partir das informações de configuração armazenadas no registry do sistema, descoberta de dispositivo plug-and-play e criação de objetos por componentes do sistema.

O executivo do Windows XP permite que qualquer objeto receba um **nome**. Um processo pode criar um objeto nomeado, enquanto um segundo processo abre um descriptor para o objeto e o compartilha com o primeiro processo. Os processos também podem compartilhar objetos duplicando descritores entre os processos, quando os objetos não precisarão ser nomeados.

Um nome pode ser permanente ou temporário. Um nome permanente representa uma entidade, como uma unidade de disco, que permanece mesmo que nenhum processo a esteja acessando. Existe um nome temporário somente enquanto um processo mantém um descriptor para o objeto.

Os nomes de objeto são estruturados como nomes de caminho no MS-DOS e no UNIX. Os diretórios do espaço de nomes são representados por um **objeto de diretório** que contém os nomes de todos os objetos no diretório. O espaço de nomes do objeto é estendido pelo acréscimo de objetos de dispositivo representando volumes que contêm sistemas de arquivo.

Os objetos são manipulados por um conjunto de funções virtuais com implementações fornecidas para cada tipo de objeto: `create()`, `open()`, `close()`, `delete()`, `query_name()`, `parse()` e `security()`. As três últimas funções precisam de alguma explicação:

- `query_name()` é chamado quando uma thread tiver uma referência a um objeto, mas deseja saber o nome do objeto.
- `parse()` é usado pelo gerenciador de objetos para procurar um objeto, dado o nome desse objeto.
- `security()` é chamado para fazer verificações de segurança sobre todas as operações com objeto, como quando um processo abre ou fecha um objeto, faz mudanças no descritor de segurança ou duplica um descritor para um objeto.

O procedimento `parse` é usado para estender o espaço de nomes abstrato para incluir arquivos. A tradução de um nome de caminho para um objeto de arquivo começa na raiz do espaço de nomes abstrato. Os componentes do nome de caminho são separados por barras invertidas ('\\') em vez das barras ('/') utilizadas no UNIX. Cada componente é pesquisado no diretório de análise atual do espaço de nomes. Os nós internos dentro do espaço de nomes são diretórios ou links simbólicos. Se um objeto folha for encontrado e não houver componentes de nome de caminho restantes, o objeto folha será retornado. Caso contrário, o procedimento `parse` do objeto folha será chamado com o nome de caminho restante.

Os procedimentos `parse` são usados apenas com um pequeno número de objetos pertencentes à GUI do Windows, o gerenciador de configuração (registry) e - principalmente - objetos de dispositivo representando sistemas de arquivos.

O procedimento `parse` para o tipo de objeto de dispositivo aloca um objeto de arquivo e inicia uma operação de E/S `open` ou `create` sobre o sistema de arquivos. Se tiver sucesso, os campos do objeto de arquivo são preenchidos para descrever o arquivo.

Resumindo, o nome de caminho até um arquivo é usado para atravessar o espaço de nomes do gerenciador de objetos, traduzindo o nome de caminho absoluto original para um par (objeto de dispositivo, nome de caminho relativo). Esse par, então, é passado ao sistema de arquivos por meio do gerenciador de E/S, que preenche o objeto de arquivo. O objeto de arquivo propriamente dito não possui nome, mas é referenciado por um descritor.

Os sistemas de arquivos do UNIX possuem **links simbólicos** que permitem vários apelidos ou aliases para o mesmo arquivo. O **objeto de link simbólico** implementado pelo gerenciador de objetos do Windows XP é usado dentro do espaço de nomes abstrato, e não para fornecer arquivos com apelidos. Mesmo assim, os links simbólicos ainda são bastante úteis. Eles são usados para

organizar o espaço de nomes, semelhante à organização do diretório /devices no UNIX. Eles também são usados para mapear letras de unidade-padrão do MS-DOS a nomes de unidade. As letras de unidade são links simbólicos que podem ser remapeados conforme a conveniência do usuário ou administrador.

As letras de unidade são o único local em que o espaço de nomes abstrato no Windows XP não é global. Cada usuário conectado possui seu próprio conjunto de letras de unidade, de modo que os usuários podem evitar interferir uns com os outros. Por outro lado, as sessões do servidor de terminal compartilham todos os processos dentro de uma sessão. BaseNamedObjects contém os objetos nomeados criados pela maioria das aplicações.

Embora o espaço de nomes não seja diretamente visível por uma rede, o método parse() do gerenciador de objetos é usado para ajudar a acessar um objeto nomeado em outro sistema. Quando um processo tenta abrir um objeto que reside em um computador remoto, o gerenciador de objetos chama o método parse para o objeto de dispositivo correspondente a um redirecionador de rede. Isso resulta em uma operação de E/S que acessa o arquivo pela rede.

Como dissemos, os objetos são instâncias de um **tipo de objeto**. O tipo de objeto especifica como as instâncias devem ser alocadas, as definições dos campos de dados e a implementação do conjunto-padrão de funções virtuais usadas para todos os objetos. Essas funções implementam operações como mapear nomes aos objetos, fechar e excluir e aplicar segurança.

O gerenciador de objetos registra duas contagens para cada objeto. O contador do ponteiro é o número de referências distintas feitas a um objeto. O código no modo protegido que se refere aos objetos precisa manter uma referência sobre o objeto para garantir que o objeto não seja excluído enquanto estiver em uso. O contador de descritores é o número de entradas na tabela de descritores que se referem a um objeto. Cada descritor também é refletido na contagem de referência.

Quando um descritor para um objeto é fechado, a rotina close do objeto é chamada. No caso de objetos de arquivo, essa chamada faz o gerenciador de E/S realizar uma operação de limpeza no fechamento do último descritor. A operação de limpeza diz ao sistema de arquivos que o arquivo não é mais acessado pelo modo usuário, de modo que as restrições de compartilhamento, locks de intervalo e outros estados específicos à rotina open correspondente podem ser removidos.

Cada fechamento de descritor remove uma referência do contador de ponteiros, mas os componentes internos do sistema podem reter referências adicionais. Quando a referência final é removida, o procedimento delete do objeto é chamado. Novamente usando objetos de arquivo como exemplo, o procedimento delete faz o gerenciador de E/S enviar ao sistema de arquivos uma operação close sobre o objeto de arquivo. Isso faz o sistema de arquivos desalocar quaisquer estruturas de dados internas alocadas para o objeto de arquivo.

Após o término do procedimento delete para um objeto temporário, o objeto é excluído da memória. O gerenciador de objetos pode tornar um objeto permanente (pelo menos, com relação ao boot atual do sistema) tomando uma referência extra contra o objeto. Esses objetos permanentes não são excluídos nem mesmo quando a última referência fora do gerenciador de objetos é removida. Quando um objeto permanente se torna temporário novamente, o gerenciador de objetos remove a referência extra. Se essa foi a última referência, o objeto é excluído. Os objetos permanentes são raros, usados principalmente para dispositivos, mapeamentos de letra de unidade e os objetos de diretório e link simbólico.

A tarefa do gerenciador de objetos é supervisionar o uso de todos os objetos gerenciados. Quando uma thread deseja usar um objeto, ela chama o método open() do gerenciador de objetos para obter uma referência ao objeto. Se o objeto estiver sendo aberto por uma API no modo usuário, a referência é inserida na tabela de objetos do processo e um descritor é retornado.

Um processo apanha um descritor criando um objeto, abrindo um objeto existente, recebendo um descritor duplicado de outro processo ou herdando um descritor de um **processo pai**, semelhante ao modo como um processo do UNIX apanha um descritor de arquivo. Esses descritores são todos armazenados na **tabela de objetos** do processo. Uma entrada na tabela de objetos contém os direitos de acesso do objeto e indica se o descritor deve ser herdado por **processos filhos**. Quando um processo termina, o Windows XP automaticamente fecha todos os descritores abertos do processo.

Descritores são uma interface padronizada para todos os tipos de objetos. Assim como um descritor de arquivos no UNIX, um descritor de objeto é um identificador exclusivo para um processo que confere a capacidade de acessar e manipular um recurso do sistema. Os descritores podem ser duplicados dentro de um processo ou então entre os processos. O último caso é usado quando se criam processos filhos ou quando contextos de execução fora do processo são implementados.

Como o gerenciador de objetos é a única entidade que gera descritores de objetos, esse é o local natural para verificar a segurança. O gerenciador de objetos verifica se um processo tem o direito de acessar um objeto quando o processo tentar abrir o objeto. O gerenciador de objetos também impõe cotas, como a quantidade máxima de memória que um processo pode usar, cobrando de um processo a memória ocupada por todos os objetos referenciados e recusando-se a alocar mais memória quando as despesas acumuladas ultrapassarem a cota do processo.

Quando o processo de login autentica um usuário, um token de acesso é anexado ao processo do

usuário. O token de acesso contém informações como a id de segurança, ids de grupo, privilégios, grupo principal e lista de controle de acesso-padrão. Os serviços e os objetos que um usuário pode acessar são determinados por esses atributos.

O token que controla o acesso está associado à thread que faz o acesso. Normalmente, o token da thread é perdido e o token do processo é utilizado como padrão, mas os serviços precisam executar código em favor de seu cliente. O Windows XP permite que as threads personifiquem um cliente temporariamente usando o token de um cliente. Assim, o token da thread não é necessariamente o mesmo token do processo.

No Windows XP, cada objeto é protegido por uma lista de controle de acesso que contém as ids de segurança e os direitos de acesso concedidos. Quando uma thread tenta acessar um objeto, o sistema compara a id de segurança no token de acesso da thread com a lista de controle de acesso do objeto para determinar se o acesso deverá ser permitido. A verificação é realizada apenas quando um objeto é aberto, de modo que não é possível negar o acesso depois de haver a abertura. Os componentes do sistema operacional executando no modo kernel contornam a verificação de acesso, pois o código no modo kernel é considerado confiável. Portanto, o código no modo kernel precisa evitar as vulnerabilidades na segurança, como deixar as verificações desativadas enquanto cria um descritor acessível ao modo usuário em um processo não confiável.

Em geral, o criador do objeto determina a lista de controle de acesso para o objeto. Se nenhuma lista for fornecida explicitamente, pode-se definir uma como default pela rotina open do tipo de objeto ou uma lista default pode ser obtida a partir do objeto token de acesso do usuário.

O token de acesso possui um campo que controla a auditoria dos acessos ao objeto. As operações auditadas são registradas no log de segurança do sistema com uma identificação do usuário. Um administrador monitora esse log para descobrir tentativas de penetrar no sistema ou acessar objetos protegidos.

22.3.3.2 Gerenciador de memória virtual

O componente do executivo que gerencia o espaço de endereços virtuais, a alocação da memória física e a paginação é o **gerenciador de memória virtual (Virtual Memory - VM)**. O projeto do gerenciador VM considera que o hardware subjacente admite o mapeamento entre virtual e físico, um mecanismo de paginação e coerência de cache transparente em sistemas multiprocessados e permite várias entradas de tabela de página para mapear o mesmo quadro de página física. O gerenciador VM no Windows XP utiliza um esquema de gerenciador baseado em página com um tamanho de página de 4 KB em processadores compatíveis com IA32 e 8 KB no IA64. As páginas de dados alocadas a um processo que não estejam na memória física são armazenadas nos **arquivos de paginação** no disco ou mapeadas diretamente para um arquivo normal em um sistema de arquivos local ou remoto. As páginas também podem ser marcadas como “zerar por demanda”, o que preenche a página com zeros antes de ser alocada, apagando, assim, o conteúdo anterior.

Em processadores IA32, cada processo possui um espaço de endereços virtual de 4 GB. Os 2 GB superiores são praticamente idênticos para todos os processos e são usados pelo Windows XP no modo kernel para acessar o código e as estruturas de dados do sistema operacional. As principais áreas da região do modo kernel que não são idênticas para todos os processos são o **automapa da tabela de página**, o **hiperespaço** e o **espaço de sessão**. O hardware referencia as tabelas de página de um processo usando os números de quadro de página físico. O gerenciador VM mapeia as tabelas de página em uma única região de 4 MB no espaço de endereços do processo, de modo que são acessadas por meio de endereços virtuais. O hiperespaço mapeia as informações do conjunto de trabalho (working set) do processo atual no espaço de endereços do modo kernel.

O espaço de sessão é usado para compartilhar o Win32 e outros drivers específicos da sessão entre todos os processos na mesma sessão do servidor de terminal, em vez de todos os processos no sistema. Os 2 GB inferiores são específicos a cada processo e acessíveis por threads no modo usuário e no modo kernel. Algumas configurações do Windows XP reservam apenas 1 GB para uso do sistema operacional, permitindo que um processo use 3 GB de espaço de endereços. A execução do sistema no modo de 3 GB reduz drasticamente a quantidade de caching de dados no kernel. No entanto, para aplicações grandes, que gerenciam sua própria E/S, como bancos de dados SQL, a vantagem de um maior espaço de endereços do modo usuário pode compensar a perda do caching.

O gerenciador VM do Windows XP utiliza um processo em duas etapas para alocar a memória do usuário. A primeira etapa *reserva* uma parte do espaço de endereços virtual do processo. A segunda etapa *promete* a alocação atribuindo espaço de memória virtual (memória física ou espaço nos arquivos de paginação). O Windows XP limita a quantidade de espaço de memória virtual que um processo consome impondo uma cota sobre a memória comprometida. Um processo descompromete a memória que não está mais em uso para liberar a memória virtual para ser usada por outros processos. As APIs usadas para reservar endereços virtuais e comprometer a memória virtual apanham um descritor sobre um objeto de processo como parâmetro. Isso permite que um processo controle a memória virtual de outro. Os subsistemas de ambiente gerenciam a memória de seus processos clientes dessa maneira.

Para melhor desempenho, o gerenciador VM permite que um processo privilegiado efetue locks de páginas selecionadas na memória física, garantindo, assim, que não serão enviadas para o arquivo

de paginação. Os processos também alocam memória física bruta e depois mapeiam regiões para o seu espaço de endereços virtual. Os processadores IA32 com o recurso de extensão de endereço físico (**Physical Address Extension - PAE**) podem ter até 64 GB de memória física em um sistema. Essa memória não pode ser toda mapeada no espaço de endereços de um processo ao mesmo tempo, mas o Windows XP a torna disponível usando APIs de extensão de janelas de endereço (Address Windowing Extension - AWE), que alocam memória física e depois associam regiões de endereços virtuais no espaço de endereços do processo a parte da memória física. A facilidade AWE é usada principalmente por aplicações muito grandes, como o banco de dados SQL.

O Windows XP implementa a memória compartilhada definindo um **objeto de seção**. Depois de obter um descriptor para um objeto de seção, um processo associa a parte da memória de que precisa ao seu espaço de endereços. Essa parte é chamada de **visão**. Um processo redefine sua visão de um objeto para obter acesso ao objeto inteiro, uma região de cada vez.

Um processo pode controlar o uso de um objeto de seção da memória compartilhada de muitas maneiras. O tamanho máximo de uma seção pode ser limitado. A seção pode ter o apoio do espaço em disco, seja no arquivo de paginação do sistema ou em um arquivo normal (um **arquivo mapeado na memória**). Uma seção pode ser *baseada*, significando que aparece no mesmo endereço virtual para todos os processos tentando acessá-la. Finalmente, a proteção de memória das páginas na seção pode ser definida como somente leitura, leitura-escrita, leitura-escrita-execução, somente execução, nenhum acesso ou cópia na escrita. Essas duas últimas definições de proteção precisam de alguma explicação:

- Uma página com nenhum acesso levanta uma exceção se for acessada; a exceção é usada, por exemplo, para verificar se um programa com defeito ultrapassa o final de um array. Tanto o alocador de memória do modo usuário quanto o alocador especial do kernel, usados pelo verificador de dispositivos, podem ser configurados para mapear cada alocação ao final de uma página, seguido por uma página com nenhum acesso, a fim de detectar overruns de buffer.
- O mecanismo *copy-on-write* (cópia quando houver uma escrita) aumenta o uso eficiente da memória física pelo gerenciador VM. Quando dois processos desejam cópias independentes de um objeto, o gerenciador VM coloca uma única cópia compartilhada na memória virtual e ativa a propriedade de cópia na escrita para essa região da memória. Se um dos processos tentar modificar dados em uma página de cópia na escrita, o gerenciador VM faz uma cópia privada da página para o processo.

A tradução de endereços virtuais no Windows XP utiliza uma tabela de página multinível. Para processadores IA32, sem as Physical Address Extensions ativadas, cada processo possui um **diretório de página** que contém 1.024 **entradas de diretório de página** (**Page-Directory Entries - PDEs**) com 4 bytes de tamanho. Cada PDE aponta para uma **tabela de página** que contém 1.024 **entradas de tabela de página** (**Page-Table Entries - PTEs**) com 4 bytes de tamanho. Cada PTE aponta para um **quadro de página** de 4 KB na memória física. O tamanho total de todas as tabelas de página para um processo é de 4 MB, de modo que o gerenciador VM pagina tabelas individuais para o disco quando for necessário. Ver, na [Figura 22.3](#), um diagrama dessa estrutura.

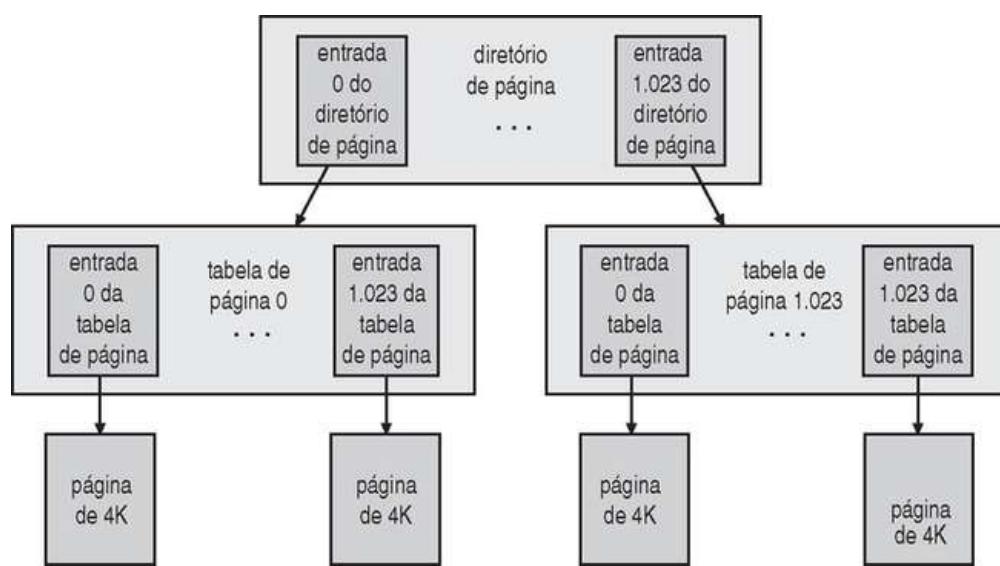


FIGURA 22.3 Layout da tabela de página.

O diretório de página e as tabelas de página são referenciados pelo hardware por meio de endereços físicos. Para melhorar o desempenho, o gerenciador VM automapeia o diretório de página e as tabelas de página em uma região de 4 MB de endereços virtuais. O automapa permite ao

gerenciador VM traduzir um endereço virtual para a PDE ou PTE correspondente sem acessos adicionais à memória. Quando um contexto de processo é alterado, uma única entrada de diretório de página precisa ser alterada para mapear as tabelas de página do novo processo. Por diversas razões, o hardware exige que cada diretório de página ou tabela de página ocupe uma única página. Assim, a quantidade de PDEs ou PTEs que cabem em uma página determina como os endereços virtuais são traduzidos.

A seguir, descrevemos como os endereços virtuais são traduzidos em endereços físicos nos processadores compatíveis com IA32 (sem PAE ativada). Um valor de 10 bits pode representar todos os valores de 0 a 1.023. Assim, um valor de 10 bits pode selecionar qualquer entrada no diretório de página ou em uma tabela de página. Essa propriedade é usada quando um ponteiro de endereço virtual é traduzido para um endereço de byte na memória física. Um endereço de memória virtual de 32 bits é dividido em três valores, como mostra a [Figura 22.4](#). Os primeiros 10 bits do endereço virtual são usados como um índice para o diretório de página. Esse endereço seleciona uma entrada do diretório de página (PDE), que contém o quadro de página físico de uma tabela de página. A unidade de gerência de memória (Memory Management Unit - MMU) utiliza os próximos 10 bits do endereço virtual para selecionar uma PTE a partir da tabela de página. A PTE especifica um quadro de página na memória física. Os 12 bits restantes do endereço virtual são o deslocamento de um byte específico no quadro de página. A MMU cria um ponteiro para o byte específico na memória física, concatenando os 20 bits da PTE com os 12 bits inferiores do endereço virtual. Assim, a PTE de 32 bits possui 12 bits para descrever o estado da página física. O hardware IA32 reserva 3 bits para uso pelo sistema operacional. O restante dos bits especifica se a página foi acessada ou escrita, os atributos de caching, o modo de acesso, se a página é global e se a PTE é válida.

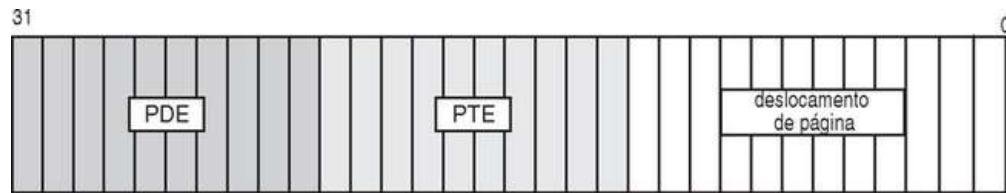


FIGURA 22.4 Tradução de endereços virtuais para físico no hardware IA32.

Processadores IA32 executando com PAE permitem o uso de PDEs e PTEs de 64 bits a fim de representar o campo maior de número de quadro de página de 24 bits. Assim, os diretórios de página de segundo nível e as tabelas de página contêm apenas 512 PDEs e PTEs, respectivamente. Fornecer 4 GB de espaço de endereços virtuais exige um nível extra de diretório de página, contendo quatro PDEs. A tradução de um endereço virtual de 32 bits utiliza 2 bits para o índice de diretório de nível superior e 9 bits para cada um dos diretórios de página de segundo nível e as tabelas de página.

Para evitar o custo adicional de traduzir cada endereço virtual pesquisando a PDE e a PTE, os processadores utilizam um **Translation Look-aside Buffer (TLB)**, que contém um cache de memória associativo para mapear páginas virtuais às PTEs. Ao contrário da arquitetura IA32, onde a TLB é mantida pela MMU de hardware, o IA64 invoca uma rotina de interceptação de software para fornecer traduções que faltam na TLB. Isso dá ao gerenciador VM flexibilidade na escolha das estruturas de dados para usar. No Windows XP, uma estrutura em árvore de três níveis é escolhida para mapear os endereços virtuais do modo usuário no IA64.

Em processadores IA64, o tamanho de página é de 8 KB, mas as PTEs ocupam 64 bits, de modo que uma página ainda contém apenas 1.024 (correspondentes a 10 bits) de PDEs ou PTEs. Portanto, com 10 bits de PDEs de alto nível, 10 bits de segundo nível, 10 bits de tabela de página e 13 bits de deslocamento de página, a parte do usuário do espaço de endereços virtual do processo para o Windows XP no IA64 é de 8 TB (correspondentes a 43 bits). A limitação de 8 TB na versão atual do Windows XP não tira proveito completo das capacidades do processador IA64, mas representa um meio-termo entre o número de referências de memória necessárias para tratar de falhas na TLB e o tamanho do espaço de endereços do modo usuário admitido.

Uma página física pode ter um dentre seis estados: válido (valid), livre (free), zerado (zeroed), standby, modificado (modified), defeituoso (bad) ou em transição (in transition).

- Uma página *válida* está em uso por um processo ativo.
- Uma página *livre* é uma página que não está referenciada em uma PTE.
- Uma página *zerada* é uma página livre que foi zerada e está pronta para uso imediato para satisfazer às falhas do tipo “zerar por demanda” (zero-on-demand).
- Uma página *modificada* é aquela que foi escrita por um processo e precisa ser enviada ao disco antes de ser alocada para outro processo.
- Páginas de *standby* são cópias das informações já armazenadas no disco. Elas podem ser páginas não modificadas, páginas modificadas já gravadas no disco ou páginas buscadas com antecedência para explorar a localidade.

- Uma página *defeituosa* é inutilizável porque foi detectado um erro de hardware.
- Finalmente, uma página *em transição* é aquela que está a caminho do disco para um quadro de página alocado na memória física.

Quando o bit de página válida em uma PTE é zero, o gerenciador VM define o formato dos outros bits. As páginas inválidas podem ter uma série de estados, representados pelos bits na PTE. As páginas do arquivo de paginação que nunca faltaram são marcadas como “zerar por demanda”. Os arquivos mapeados por meio dos objetos de seção codificam um ponteiro para esse objeto de seção. As páginas escritas no arquivo de página contêm informações suficientes para encontrar a página no disco, e assim por diante.

A estrutura real da PTE do arquivo de página aparece na [Figura 22.5](#). A PTE contém 5 bits para proteção de página, 20 bits para deslocamento do arquivo de página, 4 bits para selecionar o arquivo de paginação e 3 bits que descrevem o estado da página. Uma PTE de arquivo de página é marcada para ser um endereço virtual inválido para a MMU. Como o código executável e os arquivos mapeados na memória já possuem uma cópia no disco, eles não precisam de espaço em um arquivo de paginação. Se uma dessas páginas não estiver na memória física, a estrutura da PTE será a seguinte. O bit mais significativo é usado para especificar a proteção de página, os 28 bits seguintes são usados para indexar em uma estrutura de dados do sistema que indica um arquivo e deslocamento dentro do arquivo para essa página, e os 3 bits inferiores especificam o estado da página.

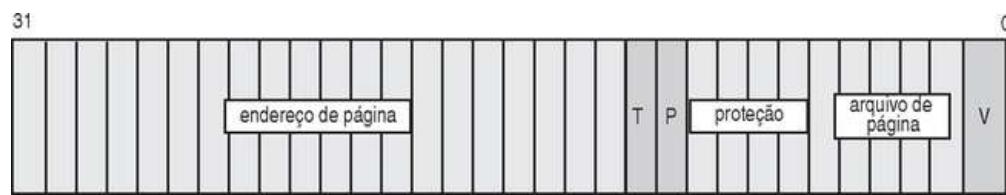


FIGURA 22.5 Entrada de tabela de página do arquivo de página. O bit válido é zero.

Endereços virtuais inválidos também podem estar em diversos estados temporários, que fazem parte dos algoritmos de paginação. Quando uma página é removida de um conjunto de trabalho do processo, ela é movida para a lista de páginas modificadas (a serem escritas no disco) ou diretamente para a lista de páginas em standby. Se for escrita para a lista de páginas em standby, a página é reivindicada sem leitura do disco, caso seja necessária novamente antes de ser movida para a lista de páginas livres. Quando possível, o gerenciador VM usará ciclos de CPU ociosos para zerar páginas na lista de páginas livres e movê-las para a lista de páginas zeradas. As páginas em transição receberam uma página física e estão esperando o término da E/S de paginação antes de a PTE ser marcada como válida.

O Windows XP utiliza objetos de seção para descrever páginas compartilháveis entre os processos. Cada processo possui seu próprio conjunto de tabelas de página virtuais, mas o objeto de seção também inclui um conjunto de tabelas de página contendo as PTEs mestras (ou protótipo). Quando uma PTE em uma tabela de página do processo é marcada como válida, ela aponta para o quadro da página física contendo a página, como precisa acontecer nos processadores IA32, onde a MMU do hardware lê as tabelas de página diretamente da memória. Contudo, quando uma página compartilhada se torna inválida, a PTE é editada para apontar para a PTE de protótipo associada ao objeto de seção.

As tabelas de página associadas a um objeto de seção são virtuais no sentido de serem criadas e aparadas conforme a necessidade. As únicas PTEs do processo necessárias são aquelas que descrevem páginas para as quais existe uma visão atualmente mapeada. Isso melhora bastante o desempenho e permite o uso mais eficiente dos endereços virtuais do kernel.

A PTE de protótipo contém o endereço do quadro de página e os bits de proteção e estado. Assim, o primeiro acesso por um processo a uma página compartilhada gera um page fault. Depois do primeiro acesso, outros acessos são realizados da maneira normal. Se um processo escreve em uma página de cópia na escrita marcada como somente leitura na PTE, o gerenciador VM faz uma cópia da página e marca a PTE como passível de escrita, e o processo não possui mais uma página compartilhada. As páginas compartilhadas nunca aparecem no arquivo de página, mas são encontradas no sistema de arquivos.

O gerenciador VM acompanha todas as páginas da memória física em um **banco de dados de quadro de página**. Existe uma entrada para cada página de memória física no sistema. A entrada aponta para a PTE, que, por sua vez, aponta para o quadro de página, de modo que o gerenciador VM pode manter o estado da página. Os quadros de página não referenciados por uma PTE válida são vinculados a listas, de acordo com o tipo de página, como zerados, modificados, livres etc.

Se uma página física compartilhada for marcada como válida para qualquer processo, não poderá ser removida da memória. O gerenciador VM mantém uma contagem de PTEs válidas para cada página no banco de dados de quadros de página. Quando a contagem cai para zero, a página física

pode ser reutilizada, assim que seu conteúdo tiver sido gravado em disco (se foi marcada como suja).

Quando ocorre um page fault, o gerenciador VM encontra uma página física para manter os dados. Para páginas do tipo “zerar por demanda”, a primeira opção é encontrar uma página já zerada. Se não houver uma disponível, uma página da lista de páginas livres ou em standby é escolhida e a página é zerada. Se a página que faltou tiver sido marcada como em transição, ela já está sendo lida do disco ou foi desmapeada ou removida e ainda está disponível na lista de páginas em standby ou modificadas. A thread espera até a E/S terminar ou, em último caso, reivindicar a página a partir da lista apropriada.

Caso contrário, uma E/S precisa ser emitida para ler a página do arquivo de paginação ou sistema de arquivos. O gerenciador VM tenta alocar uma página disponível da lista de páginas livres ou em standby. As páginas na lista de modificadas não podem ser usadas até terem sido escritas de volta no disco e transferidas para a lista em standby. Se nenhuma página estiver disponível, a thread será bloqueada até o gerenciador do conjunto de trabalho aparar as páginas da memória ou até que uma página na memória física seja desmapeada por um processo.

O Windows XP utiliza uma política de substituição FIFO (First-In, First-Out) por processo para apanhar páginas dos processos conforme apropriado. O Windows XP monitora o page fault de cada processo que está no tamanho mínimo do conjunto de trabalho e ajusta o tamanho do conjunto de trabalho de acordo. Quando um processo é iniciado, ele recebe um tamanho mínimo padrão de 50 páginas para o conjunto de trabalho. O gerenciador VM substitui e apara as páginas no conjunto de trabalho de um processo de acordo com sua idade. A idade de uma página é determinada por quantos ciclos aparando as páginas ocorreram sem a PTE. As páginas apuradas são movidas para a lista de standby ou modificadas, dependendo de o bit modificado estar definido na PTE da página.

O gerenciador VM não gera uma falha apenas na página imediatamente necessária. Pesquisas mostram que a referência de memória de uma thread costuma ter uma propriedade de **localidade**; quando uma página é usada, é provável que páginas adjacentes sejam referenciadas no futuro próximo. (Pense na execução repetitiva sobre um array ou na busca de instruções sequenciais que formam o código executável para uma thread.) Devido à localidade, quando o gerenciador VM gera uma falha de uma página, ele também precisa de algumas páginas adjacentes. Essa busca antecipada costuma reduzir a quantidade total de falhas de página. As escritas também são agrupadas para reduzir o número de operações de E/S independentes.

Além de gerenciar a memória comprometida, o gerenciador VM gerencia a memória reservada de cada processo ou o espaço de endereços virtual. Cada processo possui uma árvore larga associada, que descreve os intervalos de endereços virtuais em uso e qual é o seu uso. Isso permite ao gerenciador VM gerar falhas nas tabelas de página, conforme a necessidade. Se a PTE para um endereço falso não existir, o gerenciador VM procura o endereço na árvore de **descritores de endereço virtual (Virtual Address Descriptors - VADs)** do processo e usa essa informação para preencher a PTE que falta e buscar a página. Em alguns casos, a própria página de uma tabela de página pode não existir, e tal página precisa ser alocada e inicializada transparentemente pelo gerenciador VM.

22.3.3.3 Gerenciador de processos

O gerenciador de processos do Windows XP fornece serviços para criar, excluir e usar processos, threads e tarefas. Ele não possui conhecimento a respeito dos relacionamentos pai-filho ou hierarquias de processo; esses refinamentos são deixados para o subsistema ambiental em particular que possui o processo. O gerenciador de processos também não está envolvido no escalonamento de processos, além de definir as prioridades e as afinidades nos processos e threads, quando criados. O escalonamento de threads ocorre no despachante do kernel.

Cada processo contém uma ou mais threads. Os próprios processos podem ser reunidos em grandes unidades, denominadas **objetos de tarefa**; o uso de objetos de tarefa concede limites sobre o uso de CPU, tamanho do conjunto de trabalho e afinidades do processador, que controlam vários processos ao mesmo tempo. Os objetos de tarefa são usados para gerenciar grandes máquinas de centro de dados.

Um exemplo de criação de processo no ambiente Win32 é o seguinte:

1. Uma aplicação Win32 chama `CreateProcess()`.
2. Uma mensagem é enviada ao subsistema Win32 para notificá-lo de que o processo está sendo criado.
3. `CreateProcess()` no processo original, em seguida, chama uma API no gerenciador de processos do executivo do NT, para criar o processo.
4. O gerenciador de processos chama o gerenciador de objetos para criar um objeto de processo e retorna o descritor do objeto para o Win32.
5. O Win32 chama o gerenciador de processos novamente para criar uma thread para o processo e retorna descritores ao novo processo e à thread.

As APIs do Windows XP para manipular a memória virtual e as threads e para duplicar descritores apanham um descritor de processo para os subsistemas poderem realizar operações em favor de um novo processo, sem ter de executar diretamente no contexto do novo processo. Quando um novo

processo é criado, a thread inicial é criada e uma APC é entregue à thread para requisitar o início da execução no carregador de imagem do modo usuário. O carregador é a ntdll.dll, que é uma DLL mapeada automaticamente para cada processo recém-criado. O Windows XP também admite um estilo de criação de processo `fork()` do UNIX a fim de dar suporte ao subsistema de ambiente POSIX. Embora o ambiente Win32 chame o gerenciador de processos do processo cliente, o POSIX usa a natureza entre processos das APIs do Windows XP para criar o novo processo de dentro do processo do subsistema.

O gerenciador de processos também implementa o enfileiramento e a entrega de chamadas de procedimento assíncronas (APCs) para threads. As APCs são usadas pelo sistema para iniciar a execução da thread, completar E/S, terminar threads e processos e conectar depuradores. O código no modo usuário também pode enfileirar uma APC a uma thread para entrega de notificações tipo sinal. Para dar suporte para POSIX, o gerenciador de processos fornece APIs que enviam alertas às threads para desbloqueá-las das chamadas de sistema.

O suporte do depurador no gerenciador de processos inclui a capacidade de suspender e retomar threads e criar threads que começam em um modo suspenso. Há também APIs do gerenciador de processos que apanham e definem o contexto de registradores de uma thread e acessam a memória virtual de outro processo.

As threads podem ser criadas no processo atual; elas também podem ser injetadas em outro processo. Dentro do executivo, as threads existentes podem ser anexadas temporariamente a outro processo. Esse método é usado pelas threads trabalhadoras que precisam ser executadas no contexto do processo que origina uma requisição de trabalho.

O gerenciador de processos também admite personalificação. Uma thread executando em um processo com um token de segurança pertencente a um usuário pode definir um token específico da thread pertencente a outro usuário. Essa facilidade é fundamental para o modelo de computação cliente-servidor, no qual os serviços precisam atuar em favor de uma série de clientes, com diferentes IDs de segurança.

22.3.3.4 Facilidade de chamada de procedimento local

A implementação do Windows XP utiliza um modelo cliente-servidor. Os subsistemas de ambiente são servidores que implementam determinadas personalidades do sistema operacional. O modelo cliente-servidor é usado para implementar uma série de serviços do sistema operacional além dos subsistemas de ambiente. Gerenciamento de segurança, spooling de impressora, serviços Web, sistemas de arquivo de rede, recursos plug-and-play e muitos outros recursos são implementados por meio desse modelo. Para reduzir o uso de memória, diversos serviços normalmente são reunidos em poucos processos, que contam então com as facilidades do banco de threads do modo usuário para compartilhar as threads e esperar por mensagens (ver [Seção 22.3.3.3](#)).

O sistema operacional utiliza a facilidade de chamada de procedimento local (LPC) para passar requisições e resultados entre os processos cliente e servidor dentro de uma única máquina. Em particular, a LPC é usada para requisitar serviços dos vários subsistemas do Windows XP. A LPC é semelhante em muitos aspectos aos mecanismos de RPC usados em muitos sistemas operacionais para o processamento distribuído pelas redes, mas a facilidade LPC é otimizada para ser usada dentro de um único sistema. A implementação no Windows XP de RPC da Open Software Foundation (OSF) utiliza a LPC como um transporte na máquina local.

A LPC é um mecanismo de troca de mensagens. O processo servidor publica um objeto de porta de conexão globalmente visível. Quando um cliente deseja os serviços de um subsistema, ele abre um descritor para o objeto de porta de conexão do subsistema e envia uma requisição de conexão à porta. O servidor cria um canal e retorna um descritor ao cliente. O canal consiste em um par de portas de comunicação privada: uma para mensagens do cliente ao servidor e a outra para mensagens do servidor ao cliente. Os canais de comunicação admitem um mecanismo de callback, de modo que o cliente e o servidor podem aceitar requisições quando normalmente estariam esperando uma resposta.

Quando um canal de LPC é criado, uma dentre três técnicas de troca de mensagens precisa ser especificada:

1. A primeira técnica é adequada para pequenas mensagens (até algumas centenas de bytes). Nesse caso, a fila de mensagens da porta é usada como armazenamento intermediário, e as mensagens são copiadas de um processo para o outro.
2. A segunda técnica é para mensagens maiores. Nesse caso, um objeto de seção da memória compartilhada é criado para o canal. As mensagens enviadas por meio da fila de mensagens da porta contêm um ponteiro e informações de tamanho referentes ao objeto de seção. Isso evita a necessidade de copiar mensagens grandes. O emissor coloca dados na seção compartilhada, e o receptor os vê diretamente.
3. A terceira técnica utiliza as APIs que leem e escrevem diretamente no espaço de endereços de um processo. O mecanismo de LPC fornece funções e sincronismo para um servidor poder acessar os dados em um cliente.

O gerenciador de janelas Win32 utiliza sua própria forma de troca de mensagens, independentemente das facilidades de LPC do executivo. Quando um cliente pede uma conexão que

usa mensagens do gerenciador de janelas, o servidor estabelece três objetos: (1) uma thread do servidor dedicada para requisições de descriptor; (2) um objeto de seção de 64 KB; e (3) um objeto de par de eventos. Um *objeto de par de eventos* é um objeto de sincronismo utilizado pelo subsistema Win32 para fornecer notificação quando a thread do cliente tiver copiado uma mensagem para o servidor Win32 ou vice-versa. O objeto de seção passa as mensagens, e o objeto de par de eventos realiza o sincronismo.

O uso de mensagens do gerenciador de janelas tem diversas vantagens:

- O objeto de seção elimina a cópia de mensagens, pois representa uma região da memória compartilhada.
- O objeto de par de eventos elimina o custo adicional de usar o objeto de porta para passar mensagens contendo ponteiros e tamanhos.
- A thread do servidor dedicado elimina o custo adicional de determinar qual thread do cliente está chamando o servidor, pois existe uma thread do servidor por thread do cliente.
- O kernel fornece preferência de escalonamento a essas threads do servidor dedicado para melhorar o desempenho.

22.3.3.5 Gerenciador de E/S

O **gerenciador de E/S** é responsável pelos sistemas de arquivos, drivers de dispositivos e drivers de rede. Ele registra quais drivers de dispositivos, drivers de filtro e sistemas de arquivos estão carregados e também gerencia os buffers para requisições de E/S. Ele atua com o gerenciador VM para fornecer E/S de arquivo mapeada na memória e controla o gerenciador de cache do Windows XP, que trata do caching para o sistema de E/S inteiro. O gerenciador de E/S é assíncrono. A E/S síncrona é fornecida esperando-se explicitamente até uma operação de E/S terminar. O gerenciador de E/S fornece vários modelos de término de E/S assíncrono, incluindo a configuração de eventos, entrega de APCs para as threads iniciando e uso de portas de término de E/S, que permitem que uma única thread processe términos de E/S de muitas outras threads.

Drivers de dispositivos são organizados como uma lista para cada dispositivo (chamada pilha de driver ou E/S). O gerenciador de E/S converte as requisições que recebe para um formato-padrão, chamado **pacote de requisição de E/S (I/O Request Packet - IRP)**. Depois, ele encaminha o IRP para o primeiro driver na pilha, para processamento. Depois que o driver processa o IRP, ele chama o gerenciador de E/S para encaminhar o IRP até o driver seguinte na pilha ou, se todo o processamento tiver terminado, para completar a operação no IRP.

A requisição de E/S pode ser concluída em um contexto diferente daquele em que foi feita. Por exemplo, se um driver estiver realizando sua parte de uma operação de E/S e for forçado a ficar bloqueado por um tempo estendido, ele pode enfileirar o IRP para uma thread trabalhadora, para continuar processando no contexto do sistema. Na thread original, o driver retorna um status indicando que a requisição de E/S está pendente, de modo que a thread possa continuar executando em paralelo com a operação de E/S. Um IRP também pode ser processado em rotinas de atendimento à interrupção e completado em um contexto qualquer. Como pode ser preciso realizar algum processamento final no contexto que iniciou a E/S, o gerenciador de E/S usa uma APC para realizar o processamento final do término de E/S no contexto da thread de origem.

O modelo de pilha de dispositivo é bastante flexível. Enquanto uma pilha de drivers é montada, diversos drivers têm a oportunidade de se inserirem na pilha como **drivers de filtro**. Drivers de filtro podem examinar e potencialmente modificar cada operação de E/S. Gerenciador de montagem, gerenciador de partição, espalhamento (striping) e espelhamento (mirroring) de disco são exemplos da funcionalidade implementada por meio de drivers de filtro executados debaixo do sistema de arquivos na pilha. Os drivers de filtro do sistema de arquivos são executados acima do sistema de arquivos e têm sido usados para implementar funcionalidades do tipo gerenciamento de armazenamento hierárquico, instância única de arquivos para boot remoto e conversão dinâmica de formato. Terceiros também utilizam drivers de filtro do sistema de arquivos para implementar a detecção de vírus.

Drivers de dispositivos para Windows XP são escritos na especificação Windows Driver Model (WDM). Esse modelo estabelece todos os requisitos para os drivers de dispositivos, incluindo como dispor os drivers de filtro em camadas, compartilhar código comum para tratar requisições de energia e plug-and-play, montar a lógica de cancelamento e assim por diante.

Devido à riqueza do WDM, escrever um driver de dispositivos WDM completo para cada novo dispositivo de hardware pode ser uma quantidade de trabalho excessiva. Felizmente, o modelo de porta/miniporta torna desnecessário fazer isso. Dentro de uma classe de dispositivos semelhantes, como drivers de áudio, dispositivos SCSI e controladores Ethernet, cada instância de um dispositivo compartilha um driver comum para essa classe, chamado **driver de porta**. O driver de porta implementa as operações-padrão para a classe e depois chama rotinas específicas do dispositivo no **driver de miniporta** do dispositivo, para implementar apenas a funcionalidade específica ao dispositivo.

22.3.3.6 Gerenciador de cache

Em muitos sistemas operacionais, o caching é feito pelo sistema de arquivos. No entanto, o Windows XP fornece uma facilidade de caching centralizada. O **gerenciador de cache** trabalha de perto com o gerenciador VM para fornecer serviços de cache para todos os componentes sob o controle do gerenciador de E/S. O caching no Windows XP é baseado em arquivos e não em blocos brutos.

O tamanho do cache muda dinamicamente, de acordo com a quantidade de memória livre disponível no sistema. Lembre-se de que os 2 GB superiores do espaço de endereços de um processo compreendem a área do sistema; ela está disponível no contexto de todos os processos. O gerenciador VM aloca até metade desse espaço para o cache do sistema. O gerenciador de cache mapeia os arquivos para esse espaço de endereços e utiliza as capacidades do gerenciador VM para tratar da E/S de arquivo.

O cache é dividido em blocos de 256 KB. Cada bloco do cache pode manter uma visão (ou seja, uma região mapeada na memória) de um arquivo. Cada bloco de cache é descrito por um **bloco de controle de endereço virtual (Virtual-Address Control Block - VACB)** que armazena o endereço virtual do deslocamento do arquivo para a visão, bem como a quantidade de processos que utilizam a visão. Os VACBs residem em um único array mantido pelo gerenciador de cache.

Para cada arquivo aberto, o gerenciador de cache mantém um array de índices VACB que descreve o caching para o arquivo inteiro. Esse array possui uma entrada para cada pedaço de 256 KB do arquivo; assim, por exemplo, um arquivo de 2 MB teria um array de índices VACB com 8 entradas. Uma entrada no array de índices VACB aponta para o VACB se essa parte do arquivo estiver no cache; caso contrário, ela é nula. Quando o gerenciador de E/S recebe uma requisição de leitura de um arquivo no nível de usuário, o gerenciador de E/S envia um IRP à pilha de drivers de dispositivos em que o arquivo reside. O sistema de arquivos tenta pesquisar os dados requisitados no gerenciador de cache (a menos que a requisição peça especificamente uma leitura de não cache). O gerenciador de cache calcula qual entrada do array de índices VACB desse arquivo corresponde ao deslocamento de byte da requisição. A entrada aponta para uma visão no cache ou é inválida. Se for inválida, o gerenciador de cache aloca um bloco de cache (e a entrada correspondente no array VACB) e mapeia a visão no bloco de cache. O gerenciador de cache, então, tenta copiar dados do arquivo mapeado para o buffer de quem chamou. Se a cópia tiver sucesso, a operação termina.

Se a cópia falhar, devido a um page fault, isso faz o gerenciador VM enviar uma requisição de leitura de não cache para o gerenciador de E/S. O gerenciador de E/S envia outra requisição para a pilha de drivers, dessa vez requisitando uma operação de *paginação*, que contorna o gerenciador de cache e lê os dados diretamente do arquivo para a página alocada para o gerenciador de cache. Ao terminar, o VACB é definido de modo que aponte para a página. Os dados, agora no cache, são copiados para o buffer de quem chamou, e a requisição de E/S original é concluída. A [Figura 22.6](#) mostra uma visão geral dessas operações.

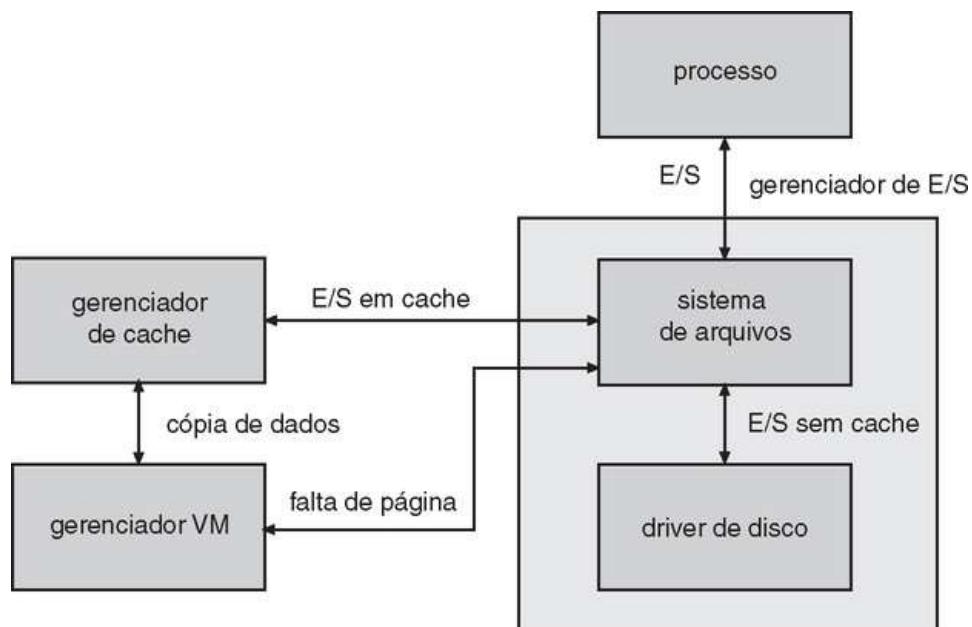


FIGURA 22.6 E/S de arquivo.

Quando for possível, para operações síncronas sobre arquivos em cache, a E/S é tratada pelo **mecanismo rápido de E/S**. Esse mecanismo é semelhante à E/S normal, baseada em IRP, mas chama dentro da pilha de drivers diretamente, em vez de passar por um IRP. Como não há IRP envolvido, a operação não deverá ser bloqueada por um período estendido e não poderá ser enfileirada para uma thread trabalhadora. Portanto, quando a operação atingir o sistema de

arquivos e chamar o gerenciador de cache, a operação falhará se as informações ainda não estiverem no cache. O gerenciador de E/S, então, tenta realizar a operação usando o caminho normal do IRP.

Uma operação de leitura no nível do kernel é semelhante, exceto que os dados podem ser acessados diretamente pelo cache, em vez de serem copiados para um buffer no espaço do usuário. Para usar os metadados do sistema de arquivos (estruturas de dados que descrevem o sistema de arquivos), o kernel utiliza a interface de mapeamento do gerenciador de cache para ler os metadados. Para modificar os metadados, o sistema de arquivos usa a interface de fixação do gerenciador de cache. A **fixação** de uma página bloqueia a página em um quadro de página da memória física, de modo que o gerenciador VM não possa mover ou paginar essa página. Depois de atualizar os metadados, o sistema de arquivos pede ao gerenciador de cache para liberar a página. Uma página modificada é marcada como suja, e assim o gerenciador VM leva a página para o disco. Os metadados são armazenados em um arquivo normal.

Para melhorar o desempenho, o gerenciador de cache mantém um pequeno histórico de requisições de leitura e, desse histórico, tenta prever requisições futuras. Se o gerenciador de cache encontrar um padrão nas três requisições anteriores, como um acesso sequencial para frente ou para trás, ele leva dados antecipadamente para o cache antes de a próxima requisição ser submetida pela aplicação. Desse modo, a aplicação encontra seus dados já em cache e não precisa esperar pela E/S em disco. As funções `OpenFile()` e `CreateFile()` da API Win32 podem receber o sinalizador `FILE_FLAG_SEQUENTIAL_SCAN`, que é uma sugestão para o gerenciador de cache tentar antecipar a leitura de 192 KB antes das requisições da thread. Normalmente, o Windows XP realiza operações de E/S em pedaços de 64 KB ou 16 páginas; assim, essa leitura antecipada corresponde a três vezes a quantidade normal.

O gerenciador de cache também é responsável por dizer ao gerenciador VM para esvaziar o conteúdo do cache. O comportamento-padrão do gerenciador de cache é o caching com escrita adiada: ele acumula escritas por 4 a 5 segundos e depois desperta a thread escritora do cache. Quando o caching de escrita direta é necessário, um processo pode marcar um sinalizador ao abrir o arquivo ou, então, o processo pode chamar uma função explícita de esvaziamento de cache.

Um processo de escrita rápida poderia potencialmente preencher todas as páginas de cache livres antes de a thread escritora do cache ter uma chance de despertar e enviar as páginas para o disco. O escritor do cache impede que um processo inunde o sistema da seguinte maneira. Quando a quantidade de memória livre no cache for baixa, o gerenciador de cache bloqueia temporariamente os processos que tentam escrever dados e desperta a thread escritora de cache para enviar as páginas para o disco. Se o processo de escrita rápida for, na realidade, um redirecionador de rede para um sistema de arquivos de rede, seu bloqueio por muito tempo poderia ultrapassar um tempo-limite e fazer as transferências na rede serem retransmitidas. Essa retransmissão desperdiçaria largura de banda da rede. Para impedir esse desperdício, os redirecionadores de rede podem instruir o gerenciador de cache a limitar o acúmulo de escritas no cache.

Como o sistema de arquivos de rede precisa mover dados entre um disco e a interface de rede, o gerenciador de cache também fornece uma interface de DMA para mover os dados diretamente. Mover dados dessa forma evita a cópia de dados passando por um buffer intermediário.

22.3.3.7 Monitor de referência de segurança

Centralizar o gerenciamento de entidades do sistema no gerenciador de objetos permite ao Windows XP usar um mecanismo uniforme para realizar a validação de acesso em tempo de execução e verificações de auditoria para cada entidade acessível ao usuário no sistema. Sempre que um processo abre um descritor para um objeto, o **monitor de referência de segurança (Security Reference Monitor - SRM)** verifica o token de segurança do processo e a lista de controle de acesso do objeto para ver se o processo tem os direitos necessários.

O SRM também é responsável por tratar os privilégios nos tokens de segurança. Privilégios especiais são exigidos para os usuários realizarem operações de backup ou restauração sobre sistemas de arquivos, depurar processos e assim por diante. As fichas também podem ser marcadas como sendo restritas em seus privilégios, de modo que não possam acessar objetos que estão disponíveis para a maioria dos usuários. Os tokens restritos são usados para restringir os danos que podem ser ocasionados pela execução de código não confiável.

Outra responsabilidade do SRM é registrar os eventos de auditoria de segurança em um log. Uma classificação de segurança C2 exige que o sistema tenha a capacidade de detectar e registrar em log todas as tentativas de acessar recursos do sistema, para ser mais fácil rastrear as tentativas de acesso não autorizado. Como o SRM é responsável por fazer verificações de acesso, ele gera a maior parte dos registros de auditoria no log de evento de segurança.

22.3.3.8 Gerenciadores de plug-and-play e energia

O sistema operacional utiliza o **gerenciador de plug-and-play (PnP)** para reconhecer e adaptar-se às mudanças na configuração do hardware. Para o PnP funcionar, tanto o dispositivo quanto o driver precisam admitir o padrão PnP. O gerenciador de PnP reconhece automaticamente os dispositivos

instalados e detecta mudanças nos dispositivos enquanto o sistema opera. O gerenciador também registra recursos usados por um dispositivo, bem como recursos em potencial que poderiam ser usados e cuida do carregamento dos drivers apropriados. Esse gerenciamento de recursos do hardware – principalmente, interrupções e intervalos de memória de E/S – tem o objetivo de definir uma configuração de hardware em que todos os dispositivos são capazes de operar.

Por exemplo, se o dispositivo B pode usar a interrupção 5 e o dispositivo A pode usar 5 ou 7, então o gerenciador de PnP atribuirá 5 a B e 7 a A. Nas versões anteriores, o usuário poderia ter de remover o dispositivo A e reconfigurá-lo para usar a interrupção 7, antes de instalar o dispositivo B. O usuário, assim, tinha de estudar os recursos do sistema antes de instalar o novo hardware e descobrir ou se lembrar de quais dispositivos estavam usando quais recursos do hardware. A proliferação de placas PCMCIA, encaixes de laptop, USB, IEEE 1394, Infiniband e outros dispositivos hot-plug também ditam o suporte de recursos configuráveis dinamicamente.

O gerenciador de PnP trata dessa reconfiguração dinâmica da seguinte maneira. Primeiro, ele apanha uma lista de dispositivos para cada driver de barramento (por exemplo, PCI, USB). Ele carrega o driver instalado (ou instala um, se for preciso) e envia uma requisição `add-device` ao driver apropriado para cada dispositivo. O gerenciador de PnP descobre os atributos de recursos ideais e envia uma requisição `start-device` para cada driver junto com a atribuição de recursos para o dispositivo. Se um dispositivo precisar ser reconfigurado, o gerenciador de PnP envia uma requisição `query-stop`, que pergunta ao driver se o dispositivo pode ser desativado temporariamente. Se o driver puder desativar o dispositivo, então todas as operações pendentes são completadas e novas operações são impedidas de iniciar. Em seguida, o gerenciador de PnP envia uma requisição `stop`; depois, ele pode reconfigurar o dispositivo com outra requisição `start-device`.

O gerenciador de PnP também aceita outras solicitações, como `query-remove`. Essa requisição é usada quando um usuário está se aprontando para ejectar um dispositivo PCCARD e opera de um modo semelhante a `query-stop`. A requisição `surprise-remove` é usada quando um dispositivo falha ou, mais provavelmente, quando um usuário remove um dispositivo PCCARD sem encerrá-lo primeiro. A requisição `remove` diz ao driver para parar de usar o dispositivo e liberar todos os recursos alocados a ele.

O Windows XP admite um gerenciamento de energia sofisticado. Embora esse recurso seja útil para os sistemas domésticos reduzirem o consumo de energia, sua aplicação principal é para promover facilidade de uso (acesso mais rápido) e estender a vida da bateria dos laptops. O sistema e os dispositivos individuais podem ser movidos para o modo de baixo consumo (denominado modo `standby` ou `sleep`) quando não estiverem em uso, de modo que a bateria seja direcionada principalmente para a retenção de dados da memória física (RAM). O sistema pode ser ligado novamente quando forem recebidos pacotes pela rede, uma linha telefônica para um modem tocar ou um usuário abrir um laptop ou pressionar um botão para acordar. O Windows XP também pode colocar um sistema em espera, armazenando o conteúdo da memória física em disco e desligando a máquina, para restaurar o sistema mais tarde, antes de a execução continuar.

Outras estratégias para reduzir o consumo de energia são aceitas também. Em vez de permitir que o sistema entre em um loop do processador quando a CPU está ociosa, o Windows XP o move para um estado que exige menos consumo de energia. Se a CPU estiver sendo pouco utilizada, o Windows XP reduz a velocidade do relógio da CPU, o que pode economizar muita energia.

22.3.3.9 Registry

O Windows XP mantém grande parte de suas informações de configuração em um banco de dados interno, chamado **registry**. Um banco de dados registry é chamado **hive**. Existem hives separados para informações do sistema, preferências de usuário-padrão, instalação de software e segurança. Como as informações no **hive do sistema** são exigidas para o boot do sistema, o gerenciador de registro é implementado como um componente do executivo.

Toda vez que o sistema dá boot com sucesso, ele salva o hive do sistema como *último estado bom conhecido*. Se o usuário instalar um software, como um driver de dispositivos, que produza uma configuração do hive do sistema que não dê boot, o usuário pode dar boot usando a configuração do último estado bom conhecido.

Os danos ao hive do sistema da instalação de aplicações e drivers de terceiros são tão comuns que o Windows XP possui um componente chamado **restauração** do sistema, que salva os hives periodicamente, além de outros estados de software como executáveis de driver e arquivos de configuração, de modo que o sistema possa ser restaurado para um estado funcionando anteriormente em casos em que o sistema dá boot mas não opera mais conforme o esperado.

22.3.3.10 Booting

O boot de um PC Windows XP começa quando o hardware é ligado e o BIOS começa a executar a partir da ROM. O BIOS identifica o **dispositivo do sistema** a ser inicializado e carrega e executa o carregador de boot, contido no início do disco. Esse carregador conhece o suficiente sobre o formato do sistema de arquivos para carregar o programa NTLDR do diretório raiz do dispositivo do sistema. O NTLDR é usado para determinar qual **dispositivo de boot** contém o sistema operacional. Em

seguida, o NTLDR carrega a biblioteca HAL, o kernel e o hive do sistema do dispositivo de boot. Do hive do sistema, ele determina quais drivers de dispositivos são necessários para inicializar o sistema (os *drivers de boot*) e os carrega. Finalmente, o NTLDR inicia a execução do kernel.

O kernel inicializa o sistema e cria dois processos. O **processo do sistema** contém todas as threads trabalhadoras internas e nunca é executado no modo usuário. O primeiro processo no modo usuário criado é o SMSS, que é semelhante ao processo INIT (inicialização) no UNIX. O SMSS continua a inicialização do sistema, incluindo o estabelecimento dos arquivos de paginação e o carregamento de drivers de dispositivos e cria os processos WINLOGON e CSRSS. O CSRSS é o subsistema Win32. O WINLOGON traz o restante do sistema, incluindo o subsistema de segurança LSASS e os serviços restantes necessários para executar o sistema.

O sistema optimiza o processo de boot carregando antecipadamente os arquivos do disco com base nos boots anteriores do sistema. Os padrões de acesso ao disco no boot também são usados para distribuir os arquivos do sistema no disco, a fim de reduzir o número de operações de E/S exigidas. A quantidade de processos exigida para iniciar o sistema é reduzida pelo agrupamento de serviços em um processo. Todas essas técnicas contribuem para uma redução drástica no tempo de boot do sistema. Naturalmente, o tempo de boot do sistema é menos importante do que era antes, devido às facilidades de espera e hibernação do Windows XP, que permitem que os usuários desliguem seu computador e depois continuem rapidamente de onde pararam.

22.4 Subsistemas de ambiente

Os subsistemas de ambiente são processos do modo usuário dispostos sobre os serviços executivos originais do Windows XP, para permitir que o Windows XP execute programas desenvolvidos para outros sistemas operacionais, incluindo Windows de 16 bits, MS-DOS e POSIX. Cada subsistema de ambiente fornece um ambiente de aplicação isolado.

O Windows XP utiliza o subsistema Win32 como ambiente operacional principal e, com isso, inicia todos os processos. Quando uma aplicação é executada, o subsistema API Win32 chama o gerenciador VM para carregar o código executável da aplicação. O gerenciador de memória retorna um status para o Win32 indicando o tipo de executável. Se ele não for um executável Win32 nativo, o ambiente API Win32 verifica se o subsistema de ambiente apropriado está sendo executado; se o subsistema não estiver em execução, ele é iniciado como um processo no modo usuário. O subsistema, então, toma o controle do boot da aplicação.

Os subsistemas de ambiente utilizam a facilidade de LPC para fornecer serviços do sistema operacional aos processos clientes. A arquitetura do subsistema Windows XP evita que as aplicações misturem rotinas da API de diferentes ambientes. Por exemplo, uma aplicação API Win32 não pode fazer uma chamada de sistema POSIX porque somente um subsistema de ambiente pode estar associado a cada processo.

Como cada subsistema é executado como um processo separado no modo usuário, uma falha em um não tem efeito sobre outros processos. A exceção é a API Win32, que fornece todas as capacidades de teclado, mouse e exibição gráfica. Se ela falhar, o sistema fica desativado e precisa ser reinicializado.

O ambiente Win32 categoriza aplicações como gráficas ou baseadas em caracteres, sendo que uma *aplicação baseada em caracteres* é aquela que acredita que a saída interativa vai para uma janela baseada em caracteres (janela de comandos). O Win32 transforma a saída de uma aplicação baseada em caracteres em uma representação gráfica na janela de comandos. Essa transformação é fácil: sempre que uma rotina de saída é chamada, o subsistema de ambiente chama uma rotina Win32 para exibir o texto. Como o ambiente API Win32 realiza essa função para todas as janelas baseadas em caracteres, ele pode transferir o texto da tela entre as janelas por meio da área de transferência (clipboard). Essa transformação funciona para aplicações do MS-DOS e também para aplicações POSIX na linha de comandos.

22.4.1 Ambiente MS-DOS

O ambiente MS-DOS não possui a complexidade dos outros subsistemas de ambiente do Windows XP. Ele é fornecido por uma aplicação API Win32 chamada **máquina DOS virtual (Virtual DOS Machine - VDM)**. Como a VDM é um processo do modo usuário, ela é paginada e despachada como qualquer outra aplicação do Windows XP. A VDM possui uma **unidade de execução de instruções** para executar ou simular as instruções do processador Intel 486. A VDM também fornece rotinas para simular o ROM BIOS do MS-DOS e os serviços de interrupção de software “int21” e possui drivers de dispositivos virtuais para a tela, teclado e portas de comunicação. A VDM é baseada no código-fonte do MS-DOS 5.0; ela separa pelo menos 620 KB de memória para a aplicação.

O shell de comandos do Windows XP é um programa que cria uma janela semelhante a um ambiente do MS-DOS. Ele pode trabalhar com executáveis de 16 e 32 bits. Quando uma aplicação MS-DOS é executada, o shell de comandos inicia um processo da VDM para executar o programa.

Se o Windows XP estiver executando em um processador compatível com IA32, as aplicações gráficas do MS-DOS executam no modo de tela inteira, e as aplicações de caractere podem executar em tela inteira ou em uma janela. Nem todas as aplicações do MS-DOS são executadas sob a VDM. Por exemplo, algumas aplicações MS-DOS acessam o hardware do disco diretamente e, por isso, não conseguem ser executadas no Windows XP, pois o acesso ao disco é restrito, a fim de proteger o sistema de arquivos. Em geral, as aplicações do MS-DOS que acessam o hardware diretamente não conseguirão operar no Windows XP.

Como o MS-DOS não é um ambiente multitarefa, algumas aplicações foram escritas de um modo que “monopolizam” a CPU. Por exemplo, o uso de loops ocupados pode causar atrasos ou pausas na execução. O escalonador no despachante do kernel detecta esses atrasos e automaticamente acelera o uso da CPU, mas isso pode fazer a aplicação problemática operar de forma incorreta.

22.4.2 Ambiente Windows de 16 bits

O ambiente de execução Win16 é fornecido por uma VDM que incorpora um software adicional, chamado *Windows on Windows* (WOW32 para aplicações de 16 bits), software que fornece as rotinas do kernel do Windows 3.1 e as rotinas stub para as funções do gerenciador de janelas e da interface de dispositivo gráfico (Graphical-Device-Interface - GDI). As rotinas de stub chamam as sub-rotinas Win32 apropriadas, convertendo (*thunking*) endereços de 16 bits para endereços de 32 bits. As aplicações que contam com a estrutura interna do gerenciador de janelas de 16 bits ou com

a GDI podem não funcionar, pois a implementação Win32 subjacente, é claro, é diferente do verdadeiro Windows de 16 bits.

O WOW32 pode realizar a multitarefa com outros processos no Windows XP, mas isso é semelhante ao Windows 3.1 de muitas maneiras. Somente uma aplicação Win16 pode ser executada de cada vez; todas as aplicações possuem uma única thread e residem no mesmo espaço de endereços, e todas elas compartilham a mesma fila de entrada. Essas características implicam que uma aplicação que deixa de receber entrada bloqueará todas as outras aplicações Win16, assim como no Windows 3.x, e que uma aplicação Win16 pode atrapalhar outras aplicações Win16, adulterando o espaço de endereços. Contudo, o usuário pode permitir que vários ambientes Win16 coexistam, usando o comando *start /aplicação Win16 separada* a partir da linha de comandos.

Existem relativamente poucas aplicações de 16 bits que os usuários precisam continuar a executar no Windows XP, mas algumas delas incluem programas de instalação (setup) comuns. Assim, o ambiente WOW32 continua a existir porque diversas aplicações de 32 bits não podem ser instaladas no Windows XP sem ele.

22.4.3 Ambiente Windows de 32 bits no IA64

O ambiente nativo para Windows no IA64 usa endereços de 64 bits e o conjunto de instruções IA64 nativo. Para executar programas IA32 nesse ambiente, é preciso haver uma camada de thunking para traduzir chamadas Win32 de 32 bits para as chamadas de 64 bits correspondentes, conforme é exigido pelas aplicações de 16 bits de tradução nos sistemas IA32. Assim, o Windows de 64 bits fornece suporte para o ambiente WOW64. As implementações do Windows de 32 bits e 64 bits são essencialmente idênticas, e o processador IA64 fornece execução direta de instruções IA32, de modo que o WOW64 consegue um nível de compatibilidade mais alto do que o WOW32.

22.4.4 Ambiente Win32

Como já dissemos, o subsistema principal no Windows XP é o Win32, que executa aplicações API Win32 e controla toda a E/S de teclado, mouse e tela. Por ser o ambiente de controle, ele é projetado para ser extremamente robusto. Vários recursos do Win32 contribuem para essa robustez. Ao contrário dos processos no ambiente Win16, cada processo API Win32 possui sua própria fila de entrada. O gerenciador de janelas despacha toda a entrada no sistema para a fila de entrada do processo apropriado, de modo que um processo que falha não bloqueia a entrada para outros processos.

O kernel do Windows XP também fornece multitarefa preemptiva, que permite ao usuário terminar aplicações que falharam ou que não são mais necessárias. O Win32 também valida todos os objetos antes de usá-los, para evitar falhas que, de outra forma, poderiam ocorrer se uma aplicação tentasse usar um descritor inválido ou errado. O subsistema API Win32 verifica o tipo do objeto que um descritor aponta antes de usar o objeto. As contagens de referência mantidas pelo gerenciador de objetos impedem que os objetos sejam excluídos enquanto ainda estão sendo usados e impedem seu uso após terem sido excluídos.

Para alcançar um alto nível de compatibilidade com sistemas Windows 95/98, o Windows XP permite que os usuários especifiquem que as aplicações individuais sejam executadas usando uma **camada de calço**, que modifica as APIs Win32 para que aproximem melhor o comportamento esperado pelas aplicações antigas. Por exemplo, algumas aplicações esperam para ver determinada versão do sistema e falham em versões novas. Constantemente, as aplicações possuem bugs latentes que se tornam expostos devido a mudanças na implementação. A execução de uma aplicação com os calços do Windows 95/98 ativados faz o sistema fornecer um comportamento muito mais próximo ao do Windows 95/98, embora com desempenho reduzido e interoperabilidade limitada com outras aplicações.

22.4.5 Subsistema POSIX

O subsistema POSIX foi projetado para executar aplicações POSIX escritas para seguir o padrão POSIX, baseado no modelo do UNIX. As aplicações POSIX podem ser iniciadas pelo subsistema API Win32 ou por outra aplicação POSIX. As aplicações POSIX utilizam o servidor do subsistema POSIX (**PSXSS.EXE**), a DLL POSIX (**PSXDLL.DLL**) e o gerenciador de sessão do console POSIX (**POSIX.EXE**).

Embora o padrão POSIX não especifique impressão, as aplicações POSIX podem usar as impressoras de forma transparente por meio do mecanismo de redirecionamento do Windows XP. As aplicações POSIX têm acesso a qualquer sistema de arquivos no sistema Windows XP; o ambiente POSIX impõe permissões do tipo UNIX sobre árvores de diretório.

Devido a questões de escalonamento, o sistema POSIX no Windows XP não vem com o sistema, mas está disponível separadamente para sistemas desktop profissionais e servidores. Ele fornece um nível de compatibilidade muito mais alto com aplicações UNIX do que as versões anteriores do NT. Das aplicações UNIX normalmente disponíveis, a maioria é compilada e executada sem mudanças com a versão mais recente do Interix.

22.4.6 Subsistemas de logon e segurança

Antes de um usuário poder acessar os objetos no Windows XP, esse usuário precisa ser autenticado pelo serviço de logon, o WINLOGON. O WINLOGON é responsável por responder à sequência de atenção segura (Ctrl-Alt-Del). A sequência de atenção segura é um mecanismo exigido para evitar que uma aplicação atue como um cavalo de Troia. Somente o WINLOGON pode interceptar essa sequência para montar uma tela de logon, alterar senhas e trancar a estação de trabalho. Para ser autenticado, um usuário precisa ter uma conta e fornecer a senha para essa conta. Como alternativa, um usuário efetua o logon usando um cartão inteligente e um número de identificação pessoal, sujeito às políticas de segurança em vigor no domínio em particular.

O subsistema de autoridade de segurança local (Local Security Authority Subsystem - LSASS) é o processo que gera as filas de acesso para representar os usuários no sistema. Ele chama um **pacote de autenticação** para realizar autenticação usando informações do subsistema de logon ou servidor de rede. Normalmente, o pacote de autenticação pesquisa a informação de conta em um banco de dados local e verifica se a senha está correta. O subsistema de segurança, então, gera o token de acesso para o ID de usuário contendo os privilégios apropriados, limites de cota e IDs de grupo. Sempre que o usuário tenta acessar um objeto no sistema, como abrir um descritor para o objeto, o token de acesso é passado ao monitor de referência de segurança, que verifica privilégios e cotas. O pacote de autenticação padrão para domínios do Windows XP é o Kerberos. O LSASS também tem a responsabilidade de implementar a política de segurança, como senhas fortes, autenticação de usuários e realização de criptografia de dados e chaves.

22.5 Sistema de arquivos

Historicamente, os sistemas MS-DOS têm usado o sistema de arquivos da tabela de alocação de arquivos (File-Allocation Table - FAT). O sistema de arquivos FAT de 16 bits possui várias limitações, incluindo a fragmentação interna, uma limitação de tamanho de 2 GB e uma falha de proteção de acesso para arquivos. O sistema de arquivos FAT de 32 bits solucionou os problemas de tamanho e fragmentação, mas seu desempenho e recursos ainda são fracos em comparação com os sistemas de arquivos modernos. O sistema de arquivos NTFS é muito melhor. Ele foi projetado para incluir muitos recursos, incluindo recuperação de dados, segurança, tolerância a falha, arquivos e sistemas de arquivos grandes, múltiplos fluxos de dados, nomes UNICODE, arquivos esparsos, criptografia, diário, cópias de sombra de volume e compactação de arquivos.

O Windows XP utiliza o NTFS como seu sistema de arquivos básico, e por isso vamos nos concentrar nele. Contudo, o Windows XP continua a usar o FAT16 para ler disquetes e outras formas de mídia removível. E, apesar das vantagens do NTFS, o FAT32 continua a ser importante por interoperabilidade de mídia com sistemas Windows 95/98. O Windows XP admite tipos de sistema de arquivos adicionais para os formatos comuns usados para mídia de CD e DVD.

22.5.1 Esquema interno do NTFS

A entidade fundamental no NTFS é um volume. Um volume é criado pelo utilitário de gerenciamento de disco lógico do Windows XP e é baseado em uma partição de disco lógica. Um volume pode ocupar uma parte de um disco, pode ocupar um disco inteiro ou pode se espalhar por vários discos.

O NTFS não lida com os setores individuais de um disco; em vez disso utiliza clusters como a unidade de alocação de disco. Um **cluster** é uma série de setores de disco cuja quantidade é uma potência de 2. O tamanho do cluster é configurado quando um sistema de arquivos NTFS é formatado. O tamanho de cluster-padrão é o tamanho do setor para volumes de até 512 MB, 1 KB para volumes de até 1 GB, 2 KB para volumes de até 2 GB e 4 KB para volumes maiores. Esse tamanho de cluster é muito menor do que aquele para o sistema de arquivos FAT de 16 bits, e o tamanho pequeno reduz a quantidade de fragmentação interna. Como exemplo, considere um disco de 1,6 GB com 16.000 arquivos. Se você usar um sistema de arquivos FAT-16, 400 MB podem ser perdidos para a fragmentação interna, pois o tamanho do cluster é de 32 KB. Sob o NTFS, somente 17 MB serão perdidos ao armazenar os mesmos arquivos.

O NTFS utiliza **números de cluster lógicos (Logical Cluster Numbers - LCNs)** como endereços de disco. Ele os atribui numerando clusters desde o início do disco até o final. Usando esse esquema, o sistema pode calcular um deslocamento de disco físico (em bytes) multiplicando o LCN pelo tamanho do cluster.

Um arquivo no NTFS não é um fluxo de bytes simples, como no MS-DOS ou no UNIX; em vez disso, ele é um objeto estruturado consistindo em **atributos** tipificados. Cada atributo de um arquivo é um fluxo de bytes independente, que pode ser criado, removido, lido e escrito. Alguns tipos de atributo são padrão para todos os arquivos, incluindo o nome do arquivo (ou nomes, se o arquivo tiver nomes alternativos, ou aliases, como um nome abreviado do MS-DOS), a hora da criação e o descritor de segurança que especifica o controle de acesso. Os dados do usuário são armazenados em *atributos de dados*.

A maioria dos arquivos de dados tradicionais possui um atributo de dados *não nomeado*, que contém todos os dados do arquivo. Entretanto, fluxos de dados adicionais podem ser criados com nomes explícitos. Por exemplo, nos arquivos do Macintosh armazenados em um servidor do Windows XP, a confluência de recursos é um fluxo de dados nomeado. As interfaces IProp do modelo de objeto comum (Component Object Model - COM) utilizam um fluxo de dados nomeado para armazenar propriedades sobre arquivos comuns, incluindo miniaturas de imagens. Em geral, os atributos podem ser acrescentados conforme a necessidade e são acessados usando uma sintaxe *nome-arquivo:atributo*. O NTFS só retorna o tamanho do atributo não nomeado em resposta a operações de consulta de arquivos, como na execução do comando `dir`.

Cada arquivo no NTFS é descrito por um ou mais registros em um array armazenado em um arquivo especial, chamado tabela mestra de arquivos (Master File Table - MFT). O tamanho de um registro é determinado quando o sistema de arquivos é criado; ele varia de 1 a 4 KB. Atributos pequenos são armazenados no próprio registro da MFT e são denominados **atributos residentes**. Atributos grandes, como os dados não nomeados, denominados **atributos não residentes**, são armazenados em uma ou mais **extensões** contíguas no disco, e um ponteiro para cada extensão é armazenado no registro da MFT. Para um arquivo pequeno, até mesmo o atributo de dados pode caber dentro do registro da MFT. Se um arquivo tiver muitos atributos - ou se ele já estiver altamente fragmentado e, portanto, muitos ponteiros são necessários para apontar todos os fragmentos -, um registro na MFT pode não ser grande o suficiente. Nesse caso, o arquivo é descrito por um registro chamado **registro-base do arquivo**, que contém ponteiros para registros adicionais, que mantêm ponteiros e atributos adicionais.

Cada arquivo em um volume NTFS possui uma ID exclusiva, chamada **referência de arquivo**. A

referência de arquivo é uma quantidade de 64 bits que consiste em um número de arquivo de 48 bits e um número de sequência de 16 bits. O número de arquivo é o número de registro (ou seja, o slot do array) na MFT que descreve o arquivo. O número de sequência é incrementado toda vez que uma entrada da MFT é reutilizada. O número de sequência permite que o NTFS realize verificações de coerência internas, como apanhar uma referência antiga para um arquivo excluído depois de a entrada MFT ter sido reutilizada para um novo arquivo.

22.5.1.1 Árvore B+ do NTFS

Como no MS-DOS e no UNIX, o espaço de nomes do NTFS é organizado como uma hierarquia de diretórios. Cada diretório utiliza uma estrutura de dados, denominada **árvore B+**, para armazenar um índice dos nomes de arquivo nesse diretório. Em uma árvore B+, o tamanho de cada caminho da raiz da árvore até a folha é igual, e o custo de reorganizar a árvore é eliminado. A **raiz de índice** de um diretório contém o nível superior da árvore B+. Para um diretório grande, esse nível superior contém ponteiros para extensões de disco que mantêm o restante da árvore. Cada entrada no diretório contém o nome e a referência do arquivo, bem como uma cópia da estampa de tempo e tamanho de arquivo atualizados, retirados dos atributos residentes do arquivo, na MFT. As cópias dessas informações são armazenadas no diretório, de modo que uma listagem de diretório pode ser gerada com eficiência. Como todos os nomes de arquivo, tamanhos e tempo de atualização estão disponíveis no próprio diretório, não é preciso reunir esses atributos das entradas da MFT para cada um dos arquivos.

22.5.1.2 Metadados do NTFS

Os metadados do volume do NTFS são todos armazenados em arquivos. O primeiro arquivo é a MFT. O segundo arquivo, usado durante a recuperação se a MFT for danificada, contém uma cópia das 16 primeiras entradas na MFT. Os próximos poucos arquivos também possuem finalidade especial. Eles incluem os arquivos descritos a seguir.

- O **arquivo de log** registra todas as atualizações de metadados ao sistema de arquivos.
- O **arquivo de volume** contém o nome do volume, a versão do NTFS que formatou o volume e um bit que informa se o volume pode ter sido adulterado e precisa ter sua coerência verificada.
- A **tabela de definição de atributo** indica quais tipos de atributos são usados no volume e que operações podem ser realizadas em cada um deles.
- O **diretório raiz** é o diretório de nível superior na hierarquia do sistema de arquivos.
- O **arquivo de mapa de bits** indica quais clusters em um volume estão alocados aos arquivos e quais estão livres.
- O **arquivo de boot** contém o código de boot para o Windows XP e precisa estar localizado em um endereço de disco em particular, para poder ser encontrado facilmente por um boot loader na ROM. O arquivo de boot também possui o endereço físico da MFT.
- O **arquivo de clusters defeituosos** registra quaisquer áreas defeituosas no volume; o NTFS usa esse registro para a recuperação de erro.

22.5.2 Recuperação

Em muitos sistemas de arquivos simples, uma falta de energia no momento errado pode danificar tanto as estruturas de dados do sistema de arquivos que o volume inteiro se torna embaralhado. Muitas versões do UNIX armazenam metadados redundantes no disco, e eles se recuperam de falhas usando o programa fsck para verificar todas as estruturas de dados do sistema de arquivos e restaurá-las forçadamente para um estado consistente. Sua restauração envolve excluir arquivos danificados e liberar clusters de dados escritos com dados do usuário, mas não foram registrados corretamente nas estruturas de metadados do sistema de arquivos. Essa verificação pode ser um processo lento e pode perder quantidades de dados significativas.

O NTFS usa uma técnica diferente para a robustez do sistema de arquivos. No NTFS, todas as atualizações à estrutura de dados do sistema de arquivos são realizadas dentro das transações. Antes de uma estrutura de dados ser alterada, a transação escreve um registro de log que contém informações de refazimento e desfazimento; depois de a estrutura de dados ter sido alterada, a transação escreve um registro commit no log para significar que a transação teve sucesso.

Após uma falha, o sistema pode restaurar as estruturas de dados do sistema de arquivos para um estado consistente, processando os registros de log, primeiro refazendo as operações para transações conformadas e depois desfazendo as operações para transações que não foram confirmadas com sucesso antes da falha. Periodicamente (a cada 5 segundos), um registro de ponto de verificação é escrito no log. O sistema não precisa manter em log os registros antes do ponto de verificação para se recuperar de uma falha. Eles podem ser descartados, de modo que o arquivo de log não cresce sem limites. Na primeira vez que um volume NTFS é acessado, depois do boot do sistema, o NTFS realiza automaticamente a recuperação do sistema de arquivos.

Esse esquema não garante que todo o conteúdo do arquivo do usuário esteja correto após uma falha; ele só garante que as estruturas de dados do sistema de arquivos (os arquivos de metadados)

não estão danificadas e refletem algum estado coerente existente antes da falha. Seria possível estender o esquema de transação para abranger os arquivos do usuário, e a Microsoft poderá fazer isso no futuro.

O log é armazenado no terceiro arquivo de metadados no início do volume. Ele é criado com um tamanho máximo fixo quando o sistema de arquivos for formatado. Ele possui duas seções: a **área de logging**, que é uma fila circular de registros de log, e a **área de reinício**, que mantém informações de contexto, como a posição na área de logging onde o NTFS deverá começar a ler durante uma recuperação. De fato, a área de reinício mantém duas cópias de suas informações, de modo que a recuperação ainda é possível se uma cópia for danificada durante a falha.

A funcionalidade do registro em log é fornecida pelo **serviço de arquivo de log** do Windows XP. Além de escrever os registros de log e realizar ações de recuperação, o serviço do arquivo de log registra o espaço livre no arquivo de log. Se o espaço livre ficar muito baixo, o serviço de arquivo de log enfileira as transações pendentes, e o NTFS encerra todas as novas operações de E/S. Depois de a operação em andamento terminar, o NTFS chama o gerenciador de cache para esvaziar todos os dados e depois reinicia o arquivo de log e realiza as transações enfileiradas.

22.5.3 Segurança

A segurança de um volume NTFS é derivada do modelo de objeto do Windows XP. Cada arquivo NTFS referencia um descritor de segurança que contém o token de acesso do proprietário do arquivo e uma lista de controle de acesso que indica os privilégios de acesso concedidos a cada usuário que possui acesso ao arquivo.

Na operação normal, o NTFS não impõe permissões na travessia de diretórios nos nomes de caminho de arquivo. Todavia, por compatibilidade com o POSIX, essas verificações podem ser ativadas. As verificações de travessia são inherentemente mais dispendiosas, pois a análise moderna dos nomes de caminho de arquivo utiliza a combinação de prefixo em vez da abertura, componente por componente, dos nomes de diretório.

22.5.4 Gerenciamento de volume e tolerância a falhas

O **FtDisk** é o driver de disco tolerante a falhas para Windows XP. Quando instalado, ele fornece várias maneiras de combinar diversas unidades de disco em um volume lógico, a fim de melhorar o desempenho, a capacidade ou a confiabilidade.

22.5.4.1 Volume set

Um modo de combinar vários discos é concatená-los logicamente para formar um volume lógico grande, como mostra a [Figura 22.7](#). No Windows XP, esse volume lógico é denominado volume set (**conjunto de volumes**), que pode consistir em até 32 partições físicas. Um conjunto de volumes que contém um volume NTFS pode ser estendido sem perturbar os dados já armazenados no sistema de arquivos. Os metadados de mapa de bits no volume NTFS são estendidos para abranger o espaço recém-acrescentado. O NTFS continua a usar o mesmo mecanismo LCN que utiliza para um único disco físico, e o driver FtDisk fornece o mapeamento de um deslocamento de volume lógico para o deslocamento em um disco em particular.

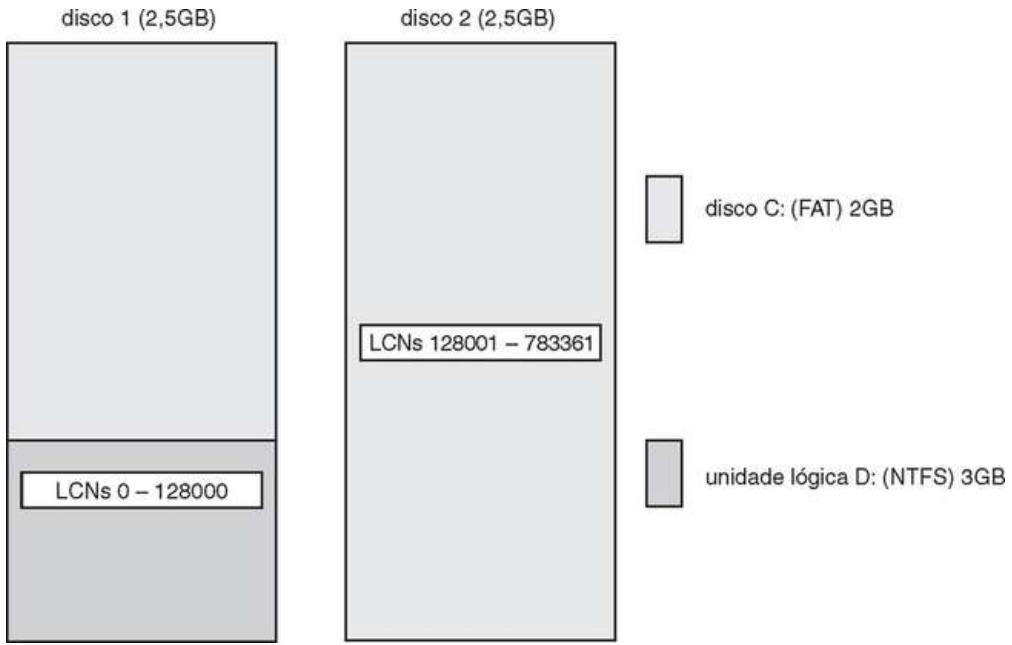


FIGURA 22.7 Conjunto de volumes em duas unidades.

22.5.4.2 Stripe set

Outra maneira de combinar várias partições físicas é intercalar seus blocos em um padrão de revezamento para formar um **stripe set**, como mostra a [Figura 22.8](#). Esse esquema também é chamado RAID nível 0, ou **disk stripe**. O FFDisk usa um tamanho de stripe de 64 KB: os primeiros 64 KB do volume lógico são armazenados na primeira partição física, os 64 KB seguintes na segunda partição física, e assim por diante, até cada partição ter contribuído com 64 KB de espaço. Depois, a alocação retorna ao primeiro disco, alocando o segundo bloco de 64 KB. Um stripe set forma um volume lógico grande, mas o esquema físico pode melhorar a largura de banda da E/S porque, para uma E/S grande, todos os discos podem transferir dados em paralelo.

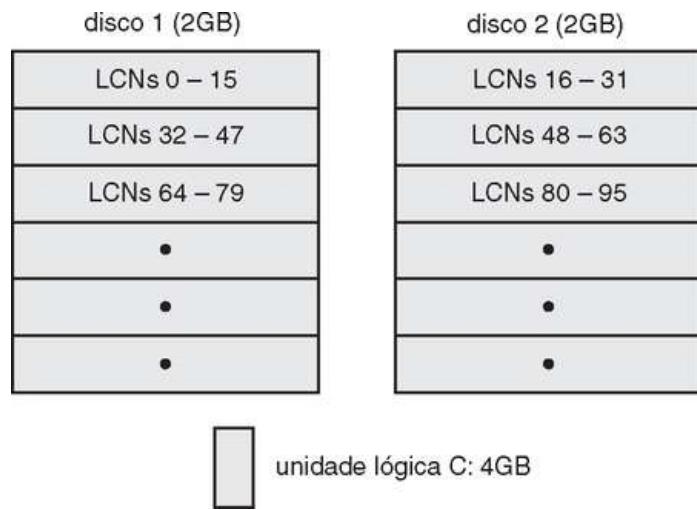


FIGURA 22.8 Stripe set em duas unidades.

22.5.4.3 Stripe set com paridade

Uma variação dessa ideia é o **stripe set com paridade**, que aparece na [Figura 22.9](#). Esse esquema também é denominado RAID nível 5. Suponha que o stripe set possua oito discos. Sete dos discos armazenarão stripes de dados, com um stripe de dados em cada disco, e o oitavo disco armazenará um stripe de paridade para cada stripe de dados. O stripe de paridade contém o resultado do XOR, byte a byte, dos stripes de dados. Se qualquer um dos oito stripes for destruído, o sistema poderá reconstruir os dados calculando o XOR dos sete restantes. Essa capacidade de reconstruir os dados torna o array de disco muito menos provável de perder dados no caso de uma falha de disco.

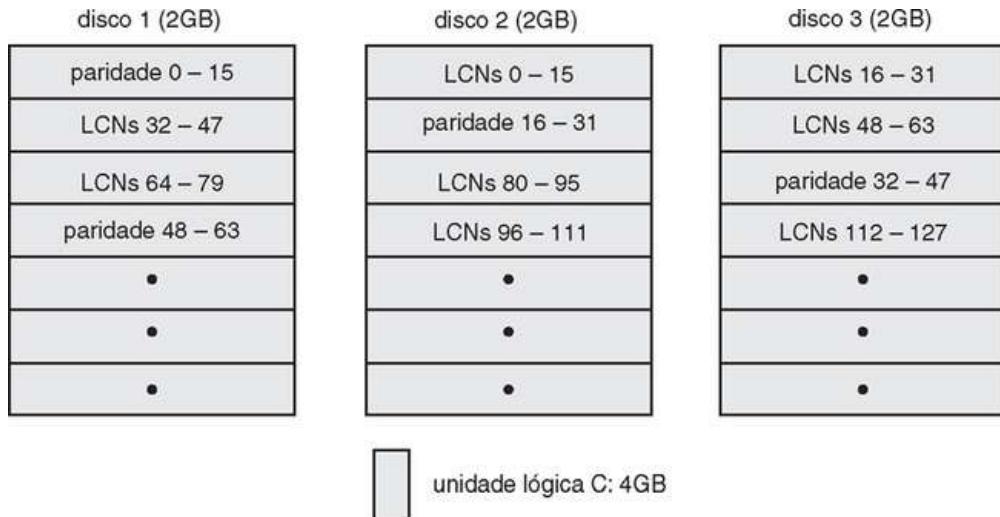


FIGURA 22.9 Stripe set com paridade em três unidades.

Observe que uma atualização em um stripe de dados também exige o novo cálculo do stripe de paridade. Sete escritas simultâneas em sete stripes de dados, portanto, também exigiriam a atualização de sete stripes de paridade. Se os stripes de paridade estivessem todos no mesmo disco, esse disco poderia ter sete vezes a carga de E/S dos discos de dados. Para evitar a criação desse gargalo, espalhamos os stripes de paridade por todos os discos, atribuindo-lhes no estilo de revezamento. Para montar um stripe set com paridade, precisamos no mínimo de três partições de mesmo tamanho, localizadas em três discos separados.

22.5.4.4 Espelhamento de disco

Um esquema ainda mais robusto é denominado **espelhamento de disco (disk mirroring)** ou RAID nível 1; ele é representado na [Figura 22.10](#). Um **mirror set (conjunto de espelhos)** compreende duas partições de mesmo tamanho em dois discos. Quando uma aplicação escreve dados em um mirror set, FdDisk escreve os dados nas duas partições. Se uma partição falhar, FdDisk possui outra cópia armazenada em segurança no espelho. Os mirror sets também podem melhorar o desempenho, pois as requisições de leitura podem ser divididas entre os dois mirrors, dando a cada um a metade da carga de trabalho. Para proteger contra a falha de um controlador de disco, podemos conectar os dois discos de um mirror set a dois controladores de disco separados. Esse arranjo é chamado de **duplex set (conjunto duplex)**.

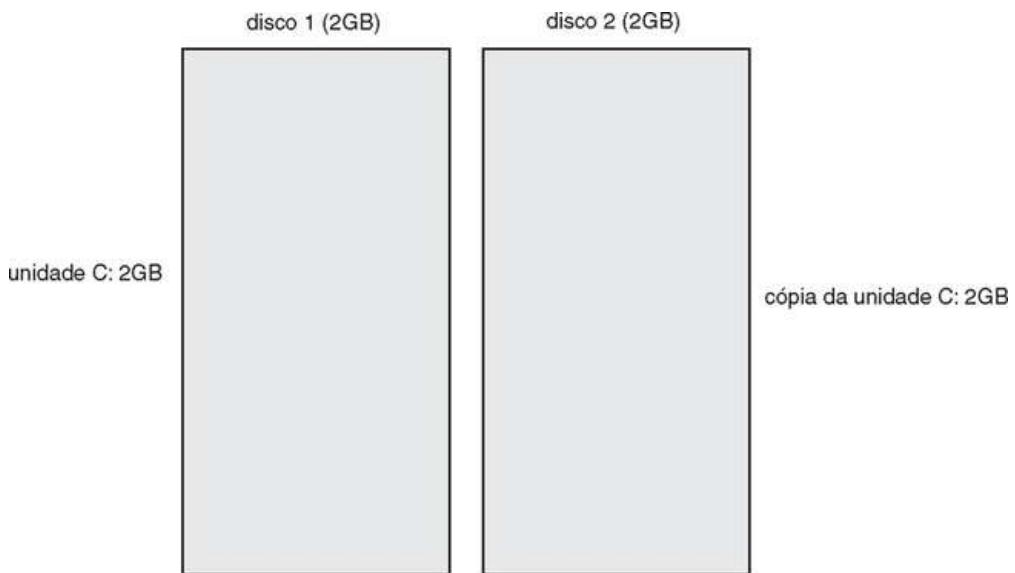


FIGURA 22.10 Mirror set em duas unidades.

22.5.4.5 Reserva de setores e remapeamento de clusters

Para lidar com setores de disco defeituosos, o FdDisk usa uma técnica de hardware chamada reserva de setores (Sector Sparing), e o NTFS utiliza uma técnica de software chamada remapeamento de

clusters (cluster remapping). A **reserva de setores** é uma capacidade do hardware fornecida por muitas unidades de disco. Quando uma unidade de disco é formatada, ela cria um mapa dos números de bloco lógicos para setores bons no disco. Ela também deixa setores extras não mapeados como reservas. Se um setor falhar, o Fdisk instrui a unidade de disco para substituí-lo por um reserva. O **remapeamento de clusters** é uma técnica de software realizada pelo sistema de arquivos. Se um bloco de disco apresentar defeito, o NTFS o substitui por um bloco diferente, não alocado, alterando quaisquer ponteiros afetados na MFT. O NTFS também anota que o bloco com defeito nunca deverá ser alocado a qualquer arquivo.

Quando um bloco de disco apresenta defeito, o resultado normal é a perda de dados. Mas a reserva de setores ou o remapeamento de clusters podem ser combinados com volumes tolerantes a falhas, para mascarar a falha de um bloco de disco. Se uma leitura falhar, o sistema reconstrói os dados que faltam lendo o espelho ou calculando a paridade XOR em um conjunto de faixas com paridade. Os dados reconstruídos são armazenados em um novo local, obtido pela reserva de setor ou pelo remapeamento de clusters.

22.5.5 Compactação e criptografia

O NTFS pode realizar compactação de dados sobre arquivos individuais ou sobre todos os arquivos de dados em um diretório. Para compactar um arquivo, o NTFS divide os dados do arquivo em **unidades de compactação**, que são blocos de 16 clusters contíguos. Quando uma unidade de compactação é escrita, um algoritmo de compactação de dados é aplicado. Se o resultado couber em menos de 16 clusters, a versão de compactação é armazenada. Na leitura, o NTFS pode determinar se os dados foram compactados; se tiverem sido, o tamanho da unidade de compactação armazenada é menor do que 16 clusters. Para melhorar o desempenho ao ler unidades de compactação contíguas, o NTFS realiza a busca prévia e descompacta antes das requisições da aplicação.

Para arquivos esparsos ou arquivos que contêm principalmente zeros, o NTFS utiliza outra técnica para economizar espaço. Os clusters que contêm zeros por nunca terem sido escritos não são alocados ou armazenados no disco. Em vez disso, lacunas são deixadas na sequência de números de clusters virtuais armazenados na entrada da MFT para o arquivo. Ao ler um arquivo, se o NTFS encontrar uma lacuna nos números de clusters virtuais, ele preenche com zero essa parte do buffer de quem chamou. Essa técnica também é usada pelo UNIX.

O NTFS aceita criptografia de arquivos. Arquivos individuais, ou diretórios inteiros, podem ser especificados para criptografia. O sistema de segurança gerencia as chaves utilizadas, e um serviço de recuperação de chave está disponível para recuperar chaves perdidas.

22.5.6 Pontos de montagem

Os pontos de montagem (mount points) são uma forma de link simbólico específico aos diretórios no NTFS. Eles fornecem um mecanismo para organizar volumes de disco que é mais flexível do que o uso de nomes globais (como letras de unidade). Um ponto de montagem é implementado como um link simbólico com dados associados, que contêm o nome verdadeiro do volume. Por fim, os pontos de montagem substituirão as letras de unidade, mas haverá uma longa transição, devido à dependência que muitas aplicações têm do esquema de letra de unidade.

22.5.7 Change Journal

O NTFS mantém um diário descrevendo todas as mudanças feitas no sistema de arquivos. Os serviços do modo usuário podem receber notificações de mudanças no diário e depois identificar quais arquivos foram alterados. O serviço de indexação de conteúdo usa o change journal para identificar arquivos que precisam ser reindexados. O serviço de replicação de arquivo o usa para identificar arquivos que precisam ser replicados pela rede.

22.5.8 Shadow copies do volume

O Windows XP implementa a capacidade de trazer um volume a um estado conhecido e depois criar uma shadow copy, que pode ser usada para o backup de uma visão coerente do volume. Fazer uma shadow copy de um volume é uma forma de cópia na escrita, onde os blocos modificados após a criação de uma shadow copy são armazenados em sua forma original na cópia. Para conseguir um estado consistente para o volume, é preciso haver cooperação das aplicações, pois o sistema não tem como saber quando os dados usados pela aplicação estão em um estado estável, do qual a aplicação poderia ser reiniciada com segurança.

A versão de servidor do Windows XP utiliza shadow copy para manter, de forma eficiente, versões antigas dos arquivos armazenados nos servidores de arquivo. Isso permite que os usuários vejam documentos armazenados nos servidores de arquivo conforme existiam em pontos anteriores no tempo. O usuário pode usar esse recurso para recuperar arquivos acidentalmente excluídos ou para

examinar uma versão anterior do arquivo, tudo sem puxar uma fita de backup.

22.6 Redes

O Windows XP admite redes peer-to-peer e redes cliente-servidor. Ele também possui facilidades para gerenciamento de rede. Os componentes da rede no Windows XP fornecem transporte de dados, comunicação entre processos, compartilhamento de arquivos por uma rede e a capacidade de enviar tarefas de impressão para impressoras remotas.

22.6.1 Interfaces de redes

Para descrever as redes no Windows XP, devemos mencionar primeiro duas das interfaces de rede internas, chamadas **especificação de interface de dispositivo de rede** (**Network Device Interface Specification - NDIS**) e **interface de driver de transporte** (**Transport Driver Interface - TDI**). A interface NDIS foi desenvolvida em 1989 pela Microsoft e pela 3Com para separar os adaptadores de rede dos protocolos de transporte, de modo que qualquer um pudesse ser alterado sem afetar o outro. O NDIS reside na interface entre as camadas de enlace de dados e rede no modelo ISO e permite que muitos protocolos operem por muitos adaptadores de rede diferentes. Em termos do modelo ISO, a TDI é a interface entre a camada de transporte (camada 4) e a camada de sessão (camada 5). Essa interface permite que qualquer componente da camada de sessão use qualquer mecanismo de transporte disponível. (Um raciocínio semelhante levou ao mecanismo de fluxos no UNIX.) A TDI admite transporte tanto baseado em conexão quanto sem conexão e possui funções para enviar qualquer tipo de dados.

22.6.2 Protocolos

O Windows XP implementa protocolos de transporte como drivers. Esses drivers podem ser carregados e descarregados do sistema dinamicamente, embora, na prática, o sistema tenha de ser reinicializado após uma mudança. O Windows XP vem com vários protocolos de rede. Em seguida, discutimos sobre diversos protocolos suportados no Windows XP para fornecer bastante funcionalidade de rede.

22.6.2.1 Bloco de mensagem do servidor

O protocolo de **bloco de mensagem do servidor** (**Server-MESSAGE-Block - SMB**) foi introduzido inicialmente no MS-DOS 3.1. O sistema usa o protocolo para enviar requisições de E/S pela rede. O protocolo SMB possui quatro tipos de mensagens. As mensagens Session control são comandos que iniciam e terminam uma conexão de redirecionador para um recurso compartilhado no servidor. Um redirecionador utiliza mensagens File para acessar arquivos no servidor. Mensagens Printer são usadas para enviar dados para uma fila de impressão remota e receber informações de status de volta, e a mensagem Message é usada para a comunicação com outra estação de trabalho. O protocolo SMB foi publicado como **sistema de arquivos comum da Internet** (**Common Internet File System - CIFS**) e é aceito em diversos sistemas operacionais.

22.6.2.2 Network Basic Input/Output System

O **sistema básico de entrada/saída da rede** (**Network Basic Input/Output System - NetBIOS**) é uma interface abstrata de hardware para redes, semelhante à interface abstrata de hardware BIOS, criada para PCs executando MS-DOS. O NetBIOS, desenvolvido no início da década de 1980, tornou-se uma interface-padrão para programação em rede. O NetBIOS é usado para estabelecer nomes lógicos na rede e conexões lógicas ou **sessões** entre dois nomes lógicos na rede e para dar suporte à transferência de dados confiável para uma sessão, por meio de solicitações NetBIOS ou SMB.

22.6.2.3 NetBIOS Extended User Interface

A **interface de usuário estendida do BIOS de rede** (**NetBIOS Extended User Interface - NetBEUI**) foi introduzida pela IBM em 1985 como um protocolo de rede simples e eficiente para até 254 máquinas. Ela é o protocolo-padrão para as redes peer-to-peer do Windows 95 e para o Windows for Workgroups. O Windows XP usa NetBEUI quando deseja compartilhar recursos com essas redes. Entre as limitações da NetBEUI estão que ela utiliza como endereço o nome real de um computador e não admite roteamento.

22.6.2.4 Transmission Control Protocol/Internet Protocol

A família de protocolos Transmission Control Protocol/Internet Protocol (TCP/IP) na Internet tornou-se a infraestrutura de rede-padrão de fato. O Windows XP usa TCP/IP para se conectar a uma grande variedade de sistemas operacionais e plataformas de hardware. A família de protocolos TCP/IP do Windows XP inclui protocolo simples de gerenciamento de rede (Simple Network-Management Protocol - SNMP), protocolo dinâmico para configuração de host (Dynamic Host-

Configuration Protocol - DHCP), serviços de nomes Internet do Windows (Windows Internet Name Service - WINS) e o suporte do NetBIOS.

22.6.2.5 Point-to-Point Tunneling Protocol

O **protocolo de tunelamento ponto a ponto (Point-to-Point Tunneling Protocol - PPTP)** é um protocolo fornecido pelo Windows XP para a comunicação entre os módulos de servidor de acesso remoto executando nas máquinas Windows XP Server e outros sistemas clientes conectados pela Internet. Os servidores de acesso remoto podem criptografar dados enviados pela conexão e admitem as **redes privadas virtuais (Virtual Private Networks - VPN)** de múltiplos protocolos pela Internet.

22.6.2.6 Protocolos Novell NetWare

Os protocolos Novell NetWare (serviço de datagrama IPX sobre a camada de transporte SPX) são bastante utilizados em LANs de PCs. O protocolo NWLink do Windows XP conecta as redes NetBIOS a NetWare. Em combinação com um redirecionador (como o Client Service for Netware da Microsoft ou o NetWare Client for Windows da Novell), esse protocolo permite que um cliente Windows XP se conecte a um servidor NetWare.

22.6.2.7 Web Distributed Authoring and Versioning Protocol

O Web Distributed Authoring and Versioning (WebDAV) é um protocolo baseado em http, para autoria colaborativa na rede. O Windows XP monta um redirecionador WebDAV no sistema de arquivos. Montar esse suporte diretamente no sistema de arquivos permite que o WebDAV trabalhe com outros recursos, como criptografia. Os arquivos pessoais agora podem ser armazenados com segurança em um local público.

22.6.2.8 Protocolo AppleTalk

O **protocolo AppleTalk** foi projetado como uma conexão de baixo custo pela Apple para permitir que os computadores Macintosh compartilhem arquivos. Os sistemas Windows XP podem compartilhar arquivos e impressoras com computadores Macintosh por meio de AppleTalk se um servidor Windows XP na rede estiver executando o pacote Windows Services for Macintosh.

22.6.3 Mecanismos de processamento distribuído

Embora o Windows XP não seja um sistema operacional distribuído, ele aceita aplicações distribuídas. Os mecanismos que dão suporte ao processamento distribuído no Windows XP incluem NetBIOS, canais nomeados e mailslots, sockets do Windows, RPCs, a Microsoft Interface Definition Language e COM.

22.6.3.1 NetBIOS

No Windows XP, aplicações NetBIOS podem se comunicar pela rede usando NetBEUI, NWLink ou TCP/IP.

22.6.3.2 Named pipes

Named pipes são um mecanismo de mensagens orientado a conexão. Foram desenvolvidos originalmente como uma interface de alto nível para conexões NetBIOS pela rede. Um processo também pode usar named pipes para se comunicar com outros processos na mesma máquina. Como os named pipes são acessados por meio da interface do sistema de arquivos, os mecanismos de segurança usados para objetos de arquivo também se aplicam a named pipes.

O formato dos nomes de pipe segue a **convenção de nome uniforme (Uniform Naming Convention - UNC)**. Um nome UNC se parece com um nome de arquivo remoto típico. O formato de um nome UNC é \\nome_servidor\name_compartilhamento\x\y\z, onde o nome_servidor identifica um servidor na rede; nome_compartilhamento identifica qualquer recurso que se torna disponível aos usuários da rede, como diretórios, arquivos, named pipes e impressoras, e a parte \x\y\z é um nome de caminho de arquivo normal.

22.6.3.3 Mailslots

Mailslots são um mecanismo de mensagem sem conexão. Eles não são confiáveis quando acessados pela rede porque uma mensagem enviada a um mailslot pode se perder antes que o destinatário desejado a receba. Mailslots são usados para aplicações de broadcast, como localizar componentes na rede; eles também são usados pelo serviço Windows Computer Browser.

22.6.3.4 Winsock

Winsock é a API de sockets do Windows XP. O Winsock é uma interface da camada de sessão bastante compatível com sockets UNIX, mas tem algumas extensões do Windows XP acrescentadas.

Ele fornece uma interface padronizada para muitos protocolos de transporte que podem ter diferentes esquemas de endereçamento, de modo que qualquer aplicação Winsock pode executar em qualquer pilha de protocolo compatível com Winsock.

22.6.3.5 Remote procedure call

Uma **remote procedure call (RPC)** é um mecanismo cliente-servidor que permite que uma aplicação em uma máquina faça uma chamada de procedimento ao código em outra máquina. O cliente chama um procedimento local - uma **rotina stub** - que empacota seus argumentos em uma mensagem e os envia pela rede até um processo servidor específico. A rotina stub no cliente, em seguida, é bloqueada. Enquanto isso, o servidor desempacota a mensagem, chama o procedimento, empacota os resultados do retorno em uma mensagem e os envia de volta ao stub do cliente. O stub do cliente é desbloqueado, recebe a mensagem, desempacota os resultados da RPC e os retorna a quem chamou. Esse empacotamento de argumentos às vezes é chamado de **marshalling**.

O mecanismo de RPC do Windows XP segue o padrão de ambiente de computação distribuído, bastante utilizado para as mensagens de RPC, de modo que os programas escritos para usar RPCs do Windows XP são altamente portáveis. O padrão RPC é detalhado. Ele esconde muitas das diferenças arquitetônicas entre os computadores, como os tamanhos dos números binários e a ordem dos bytes e bits nas palavras do computador, especificando formatos de dados-padrão para mensagens de RPC.

O Windows XP pode enviar mensagens de RPC usando NetBIOS, ou Winsock em redes TCP/IP, ou canais nomeados em redes LAN Manager. A facilidade LPC, discutida anteriormente, é semelhante à RPC, exceto que, no caso da LPC, as mensagens são passadas entre dois processos em execução no mesmo computador.

22.6.3.6 Microsoft Interface Definition Language

É cansativo e passível de erros escrever o código para empacotar e transmitir argumentos no formato-padrão, desempacotar e executar o procedimento remoto, empacotar e enviar os resultados do retorno e desempacotá-los e retorná-los à rotina que chamou. Entretanto, felizmente, grande parte desse código pode ser gerada de forma automática a partir de uma simples descrição de argumentos e resultados de retorno.

O Windows XP fornece a **Microsoft Interface Definition Language** para descrever nomes, argumentos e resultados do procedimento remoto. O compilador para essa linguagem gera arquivos de cabeçalho que declararam os stubs para os procedimentos remotos, assim como os tipos de dados para os argumentos e mensagens do valor de retorno. Ele também gera o código-fonte para as rotinas stub usadas no lado cliente e também para o desempacotador e despachante no lado do servidor. Quando a aplicação é vinculada, as rotinas de stub são incluídas. Quando a aplicação executa o stub RPC, o código gerado cuida do resto.

22.6.3.7 Component Object Model

O **modelo de objeto (comum) Component Object Model - COM** é um mecanismo para comunicação entre processos desenvolvidos para o Windows. Objetos COM fornecem uma interface bem definida para manipular os dados no objeto. Por exemplo, o COM é a infraestrutura usada pela tecnologia de **link e incorporação de objetos (Object Linking and Embedding - OLE)** para inserir planilhas em documentos do Word. O Windows XP possui uma extensão distribuída, chamada **DCOM**, que pode ser usada por uma rede utilizando RPC, para fornecer um método transparente de desenvolvimento de aplicações distribuídas.

22.6.4 Redirecionadores e servidores

No Windows XP, uma aplicação pode usar a API de E/S do Windows XP para acessar arquivos a partir de um computador remoto como se fossem locais, desde que o computador remoto esteja executando um servidor CIFS, como o fornecido pelo Windows XP ou sistemas Windows anteriores. Um **redirecionador** é o objeto no cliente que encaminha requisições de E/S aos arquivos remotos, onde são satisfeitas por um servidor. Por desempenho e segurança, os redirecionadores e servidores executam no modo kernel. Mais detalhadamente, o acesso a um arquivo remoto acontece da seguinte maneira:

1. A aplicação chama o gerenciador de E/S para requisitar que um arquivo seja aberto com um nome de arquivo no formato UNC padrão.
2. O gerenciador de E/S monta um pacote de requisição de E/S, conforme descrevemos na Seção 22.3.3.5.
3. O gerenciador de E/S reconhece que o acesso é para um arquivo remoto e chama um driver denominado **provedor múltiplo de convenção universal de nomes (Multiple Universal-naming-convention Provider - MUP)**.
4. O MUP envia o pacote de requisição de E/S de forma assíncrona a todos os redirecionadores registrados.

5. Um redirecionador que possa satisfazer a requisição responde ao MUP. Para evitar a mesma pergunta a todos os redirecionadores no futuro, o MUP usa um cache para lembrar qual redirecionador pode tratar desse arquivo.
6. O redirecionador envia a requisição da rede ao sistema remoto.
7. Os drivers de rede do sistema remoto recebem a requisição e a passam para o driver do servidor.
8. O driver do servidor entrega a requisição ao driver apropriado no sistema de arquivos local.
9. O driver de dispositivos apropriado é chamado para acessar os dados.
10. Os resultados são retornados ao driver do servidor, que envia os dados de volta para o redirecionador requisitante. O redirecionador, em seguida, retorna os dados à aplicação que chamou por meio do gerenciador de E/S.

Acontece um processo semelhante para aplicações que usam a API de rede API Win32, em vez dos serviços UNC, exceto que um módulo, denominado roteamento de provedor múltiplo, é utilizado no lugar de um MUP.

Por questão de portabilidade, os redirecionadores e servidores utilizam a API TDI para o transporte da rede. As requisições em si são expressas em um protocolo de nível mais alto, que como padrão é o protocolo SMB, mencionado na [Seção 22.6.2](#). A lista de redirecionadores é mantida no banco de dados de registro do sistema.

22.6.4.1 Sistema de arquivos distribuído

Os nomes UNC nem sempre são convenientes, pois vários servidores de arquivos podem estar disponíveis para atender ao mesmo conteúdo, e os nomes UNC incluem explicitamente o nome do servidor. O Windows XP admite um protocolo de **sistema de arquivos distribuído (Distributed File System - DFS)** que permite a um administrador de rede enviar arquivos de vários servidores usando um único espaço de nomes distribuído.

22.6.4.2 Redirecionamento de pastas e caching no cliente

Para melhorar a experiência do PC para usuários de empresas, que constantemente trocam de computador, o Windows XP permite que os administradores deem aos usuários **roaming profiles**, que mantêm as preferências do usuário e outras configurações nos servidores. O **redirecionamento de pastas** é utilizado para armazenar automaticamente os documentos de um usuário e outros arquivos em um servidor. Isso funciona bem até um dos computadores não estar mais conectado à rede, como quando um usuário leva um laptop para um avião. Para dar aos usuários acesso off-line a seus arquivos redirecionados, o Windows XP utiliza o **caching no cliente (Client-Side Caching - CSC)**. O CSC é usado quando o usuário estiver on-line, para manter cópias dos arquivos do servidor na máquina local, a fim de melhorar o desempenho. Os arquivos são levados para o servidor quando forem alterados. Se o computador estiver desconectado, os arquivos ainda estarão disponíveis, e a atualização do servidor será adiada até a próxima vez que o computador estiver on-line.

22.6.5 Domínios

Muitos ambientes em rede possuem grupos naturais de usuários, como alunos em um laboratório de computação na escola ou funcionários em um departamento de uma empresa. Constantemente, queremos que todos os membros do grupo sejam capazes de acessar recursos compartilhados em seus diversos computadores no grupo. Para controlar os direitos de acesso globais dentro desses grupos, o Windows XP utiliza o conceito de domínio. Antes, esses domínios não tinham qualquer relacionamento com o DNS (sistema de nomes de domínio) que mapeia os nomes de host da Internet aos endereços IP. Contudo, agora, eles estão bastante relacionados.

Especificamente, um domínio do Windows XP é um grupo de estações de trabalho e servidores do Windows XP que compartilham uma política de segurança e um banco de dados de usuário comuns. Como o Windows XP agora usa o protocolo Kerberos para relacionamento de confiança e autenticação, um domínio do Windows XP é a mesma coisa que um domínio Kerberos. As versões anteriores do NT usavam a ideia de controladores de domínio principal e de backup; agora, todos os servidores em um domínio são controladores de domínio. Além disso, as versões anteriores exigiam a configuração de relacionamentos de confiança unidirecionais entre os domínios. O Windows XP utiliza um enfoque hierárquico baseado no DNS e permite relacionamentos de confiança transitivos, que podem subir e descer na hierarquia. Esse enfoque reduz a quantidade de relacionamentos de confiança exigidos para n domínios de $n * (n - 1)$ para $O(n)$. As estações de trabalho no domínio confiam no controlador de domínio para prestar informações corretas sobre os direitos de acesso de cada usuário (por meio do token de acesso do usuário). No entanto, todos os usuários retêm a capacidade de restringir o acesso às suas próprias estações de trabalho, não importa o que o controlador de domínio possa dizer.

22.6.5.1 Árvores e florestas de domínios

Como uma empresa pode ter muitos departamentos e uma escola pode ter muitas turmas, normalmente é preciso gerenciar múltiplos domínios dentro de uma única organização. Uma **árvore de domínios** é uma hierarquia de nomes de DNS contígua. Por exemplo, *bell-labs.com* poderia ser a raiz da árvore, com *research.bell-labs.com* e *pez.bell-labs.com* como domínios filhos *research* e *pez*. Uma **floresta** é um conjunto de nomes não contíguos. Um exemplo seria a árvore *bell-labs.com* e/ou *lucent.com*. Entretanto, uma floresta pode ser composta de uma única árvore de domínios.

22.6.5.2 Relacionamentos de confiança

Os relacionamentos de confiança podem ser configurados entre domínios de três maneiras: unidirecional, transitivo e enlace cruzado. As versões do NT até a versão 4.0 só permitiam relacionamentos de confiança unidirecionais. Um **relacionamento de confiança unidirecional** é exatamente o que seu nome sugere: o domínio A é informado de que pode confiar no domínio B. Todavia, B não confiaria em A, a menos que outro relacionamento fosse configurado. Sob um **relacionamento de confiança transitivo**, se A confia em B e B confia em C, então A, B e C confiam um no outro, pois os relacionamentos de confiança transitivos são bidirecionais como padrão. Os relacionamentos de confiança transitivos são ativados como padrão para novos domínios em uma árvore e só podem ser configurados entre os domínios dentro de uma floresta. O terceiro tipo, um **relacionamento de confiança de enlace cruzado**, é útil para reduzir o tráfego de autenticação. Suponha que os domínios A e B sejam nós de folha e que os usuários em A normalmente utilizem os recursos em B. Se um relacionamento de confiança transitivo for utilizado, as solicitações de autenticação precisarão atravessar até o ancestral comum dos dois nós de folha; mas se A e B tiverem um relacionamento de confiança de enlace cruzado, as autenticações serão enviadas diretamente para o outro nó.

22.6.6 Active Directory

Active Directory é a implementação no Windows XP dos serviços do **protocolo leve de acesso ao diretório (Lightweight Directory-Access Protocol - LDAP)**. O Active Directory armazena as informações de topologia sobre o domínio, mantém as contas e senhas de usuário e grupo baseadas em domínio e fornece um armazenamento baseado em domínio para tecnologias como **políticas de grupo** e **intellimirror**.

Os administradores utilizam políticas de grupo para estabelecer padrões uniformes para preferências e software desktop. Para muitos grupos de tecnologia de informação corporativos, a uniformidade reduz drasticamente o custo da computação. O intellimirror é usado em conjunto com políticas de grupo, para especificar que software deverá estar disponível a cada classe de usuário, mesmo instalando-o de maneira automática por demanda de um servidor corporativo.

22.6.7 Tradução de nomes nas redes TCP/IP

Em uma rede IP, a **tradução de nomes** é o processo de converter um nome de computador para um endereço IP, como na tradução de www.bell-labs.com para 135.104.1.14. O Windows XP fornece vários métodos de resolução de nome, incluindo serviços de nomes Internet do Windows (Windows Internet Name Service - WINS), tradução de nomes de broadcast, sistema de nomes de domínio (Domain Name System - DNS), um arquivo de hosts e um arquivo LMHOSTS. A maior parte desses métodos é utilizada por muitos sistemas operacionais, de modo que descreveremos apenas o WINS aqui.

Sob o WINS, dois ou mais servidores WINS mantêm um banco de dados dinâmico de links de endereços de nome para IP, e o software cliente para consultar os servidores. Pelo menos dois servidores são usados, de modo que o serviço WINS possa sobreviver a uma falha do servidor e a carga de trabalho da tradução de nomes possa se espalhar por várias máquinas.

O WINS utiliza o protocolo dinâmico para configuração de host (Dynamic Host-Configuration Protocol - DHCP). O DHCP atualiza configurações de endereço automaticamente no banco de dados WINS, sem intervenção do usuário ou do administrador, da seguinte maneira. Quando um cliente DHCP é iniciado, ele transmite uma mensagem discover por broadcast. Cada servidor DHCP que recebe a mensagem responde com uma mensagem offer, que contém um endereço IP e informações de configuração para o cliente. O cliente escolhe uma das configurações e envia uma mensagem request ao servidor DHCP selecionado. O servidor DHCP responde com o endereço IP e informações de configuração que informou anteriormente e com uma **locação** para esse endereço. A locação dá ao cliente o direito de usar o endereço IP por um período especificado. Quando o tempo de locação estiver pela metade, o cliente tenta renovar a locação para o endereço. Se a locação não for renovada, o cliente precisa obter uma nova.

22.7 Interface do programador

A API Win32 é a interface fundamental para as capacidades do Windows XP. Esta seção descreve cinco aspectos principais da API Win32: acesso aos objetos do kernel, compartilhamento de objetos entre os processos, gerência de processos, comunicação entre processos e gerência de memória.

22.7.1 Acesso aos objetos do kernel

O kernel do Windows XP fornece muitos serviços que os programas de aplicação podem utilizar. Os programas de aplicação obtêm esses serviços manipulando objetos do kernel. Um processo recebe acesso a um objeto do kernel chamado xxx chamando a função CreateXXX para abrir um descritor para xxx. Esse descritor é exclusivo do processo. Dependendo de qual objeto está sendo objeto, se a função Create() falhar, ela pode retornar 0 ou uma constante especial, chamada INVALID_HANDLE_VALUE. Um processo pode fechar qualquer descritor chamando a função CloseHandle(), e o sistema pode excluir o objeto se a contagem de processos que utilizam o objeto cair para 0.

22.7.2 Compartilhando objetos entre processos

O Windows XP fornece três maneiras de compartilhar objetos entre processos. A primeira delas é quando um processo filho herda um descritor para o objeto. Quando o pai chama a função CreateXXX, ele fornece uma estrutura SECURITIES_ATTRIBUTES com o campo bInheritHandle definido como TRUE. Esse campo cria um descritor que pode ser herdado. Em seguida, o processo filho é criado, passando um valor TRUE para o argumento bInheritHandle da função CreateProcess(). A [Figura 22.11](#) mostra um código que demonstra a criação de um descritor de semáforo herdado por um processo filho.

```
SECURITY_ATTRIBUTES sa;
sa.nlength = sizeof(sa);
sa.lpSecurityDescriptor = NULL;
sa.bInheritHandle = TRUE;
Handle a_semaphore = CreateSemaphore(&sa,1,1,NULL);
char command_line[132];
ostringstream ostrstream(command_line, sizeof(command_line));
ostringstream << a_semaphore << ends;
CreateProcess("outro_processo.exe",command_line,
    NULL,NULL,TRUE,...);
```

FIGURA 22.11 Código para um filho compartilhar um objeto herdando um descritor.

Supondo que o processo filho saiba quais descritores são compartilhados, o pai e o filho podem conseguir a comunicação entre processos por meio dos objetos compartilhados. No exemplo da [Figura 22.11](#), o processo filho apanha o valor do descritor no primeiro argumento da linha de comandos e depois compartilha o semáforo com o processo pai.

A segunda maneira de compartilhar objetos é quando um processo dá um nome ao objeto quando ele for criado, e o segundo processo abre o nome. Esse método possui duas desvantagens: o Windows XP não fornece um meio de verificar se um objeto com o nome escolhido já existe, e o espaço de nomes do objeto é global, sem considerar o tipo do objeto. Por exemplo, duas aplicações poderão criar um objeto chamado *canal* quando dois objetos distintos - e possivelmente diferentes - são desejados.

Os objetos nomeados possuem a vantagem de que processos não relacionados podem compartilhá-los com facilidade. O primeiro processo chama uma das funções CreateXXX e fornece um nome no parâmetro lpszName. O segundo processo apanha um descritor para compartilhar o objeto chamando OpenXXX (ou CreateXXX) com o mesmo nome, como mostramos no exemplo da [Figura 22.12](#).

```

// Processo A
. . .
HANDLE a_semaphore = CreateSemaphore(NULL,1,1,"MeuSEM1");
. . .

// Processo B
. . .
HANDLE b_semaphore = OpenSemaphore(SEMAPHORE_ALL_ACCESS,
    FALSE, "MeuSEM1");

```

FIGURA 22.12 Código para compartilhar um objeto por pesquisa de nome.

Uma terceira forma de compartilhar objetos é por meio da função `DuplicateHandle()`. Esse método exige que algum outro método de comunicação entre processos passe o descritor duplicado. Dado um descritor para um processo e o valor de um descritor dentro desse processo, um segundo processo pode obter um descritor para o mesmo objeto e, desse modo, compartilhá-lo. Um exemplo desse método aparece na [Figura 22.13](#).

```

// Processo A deseja dar ao Processo B acesso a um semáforo

// Processo A
HANDLE a_semaphore = CreateSemaphore(NULL,1,1,NULL);
// envia o valor do semáforo para o Processo B
// usando uma mensagem ou memória compartilhada
. . .

// Processo B
HANDLE process_a = OpenProcess(PROCESS_ALL_ACCESS, FALSE,
    process_id_of_A);
HANDLE b_semaphore;
DuplicateHandle(process_a,a_semaphore,
    GetCurrentProcess( ),&b_semaphore,
    0,FALSE,DUPPLICATE_SAME_ACCESS);
// usa b_semaphore para acessar o semáforo

```

FIGURA 22.13 Código para compartilhar um objeto passando um descritor.

22.7.3 Gerência de processos

No Windows XP, um **processo** é uma instância em execução de uma aplicação, e uma **thread** é uma unidade de código que pode ser agendada pelo sistema operacional. Assim, um processo contém uma ou mais threads. Um processo é iniciado quando algum outro processo chama a rotina `CreateProcess()`. Essa rotina carrega quaisquer DLLs usadas pelo processo e cria uma **thread primária**. Threads adicionais podem ser criadas pela função `CreateThread()`. Cada thread é criada com sua própria pilha, cujo padrão é de 1 MB, a menos que seja especificado de outra forma em um argumento de `CreateThread()`. Como algumas funções C em tempo de execução mantêm o estado em variáveis estáticas, como `errno`, uma aplicação multithread precisa se proteger contra o acesso não sincronizado. A função empacotadora `beginthreadex()` fornece o sincronismo apropriado.

22.7.3.1 Descritores de instância

Cada biblioteca de vínculo dinâmico (Dynamic Link Library - DLL) ou arquivo executável carregado no espaço de endereços de um processo é identificado por um **descritor de instância**. O valor do descritor de instância, na realidade, é o endereço virtual onde o arquivo é carregado. Uma aplicação pode apanhar o descritor para um módulo em seu espaço de endereço passando o nome do módulo para `GetModuleHandle()`. Se `NULL` for passado como nome, o endereço de base do processo é retornado. Os 64 KB inferiores do espaço de endereços não são usados, de modo que um programa

com falhas que tente desreferenciar um ponteiro NULL receberá um erro por violação de acesso.

As prioridades no ambiente API Win32 são baseadas no modelo de escalonamento do Windows XP, mas nem todos os valores de prioridade podem ser escolhidos. A API Win32 utiliza quatro classes de prioridade:

1. IDLE_PRIORITY_CLASS (nível de prioridade 4).
2. NORMAL_PRIORITY_CLASS (nível de prioridade 8).
3. HIGH_PRIORITY_CLASS (nível de prioridade 13).
4. REALTIME_PRIORITY_CLASS (nível de prioridade 24).

Os processos normalmente são membros da NORMAL_PRIORITY_CLASS, a menos que o pai do processo seja da IDLE_PRIORITY_CLASS ou outra classe tenha sido especificada quando `CreateProcess` foi chamado. A classe de prioridade de um processo pode ser mudada com a função `SetPriorityClass()` ou por um argumento sendo passado ao comando `START`. Por exemplo, o comando `START/REALTIME cbsrvr.exe` executaria o programa `cbsrvr` na REALTIME_PRIORITY_CLASS. Somente os usuários com o privilégio para *aumentar prioridade de escalonamento* podem mover um processo para a REALTIME_PRIORITY_CLASS. Administradores e usuários especialistas possuem esse privilégio como padrão.

22.7.3.2 Regra de escalonamento

Quando um usuário estiver executando em um programa interativo, o sistema precisa fornecer um desempenho especialmente bom para o processo. Por esse motivo, o Windows XP possui uma regra de escalonamento especial para os processos na NORMAL_PRIORITY_CLASS. O Windows XP faz distinção entre o processo de primeiro plano selecionado atualmente na tela e os processos de segundo plano que não estão selecionados atualmente. Quando um processo passa para o primeiro plano, o Windows XP aumenta o quantum de escalonamento por algum fator - normalmente, por 3. (Esse fator pode ser alterado por meio da opção de desempenho na seção do sistema do painel de controle.) Esse aumento dá ao processo de primeiro plano três vezes mais tempo para executar antes de ocorrer uma preempção de tempo compartilhado.

22.7.3.3 Prioridades de thread

Uma thread começa com uma prioridade inicial determinada por sua classe. A prioridade pode ser alterada pela função `SetThreadPriority()`. Essa função apanha um argumento que especifica uma prioridade relativa à prioridade de base de sua classe:

- THREAD_PRIORITY_LOWEST: base - 2.
- THREAD_PRIORITY_BELOW_NORMAL: base - 1.
- THREAD_PRIORITY_NORMAL: base + 0.
- THREAD_PRIORITY_ABOVE_NORMAL: base + 1.
- THREAD_PRIORITY_HIGHEST: base + 2.

Duas outras designações também são usadas para ajustar a prioridade. Lembre-se, como vimos na Seção 22.3.2.1, de que o kernel possui duas classes de prioridade: 16-31 para a classe de tempo real e 0-15 para a classe de prioridade variável. `THREAD_PRIORITY_IDLE` define a prioridade como 16 para threads de tempo real e como 1 para threads de prioridade variável. `THREAD_PRIORITY_TIME_CRITICAL` define a prioridade como 31 para threads de tempo real e como 15 para threads de prioridade variável.

Conforme discutimos na Seção 22.3.2.1, o kernel ajusta a prioridade de uma thread dinamicamente, dependendo de a thread ser I/O bound ou CPU bound. A API Win32 fornece um método para desativar esse ajuste, por meio das funções `SetProcessPriorityBoost()` e `SetThreadPriorityBoost()`.

22.7.3.4 Sincronismo de thread

Uma thread pode ser criada em um **estado suspenso**: a thread não é executada até outra thread torná-la legível por meio da função `ResumeThread()`. A função `SuspendThread()` faz o contrário. Essas funções definem um contador, de modo que, se uma thread for suspensa duas vezes, ela precisa ser retomada duas vezes antes de poder executar. Para sincronizar o acesso simultâneo aos objetos compartilhados pelas threads, o kernel fornece objetos de sincronismo, como semáforos e mutexes.

Além disso, o sincronismo das threads pode ser conseguido usando as funções `WaitForSingleObject()` ou `WaitForMultipleObjects()`. Outro método de sincronismo na API Win32 é a seção crítica. Uma seção crítica é uma região de código sincronizada que só pode ser executada por uma thread de cada vez. Uma thread estabelece uma seção crítica chamando `InitializeCriticalSection()`. A aplicação precisa chamar `EnterCriticalSection()` antes de entrar na seção crítica e `LeaveCriticalSection()` depois de sair. Essas duas rotinas garantem que, se várias threads tentarem entrar na seção crítica ao mesmo tempo, somente uma thread de cada vez terá permissão para prosseguir, e as outras esperarão na rotina `EnterCriticalSection()`. O mecanismo de seção crítica é mais rápido do que o uso de objetos de sincronismo do kernel, pois evita a alocação de objetos do kernel até encontrar contenção para a seção crítica.

22.7.3.5 Fibers

Uma **fiber** é um código no modo usuário escalonado de acordo com um algoritmo de escalonamento definido pelo usuário. Um processo pode ter várias fibers, assim como pode ter várias threads. Uma diferença importante entre as threads e as fibers é que as threads podem ser executadas concorrentemente, mas somente uma fiber de cada vez tem permissão para executar, até mesmo em hardware multiprocessador. Esse mecanismo foi incluído no Windows XP para facilitar o transporte daquelas aplicações UNIX legadas escritas para um modelo de execução de fiber.

O sistema cria uma fibra chamando `ConvertThreadToFiber()` ou `CreateFiber()`. A principal diferença entre essas funções é que `CreateFiber()` não inicia a execução da fibra criada. Para iniciar a execução, a aplicação precisa chamar `SwitchToFiber()`. A aplicação pode terminar uma fibra chamando `DeleteFiber()`.

22.7.3.6 Banco de threads

A criação/exclusão repetida de threads pode ser dispendiosa para aplicações e serviços que realizam pequenas quantidades de trabalho em cada instanciação. O banco de threads fornece programas no modo usuário com três serviços: uma fila à qual as requisições de trabalho podem ser submetidas (por meio da API `QueueUserWorkItem()`), uma API que pode ser usada para vincular callbacks a descritores passíveis de espera (`RegisterWaitForSingleObject()`) e APIs para vincular callbacks a tempos-limite (`CreateTimerQueue()` e `CreateTimerQueueTimer()`).

O objetivo do banco de threads é aumentar o desempenho. As threads são relativamente dispendiosas, e um processador só pode executar uma coisa de cada vez, não importa quantas threads sejam usadas. O banco de threads tenta reduzir a quantidade de threads pendentes adiando ligeiramente as requisições de trabalho (reutilizando cada thread para muitas requisições), enquanto fornece threads suficientes para utilizar com eficiência as CPUs da máquina. As APIs de espera e o callback de temporizador permitem que o banco de threads reduza ainda mais a quantidade de threads em um processo, usando muito menos threads do que seriam necessárias se um processo tivesse de dedicar uma thread para atender a cada descritor passível de espera ou tempo-limite.

22.7.4 Comunicação entre processos

As aplicações API Win32 tratam da comunicação entre processos de diversas maneiras. Uma maneira é compartilhando objetos do kernel. Outra é trocar mensagens, uma técnica particularmente popular para aplicações GUI do Windows. Uma thread pode enviar uma mensagem para outra thread ou para uma janela chamando `PostMessage()`, `PostThreadMessage()`, `SendMessage()`, `SendThreadMessage()` ou `SendMessageCallback()`. A diferença entre *postar* uma mensagem e *enviar* uma mensagem é que as rotinas de postagem são assíncronas: elas retornam imediatamente, e a thread que chama não sabe quando a mensagem será entregue. As rotinas de envio são síncronas: elas ficam bloqueadas por quem chamou até a mensagem ter sido entregue e processada.

Além de enviar uma mensagem, uma thread também pode enviar dados com a mensagem. Como os processos possuem espaços de endereço separados, os dados precisam ser copiados. O sistema os copia chamando `SendMessage()` para enviar uma mensagem do tipo `WM_COPYDATA` com uma estrutura de dados `COPYDATASTRUCT` que contém o tamanho e o endereço dos dados a serem transferidos. Quando a mensagem é enviada, o Windows XP copia os dados para um novo bloco de memória e dá o endereço virtual do novo bloco ao processo receptor.

Ao contrário das threads no ambiente Windows de 16 bits, cada thread da API Win32 possui sua própria fila de entrada, da qual a thread recebe mensagens. (Toda a entrada é recebida por meio de mensagens.) Essa estrutura é mais confiável do que a fila de entrada compartilhada de janelas de 16 bits porque, com filas separadas, não é mais possível para uma aplicação presa bloquear a entrada para as outras aplicações. Se uma aplicação API Win32 não chamar `GetMessage()` para tratar de eventos em sua fila de entrada, a fila se enche, e depois de 5 segundos o sistema marca a aplicação como “não respondendo”.

22.7.5 Gerência de memória

A API Win32 fornece várias maneiras para uma aplicação usar a memória: memória virtual, arquivos mapeados na memória e armazenamento local a thread.

22.7.5.1 Memória virtual

Uma aplicação chama `VirtualAlloc()` para reservar ou confirmar a memória virtual e `VirtualFree()` para cancelar ou liberar a memória. Essas funções permitem que a aplicação especifique o endereço virtual em que a memória é alocada. Elas operam sobre múltiplos do tamanho de página da memória, e o endereço inicial de uma região alocada precisa ser maior do que `0x10000`. Alguns exemplos dessas funções aparecem na [Figura 22.14](#).

```

// aloca 16 MB no topo do nosso espaço de endereços
void *buf = VirtualAlloc(0,0x1000000,MEM_RESERVE | MEM_TOP_DOWN,
    PAGE_READWRITE);
// confirma os 8 MB superiores do espaço alocado
VirtualAlloc(buf + 0x800000,0x800000,MEM_COMMIT,PAGE_READWRITE);
// do something with the memory
. .
// agora desconfirma a memória
VirtualFree(buf + 0x800000,0x800000,MEM_DECOMMIT);
// libera todo o espaço de endereços virtuais alocado
VirtualFree(buf,0,Mem_RELEASE);

```

FIGURA 22.14 Fragmentos de código para alocar memória virtual.

Um processo pode bloquear algumas de suas páginas confirmadas na memória física chamando `VirtualLock()`. O número máximo de páginas que um processo pode bloquear é 30, a menos que o processo primeiro chame `SetProcessWorkingSetSize()` para aumentar o tamanho máximo do conjunto de trabalho.

22.7.5.2 Arquivos mapeados na memória

Outra maneira de uma aplicação usar a memória é pelo mapeamento de memória de um arquivo com seu espaço de endereços. O mapeamento de memória também é uma forma conveniente de como dois processos podem compartilhar memória: os dois processos mapeiam o mesmo arquivo em sua memória virtual. O mapeamento de memória é um processo em estágios múltiplos, como você pode ver no exemplo da [Figura 22.15](#).

```

// abre o arquivo ou cria, se não existir
HANDLE hfile = CreateFile("algum arquivo",GENERIC_READ | GENERIC_WRITE,
    FILE_SHARE_READ | FILE_SHARE_WRITE,NULL,
    OPEN_ALWAYS, FILE_ATTRIBUTE_NORMAL, NULL);
// cria o arquivo mapeando 8 MB em tamanho
HANDLE hmap = CreateFileMapping(hfile,PAGE_READWRITE,
    SEC_COMMIT,0,0x800000,"SHM_1");
// agora obtém uma visão do espaço mapeado
void *buf = MapViewOfFile(hmap,FILE_MAP_ALL_ACCESS,0,0,0, 0x800000);
// faz alguma coisa com o arquivo mapeado
. .
// agora desmapeia o arquivo
UnMapViewOfFile(buf);
CloseHandle(hmap);
CloseHandle(hfile);

```

FIGURA 22.15 Código para variável dinâmica no armazenamento local a thread.

Se um processo quiser mapear algum espaço de endereços só para compartilhar uma região da memória com outro processo, nenhum arquivo é necessário. O processo chama `CreateFileMapping()` com um descritor de arquivo `0xffffffff` e um tamanho específico. O objeto de mapeamento de arquivo resultante pode ser compartilhado por herança, por pesquisa de nome ou por duplicação.

22.7.5.3 Pilhas heaps

As pilhas heaps fornecem uma terceira forma para as aplicações utilizarem a memória. Uma pilha heap no ambiente Win32 é uma região reservada do espaço de endereços. Quando um processo API Win32 é inicializado, ele é criado com uma **pilha heap padrão** de 1 MB. Como muitas funções API Win32 utilizam a pilha heap padrão, o acesso à pilha heap é sincronizado para proteger as estruturas de dados de alocação de espaço da pilha heap contra danos ocasionados pelas atualizações simultâneas vindas de diversas threads.

A API Win32 fornece várias funções de gerenciamento de pilha heap, de modo que um processo

possa alocar e controlar uma pilha heap particular. Essas funções são `HeapCreate()`, `HeapAlloc()`, `HeapRealloc()`, `HeapSize()`, `HeapFree()` e `HeapDestroy()`. A API Win32 também fornece as funções `HeapLock()` e `HeapUnlock()` para permitir que uma thread obtenha acesso exclusivo à pilha heap. Ao contrário de `VirtualLock()`, essas funções realizam apenas sincronismo; elas não bloqueiam páginas na memória física.

22.7.5.4 Armazenamento local a thread

A quarta forma como as aplicações utilizam a memória é por meio de um mecanismo de armazenamento local à thread. As funções que contam com dados globais ou estáticos não funcionam bem em um ambiente multithread. Por exemplo, a função C em tempo de execução `strtok()` utiliza uma variável estática para registrar sua posição atual enquanto analisa uma string. Para duas threads concorrentes executarem `strtok()` corretamente, elas precisam de variáveis de *posição atual* separadas. O mecanismo de armazenamento local à thread aloca armazenamento global para cada thread. Ele fornece métodos dinâmicos e estáticos para a criação do armazenamento local à thread. O método dinâmico é ilustrado na [Figura 22.16](#).

```
// reserva um slot para uma variável
DWORD var_index = TlsAlloc( );
// define-a com valor 10
TlsSetValue(var_index,10);
// apanha o valor de volta
int var = TlsGetValue(var_index);
// libera o índice
TlsFree(var_index);
```

FIGURA 22.16 Fragmentos de código para mapeamento de memória de um arquivo.

Para usar uma variável estática local à thread, a aplicação declara a variável da seguinte maneira, para garantir que cada thread tenha sua própria cópia particular:

```
_declspec(thread) DWORD cur_pos = 0;
```

22.8 Resumo

A Microsoft projetou o Windows XP para ser um sistema operacional extensível e portável, capaz de tirar proveito de novas técnicas e hardware. O Windows XP admite vários ambientes operacionais e multiprocessamento simétrico, incluindo processadores de 32 bits e 64 bits, além de arquiteturas NUMA. O uso de objetos do kernel para fornecer serviços básicos, além do suporte para computação cliente-servidor, permite que o Windows XP aceite uma grande variedade de ambientes de aplicação. Por exemplo, o Windows XP pode executar programas compilados para MS-DOS, Win16, Windows 95, Windows XP e POSIX. Ele fornece memória virtual, caching integrado e escalonamento preemptivo. O Windows XP admite um modelo de segurança mais forte do que aqueles dos sistemas operacionais anteriores da Microsoft e inclui recursos de internacionalização. O Windows XP funciona em uma grande variedade de computadores, de modo que os usuários podem escolher e atualizar o hardware de acordo com seus orçamentos e requisitos de desempenho, sem ter de mudar as aplicações com que trabalham.

Exercícios práticos

- 22.1. Que tipo de sistema operacional é o Windows XP? Descreva dois de seus principais recursos.
- 22.2. Liste os objetivos de projeto do Windows XP. Descreva dois deles com detalhes.
- 22.3. Descreva o processo de booting para um sistema Windows XP.
- 22.4. Descreva as três principais camadas arquitetônicas do Windows XP.
- 22.5. Qual é a função do gerenciador de objetos?
- 22.6. Que tipos de serviços o gerenciador de processos oferece? O que é uma chamada de procedimento local?
- 22.7. Quais são as responsabilidades do gerenciador de E/S?
- 22.8. Descreva dois processos do modo usuário que permitem ao Windows XP executar programas desenvolvidos para outros sistemas operacionais.
- 22.9. Que tipos de rede o Windows XP admite? Como o Windows XP implementa os protocolos de transporte? Descreva dois protocolos de rede.
- 22.10. Como o espaço de nomes do NTFS é organizado? Descreva essa organização.
- 22.11. Como o NTFS lida com estruturas de dados? Como o NTFS se recupera de uma falha do sistema? O que é garantido após uma recuperação do sistema?
- 22.12. Como o Windows XP aloca a memória do usuário?
- 22.13. Descreva algumas maneiras como uma aplicação pode usar a memória por meio da API Win32.

Exercícios

- 22.14. Sob quais circunstâncias seria usada a facilidade de chamadas de procedimento adiadas no Windows XP?
- 22.15. O que é um descritor e como um processo obtém um descritor?
- 22.16. Descreva o esquema de gerenciamento do gerenciador de memória virtual. Como o gerenciador VM melhora o desempenho?
- 22.17. Descreva uma aplicação útil da facilidade de página com nenhum acesso, fornecida no Windows XP.
- 22.18. Os processadores IA64 contêm registradores que podem ser usados para endereçar um espaço de endereços de 64 bits. Contudo, o Windows XP limita o espaço de endereços dos programas do usuário a 8 TB, o que corresponde a 43 bits. Por que foi tomada essa decisão?
- 22.19. Descreva as três técnicas usadas para a comunicação de dados em uma chamada de procedimento local. Quais opções diferentes são mais importantes para a aplicação de diferentes técnicas de passagem de mensagem?
- 22.20. O que gerencia o cache no Windows XP? Como o cache é gerenciado?
- 22.21. Qual é a finalidade do ambiente de execução Win16? Que limitações são impostas aos programas que executam dentro desse ambiente? Quais são as garantias de proteção fornecidas entre diferentes aplicações executando dentro do ambiente Win16? Quais são as garantias de proteção oferecidas entre uma aplicação executando dentro do ambiente Win16 e uma aplicação de 32 bits?
- 22.22. Descreva dois processos no modo usuário que o Windows XP fornece para permitir a execução de programas desenvolvidos para outros sistemas operacionais.
- 22.23. Qual é a diferença entre a estrutura de diretórios NTFS e a estrutura de diretórios usada nos sistemas operacionais Unix?
- 22.24. O que é um processo e como ele é gerenciado no Windows XP?
- 22.25. O que é a abstração de fibra fornecida pelo Windows XP? Qual é a diferença entre ela e a abstração de threads?

Notas bibliográficas

[Solomon e Russinovich \[2000\]](#) dão uma visão geral do Windows XP e pormenores técnicos consideráveis sobre os detalhes internos e componentes do sistema. [Tate \[2000\]](#) é uma boa referência sobre o uso do Windows XP. O Microsoft Windows XP Server Resource Kit ([Microsoft \[2000b\]](#)) é um conjunto de seis volumes, útil para se obter informações sobre uso e implantação do Windows XP. A Microsoft Developer Network Library ([Microsoft \[2000a\]](#)), emitida trimestralmente, fornece inúmeras informações sobre o Windows XP e outros produtos da Microsoft.

[Iseminger \[2000\]](#) fornece uma boa referência sobre o Active Directory do Windows XP. [Richter \[1997\]](#) apresenta uma discussão detalhada sobre a escrita de programas que utilizam a API Win32. Em Silberschatz e outros [2001] há uma boa discussão sobre árvores B+.

CAPÍTULO 23

Sistemas operacionais marcantes

Agora que estão entendidos os conceitos fundamentais dos sistemas operacionais (escalonamento de CPU, gerência de memória, processos, e assim por diante), podemos examinar como esses conceitos foram aplicados em vários sistemas operacionais mais antigos e altamente influentes. Alguns deles (como o XDS-940 e o sistema THE) foram sistemas de uso exclusivo; outros (como o OS/360) foram bastante utilizados. A ordem da apresentação destaca as semelhanças e as diferenças dos sistemas; ela não é estritamente cronológica ou ordenada por importância. O estudante de sistemas operacionais que leva o assunto a sério deverá se familiarizar com todos esses sistemas.

Enquanto descrevemos os primeiros sistemas, incluímos referências para leitura mais profunda. Os artigos escritos pelos projetistas dos sistemas são importantes, tanto por seu conteúdo técnico quanto por seu estilo.

OBJETIVOS DO CAPÍTULO

- Explicar como os recursos do sistema operacional migraram com o tempo, desde grandes sistemas de computação até os menores.
- Discutir os recursos de diversos sistemas operacionais historicamente importantes.

23.1 Migração de recursos

Um motivo para estudar as antigas arquiteturas e sistemas operacionais é que um recurso outrora utilizado apenas em imensos sistemas por fim pode ter aparecido em sistemas muito pequenos. Na verdade, um exame dos sistemas operacionais para mainframes e microcomputadores mostra que muitos recursos que existiam apenas em mainframes foram adotados para microcomputadores. Os mesmos conceitos de sistema operacional, portanto, são apropriados para diversas classes de computadores: mainframes, minicomputadores, microcomputadores e portáteis. Para entender os sistemas operacionais modernos, então, você precisa reconhecer o tema da migração de recursos e a longa história de muitos recursos do sistema operacional, como mostra a [Figura 23.1](#).

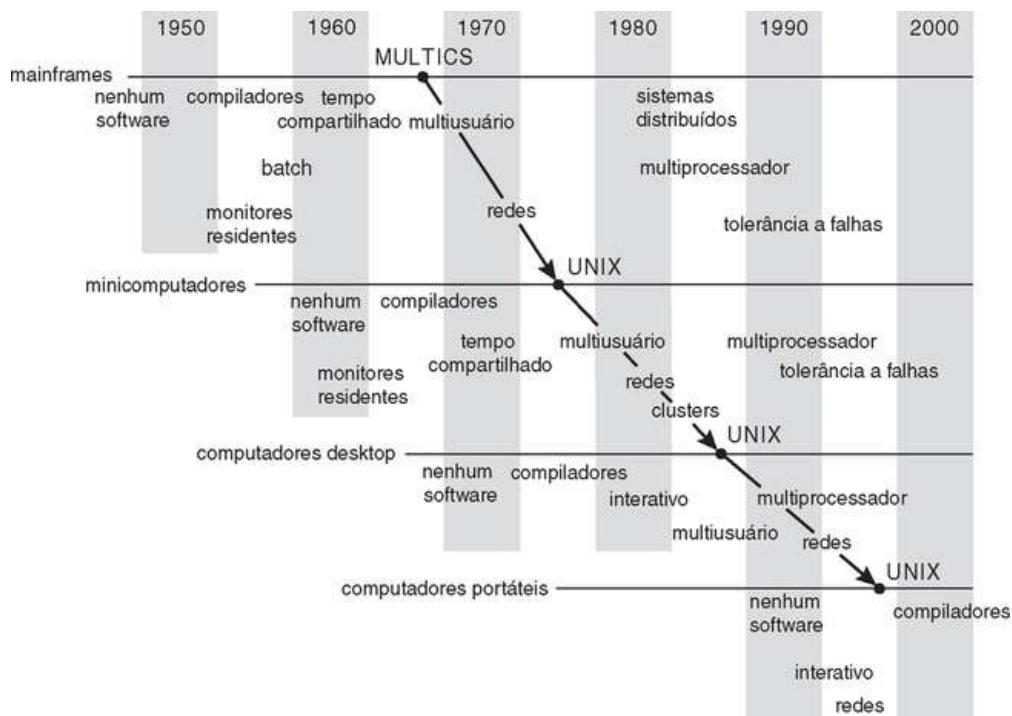


FIGURA 23.1 Migração dos conceitos e recursos do sistema operacional.

Um bom exemplo da migração de recursos começou com o sistema operacional Multiplexed Information and Computing Services (MULTICS). MULTICS foi desenvolvido de 1965 a 1970 no Massachusetts Institute of Technology (MIT) como um **utilitário** de computação. Ele era executado em um computador mainframe grande e complexo (o GE 645). Muitas das ideias que foram desenvolvidas para o MULTICS foram mais tarde usadas na Bell Laboratories (um dos parceiros originais no desenvolvimento do MULTICS) no projeto do UNIX. O sistema operacional UNIX foi projetado por volta de 1970 para um minicomputador PDP-11. Por volta de 1980, os recursos do UNIX tornaram-se a base para sistemas operacionais tipo UNIX em microcomputadores; e esses recursos estão incluídos em vários sistemas operacionais mais recentes para microcomputadores, como Microsoft Windows, Windows XP e o sistema operacional Mac OS X. O Linux inclui alguns desses mesmos recursos, e agora eles também podem ser encontrados em PDAs.

23.2 Primeiros sistemas

Voltamos nossa atenção agora para uma abordagem histórica dos primeiros sistemas de computação. Devemos observar que a história da computação começa muito antes dos “computadores”, com ábacos e calculadoras, conforme descrito em [Frah \[2001\]](#) e mostrado graficamente em [Frauenfelder \[2005\]](#). Porém, começamos nossa discussão com os computadores do século XX.

Antes da década de 1940, os dispositivos de computação eram projetados e implementados para realizar tarefas específicas, fixas. A modificação de uma dessas tarefas exigia muito esforço e trabalho manual. Tudo isso mudou na década de 1940, quando Alan Turing e John von Neumann (e seus colegas), separados e em conjunto, trabalharam com a ideia de um computador com **programa armazenado** e de uso geral. Essa máquina possui locais para armazenamento de programas e armazenamento de dados, onde o armazenamento de programas oferece instruções sobre o que fazer com os dados.

Este conceito fundamental da computação gerou rapidamente uma série de computadores de uso geral, mas grande parte da história dessas máquinas foi ocultada pelo tempo e pelos segredos do seu desenvolvimento durante a Segunda Guerra Mundial. É provável que o primeiro computador de uso geral com programa armazenado funcionando fosse o Manchester Mark 1, que funcionou com sucesso em 1949. O primeiro computador comercial foi seu sucessor, o Ferranti Mark 1, que passou a ser vendido em 1951. Esses primeiros esforços de computação são descritos por [Rojas e Hashagen \[2000\]](#) e [Ceruzzi \[1998\]](#).

Os primeiros computadores eram máquinas fisicamente enormes, executadas a partir de consoles. O programador, que também era o operador do sistema de computador, escrevia um programa e depois operava o programa diretamente a partir do console do operador. Primeiro, o programa era carregado manualmente na memória, a partir de chaves no painel frontal (uma instrução por vez), a partir de fita de papel ou de cartões perfurados. Depois, os botões apropriados seriam pressionados para definir o endereço de boot e iniciar a execução do programa. Enquanto o programa era executado, o programador/operador monitorava sua execução pelas luzes do visor no console. Se fossem descobertos erros, o programador poderia interromper o programa, examinar o conteúdo da memória e dos registradores e depurar o programa diretamente pelo console. A saída era impressa ou perfurada em fita de papel ou cartões, para posterior impressão.

23.2.1 Sistemas computadorizados dedicados

Com o passar do tempo, foram desenvolvidos software e hardware adicionais. Leitoras de cartões, impressoras de linhas e fita magnética tornaram-se comuns. Montadores (assemblers), carregadores (loaders) e ligadores (linkers) foram projetados para facilitar a tarefa de programação. Bibliotecas de funções comuns foram criadas. Funções comuns poderiam ser copiadas para um novo programa sem precisarem ser escritas novamente, oferecendo reutilização do software.

As rotinas que realizavam E/S eram especialmente importantes. Cada novo dispositivo de E/S tinha suas próprias características, exigindo uma programação cuidadosa. Uma sub-rotina especial – chamada driver de dispositivos – era escrita para cada dispositivo de E/S. Um driver de dispositivos sabe como devem ser usados os buffers, sinalizadores, registradores, bits de controle e bits de status para determinado dispositivo. Cada tipo de dispositivo possui seu próprio driver. Uma tarefa simples, como ler um caractere de um leitor de fita de papel, poderia envolver sequências complexas de operações específicas ao dispositivo. Em vez de escrever o código necessário em todos os programas, o driver de dispositivos era usado a partir da biblioteca.

Mais tarde, apareceram compiladores para FORTRAN, COBOL e outras linguagens, facilitando muito a tarefa de programação, mas tornando a operação do computador mais complexa. Por exemplo, para preparar um programa em FORTRAN para execução, o programador primeiro precisaria carregar o compilador FORTRAN no computador. O compilador normalmente era mantido em fita magnética, de modo que a fita correta teria de ser montada em uma unidade de fita. O programa seria lido por meio de uma leitora de cartões e escrito em outra fita. O compilador FORTRAN produziria uma saída em linguagem assembly, que, em seguida, precisava ser montada. Esse procedimento exigia a montagem de outra fita com o assembler. A saída do assembler precisaria ser vinculada a rotinas de biblioteca de suporte. Finalmente, a forma de objeto binário do programa estaria pronta para ser executada. Ela poderia ser carregada na memória e depurada do console, como antes.

Uma quantidade significativa de **tempo de preparação** poderia estar envolvida na execução de uma tarefa. Cada tarefa (ou job) consistia em muitas etapas separadas:

1. Carregar a fita do compilador FORTRAN.
2. Executar o compilador.
3. Descarregar a fita do compilador.
4. Carregar a fita do assembler.
5. Executar o assembler.

6. Descarregar a fita do assemblér.
7. Carregar o programa objeto.
8. Executar o programa objeto.

Se houvesse um erro durante qualquer etapa, o programador/operador poderia ter de começar novamente do início. Cada etapa da tarefa poderia envolver o carregamento e o descarregamento de fitas magnéticas, fitas de papel e cartões perfurados.

O tempo de preparação da tarefa era um problema real. Enquanto as fitas estavam sendo montadas ou o programador estava operando o console, a CPU ficava ociosa. Lembre-se de que, nos primeiros dias, havia poucos computadores disponíveis e eles eram muito caros. Um computador poderia custar milhões de dólares, sem incluir os custos operacionais de energia, refrigeração, programadores e assim por diante. Assim, o tempo do computador era extremamente valioso, e os proprietários queriam que seus computadores fossem usados o máximo possível. Eles precisavam de alta **utilização** para obter o máximo que pudessem de seus investimentos.

23.2.2 Sistemas computadorizados compartilhados

A solução foi dupla. Primeiro, um operador de computador profissional era contratado. O programador não operava mais a máquina. Assim que uma tarefa terminava, o operador poderia iniciar a seguinte. Como o operador tinha mais experiência com a montagem de fitas do que um programador, o tempo de preparação era reduzido. O programador fornecia os cartões ou fitas que fossem necessários, além de uma pequena descrição de como a tarefa deveria ser executada. Naturalmente, o operador não poderia depurar um programa incorreto no console, pois ele não entenderia o programa. Portanto, no caso do erro de programa, era feita uma cópia da memória e dos registradores, e o programador tinha de depurar a cópia. A cópia (ou dump) da memória e dos registradores permitia que o operador continuasse com a tarefa seguinte, mas deixava o programador com o problema de depuração mais difícil.

Segundo, tarefas com necessidades semelhantes eram reunidas e executadas no computador como um grupo, a fim de reduzir o tempo de preparação. Por exemplo, suponha que o operador recebesse uma tarefa em FORTRAN, uma tarefa em COBOL e outra tarefa em FORTRAN. Se ele as executasse nessa ordem, teria de se preparar para FORTRAN (carregar as fitas de compilador e assim por diante), depois se preparar para COBOL, e depois se preparar para FORTRAN novamente. Contudo, se ele executasse dois programas em FORTRAN como um lote, poderia se preparar apenas uma vez para FORTRAN, economizando tempo de operador.

Mas ainda há problemas. Por exemplo, quando uma tarefa terminava, o operador teria de notar que ela parou (observando o console), determinar *por que* ela parou (término normal ou anormal), copiar a memória e os registradores (se fosse necessário), carregar o dispositivo apropriado com a próxima tarefa e reiniciar o computador. Durante essa transição de uma tarefa para a seguinte, a CPU ficava ociosa.

Para contornar esse tempo ocioso, foi desenvolvida a **sequência automática de tarefas**; com essa técnica, os primeiros sistemas operacionais rudimentares foram criados. Um pequeno programa, chamado **monitor residente**, era criado para transferir o controle automaticamente de uma tarefa para a seguinte (Figura 23.2). O monitor residente sempre está na memória (ou *residente*).

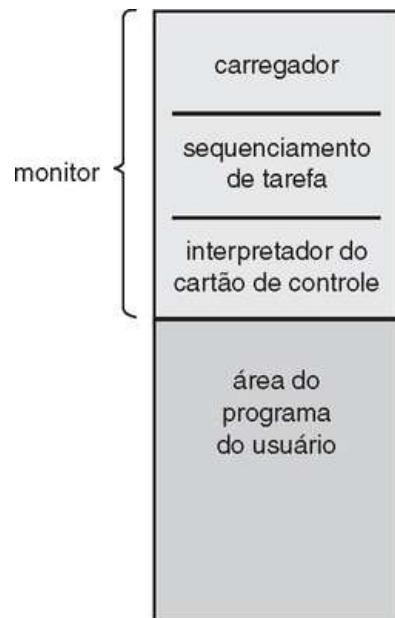


FIGURA 23.2 Layout de memória para um monitor residente.

Quando o computador era ligado, o monitor residente era chamado e transferia o controle para um programa. Quando o programa terminava, ele retornava o controle para o monitor residente, que então continuava no programa seguinte. Assim, o monitor residente automaticamente fazia a sequência de um programa para outro e de uma tarefa para outra.

Mas como o monitor residente saberia qual programa executar? Anteriormente, o operador recebeu uma curta descrição de quais programas tiveram de ser executados sobre quais dados. **Cartões de controle (control cards)** foram introduzidos para fornecer essa informação diretamente ao monitor. A ideia é simples: além do programa ou dados para uma tarefa, o programador incluía os cartões de controle, que continham diretivas para o monitor residente, indicando o programa a executar. Por exemplo, um programa normal do usuário poderia exigir um de três programas para executar: o compilador FORTRAN (FTN), o assembler (ASM) ou o programa do usuário (RUN). Poderíamos usar um cartão de controle separado para cada um destes:

\$FTN - Executar o compilador FORTRAN.

\$ASM - Executar o assembler.

\$RUN - Executar o programa do usuário.

Esses cartões dizem ao monitor residente qual programa deve ser executado.

Podemos usar dois cartões de controle adicionais para definir os limites de cada tarefa:

\$JOB - Primeiro cartão de uma tarefa.

\$END - Último cartão de uma tarefa.

Esses dois cartões poderiam ser úteis na contabilidade dos recursos da máquina usados pelo programador. Parâmetros podem ser usados para definir o nome da tarefa, o número da conta a ser cobrada e assim por diante. Outros cartões de controle podem ser definidos para outras funções, como pedir ao operador para carregar ou descarregar uma fita.

Um problema com os cartões de controle é como distingui-los de cartões de dados ou de programa. A solução comum é identificá-los por um caractere ou padrão especial no cartão. Vários sistemas usavam o caractere de cifrão (\$) na primeira coluna para identificar um cartão de controle. Outros usavam um código diferente. A linguagem de controle de tarefa (Job Control Language - JCL) da IBM usava barras (/) nas duas primeiras colunas. A [Figura 23.3](#) mostra um grupo de cartões de exemplo preparado para um sistema batch (em lote) simples.

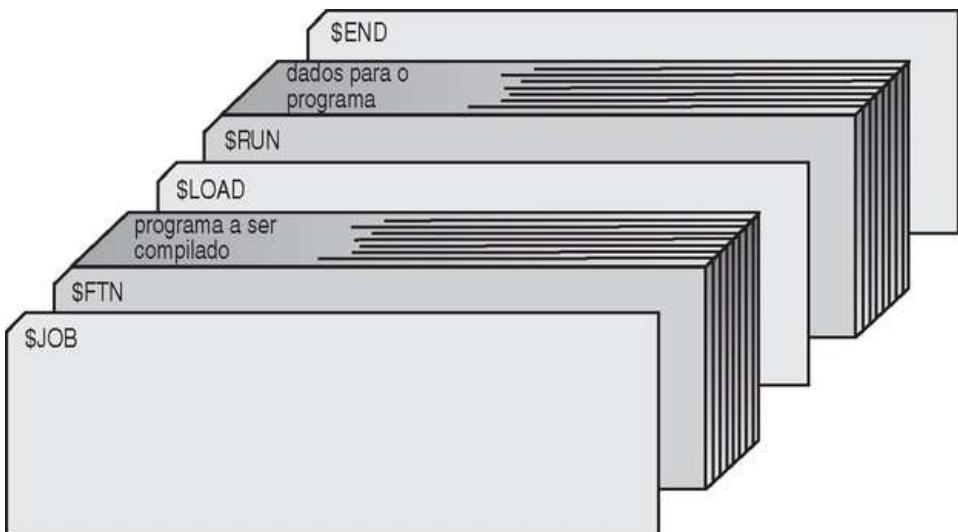


FIGURA 23.3 Grupo de cartões para um sistema batch simples.

Um monitor residente, assim, possui várias partes identificáveis:

- O **interpretador do cartão de controle (control-card interpreter)**, que é responsável por ler e executar as instruções sobre os cartões no ponto de execução.
- O **carregador (loader)**, que é invocado pelo interpretador do cartão de controle, para carregar programas do sistema e programas de aplicação para a memória nos intervalos.
- Os **drivers de dispositivos (device drivers)**, que são usados pelo interpretador do cartão de controle e pelo carregador para os dispositivos de E/S do sistema. Normalmente, o sistema e os programas de aplicação são vinculados a esses mesmos drivers de dispositivos, oferecendo continuidade em sua operação, além de economizar espaço de memória e tempo de programação.

Esses sistemas batch funcionam muito bem. O monitor residente oferece sequência de tarefas automática, conforme indicado pelos cartões de controle. Quando um cartão de controle indica que um programa deve ser executado, o monitor carrega o programa na memória e transfere o controle para ele. Quando o programa termina, ele transfere o controle de volta ao monitor, que lê o próximo cartão de controle, carrega o programa apropriado, e assim por diante. Esse ciclo é repetido até todos os cartões de controle serem interpretados para a tarefa. Depois, o monitor continua automaticamente com a próxima tarefa.

A troca para sistemas batch com sequência de tarefa automática foi feita para melhorar o desempenho. O problema, colocando de forma simples, é que os seres humanos são muito mais lentos que o computador. Consequentemente, é desejável substituir a operação humana pelo software do sistema operacional. A sequência automática de tarefas elimina a necessidade humana no tempo de preparação e sequência de tarefas.

Entretanto, até mesmo com esse arranjo, a CPU normalmente está ociosa. O problema é a velocidade dos dispositivos de E/S mecânicos, intrinsecamente mais lentos do que os dispositivos eletrônicos. Até mesmo uma CPU lenta trabalha na faixa de microssegundos, com milhares de instruções executadas por segundo. Uma leitora de cartões rápida, por outro lado, poderia ler 1.200 cartões por minuto (ou 20 cartões por segundo). Assim, a diferença na velocidade entre a CPU e seus dispositivos de E/S pode ser de três ordens de grandeza ou mais. Com o tempo, as melhorias na tecnologia resultaram em dispositivos de E/S mais rápidos. Infelizmente, as velocidades de CPU aumentaram ainda mais, de modo que o problema não apenas não foi resolvido, mas também aumentou.

23.2.3 E/S sobreposta

Uma solução comum para o problema de E/S foi substituir as leitoras de cartões (dispositivos de entrada) e impressoras de linhas (dispositivos de saída) lentas por unidades de fita magnética. A maioria dos sistemas de computador no final da década de 1950 e início de 1960 eram sistemas batch, lendo de leitoras de cartões e escrevendo em impressoras de linhas ou perfuradoras de cartões. Contudo, a CPU não lia diretamente dos cartões; em vez disso, os cartões primeiro eram copiados para uma fita magnética, por meio de um dispositivo separado. Quando a fita estivesse suficientemente cheia, ela era retirada e levada para o computador. Quando um cartão era necessário para a entrada em um programa, o registro equivalente era lido da fita. De modo semelhante, a saída era gravada em fita, e o conteúdo da fita era impresso mais tarde. As leitoras de cartões e impressoras de linhas eram operadas off-line, e não pelo computador principal ([Figura 23.4](#)).

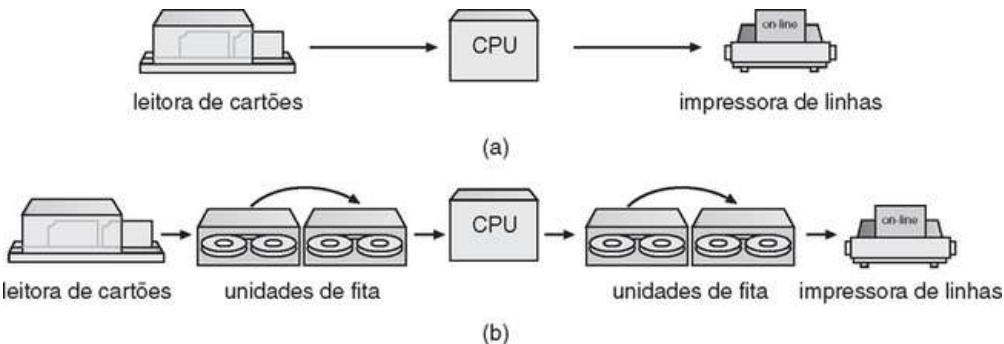


FIGURA 23.4 Operação de dispositivos de E/S (a) on-line e (b) off-line.

Uma vantagem óbvia da operação off-line era que o computador principal não estava mais restrito pela velocidade das leitoras de cartões e impressoras de linhas, mas estava limitado apenas pela velocidade das unidades de fita magnética muito mais rápidas. A técnica de uso de fita magnética para toda a E/S poderia ser aplicada com um equipamento semelhante (como leitora de cartões, perfuradoras de cartões, plotadoras, fita de papel, impressoras).

O ganho real na operação off-line veio da possibilidade de usar vários sistemas de leitora para fita e de fita para impressora para uma única CPU. Se a CPU pode processar a entrada duas vezes mais rapidamente do que a leitora pode ler cartões, então duas leitoras trabalhando ao mesmo tempo podem produzir fita suficiente para manter a CPU ocupada. No entanto, há uma desvantagem também - um atraso maior na execução de uma tarefa em particular. Primeiro, a tarefa precisa ser lida para a fita. Depois, ela precisa esperar que tarefas suficientes sejam lidas para a fita para "preenchê-la". A fita precisa ser rebobinada, descarregada, transportada à mão para a CPU e montada em uma unidade de fita livre. Logicamente, esse processo não é excessivo para sistemas batch. Muitas tarefas semelhantes podem ser colocadas em lote em uma fita antes de serem levadas para o computador.

Embora a preparação off-line de tarefas continue por algum tempo, ela foi substituída na maioria dos sistemas. Os sistemas de disco se tornaram amplamente disponíveis e melhoraram bastante a operação off-line. O problema com os sistemas de fita foi que a leitora de cartões não poderia escrever para uma extremidade da fita enquanto a CPU lia de outra. A fita inteira tinha de ser escrita antes de ser rebobinada e lida, pois as fitas, por natureza, são **dispositivos de acesso sequencial**. Os sistemas de disco eliminaram esse problema, por serem dispositivos de acesso aleatório. Como a cabeça é movida de uma área do disco para outra, ela pode passar rapidamente de uma área do disco sendo usada pela leitora de cartões para armazenar novos cartões para a posição necessária pela CPU para ler o "próximo" cartão.

Em um sistema de disco, os cartões são lidos diretamente da leitora de cartões para o disco. O local das imagens do cartão é registrado em uma tabela mantida pelo sistema operacional. Quando uma tarefa é executada, o sistema operacional satisfaz suas requisições para entrada de leitora de cartões lendo do disco. De modo semelhante, quando a tarefa requisita a impressora para gerar uma linha, essa linha é copiada em um buffer do sistema e gravada no disco. Quando a tarefa termina, a saída é impressa. Essa forma de processamento é denominada **spooling** (Figura 23.4); o nome é um acrônimo para "simultaneous peripheral operation on-line" (operação on-line simultânea de periférico). O spooling utiliza o disco como um grande buffer, para o máximo de leitura antecipada em dispositivos de entrada e para armazenar arquivos de saída até os dispositivos de saída serem capazes de aceitá-las.

O spooling também é usado para processamento de dados em sites remotos. A CPU envia um dado por caminhos de comunicação para uma impressora remota (ou aceita uma tarefa de entrada inteira de uma leitora de cartões remota). O processamento remoto é feito em sua própria velocidade, sem intervenção da CPU. A CPU só precisa ser notificada quando o processamento terminar, de modo que possa colocar o próximo lote de dados em spool.

O spooling sobrepõe a E/S de uma tarefa com a computação de outras tarefas. Até mesmo em um sistema simples, o spooler pode ler a entrada de uma tarefa enquanto imprime a saída de uma tarefa diferente. Durante esse tempo, outra tarefa ainda (ou tarefas) pode ser executada, lendo seus "cartões" do disco e "imprimindo" suas linhas de saída no disco.

O spooling possui um efeito direto sobre o desempenho do sistema. Ao custo de algum espaço em disco e algumas tabelas, o cálculo de uma tarefa pode sobrepor-se à E/S de outras tarefas. Assim, o spooling pode manter tanto a CPU quanto os dispositivos de E/S trabalhando em velocidades muito mais altas. O spooling leva naturalmente à multiprogramação, que é a base de todos os sistemas operacionais modernos.

23.3 Atlas

O sistema operacional Atlas ([Kilburn e outros \[1961\]](#), [Howarth e outros \[1961\]](#)) foi criado na Universidade de Manchester, na Inglaterra, no final da década de 1950 e início da década de 1960. Muitos de seus recursos básicos que eram novidade na época se tornaram padrão dos sistemas operacionais modernos. Drivers de dispositivos eram uma parte importante do sistema. Além disso, as chamadas de sistema eram acrescentadas por um conjunto de instruções especiais, chamadas *códigos extras*.

O Atlas era um sistema operacional batch com spooling. O spooling permitia ao sistema escalonar tarefas de acordo com a disponibilidade dos dispositivos periféricos, como unidades de fita magnética, leitoras de fita de papel, perfuradoras de fita de papel, impressoras de linha, leitoras de cartão e perfuradoras de cartão.

Contudo, o recurso mais marcante do Atlas era a gerência de memória. A **memória de núcleo (Core Memory)** era nova e cara na época. Muitos computadores, como o IBM 650, usavam um tambor para a memória principal. O sistema Atlas usava um tambor para sua memória principal, mas tinha uma pequena quantidade de memória de núcleo, usada como cache para o tambor. A paginação por demanda era usada para transferir informações automaticamente entre a memória de núcleo e o tambor.

O sistema Atlas usava um computador britânico com palavras de 48 bits. Os endereços eram de 24 bits, mas eram codificados em decimal, o que permitia apenas 1 milhão de palavras endereçadas. Naquela época, isso era um espaço de endereços extremamente grande. A memória física para o Atlas era um tambor de 98 KB de palavras e 16 KB de palavras de núcleo. A memória era dividida em páginas de 512 palavras, oferecendo 32 quadros na memória física. Uma memória associativa de 32 registradores implementava o mapeamento de um endereço virtual para um endereço físico.

Se houvesse falta de página, um algoritmo de substituição de página era chamado. Um quadro de memória sempre era mantido vazio, de modo que uma transferência de tambor pudesse ser iniciada imediatamente. O algoritmo de substituição de página tentava prever o comportamento de acesso à memória futura com base no comportamento passado. Um bit de referência para cada quadro era definido sempre que ele fosse acessado. Os bits de referência eram lidos para a memória a cada 1.024 instruções, e os últimos 32 valores desses bits eram retidos. Esse histórico era usado para definir o tempo desde a referência mais recente (t_1) e o intervalo entre as duas últimas referências (t_2). As páginas eram escolhidas para substituição na seguinte ordem:

1. Qualquer página com $t_1 > t_2 + 1$. Essa página é considerada não estando mais em uso.
2. Se $t_1 \leq t_2$ para todas as páginas, então substitua essa página pelo maior entre t_2 e t_1 .

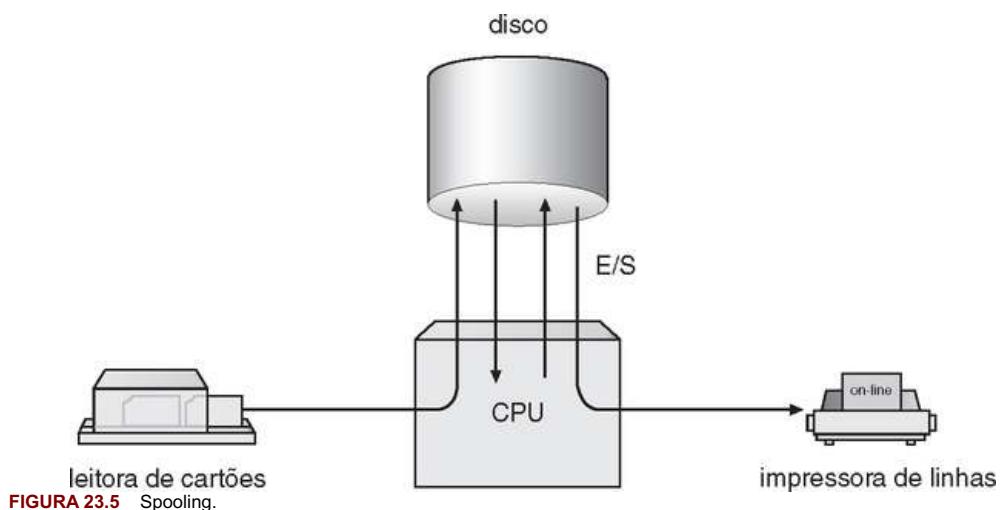


FIGURA 23.5 Spooling.

O algoritmo de substituição de página considera que os programas acessam a memória em loops. Se o tempo entre as duas últimas referências for t_2 , então outra referência é esperada t_2 unidades de tempo depois. Se uma referência não ocorrer ($t_1 > t_2$), considera-se que a página não está mais sendo usada, e a página é substituída. Se todas as páginas ainda estiverem em uso, então a página que não será necessária pelo maior tempo é substituída. O tempo para a próxima referência é esperado sendo $t_2 - t_1$.

23.4 XDS-940

O sistema operacional XDS-940 ([Lichtenberger e Pirtle \[1965\]](#)) foi projetado na Universidade da Califórnia em Berkeley. Assim como o sistema Atlas, ele usava a paginação para gerência de memória. Entretanto, ao contrário do sistema Atlas, era um sistema de tempo compartilhado.

A paginação era usada apenas para relocação; não era usada para paginação por demanda. A memória virtual de qualquer processo do usuário era de apenas 16 KB de palavras, enquanto a memória física era de 64 KB de palavras. Cada página era composta de 2 KB de palavras. A tabela de página era mantida em registradores. Como a memória física era maior do que a memória virtual, vários processos do usuário poderiam estar na memória ao mesmo tempo. A quantidade de usuários poderia ser aumentada compartilhando páginas quando tivessem código reentrante somente de leitura. Os processos eram mantidos em um tambor e eram trocados para dentro e para fora da memória conforme a necessidade.

O sistema XDS-940 foi construído a partir de um XDS-930 modificado. As modificações eram típicas das mudanças feitas a um computador básico para permitir que um sistema operacional fosse escrito corretamente. Um modo usuário-monitor foi acrescentado. Certas instruções, como I/O e Halt, foram definidas como privilegiadas. Uma tentativa de executar uma instrução privilegiada no modo usuário causaria uma interceptação para o sistema operacional.

Uma instrução de chamada de sistema foi acrescentada ao conjunto de instruções do modo usuário. Essa instrução foi usada para criar novos recursos, como arquivos, permitindo que o sistema operacional gerenciasse os recursos físicos. Os arquivos, por exemplo, eram alocados em blocos de 256 palavras no tambor. Um mapa de bits era usado para gerenciar blocos de tambor livres. Cada arquivo tinha um bloco de índice com ponteiros para os blocos de dados reais. Os blocos de índice eram encadeados.

O sistema XDS-940 também fornecia chamadas do sistema para permitir que os processos criassem, iniciassem, suspendessem e destruissem subprocessos. Um programador poderia construir um sistema de processos. Processos separados poderiam compartilhar memória para comunicação e sincronismo. A criação de processos definia uma estrutura de árvore, na qual um processo é a raiz e seus subprocessos são nós abaixo dele na árvore. Cada um dos subprocessos poderia, por sua vez, criar mais subprocessos.

23.5 THE

O sistema operacional THE ([Dijkstra \[1968\]](#), [McKeag e Wilson \[1976\]](#)) foi projetado na Technische Hogeschool, em Eindhoven, nos Países Baixos. Ele era um sistema batch executando em um computador holandês, o EL X8, com 32 KB de palavras de 27 bits. O sistema foi mencionado principalmente por seu projeto claro, em especial sua estrutura de camadas, seu uso fácil de um conjunto de processos simultâneos, empregando semáforos para sincronismo.

Todavia, ao contrário do sistema XDS-940, o conjunto de processos no sistema THE era estático. O próprio sistema operacional foi projetado como um conjunto de processos em cooperação. Além disso, cinco processos do usuário foram criados servindo como agentes ativos para compilar, executar e imprimir programas do usuário. Quando uma tarefa terminava, o processo retornava à fila de entrada para selecionar outra tarefa.

Um algoritmo de escalonamento de CPU por prioridade foi usado. As prioridades eram recalculadas a cada 2 segundos e eram inversamente proporcionais à quantidade de tempo de CPU usada recentemente (nos últimos 8 a 10 segundos). Esse esquema dava prioridade mais alta para processos voltados para E/S e para processos novos.

A gerência de memória era limitada pela falta de suporte do hardware. Entretanto, como o sistema era limitado e os programas do usuário só poderiam ser escritos em Algol, um esquema de página de software era utilizado. O compilador Algol gerava automaticamente chamadas para rotinas do sistema, que se certificavam de que a informação requisitada estivesse na memória, fazendo a troca, se necessário. O armazenamento de apoio era um tambor de 512 KB de palavras. Uma página de 512 palavras era usada, com uma estratégia de substituição de página LRU.

Outra grande preocupação do sistema THE era o controle de deadlock. O algoritmo do banqueiro era usado para evitar deadlock.

Bastante relacionado com o sistema THE é o sistema Venus ([Liskov \[1972\]](#)). O sistema Venus também foi um projeto estruturado em camadas, usando semáforos para sincronizar processos. Contudo, os níveis mais baixos do projeto foram implementados em microcódigo, oferecendo um sistema muito mais rápido. A gerência de memória foi mudada para uma memória com segmentação paginada. O sistema também foi projetado como um sistema de tempo compartilhado, em vez de um sistema batch.

23.6 RC 4000

O sistema RC 4000, como o sistema THE, foi notável principalmente por seus conceitos de projeto. Ele foi projetado para o computador dinamarquês RC 4000 pela Regnecentralen, particularmente por Brinch-Hansen ([Birnch-Hansen \[1970\]](#), [Birnch-Hansen \[1973\]](#)). O objetivo não foi projetar um sistema batch ou um sistema de tempo compartilhado, ou qualquer outro sistema específico. Em vez disso, o objetivo foi criar um núcleo (ou kernel) de sistema operacional, sobre o qual um sistema operacional completo pudesse ser montado. Assim, a estrutura do sistema foi em camadas, e somente os níveis mais baixos - inclusive o kernel - foram fornecidos.

O kernel aceitava uma coleção de processos simultâneos. Um escalonador de CPU round-robin foi usado. Embora os processos pudessem compartilhar memória, o principal mecanismo de comunicação e sincronismo era o **sistema de mensagens** fornecido pelo kernel. Os processos poderiam se comunicar entre si trocando mensagens de tamanho fixo, com oito palavras de extensão. Todas as mensagens eram armazenadas em buffers, dentro de um banco de buffers comum. Quando um buffer de mensagem não era mais necessário, ele era retornado ao banco comum.

Uma **fila de mensagens (message queue)** era associada a cada processo. Ela continha todas as mensagens enviadas a esse processo, mas que ainda não haviam sido recebidas. As mensagens eram removidas da fila na ordem FIFO. O sistema admitia quatro primitivas de operações, executadas de forma atômica:

- **send-message (in receptor, in mensagem, out buffer).**
- **wait-message (out emissor, out mensagem, out buffer).**
- **send-answer (out resultado, in mensagem, in buffer).**
- **wait-answer (out resultado, out mensagem, in buffer).**

As duas últimas operações permitiam que os processos trocassem várias mensagens de uma só vez.

Essas primitivas exigiam que um processo atendesse a sua fila de mensagens em uma ordem FIFO e que ficasse bloqueado enquanto outros processos estivessem tratando de suas mensagens. Para remover essas restrições, os desenvolvedores forneceram duas primitivas de comunicação adicionais. Eles permitiram que um processo esperasse pela chegada da mensagem seguinte ou respondesse e atendesse a sua fila em qualquer ordem:

- **wait-event (in buffer-anterior, out buffer-seguinte, out resultado).**
- **get-event (out buffer).**

Os dispositivos de E/S também eram tratados como processos. Os drivers de dispositivos eram código que convertia as interrupções e registradores do dispositivo em mensagens. Assim, um processo escreveria em um terminal enviando uma mensagem a esse terminal. O driver de dispositivos receberia a mensagem e enviaria o caractere no terminal. Um caractere de entrada interromperia o sistema e transferiria para um driver de dispositivos. O driver de dispositivos criaria uma mensagem do caractere entrado e a enviaria para um processo aguardando.

23.7 CTSS

O sistema Compatible Time-Sharing System (CTSS) ([Corbato e outros \[1962\]](#)) - foi projetado no MIT como um sistema de tempo compartilhado experimental. Ele foi implementado em um IBM 7090 e, por fim, admitia até 32 usuários interativos. Os usuários recebiam um conjunto de comandos interativos que permitia manipular arquivos e compilar e executar programas por meio de um terminal.

O 7090 tinha uma memória de 32 KB, composta de palavras de 36 bits. O monitor usava 5 KB palavras, deixando 27 KB para os usuários. As imagens de memória do usuário eram trocadas entre a memória e um tambor rápido. O escalonamento de CPU empregava um algoritmo de fila multinível com feedback. O quantum de tempo para o nível i era $2 * i$ unidades de tempo. Se um programa não terminasse seu burst de CPU em um quantum de tempo, ele era movido para o próximo nível abaixo na fila, recebendo o dobro do tempo. O programa no nível mais alto (com o menor quantum) era executado em primeiro lugar. O nível inicial de um programa era determinado por seu tamanho, de modo que o quantum de tempo era pelo menos o tempo de troca.

O CTSS foi extremamente bem-sucedido e esteve em uso até 1972. Embora sendo limitado, ele teve sucesso ao demonstrar que o tempo compartilhado era um modo conveniente e prático de computação. Um resultado do CTSS foi o maior desenvolvimento de sistemas de tempo compartilhado. Outro resultado foi o desenvolvimento do MULTICS.

23.8 MULTICS

O sistema operacional MULTICS ([Corbato e Vyssotsky \[1965\]](#), [Organick \[1972\]](#)) foi projetado no MIT como uma extensão natural do CTSS. O CTSS e outros antigos sistemas de tempo compartilhado tiveram tanto sucesso que criaram um desejo imediato de prosseguir rapidamente para sistemas maiores e melhores. À medida que computadores maiores se tornavam disponíveis, os projetistas do CTSS começavam a criar um utilitário de tempo compartilhado. O serviço de computação seria fornecido como a energia elétrica. Grandes sistemas de computador seriam conectados por fios telefônicos aos terminais em escritórios e lares por toda uma cidade. O sistema operacional seria um sistema de tempo compartilhado executando continuamente, com um vasto sistema de arquivos de programas e dados compartilhados.

O MULTICS foi projetado por uma equipe do MIT, GE (que mais tarde vendeu seu departamento de computador para a Honeywell) e Bell Laboratories (que saiu do projeto em 1969). O computador GE 635 básico foi modificado para um novo sistema de computador, chamado GE 645, principalmente pelo acréscimo do hardware de memória com segmentação paginada.

No MULTICS, um endereço virtual era composto por um número de segmento de 18 bits e um deslocamento de palavra de 16 bits. Os segmentos eram então paginados em páginas de 1 KB de palavras. Foi usado o algoritmo de substituição de palavra da segunda chance.

O espaço de endereços virtual segmentado foi mesclado ao sistema de arquivos; cada segmento era um arquivo. Os segmentos eram endereçados pelo nome do arquivo. O próprio sistema de arquivos foi uma estrutura de árvore multinível, permitindo que os usuários criassem suas próprias estruturas de subdiretório.

Assim como o CTSS, o MULTICS usava uma fila multinível com feedback para escalonamento de CPU. A proteção era realizada por uma lista de acesso associada a cada arquivo e um conjunto de anéis de proteção para os processos em execução. O sistema, escrito quase todo em PL/1, compreendia cerca de 300.000 linhas de código. Ele foi estendido para um sistema multiprocessado, permitindo que uma CPU fosse retirada de serviço para manutenção enquanto o sistema continuava funcionando.

23.9 IBM OS/360

A maior linha de desenvolvimento de sistema operacional, sem dúvida alguma, é a dos computadores IBM. Os primeiros computadores IBM, como o IBM 7090 e o IBM 7094, são exemplos clássicos do desenvolvimento de sub-rotinas de E/S comuns, seguido por um monitor residente, instruções privilegiadas, proteção de memória e processamento batch simples. Esses sistemas foram desenvolvidos separadamente, geralmente em locais independentes. Como resultado, a IBM encarou muitos computadores diferentes, com linguagens diferentes e software de sistemas diferentes.

O IBM/360 foi projetado para mudar essa situação. Foi projetado como uma família de computadores estendendo-se por uma completa gama de máquinas, desde as comerciais pequenas até grandes máquinas científicas. Somente um conjunto de software seria necessário para esses sistemas, todos usando o mesmo sistema operacional: OS/360 ([Mealy e outros \[1966\]](#)). Esse arranjo tinha como finalidade reduzir os problemas de manutenção para a IBM e permitir que os usuários passassem programas e aplicações de um sistema IBM para outro.

Infelizmente, o OS/360 tentou ser tudo para todas as pessoas. Como resultado, ele não realizou nenhuma de suas tarefas especialmente bem. O sistema de arquivos incluía um campo de tipo, que definia o tipo de cada arquivo, e diferentes tipos de arquivos foram definidos para registros de tamanho fixo e variável, e para arquivos bloqueados e desbloqueados. A alocação contígua foi usada, de modo que o usuário tinha de adivinhar o tamanho de cada arquivo de saída. A Job Control Language (JCL) acrescentava parâmetros para cada opção possível, tornando-a incompreensível para o usuário comum.

As rotinas de gerência de memória foram prejudicadas pela arquitetura. Embora fosse usado um modo de endereçamento com registrador de base, o programa poderia acessar e modificar o registrador de base, de modo que endereços absolutos eram gerados pela CPU. Esse arranjo impedia a relocação dinâmica; o programa estava preso à memória física no momento da carga. Duas versões separadas do sistema operacional foram produzidas: o OS/MFT usava regiões fixas e o OS/MVT usava regiões variáveis.

O sistema foi escrito em linguagem assembly por milhares de programadores, resultando em milhões de linhas de código. O próprio sistema operacional exigiu grande quantidade de memória para seu código e tabelas. O custo adicional do sistema operacional normalmente consumia metade do total de ciclos da CPU. Com o passar dos anos, novas versões foram lançadas para acrescentar novos recursos e reparar erros. Contudo, o reparo de um erro em geral causava outro em alguma parte remota do sistema, de modo que a quantidade de erros conhecidos no sistema era quase constante.

A memória virtual foi acrescentada ao OS/360 com a mudança para a arquitetura IBM 370. O hardware básico fornecia uma memória virtual com segmentação paginada. Novas versões do sistema operacional usavam esse hardware de diferentes maneiras. O OS/VS1 criou um grande espaço virtual de endereços e executava OS/MFT nessa memória virtual. Assim, o próprio sistema operacional era paginado, bem como os programas do usuário. O OS/VS2 Release 1 executava OS/MVT na memória virtual. Por fim, o OS/VS2 Release 2, atualmente denominado MVS, fornecia a cada usuário sua própria memória virtual.

O MVS ainda é um sistema operacional batch. O sistema CTSS foi executado em um IBM 7094, mas os desenvolvedores no MIT decidiram que o espaço de endereços do 360, o sucessor da IBM para o 7094, era muito pequeno para o MULTICS e, por isso, trocaram de fornecedores. A IBM, então, decidiu criar seu próprio sistema de tempo compartilhado, o TSS/360 ([Lett e Konigsford \[1968\]](#)). Assim como o MULTICS, o TSS/360 deveria ser um grande utilitário de tempo compartilhado. A arquitetura básica do 360 foi modificada no modelo 67 para fornecer memória virtual. Várias instalações adquiriram o 360/67 em antecipação ao TSS/360.

No entanto, o TSS/360 foi adiado, de modo que outros sistemas de tempo compartilhado foram desenvolvidos como sistemas temporários, até o TSS/360 estar disponível. Uma opção de tempo compartilhado (TSO) foi acrescentada ao OS/360. O Cambridge Scientific Center da IBM desenvolveu o CMS como um sistema de único usuário e o CP/67 para fornecer uma máquina virtual onde ele pudesse ser executado ([Meyer e Seawright \[1970\]](#), Parmelee e outros [1972]).

Quando o TSS/360 por fim foi entregue, ele foi um desastre. Era muito grande e muito lento. Como resultado, nenhuma instalação passou do seu sistema temporário para o TSS/360. Hoje o tempo compartilhado nos sistemas IBM é fornecido principalmente pelo TSO sob MVS ou por CMS sob CP/67 (renomeado VM).

Nem o TSS/360 nem o MULTICS obtiveram sucesso comercial. O que saiu errado? Parte do problema foi que esses sistemas avançados eram muito grandes e muito complexos para serem entendidos. Outro problema foi a suposição de que o poder de computação estaria disponível de uma origem grande e remota. Os minicomputadores apareceram e diminuíram a necessidade de grandes sistemas monolíticos. Eles foram acompanhados de estações de trabalho e, depois, pelos computadores pessoais, que colocaram o poder de computação cada vez mais perto dos usuários finais.

23.10 TOPS-20

A DEC criou muitos sistemas de computação marcantes durante a sua história. Provavelmente, o sistema operacional mais famoso associado à DEC seja o VMS ([Kenah e outros \[1988\]](#)), um sistema popular orientado a negócios, que ainda é usado hoje como OpenVMS, um produto da Hewlett-Packard. Porém, talvez o mais marcante dos sistemas operacionais da DEC tenha sido o TOPS-20.

O TOPS-20 foi iniciado como um projeto de pesquisa na Bolt, Beranek, and Newman (BBN) por volta de 1970. A BBN apanhou o PDP-10 orientado a negócios da DEC, rodando o TOPS-10, acrescentou um sistema de página de memória por hardware para implementar a memória virtual e escreveu um novo sistema operacional para esse computador, a fim de tirar proveito dos novos recursos de hardware. O resultado foi o TENEX ([Bobrow e outros \[1972\]](#)), um sistema de tempo compartilhado de uso geral. A DEC, então, comprou os direitos do TENEX e criou um novo computador com um paginador de hardware embutido. O sistema resultante foi o DECSYSTEM-20 e o sistema operacional TOPS-20. A [Figura 23.6](#) é uma foto do paginador da BBN. Observe que ele tinha o tamanho de um rack de “19 polegadas” padrão de um centro de dados. Esse rack pode manter centenas de CPUs (cada uma contendo um paginador de hardware) hoje em dia.

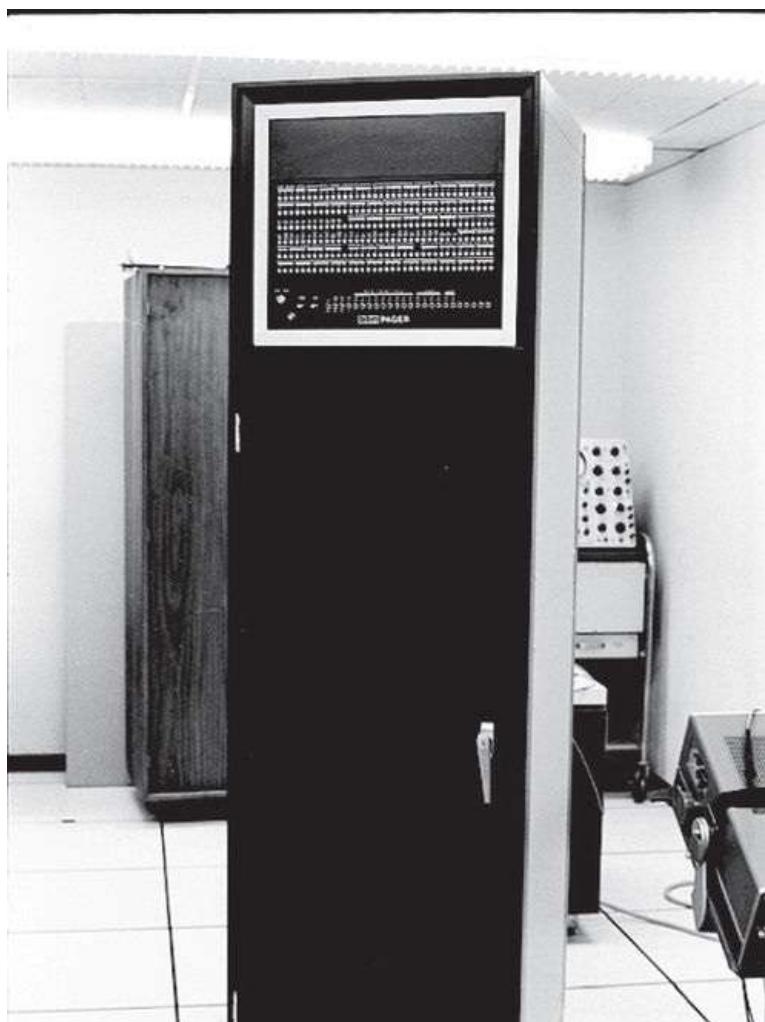


FIGURA 23.6 O acréscimo do paginador de hardware da BBN ao PDP-10.

O TOPS-20 tinha um interpretador de linha de comando avançado, que fornecia ajuda ao usuário quando fosse necessário. Isso, em combinação com a potência do computador e seu preço razoável, fez do DECSYSTEM-20 o sistema de tempo compartilhado mais popular de sua época. Em 1984, a DEC parou de trabalhar nessa linha de computadores PDP-10 de 36 bits para se concentrar nos sistemas VAX de 32 bits rodando VMS.

23.11 CP/M e MS-DOS

Os primeiros computadores não profissionais geralmente eram montados por meio de kits e rodavam um único programa de cada vez. Os sistemas evoluíam para sistemas mais avançados à medida que os componentes do computador eram aperfeiçoados. Um antigo sistema operacional “padrão” para esses computadores na década de 1970 foi o **CP/M**, abreviação de Control Program/Monitor, escrito por Gary Kindall, da Digital Research, Inc. O CP/M rodava principalmente na primeira CPU para “computador pessoal”, o Intel 8080 de 8 bits. O CP/M admitia originalmente apenas 64 KB de memória e executava apenas um programa por vez. Naturalmente, ele era baseado em texto, com um interpretador de comandos. O interpretador de comandos era semelhante aos de outros sistemas operacionais da época, como o TOPS-10, da DEC.

Quando a IBM entrou no mercado de computadores pessoais, ela decidiu pedir a Bill Gates e companhia para que escrevessem um novo sistema operacional para a sua CPU de 16 bits escolhida - o Intel 8086. Esse sistema operacional, **MS-DOS**, era semelhante ao CP/M, porém tinha um conjunto mais rico de comandos internos, novamente modelados principalmente a partir do TOPS-10. O MS-DOS tornou-se o sistema operacional de computador pessoal mais popular de sua época, começando em 1981 e continuando o desenvolvimento até o ano 2000. Ele aceitava 640 KB de memória, com a capacidade de endereçar a memória “estendida” e “expandida” para ultrapassar esse limite. Contudo, ele não possuía os recursos fundamentais dos sistemas operacionais atuais, especialmente memória protegida.

23.12 Macintosh Operating System e Windows

Com o surgimento das CPUs de 16 bits, os sistemas operacionais para computadores pessoais poderiam se tornar mais avançados, contendo mais recursos e mais fáceis de usar. O computador **Apple Macintosh** foi, sem dúvida, o primeiro computador com uma interface gráfica (GUI) projetada para usuários domésticos ([Apple \[1987\]](#)). Certamente, ele foi o mais bem-sucedido por um tempo, a partir do seu lançamento em 1984. Ele usava um mouse para apontar e selecionar na tela, e vinha com muitos programas utilitários que tiravam proveito da nova interface com o usuário. As unidades de disco rígido eram relativamente caras em 1984, de modo que ele vinha apenas com uma unidade de disquete de 400 KB como padrão.

O sistema Mac original rodava apenas em computadores Apple e lentamente foi perdendo terreno para o Microsoft Windows (a partir da versão 1.0 em 1985), que era licenciado para rodar em muitos computadores diferentes, de diversas empresas. À medida que as CPUs evoluíram para chips de 32 bits com recursos avançados, como memória protegida e troca de contexto, esses sistemas operacionais acrescentavam recursos que anteriormente só existiam em mainframes e minicomputadores. Com o tempo, os computadores pessoais se tornaram tão poderosos quanto aqueles sistemas, tornando-se mais úteis para muitos propósitos. Os minicomputadores sumiram, substituídos por “servidores” de uso genérico e especial. Embora os computadores pessoais continuem a aumentar sua capacidade e seu desempenho, os servidores tendem a permanecer à frente deles em quantidade de memória, espaço em disco e quantidade e velocidade de CPUs disponíveis. Hoje, os servidores normalmente operam em centros de dados ou salas de máquinas, com os computadores pessoais nas mesas e comunicando-se entre si e com os servidores por meio de uma rede.

A rivalidade no setor dos desktops entre Apple e Microsoft continua ainda hoje, com novas versões do Windows e do Mac OS tentando superar um ao outro em recursos, usabilidade e funcionalidade de aplicações. Outros sistemas operacionais, como o AmigaOS e o OS/2, apareceram com o passar do tempo, mas não se tornaram concorrentes dos dois principais sistemas operacionais de desktop. Nesse meio-tempo, o Linux (em suas muitas formas) continua a ganhar popularidade entre os usuários mais técnicos - e até mesmo com usuários não técnicos, em sistemas como a rede de computadores conectados para crianças **One Laptop per Child (OLPC)** - <http://laptop.org/>.

Para obter mais informações sobre esses sistemas operacionais e sua história, consulte [Freiberger e Swaine \[2000\]](#).

23.13 Mach

O sistema operacional Mach tem como ancestral o sistema operacional Accent, desenvolvido na Carnegie Mellon University - CMU ([Rashid e Robertson \[1981\]](#)). O sistema de comunicação e a filosofia do Mach são derivados do Accent, mas muitas outras partes significativas do sistema (por exemplo, o sistema de memória virtual, o gerenciamento de tarefa e threads) foram desenvolvidas do zero ([Rashid \[1986\]](#), [Tevanian e outros \[1989\]](#) e [Accetta e outros \[1986\]](#)). O escalonador do Mach foi descrito com detalhes por [Tevanian e outros \[1987a\]](#) e [Black \[1990\]](#). Uma antiga versão do sistema de memória compartilhada e mapeamento de memória do Mach foi apresentada por [Tevanian e outros \[1987b\]](#).

O sistema operacional Mach foi projetado visando aos três objetivos fundamentais a seguir:

1. Simular o UNIX 4.3BSD para que os arquivos executáveis de um sistema UNIX possam ser executados corretamente sob o Mach.
2. Ser um sistema operacional moderno que aceite muitos modelos de memória, computação paralela e distribuída.
3. Ter um kernel mais simples e mais fácil de modificar do que o 4.3BSD.

O desenvolvimento do Mach seguiu um caminho evolucionário dos sistemas BSD UNIX. O código do Mach foi desenvolvido inicialmente dentro do kernel do 4.2BSD, com os componentes do kernel do BSD sendo substituídos por componentes do Mach à medida que os componentes do Mach eram completados. Os componentes do BSD foram atualizados para o 4.3BSD quando ele ficou disponível. Em 1986, os sistemas de memória virtual e comunicação estavam rodando na família de computadores VAX da DEC, incluindo versões VAX com multiprocessadores. As versões para o RT/PC da IBM e para estações de trabalho SUN 3 vieram pouco tempo depois. Em 1987, foram concluídas as versões para multiprocessadores Encore Multimax e Sequent Balance, incluindo suporte para tarefa e thread, bem como as primeiras versões oficiais do sistema, Release 0 e Release 1.

Por meio da Release 2, o Mach forneceu compatibilidade com os sistemas BSD correspondentes, incluindo grande parte do código do BSD no kernel. Os novos recursos e capacidades do Mach tornaram os kernels nessas versões maiores do que os kernels do BSD correspondentes. O Mach 3 passou o código do BSD para fora do kernel, deixando o microkernel muito menor. Esse sistema implementa apenas os recursos básicos do Mach no kernel; todo o código específico do UNIX foi retirado para executar em servidores no modo usuário. A retirada do kernel de todo o código específico do UNIX permite a substituição do BSD por outro sistema operacional ou a execução simultânea de múltiplas interfaces do sistema operacional em cima do microkernel. Além do BSD, as implementações do modo usuário foram desenvolvidas para DOS, o sistema operacional Macintosh e o OSF/1. Essa técnica possui semelhanças com o conceito de máquina virtual, mas a máquina virtual é definida pelo software (a interface do núcleo do Mach) e não pelo hardware. Na Release 3.0, o Mach tornou-se disponível em uma grande variedade de sistemas, incluindo máquinas SUN, Intel, IBM e DEC de único processador, e sistemas DEC, Sequent e Encore de multiprocessadores.

O Mach recebeu maior atenção do setor quando a Open Software Foundation (OSF) anunciou, em 1989, que usaria o Mach 2.5 como a base para o seu novo sistema operacional, OSF/1. (O Mach 2.5 também é a base para o sistema operacional na estação de trabalho NeXT, a ideia brilhante de Steve Jobs, da famosa Apple Computer.) A versão inicial do OSF/1 ocorreu um ano depois e esse sistema competiu com o UNIX System V, Release 4, o sistema operacional preferido naquela época entre os membros da UNIX International. Os membros da OSF incluíam empresas tecnológicas importantes, como IBM, DEC e HP. Desde então, a OSF tem mudado sua direção, e apenas o Unix da DEC é baseado no kernel do Mach.

Ao contrário do UNIX, desenvolvido sem considerar o multiprocessamento, o Mach incorpora o suporte integral de multiprocessamento. Esse suporte também é extremamente flexível, variando desde sistemas de memória compartilhada até sistemas sem qualquer memória compartilhada entre os processadores. O Mach usa processos leves, na forma de múltiplas threads dentro de uma tarefa (ou espaço de endereços), para dar suporte ao multiprocessamento e à computação paralela. Seu extenso uso de mensagens como único método de comunicação garante que os mecanismos de proteção sejam completos e eficientes. Integrando mensagens com o sistema de memória virtual, o Mach também garante que as mensagens possam ser tratadas de forma eficiente. Finalmente, fazendo o sistema de memória virtual utilizar mensagens para se comunicar com os daemons que gerenciam o armazenamento de apoio, o Mach oferece grande flexibilidade no projeto e na implementação dessas tarefas de gerenciamento de objetos da memória. Oferecendo chamadas do sistema de baixo nível, ou primitivos dos quais podem ser montadas funções mais complexas, o Mach reduz o tamanho do kernel, enquanto permite a simulação do sistema operacional no nível do usuário, de modo semelhante aos sistemas de máquina virtual da IBM.

Algumas edições anteriores deste livro incluíram um capítulo inteiro sobre o Mach. Esse capítulo, conforme apareceu na quarta edição original, está disponível na Web (<http://www.os-book.com>).

23.14 Outros sistemas

Naturalmente, existem outros sistemas operacionais, e a maioria possui propriedades interessantes. O sistema operacional MCP, para a família de computadores Burroughs ([McKeag e Wilson \[1976\]](#)), foi o primeiro a ser escrito em uma linguagem de programação de sistemas. Ele admitia segmentação e múltiplas CPUs. O sistema operacional SCOPE, para o CDC 6600 ([McKeag e Wilson \[1976\]](#)), também foi um sistema para múltiplas CPUs. A coordenação e o sincronismo entre os múltiplos processos foram surpreendentemente bem projetados.

A história está repleta de sistemas operacionais que se adequaram a um propósito em uma época (seja ela curta ou longa) e depois, quando desapareciam, eram substituídos por sistemas operacionais que tinham mais recursos, com suporte para hardware mais novo, que eram mais fáceis de usar ou mais bem comercializados. Estamos certos de que essa tendência continuará no futuro.

Exercícios

- 23.1. Discuta que considerações o operador de computador levava em conta ao decidir sobre as sequências em que os programas seriam executados nos primeiros sistemas computadorizados que eram operados manualmente.
- 23.2. Que otimizações foram usadas para reduzir a discrepância entre as velocidades de CPU e E/S nos primeiros sistemas computadorizados?
- 23.3. Considere o algoritmo de substituição de página usado pelo Atlas. De que maneiras ele é diferente do algoritmo de relógio, discutido na [Seção 9.4.5.2](#)?
- 23.4. Considere a fila multinível com feedback usada pelo CTSS e o MULTICS. Suponha que um programa use consistentemente sete unidades de tempo toda vez que for escalonado, antes de realizar uma operação de E/S, e depois seja bloqueado. Quantas unidades de tempo são alocadas a esse programa quando ele é escalonado para execução em diferentes pontos no tempo?
- 23.5. Quais são as implicações do suporte à funcionalidade do BSD nos servidores no modo usuário dentro do sistema operacional Mach?
- 23.6. A que conclusões podemos chegar sobre a evolução dos sistemas operacionais? O que faz alguns sistemas operacionais ganharem popularidade e outros desaparecerem?

Bibliografia

- Accetta, M., Baron, R., Bolosky, W., Golub, D. B., Rashid, R., Tevanian, A., Young, M., "Mach: A New Kernel Foundation for Development". *Proceedings of the Summer USENIX Conference* 1986; 93-112.
- Adl-Tabatabai, A. R., Kozyrakis, C., Saha, B. "Unlocking Concurrency". *Queue*. 2007; 4(10):24-33.
- Agrawal, D. P., Abbadi, A. E. "An Efficient and Fault-Tolerant Solution of Distributed Mutual Exclusion". *ACM Transactions on Computer Systems*. 1991; 9(1):1-20.
- Agre, P. E. "P2P and the Promise of Internet Equality". *Communications of the ACM*. 2003; 46(2):39-42.
- Ahituv, N., Lapid, Y., Neumann, S. "Processing Encrypted Data". *Communications of the ACM*. 1987; 30(9):777-780.
- Ahmed, I. "Cluster Computing: A Glance at Recent Events". *IEEE Concurrency*. 8(1), 2000.
- Akl, S. G. "Digital Signatures: A Tutorial Survey". *Computer*. 1983; 16(2):15-24.
- Akyurek, S., Salem, K. "Adaptive Block Rearrangement". *Proceedings of the International Conference on Data Engineering*. 1993; 182-189.
- Alt, H. "Removable Media in Solaris". *Proceedings of the Winter USENIX Conference*. 1993; 281-287.
- Anderson, T. E. "The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors". *IEEE Trans. Parallel Distrib. Syst.*.. 1990; 1(1):6-16.
- Anderson, T. E., Lazowska, E. D., Levy, H. M. "The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors". *IEEE Transactions on Computers*. 1989; 38(12):1631-1644.
- Anderson, T. E., Bershad, B. N., Lazowska, E. D., Levy, H. M. "Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1991; 95-109.
- Anderson, T. E., Dahlin, M. D., Neefe, J. M., Petterson, D. A., Roselli, D. S., Wang, R. Y. "Serverless Network File Systems". *Proceedings of the ACM Symposium on Operating System Principles*. 1995; 109-126.
- Anderson, T. E., Chase, J., Vahdat, A. "Interposed Request Routing for Scalable Network Storage". *Proceedings of the Fourth Symposium on Operating Systems Design and Implementation*. 2000.
- Apple Technical Introduction to the Macintosh Family. Addison-Wesley, 1987.
- Asthana, P., Finkelstein, B. "Superdense Optical Storage". *IEEE Spectrum*. 1995; 32(8):25-31.
- Audsley, N. C., Burns, A., Richardson, M. F., Wellings, A. J. "Hard Real-Time Scheduling: The Deadline Monotonic Approach". *Proceedings of the IEEE Workshop on Real-Time Operating Systems and Software*. 1991.
- Axelsson, S., "The Base-Rate Fallacy and Its Implications for Intrusion Detection". *Proceedings of the ACM Conference on Computer and Communications Security* 1999; 1-7
- Babaoglu, O., Marzullo, K. "Consistent Global States of Distributed Systems: Fundamental Concepts and Mechanisms". *Distributed Systems*. 1993; 55-96.
- Bach, M.J.The Design of the UNIX Operating System. Prentice Hall, 1987.
- Back, G., Tullman, P., Stoller, L., Hsieh, W. C., Lepreau, J. "Techniques for the Design of Java Operating Systems". *2000 USENIX Annual Technical Conference*. 2000.
- Baker, M. G., Hartman, J. H., Kupter, M. D., Shirriff, K. W., Ousterhout, J. K. "Measurements of a Distributed File System". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1991; 198-212.
- Balakrishnan, H., Kaashoek, M. F., Karger, D., Morris, R., Stoica, I. "Looking Up Data in P2P Systems". *Communications of the ACM*. 2003; 46:43-48. [Número 2].
- Baldwin, J. "Locking in the Multithreaded FreeBSD Kernel". *USENIX BSD*. 2002.
- Barnes, G. "A Method for Implementing Lock-Free Shared Data Structures". *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*. 1993; 261-270.
- Barrera, J. S. "A Fast Mach Network IPC Implementation". *Proceedings of the USENIX Mach Symposium*. 1991; 1-12.
- Basu, A., Buch, V., Vogels, W., von Eicken, T. "U-Net: A User-Level Network Interface for Parallel and Distributed Computing". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1995.
- Bays, C. "A Comparison of Next-Fit, First-Fit and Best-Fit". *Communications of the ACM*. 1977; 20(3):191-192.
- Belady, L. A. "A Study of Replacement Algorithms for a Virtual-Storage Computer". *IBM Systems Journal*. 1966; 5(2):78-101.
- Belady, L. A., Nelson, R. A., Shedler, G. S. "An Anomaly in Space-Time Characteristics of Certain Programs Running in a Paging Machine". *Communications of the ACM*. 1969; 12(6):349-353.
- Bellovin, S. M. "Security Problems in the TCP/IP Protocol Suite". *Computer Communications Review*. 1989; 19(2):32-48.
- Ben-Ari, M. Principles of Concurrent and Distributed Programming. Prentice Hall, 1990.
- Benjamin, C. D. "The Role of Optical Storage Technology for NASA". *Proceedings, Storage and Retrieval Systems and Applications*. 1990; 10-17.
- Bernstein, P. A., Goodman, N. "Time-Stamp-Based Algorithms for Concurrency Control in Distributed Database Systems". *Proceedings of the International Conference on Very Large Databases*. 1980; 285-300.
- Bernstein, A., Hadzilacos, V., Goodman, N. Concurrency Control and Recovery in Database Systems. Addison-Wesley, 1987.
- Bershad, B. "Practical Considerations for Non-Blocking Concurrent Objects". *IEEE International Conference on Distributed Computing Systems*. 1993; 264-273.
- Bershad, B. N., Pinkerton, C. B. "Watchdogs: Extending the Unix File System". *Proceedings of the Winter USENIX Conference*. 1988.
- Bershad, B. N., Anderson, T. E., Lazowska, E. D., Levy, H. M. "Lightweight Remote Procedure Call". *ACM Transactions on Computer Systems*. 1990; 8(1):37-55.
- Bershad, B. N., Savage, S., Pardyak, P., Sirer, E. G., Fiuczynski, M., Becker, D., Eggers, S., Chambers, C. "Extensibility, Safety and Performance in the SPIN Operating System". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1995; 267-284.
- Beveridge, J., Wiener, R. Multithreading Applications in Win32. Addison-Wesley, 1997.
- Binding, C. "Cheap Concurrency in C". *SIGPLAN Notices*. 1985; 20(9):21-27.
- Birrell, A.D. "An Introduction to Programming with Threads". Technical Report 35. DEC-SRC, 1989
- Birrell, A. D., Nelson, B. J. "Implementing Remote Procedure Calls". *ACM Transactions on Computer Systems*. 1984; 2(1):39-59.
- Blaauw, G., Brooks, F.Computer Architecture: Concepts and Evolution. Addison-Wesley, 1997.
- Black, D. L. "Scheduling Support for Concurrency and Parallelism in the Mach Operating System". *IEEE Computer*. 1990; 23(5):35-43.
- Bobrow, D. G., Burchfiel, J. D., Murphy, D. L., Tomlinson, R. S. "TENEX, a Paged Time Sharing System for the PDP-10". *Communications of the ACM*. 15(3), 1972.
- Bolosky, W. J., Fitzgerald, R. P., Douceur, J. R. "Distributed Schedule Management in the Tiger Video Fileserver". *Proceedings of the*

- ACM Symposium on Operating Systems Principles*. 1997; 212–223.
- Bonwick, J. "The Slab Allocator: An Object-Caching Kernel Memory Allocator". *USENIX Summer*. 1994; 87–98.
- Bonwick, J., Adams, J. "Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources". *Proceedings of 2001 USENIX Annual Technical Conference*. 2001.
- Bovet, D.P., Cesati, M.Understanding the Linux Kernel, Segunda Edição. O'Reilly & Associates, 2002.
- Bovet, D., Cesati, M.Understanding the Linux Kernel, Terceira Edição. O'Reilly & Associates, 2006.
- Brain, M.Win32 System Services, Segunda Edição. Prentice Hall, 1996.
- Brent, R. "Efficient Implementaion of the First-Fit Strategy for Dynamic Storage Allocation". *ACM Transactions on Programming Languages and Systems*. 1989; 11(3):388–403.
- Brereton, O. P. "Management of Replicated Files in a UNIX Environment". *Software - Practice and Experience*. 1986; 16:771–780.
- Brinch-Hansen, P. "The Nucleus of a Multiprogramming System". *Communications of the ACM*. 1970; 13(4):238–241. [e 250.]
- Brinch-Hansen, P. "Structured Multiprogramming". *Communications of the ACM*. 1972; 15(7):574–578.
- Brinch-Hansen, P.Operating System Principles. Prentice Hall, 1973.
- Brookshear, J.G.Computer Science: An Overview, Sétima Edição. Addison-Wesley, 2003.
- Brownbridge, D. R., Marshall, L. F., Randell, B. "The Newcastle Connection or UNIXes of the World Unite!". *Software - Practice and Experience*. 1982; 12(12):1147–1162.
- Burns, J. E. "Mutual Exclusion with Linear Waiting Using Binary Shared Variables". *SIGACT News*. 1978; 10(2):42–47.
- Butenhof, D.Programming with POSIX Threads. Addison-Wesley, 1997.
- Buyya, R.High Performance Cluster Computing: Architectures and Systems. Prentice Hall, 1999.
- Callaghan, B.NFS Illustrated. Addison-Wesley, 2000.
- Cantrill, B. M., Shapiro, M. W., Leventhal, A. H. "Dynamic Instrumentation of Production Systems". *2004 USENIX Annual Technical Conference*. 2004.
- Carr, W. R., Hennessy, J. L. "WSClock - A Simple and Effective Algorithm for Virtual Memory Management". *Proceedings of the ACM Symposium on Operating System Principles*. 1981; 87–95.
- Carvalho, O. S., Roucair, G. "On Mutual Exclusion in Computer Networks". *Communications of the ACM*. 1983; 26(2):146–147.
- Ceruzzi, P.E. A History of Modern Computing. MIT Press, 1998
- Chandy, K. M., Lamport, L. "Distributed Snapshots: Determining Global States of Distributed Systems". *ACM Transactions on Computer Systems*. 1985; 3(1):63–75.
- Chang, E. "N-Philosophers: An Exercise in Distributed Control". *Computer Networks*. 1980; 4(2):71–76.
- Chang, A., Mergen, M. F. "801 Storage: Architecture and Programming". *ACM Transactions on Computer Systems*. 1988; 6(1):28–50.
- Chase, J. S., Levy, H. M., Feeley, M. J., Lazowska, E. D. "Sharing and Protection in a Single-Address-Space Operating System". *ACM Transactions on Computer Systems*. 1994; 12(4):271–307.
- Chen, P. M., Lee, E. K., Gibson, G. A., Katz, R. H., Patterson, D. A. "RAID: High-Performance, Reliable Secondary Storage". *ACM Computing Survey*. 1994; 26(2):145–185.
- Cheriton, D. "The V Distributed System". *Communications of the ACM*. 1988; 31(3):314–333.
- Cheriton, D. R., Malcolm, M. A., Melen, L. S., Sager, G. R. "Thoth, a Portable Real-Time Operating System". *Communications of the ACM*. 1979; 22(2):105–115.
- Cheswick, W., Bellovin, S., Rubin, A.Firewalls and Internet Security: Repeling the Wily Hacker. segunda edição, Addison-Wesley, 2003.
- Cheung, W. H., Loong, A. H.S. "Exploring Issues of Operating Systems Structuring: From Microkernel to Extensible Systems". *Operating Systems Review*. 1995; 29(4):4–16.
- Chi, C. S. "Advances in Computer Mass Storage Technology". *Computer*. 1982; 15(5):60–74.
- Coffman, E. G., Elphick, M. J., Shoshani, A. "System Deadlocks". *Computing Surveys*. 1971; 3(2):67–78.
- Cohen, E. S., Jefferson, D. "Protection in the Hydra Operating System". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1975; 141–160.
- Cohen, A., Woodring, M.Win32 Multithreaded Programming. O'Reilly & Associates, 1997.
- Comer, D.Internetworking with TCP/IP, Volume II, Terceira Edição. Prentice Hall, 1999.
- Comer, D.Internetworking with TCP/IP, Volume I, Quarta Edição. Prentice Hall, 2000.
- Corbato, F. J., Vyssotsky, V. A. "Introduction and Overview of the MULTICS System". *Proceedings of the AFIPS Fall Joint Computer Conference*. 1965; 185–196.
- Corbato, F. J., Merwin-Daggett, M., Daley, R. C. "An Experimental Time-Sharing System". *Proceedings of the AFIPS Fall Joint Computer Conference*. 1962; 335–344.
- Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.Introduction to Algorithms, Segunda Edição. MIT Press, 2001.
- Coulouris, G., Dollimore, J., Kindberg, T.Distributed Systems Concepts and Designs, Terceira Edição. Addison Wesley, 2001.
- Courtois, P. J., Heymans, F., Parnas, D. L. "Concurrent Control with 'Readers' and 'Writers'". *Communications of the ACM*. 1971; 14(10):667–668.
- Culler, D.E., Singh, J.P., Gupta, A.Parallel Computer Architecture: A Hardware/Software Approach. Morgan Kaufmann Publishers Inc., 1998.
- Custer, H.Inside the Windows NT File System. Microsoft Press, 1994.
- Dabek, F., Kaashoek, M. F., Karger, D., Morris, R., Stoica, I. "Wide-Area Cooperative Storage with CFS". *Proceedings of the ACM Symposium on Operating Systems Principles*. 2001; 202–215.
- Daley, R. C., Dennis, J. B. "Virtual Memory, Processes, and Sharing in Multics". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1967; 121–128.
- Davcev, D., Burkhard, W. A. "Consistency and Recovery Control for Replicated Files". *Proceedings of the ACM Symposium on Operating Systems Principles* 1985; 87–96
- Davies, D. W. "Applying the RSA Digital Signature to Electronic Mail". *Computer*. 1983; 16(2):55–62.
- deBrujin, N. G. "Additional Comments on a Problem in Concurrent Programming and Control". *Communications of the ACM*. 1967; 10(3):137–138.
- Deitel, H.M.An Introduction to Operating Systems, Segunda Edição. Addison-Wesley, 1990.
- Denning, P. J. "The Working Set Model for Program Behavior". *Communications of the ACM*. 1968; 11(5):323–333.
- Denning, P. J. "Working Sets Past and Present". *IEEE Transactions on Software Engineering*. 1980; SE-6(1):64–84.
- Denning, D.E.Cryptography and Data Security. Addison-Wesley, 1982.
- Denning, D. E. "Protecting Public Keys and Signature Keys". *Computer*. 1983; 16(2):27–35.
- Denning, D. E. "Digital Signatures with RSA and Other Public-Key Cryptosystems". *Communications of the ACM*. 1984; 27(4):388–392.
- Denning, D. E., Denning, P. J. "Data Security". *ACM Comput. Surv.* 1979; 11(3):227–249.
- Dennis, J. B. "Segmentation and the Design of Multiprogrammed Computer Systems". *Communications of the ACM*. 1965; 8(4):589–602.
- Dennis, J. B., Horn, E. C.V. "Programming Semantics for Multiprogrammed Computations". *Communications of the ACM*. 1966;

- 9(3):143-155.
- Di Pietro, R., Mancini, L. V. "Security and Privacy Issues of Handheld and Wearable Wireless Devices". *Communications of the ACM*. 2003; 46(9):74-79.
- Diffie, W., Hellman, M. E. "New Directions in Cryptography". *IEEE Transactions on Information Theory*. 1976; 22(6):644-654.
- Diffie, W., Hellman, M. E. "Privacy and Authentication". *Proceedings of the IEEE*. 1979; 397-427.
- Dijkstra, E.W."Cooperating Sequential Processes". Technical Report. Technological University, Eindhoven, the Netherlands, 1965.
- Dijkstra, E. W. "Solution of a Problem in Concurrent Programming Control". *Communication of the ACM*. 1965; 8(9):569.
- Dijkstra, E. W. "The Structure of the THE Multiprogramming System". *Communications of the ACM*. 1968; 11(5):341-346.
- Dijkstra, E. W. "Hierarchical Ordering of Sequential Processes". *Acta Informatica*. 1971; 1(2):115-138.
- Trusted Computer System Evaluation Criteria. Department of Defense, 1985.
- Dougan, C., Mackerras, P., Yodaiken, V. "Optimizing the Idle Task and Other MMU Tricks". *Proceedings of the Symposium on Operating System Design and Implementation*. 1999.
- Douglis, F., Ousterhout, J. K. "Transparent Process Migration: Design Alternatives and the Sprite Implementation". *Software - Practice and Experience*. 1991; 21(8):757-785.
- Douglis, F., Kaashoek, F., Li, K., Caceres, R., Marsh, B., Tauber, J. A., "Storage Alternatives for Mobile Computers". *Proceedings of the ACM Symposium on Operating Systems Design and Implementation* 1994; 25-37
- Douglis, F., Krishnan, P., Bershad, B. "Adaptive Disk Spin-Down Policies for Mobile Computers". *Proceedings of the USENIX Symposium on Mobile and Location Independent Computing*. 1995; 121-137.
- Draves, R. P., Bershad, B. N., Rashid, R. F., Dean, R. W. "Using Continuations to Implement Thread Management and Communication in Operating Systems". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1991; 122-136.
- Druschel, P., Peterson, L. L. "Fbufs: A High-Bandwidth Cross-Domain Transfer Facility". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1993; 189-202.
- Eastlake, D. "Domain Name System Security Extensions". *Network Working Group, Request for Comments*: 2535. 1999.
- Eisenberg, M. A., McGuire, M. R. "Further Comments on Dijkstra's Concurrent Programming Control Problem". *Communications of the ACM*. 1972; 15(11):999.
- Ekanadham, K., Bernstein, A. J. "Conditional Capabilities". *IEEE Transactions on Software Engineering*. 1979; SE-5(5):458-464.
- Engelschall, R. "Portable Multithreading: The Signal Stack Trick For User-Space Thread Creation". *Proceedings of the 2000 USENIX Annual Technical Conference*. 2000.
- Eswaran, K. P., Gray, J. N., Lorie, R. A., Traiger, I. L. "The Notions of Consistency and Predicate Locks in a Database System". *Communications of the ACM*. 1976; 19(11):624-633.
- Fang, Z., Zhang, L., Carter, J. B., Hsieh, W. C., McKee, S. A. "Reevaluating Online Superpage Promotion with Hardware Support". *Proceedings of the International Symposium on High-Performance Computer Architecture*. 50(5), 2001.
- Farley, J. Java Distributed Computing. O'Reilly & Associates, 1998.
- Farrow, R. "Security for Superusers, or How to Break the UNIX System". *UNIX World*. 1986; 65-70.
- R. Farrow. "Security Issues and Strategies for Users". *UNIX World (abril de 1986)*, p. 65-71.
- Fidge, C., "Logical Time in Distributed Computing Systems". *Computer*, 1991;24(8):28-33
- R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach e T. Berners-Lee. "Hypertext Transfer Protocol - HTTP/1.1" (1999).
- Filipski, A., Hanko, J. "Making UNIX Secure". *Byte*. abril de 1986; 113-128.
- Fisher, J. A. "Trace Scheduling: A Technique for Global Microcode Compaction". *IEEE Transactions on Computers*. 1981; 30(7):478-490.
- Folk, M.J., Zoellick, B. File Structures. Addison-Wesley, 1987.
- Forrest, S., Hofmeyr, S. A., Longstaff, T. A. "A Sense of Self for UNIX Processes". *Proceedings of the IEEE Symposium on Security and Privacy*. 1996; 120-128.
- Fortier, P.J. Handbook of LAN Technology. McGraw-Hill, 1989.
- Frah, G. The Universal History of Computing. John Wiley and Sons, 2001.
- Frauenfelder, M. The Computer - An Illustrated History. Carlton Books, 2005.
- Freedman, D. H. "Searching for Denser Disks". *Infosystems*. 1983; 56.
- Freiberger, P., Swaine, M. Fire in the Valley - The Making of the Personal Computer. McGraw-Hill, 2000.
- Fuhrt, B. "Multimedia Systems: An Overview". *IEEE MultiMedia*. 1994; 1:47-59. [Número 1].
- Fujitani, L. "Laser Optical Disk: The Coming Revolution in On-Line Storage". *Communications of the ACM*. 1984; 27(6):546-554.
- Gait, J. "The Optical File Cabinet: A Random-Access File System for Write-On Optical Disks". *Computer*. 21(6), 1988.
- Ganapathy, N., Schimmel, C. "General Purpose Operating System Support for Multiple Page Sizes". *Proceedings of the USENIX Technical Conference*. 1998.
- Ganger, G. R., Engler, D. R., Kaashoek, M. F., Briceno, H. M., Hunt, R., Pinckney, T. "Fast and Flexible Application-Level Networking on Exokernel Systems". *ACM Transactions on Computer Systems*. 2002; 20(1):49-83.
- Garcia-Molina, H. "Elections in Distributed Computing Systems". *IEEE Transactions on Computers*. C-31(1), 1982.
- Garfinkel, S., Spafford, G., Schwartz, A. Practical UNIX & Internet Security. O'Reilly & Associates, 2003.
- Ghemawat, S., Gobioff, H., Leung, S.-T. "The Google File System". *Proceedings of the ACM Symposium on Operating Systems Principles*. 2003.
- Gibson, G., Nagle, D., Amiri, K., Chang, F., Gobioff, H., Riedel, E., Rochberg, D., Zelenka, J. "Filesystems for Network-Attached Secure Disks". Technical Report. Carnegie-Mellon University, 1997.
- Gibson, G. A., Nagle, D., Amiri, K., Chang, F. W., Feinberg, E. M., Gobioff, H., Lee, C., Ozceri, B., Riedel, E., Rochberg, D., Zelenka, J. "File Server Scaling with Network-Attached Secure Disks". *Measurement and Modeling of Computer Systems*. 1997; 272-284.
- Gifford, D. K. "Cryptographic Sealing for Information Secrecy and Authentication". *Communications of the ACM*. 1982; 25(4):274-286.
- Goetz, B., Peirls, T., Bloch, J., Bowbeer, J., Holmes, D., Lea, D. Java Concurrency in Practice. Addison-Wesley, 2006.
- Goldberg, I., Wagner, D., Thomas, R., Brewer, E. A. "A Secure Environment for Untrusted Helper Applications". *Proceedings of the 6th Usenix Security Symposium*. 1996.
- Golden, D., Pechura, M. "The Structure of Microcomputer File Systems". *Communications of the ACM*. 1986; 29(3):222-230.
- Golding, R. A., Bosch, P., Staelin, C., Sullivan, T., Wilkes, J. "Idleness is Not Sloth". *USENIX Winter*. 1995; 201-212.
- Golm, M., Felser, M., Wawersich, C., Kleinoder, J. "The JX Operating System". *2002 USENIX Annual Technical Conference*. 2002.
- Gong, L. "Peer-to-Peer Networks in Action". *IEEE Internet Computing*. 6(1), 2002.
- Gong, L., Mueller, M., Prafullchandra, H., Schemers, R. "Going Beyond the Sandbox: An Overview of the New Security Architecture in the Java Development Kit 1.2". *Proceedings of the USENIX Symposium on Internet Technologies and Systems*. 1997.
- Goodman, J. R., Vrenon, M. K., Woest, P. J. "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors". *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 1989; 64-75.
- Goodrich, M.T., Tamassia, R. Data Structures and Algorithms in Java, Quarta Edição. John Wiley and Sons, 2006.

- Gosling, J., Joy, B., Steele, G., Bracha, G. *The Java Language Specification*, Terceira Edição. Addison-Wesley, 2005.
- Govindan, R., Anderson, D. P. "Scheduling and IPC Mechanisms for Continuous Media". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1991; 68-80.
- Grampp, F. T., Morris, R. H. "UNIX Operating-System Security". *AT&T Bell Laboratories Technical Journal*. 1984; 63(8):1649-1672.
- Gray, J. N. "The Transaction Concept: Virtues and Limitations". *Proceedings of the International Conference on Very Large Databases*. 1981; 144-154.
- Gray, J. *Interprocess Communications in UNIX*. Prentice Hall, 1997.
- Gray, J. N., McJones, P. R., Blasgen, M. "The Recovery Manager of the System R Database Manager". *ACM Computing Survey*. 1981; 13(2):223-242.
- Greenawalt, P. "Modeling Power Management for Hard Disks". *Proceedings of the Symposium on Modeling and Simulation of Computer Telecommunication Systems*. 1994; 62-66.
- Grosshans, D. *File Systems Design and Implementation*. Prentice Hall, 1986.
- Grosso, W. *Java RMI*. O'Reilly & Associates, 2002.
- Habermann, A. N., "Prevention of System Deadlocks". *Communications of the ACM*, 1969; 12(7):373-377 385.
- Hall, L., Shmoys, D., Wein, J. "Scheduling To Minimize Average Completion Time: Off-line and On-line Algorithms". *SODA: ACM-SIAM Symposium on Discrete Algorithms*. 1996.
- Hamacher, C., Vranesic, Z., Zaky, S. *Computer Organization*, Quinta Edição. McGraw-Hill, 2002.
- Han, K., Ghosh, S. "A Comparative Analysis of Virtual Versus Physical Process-Migration Strategies for Distributed Modeling and Simulation of Mobile Computing Networks". *Wireless Networks*. 1998; 4(5):365-378.
- Harchol-Balter, M., Downey, A. B. "Exploiting Process Lifetime Distributions for Dynamic Load Balancing". *ACM Transactions on Computer Systems*. 1997; 15(3):253-285.
- Harish, V. C., Owens, B. "Dynamic Load Balancing DNS". *Linux Journal*. 1999(64), 1999.
- Harker, J. M., Brede, D. W., Pattison, R. E., Santana, G. R., Taft, L. G. "A Quarter Century of Disk File Innovation". *IBM Journal of Research and Development*. 1981; 25(5):677-689.
- Harold, E. *Java Network Programming*. O'Reilly & Associates, 2000.
- Harold, E. *Java Network Programming*, Terceira Edição. O'Reilly & Associates, 2005.
- Harrison, M. A., Ruzzo, W. L., Ullman, J. D. "Protection in Operating Systems". *Communications of the ACM*. 1976; 19(8):461-471.
- Hart, J.M. *Windows System Programming*, Terceira Edição. Addison-Wesley, 2005.
- Hartman, J. H., Ousterhout, J. K. "The Zebra Striped Network File System". *ACM Transactions on Computer Systems*. 1995; 13(3):274-310.
- Havender, J. W. "Avoiding Deadlock in Multitasking Systems". *IBM Systems Journal*. 1968; 7(2):74-84.
- Hecht, M. S., Johri, A., Aditham, R., Wei, T. J. "Experience Adding C2 Security Features to UNIX". *Proceedings of the Summer USENIX Conference*. 1988; 133-146.
- Hennessy, J.L., Patterson, D.A. *Computer Architecture: A Quantitative Approach*, Quarta Edição. Morgan Kaufmann Publishers, 2007.
- Henry, G. "The Fair Share Scheduler". *AT&T Bell Laboratories Technical Journal*. 1984.
- Herlihy, M. "A Methodology for Implementing Highly Concurrent Data Objects". *ACM Transactions on Programming Languages and Systems*. 1993; 15(5):745-770.
- Herlihy, M., Moss, J. E.B. "Transactional Memory: Architectural Support For Lock-Free Data Structures". *Proceedings of the Twentieth Annual International Symposium on Computer Architecture*. 1993.
- Hitchens, R. *Java NIO*. O'Reilly & Associates, 2002.
- Hitz, D., Lau, J., Malcolm, M. "File System Design for an NFS File Server Appliance". Technical Report, NetApp. http://www.netapp.com/tech_library/3002.html. [(1995).].
- Hoagland, A. S. "Information Storage Technology - A Look at the Future". *Computer*. 1985; 18(7):60-68.
- C. A. R. Hoare. "Towards a Theory of Parallel Programming", em Hoare e Perrott 1972 (1972), p. 61-71.
- Hoare, C. A.R. "Monitors: An Operating System Structuring Concept". *Communications of the ACM*. 1974; 17(10):549-557.
- Holt, R. C. "Comments on Prevention of System Deadlocks". *Communications of the ACM*. 1971; 14(1):36-38.
- Holt, R. C. "Some Deadlock Properties of Computer Systems". *Computing Surveys*. 1972; 4(3):179-196.
- Holub, A. *Taming Java Threads*. Apress, 2000.
- Howard, J. H., Kazar, M. L., Menees, S. G., Nichols, D. A., Satyanarayanan, M., Sidebotham, R. N., "Scale and Performance in a Distributed File System". *ACM Transactions on Computer Systems*, 1988;6(1):55-81
- Howarth, D. J., Payne, R. B., Sumner, F. H. "The Manchester University Atlas Operating System, Part II: User's Description". *Computer Journal*. 1961; 4(3):226-229.
- Hsiao, D.K., Kerr, D.S., Madnick, S.E. *Computer Security*. Academic Press, 1979.
- Hu, Y.-C., Perrig, A. "SPV: A Secure Path Vector Routing Scheme for Securing BGP". *Proceedings of the Annual International Conference on Mobile Computing and Networking*. 2004.
- Hu, Y.-C., Perrig, A., Johnson, D. "Ariadne: A Secure On-Demand Routing Protocol for Ad Hoc Networks". *Proceedings of the Annual International Conference on Mobile Computing and Networking*. 2002.
- Hyman, D. *The Columbus Chicken Statue and More Bonehead Legislation*. S. Greene Press, 1985.
- Iacobucci, E. *OS/2 Programmer's Guide*. Osborne McGraw-Hill, 1988.
- Technical Reference. IBM Corporation, 1983.
- Iliffe, J. K., Jodeit, J. G. "A Dynamic Storage Allocation System". *Computer Journal*. 1962; 5(3):200-209.
- iAPX 286 Programmer's Reference Manual. Intel Corporation, 1985.
- iAPX 86/88, 186/188 User's Manual Programmer's Reference. Intel Corporation, 1985.
- iAPX 386 Programmer's Reference Manual. Intel Corporation, 1986.
- i486 Microprocessor Programmer's Reference Manual. Intel Corporation, 1990.
- Pentium Processor User's Manual, Volume 3: Architecture and Programming Manual. Intel Corporation, 1993.
- Iseminger, D. *Active Directory Services for Microsoft Windows 2000*. Technical Reference. Microsoft Press, 2000.
- Jacob, B., Mudge, T. "Software-Managed Address Translation". *Proceedings of the International Symposium on High Performance Computer Architecture and Implementation*. 1997.
- Jacob, B., Mudge, T. "Virtual Memory in Contemporary Microprocessors". *IEEE Micro Magazine*. 1998; 18:60-75.
- Jacob, B., Mudge, T. "Virtual Memory: Issues of Implementation". *IEEE Computer Magazine*. 1998; 31(6):33-43.
- Jacob, B., Mudge, T. "Uniprocessor Virtual Memory Without TLBs". *IEEE Transactions on Computers*. 50(5), 2001.
- Jacobson, D.M., Wilkes, J. "Disk Scheduling Algorithms Based on Rotational Position". Technical report. Hewlett-Packard Laboratories, 1991.
- Jensen, E. D., Locke, C. D., Tokuda, H. "A Time-Driven Scheduling Model for Real-Time Operating Systems". *Proceedings of the IEEE Real-Time Systems Symposium*. 1985; 112-122.
- Johnstone, M. S., Wilson, P. R. "The Memory Fragmentation Problem: Solved?". *Proceedings of the First International Symposium on Memory Management*. 1998; 26-36.

- Jones, A. K., Liskov, B. H. "A Language Extension for Expressing Constraints on Data Access". *Communications of the ACM*. 1978; 21(5):358-367.
- Jul, E., Levy, H., Hutchinson, N., Black, A. "Fine-Grained Mobility in the Emerald System". *ACM Transactions on Computer Systems*. 1988; 6(1):109-133.
- Kaashoek, M. F., Engler, D. R., Ganger, G. R., Briceno, H. M., Hunt, R., Mazieres, D., Pinckney, T., Grimm, R., Jannotti, J., Mackenzie, K. "Application Performance and Flexibility on Exokernel Systems". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1997; 52-65.
- Katz, R. H., Gibson, G. A., Patterson, D. A., "Disk System Architectures for High Performance Computing". *Proceedings of the IEEE* 1989;
- Kay, J., Lauder, P. "A Fair Share Scheduler". *Communications of the ACM*. 1988; 31(1):44-55.
- Kenah, L.J., Goldenberg, R.E., Bate, S.F.VAX/VMS Internals and Data Structures. Digital Press, 1988.
- Kent, S., Lynn, C., Seo, K. "Secure Border Gateway Protocol (Secure-BGP)". *IEEE Journal on Selected Areas in Communications*. 2000; 18(4):582-592.
- Kenville, R. F. "Optical Disk Data Storage". *Computer*. 1982; 15(7):21-26.
- Kessels, J. L.W. "An Alternative to Event Queues for Synchronization in Monitors". *Communications of the ACM*. 1977; 20(7):500-503.
- Kieburz, R. B., Silberschatz, A. "Capability Managers". *IEEE Transactions on Software Engineering*. 1978; SE-4(6):467-477.
- Kieburz, R. B., Silberschatz, A. "Access Right Expressions". *ACM Transactions on Programming Languages and Systems*. 1983; 5(1):78-96.
- Kilburn, T., Howarth, D. J., Payne, R. B., Sumner, F. H. "The Manchester University Atlas Operating System, Part I: Internal Organization". *Computer Journal*. 1961; 4(3):222-225.
- Kim, G. H., Spafford, E. H. "The Design and Implementation of Tripwire: A File System Integrity Checker". *Technical Report, Purdue University*. 1993.
- King, R. P. "Disk Arm Movement in Anticipation of Future Requests". *ACM Transactions on Computer Systems*. 1990; 8(3):214-229.
- Kistler, J., Satyanarayanan, M. "Disconnected Operation in the Coda File System". *ACM Transactions on Computer Systems*. 1992; 10(1):3-25.
- Kleinrock, L. Queueing Systems, Volume II: Computer Applications. Wiley-Interscience, 1975.
- Knapp, E. "Deadlock Detection in Distributed Databases". *Computing Surveys*. 1987; 19(4):303-328.
- Knowlton, K. C. "A Fast Storage Allocator". *Communications of the ACM*. 1965; 8(10):623-624.
- Knuth, D. E. "Additional Comments on a Problem in Concurrent Programming Control". *Communications of the ACM*. 1966; 9(5):321-322.
- Knuth, D.E.The Art of Computer Programming, Volume 1: Fundamental Algorithms, Segunda Edição. Addison-Wesley, 1973.
- Knuth, D.E.The Art of Computer Programming, Volume 3: Sorting and Searching, Segunda Edição. Addison-Wesley, 1998.
- Koch, P. D.L. "Disk File Allocation Based on the Buddy System". *ACM Transactions on Computer Systems*. 1987; 5(4):352-370.
- Kongetira, P., Aingaran, K., Olukotun, K. "Niagara: A 32-Way Multithreaded SPARC Processor". *IEEE Micro Magazine*. 2005; 25(2):21-29.
- Kopetz, H., Reisinger, J. "The Non-Blocking Write Protocol NBW: A Solution to a Real-Time Synchronization Problem". *IEEE Real-Time Systems Symposium*. 1993; 131-137.
- Kosaraju, S. "Limitations of Dijkstra's Semaphores Primitives and Petri Nets". *Operating Systems Review*. 1973; 7(4):122-126.
- Kozierok, C. The TCP/IP Guide. No Starch Press, 2005.
- Kramer, S. M. "Retaining SUID Programs in a Secure UNIX". *Proceedings of the Summer USENIX Conference*. 1988; 107-118.
- Kubiatowicz, J., Bindel, D., Chen, Y., Czerwinski, S., Eaton, P., Geels, D., Gummadi, R., Rhea, S., Weatherspoon, H., Weimer, W., Wells, C., Zhao, B. "OceanStore: An Architecture for Global-Scale Persistent Storage". *Proc. of Architectural Support for Programming Languages and Operating Systems*. 2000.
- Kurose, J., Ross, K. Computer Networking - A Top-Down Approach Featuring the Internet, Segunda Edição. Addison-Wesley, 2005.
- Lamport, L., "A New Solution of Dijkstra's Concurrent Programming Problem". *Communications of the ACM*, 1974;17(8):453-455
- Lamport, L. "Synchronization of Independent Processes". *Acta Informatica*. 1976; 7(1):15-34.
- Lamport, L. "Concurrent Reading and Writing". *Communications of the ACM*. 1977; 20(11):806-811.
- Lamport, L. "The Implementation of Reliable Distributed Multiprocess Systems". *Computer Networks*. 1978; 2(2):95-114.
- Lamport, L. "Time, Clocks and the Ordering of Events in a Distributed System". *Communications of the ACM*. 1978; 21(7):558-565.
- Lamport, L. "Password Authentication with Insecure Communications". *Communications of the ACM*. 1981; 24(11):770-772.
- Lamport, L. "The Mutual Exclusion Problem". *Communications of the ACM*. 1986; 33(2):313-348.
- Lamport, L. "A Fast Mutual Exclusion Algorithm". *ACM Transactions on Computer Systems*. 1987; 5(1):1-11.
- Lamport, L. "The Mutual Exclusion Problem Has Been Solved". *Communications of the ACM*. 1991; 34(1):110.
- Lamport, L., Shostak, R., Pease, M. "The Byzantine Generals Problem". *ACM Transactions on Programming Languages and Systems*. 1982; 4(3):382-401.
- Lampson, B. W. "Dynamic Protection Structures". *Proceedings of the AFIPS Fall Joint Computer Conference*. 1969; 27-38.
- Lampson, B. W. "Protection". *Proceedings of the Fifth Annual Princeton Conference on Information Systems Science*. 1971; 437-443.
- Lampson, B. W. "A Note on the Confinement Problem". *Communications of the ACM*. 1973; 10(16):613-615.
- Lampson, B. W., Redell, D. D. "Experience with Processes and Monitors in Mesa". *Proceedings of the 7th ACM Symposium on Operating Systems Principles (SOSP)*. 1979; 43-44.
- Lampson, B., Sturgis, H. "Crash Recovery in a Distributed Data Storage System". Technical Report. Xerox Research Center, 1976.
- Landwehr, C. E. "Formal Models of Computer Security". *Computing Surveys*. 1981; 13(3):247-278.
- Lann, G. L. "Distributed Systems - Toward a Formal Approach". *Proceedings of the IFIP Congress*. 1977; 155-160.
- Larson, P., Kajla, A. "File Organization: Implementation of a Method Guaranteeing Retrieval in One Access". *Communications of the ACM*. 1984; 27(7):670-677.
- Lauzac, S., Melhem, R., Mosse, D. "An Improved Rate-Monotonic Admission Control and Its Applications". *IEEE Transactions on Computers*. 52(3), 2003.
- Lee, J. "An End-User Perspective on File-Sharing Systems". *Communications of the ACM*. 2003; 46(2):49-53.
- Lee, E. K., Thekkath, C. A. "Petal: Distributed Virtual Disks". *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*. 1996; 84-92.
- Leffler, S.J., McKusick, M.K., Karels, M.J., Quarterman, J.S. The Design and Implementation of the 4.3BSD UNIX Operating System. Addison-Wesley, 1989.
- Lehmann, F. "Computer Break-Ins". *Communications of the ACM*. 1987; 30(7):584-585.
- Lehoczyk, L. Sha, Ding, Y. "The Rate Monotonic Scheduling Algorithm: Exact Characterization and Average Case Behaviour". *Proceedings of the 10th IEEE Real-Time Systems Symposium*. 1989.
- Lempel, A. "Cryptography in Transition". *Computing Surveys*. 1979; 11(4):286-303.
- Leslie, I. M., McAuley, D., Black, R., Roscoe, T., Barham, P. T., Evers, D., Fairbairns, R., Hyden, E., "The Design and Implementation of an Operating System to Support Distributed Multimedia Applications". *IEEE Journal of Selected Areas in Communications*,

- 1996;14(7):1280-1297
- Lett, A. L., Konigsford, W. L. "TSS/360: A Time-Shared Operating System". *Proceedings of the AFIPS Fall Joint Computer Conference*. 1968; 15-28.
- Levin, R., Cohen, E. S., Corwin, W. M., Pollack, F. J., Wulf, W. A. "Policy/Mechanism Separation in Hydra". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1975; 132-140.
- Levine, G. "Defining Deadlock". *Operating Systems Review*. 37(1), 2003.
- Levy, S. Hackers. Penguin Books, 1994.
- Lewis, B., Berg, D. Multithreaded Programming with Pthreads. Sun Microsystems Press, 1998.
- Lewis, B., Berg, D. Multithreaded Programming with Java Technology. Sun Microsystems Press, 2000.
- Lichtenberger, W. W., Pirtle, M. W. "A Facility for Experimentation in Man-Machine Interaction". *Proceedings of the AFIPS Fall Joint Computer Conference*. 1965; 589-598.
- Lindholm, T., Yellin, F. The Java Virtual Machine Specification. Segunda Edição, Addison-Wesley, 1999.
- Ling, Y., Mullen, T., Lin, X. "Analysis of Optimal Thread Pool Size". *Operating System Review*. 34(2), 2000.
- Lipner, S. "A Comment on the Confinement Problem". *Operating System Review*. 1975; 9(5):192-196.
- Lipton, R. "On Synchronization Primitive Systems". Tese de PhD. Carnegie-Mellon University, 1974.
- Liskov, B. H. "The Design of the Venus Operating System". *Communications of the ACM*. 1972; 15(3):144-149.
- Liu, C. L., Layland, J. W. "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment". *Communications of the ACM*. 1973; 20(1):46-61.
- Lobel, J. Foiling the System Breakers: Computer Security and Access Control. McGraw-Hill, 1986.
- Loo, A. W. "The Future of Peer-to-Peer Computing". *Communications of the ACM*. 2003; 46(9):56-61.
- Love, R. Linux Kernel Development, Segunda Edição. Developer's Library, 2005.
- Lowney, P. G., Freudenberger, S. M., Karzes, T. J., Lichtenstein, W. D., Nix, R. P., O'Donnell, J. S., Ruttenberg, J. C. "The Multiflow Trace Scheduling Compiler". *Journal of Supercomputing*. 1993; 7(1-2):51-142.
- Ludwig, M. The Giant Black Book of Computer Viruses. Segunda Edição, American Eagle Publications, 1998.
- Ludwig, M. The Little Black Book of Email Viruses. American Eagle Publications, 2002.
- Lumb, C., Schindler, J., Ganger, G. R., Nagle, D. F., Riedel, E. "Towards Higher Disk Head Utilization: Extracting Free Bandwidth From Busy Disk Drives". *Symposium on Operating Systems Design and Implementation*. 2000.
- Maekawa, M. "A Square Root Algorithm for Mutual Exclusion in Decentralized Systems". *ACM Transactions on Computer Systems*. 1985; 3(2):145-159.
- Maher, C., Goldick, J. S., Kerby, C., Zumach, B. "The Integration of Distributed File Systems and Mass Storage Systems". *Proceedings of the IEEE Symposium on Mass Storage Systems*. 1994; 27-31.
- Marsh, B. D., Scott, M. L., LeBlanc, T. J., Markatos, E. P. "First-Class User-Level Threads". *Proceedings of the 13th ACM Symposium on Operating Systems Principles*. 1991; 110-121.
- Mattern, F. "Virtual Time and Global States of Distributed Systems". *Workshop on Parallel and Distributed Algorithms*. 1988.
- Mattson, R. L., Gecsei, J., Slutz, D. R., Traiger, I. L., "Evaluation Techniques for Storage Hierarchies". *IBM Systems Journal*, 1970;9(2):78-117
- Mauro, J., McDougall, R. Solaris Internals: Core Kernel Architecture. Prentice Hall, 2007.
- McCanne, S., Jacobson, V. "The BSF Packet Filter: A New Architecture for User-level Packet Capture". *USENIX Winter*. 1993; 259-270.
- McDougall, R., Laudon, J. "Multi-Core Processors are Here". *USENIX; login: The USENIX Magazine*. 2006; 31(5):32-39.
- McDougall, R., Mauro, J. Solaris Internals, Segunda Edição. Prentice Hall, 2007.
- McGraw, J. R., Andrews, G. R. "Access Control in Parallel Programs". *IEEE Transactions on Software Engineering*. 1979; SE-5(1):1-9.
- McKeag, R. M., Wilson, R. Studies in Operating Systems. Academic Press, 1976.
- McKeon, B. "An Algorithm for Disk Caching with Limited Memory". *Byte*. 1985; 10(9):129-138.
- McKusick, M. K., Neville-Neil, G. V. The Design and Implementation of the FreeBSD UNIX Operating System. AddisonWesley, 2005.
- McKusick, M. K., Joy, W. N., Leffler, S. J., Fabry, R. S. "A Fast File System for UNIX". *ACM Transactions on Computer Systems*. 1984; 2(3):181-197.
- McKusick, M. K., Bostic, K., Karels, M. J. The Design and Implementation of the 4.4 BSD UNIX Operating System. John Wiley and Sons, 1996.
- McNairy, C., Bhatia, R. "Montecito: A Dual-Core, Dual-Threaded Itanium Processor". *IEEE Micro Magazine*. 2005; 25(2):10-20.
- McVoy, L. W., Kleiman, S. R. "Extent-like Performance from a UNIX File System". *Proceedings of the Winter USENIX Conference*. 1991; 33-44.
- Mealy, G. H., Witt, B. I., Clark, W. A. "The Functional Structure of OS/360". *IBM Systems Journal*. 5(1), 1966.
- Mellor-Crummey, J. M., Scott, M. L. "Algorithms for Scalable Synchronization on Shared-Memory Multiprocessors". *ACM Transactions on Computer Systems*. 1991; 9(1):21-65.
- Menasce, D., Muntz, R. R. "Locking and Deadlock Detection in Distributed Data Bases". *IEEE Transactions on Software Engineering*. 1979; SE-5(3):195-202.
- Mercer, C. W., Savage, S., Tokuda, H. "Processor Capacity Reserves: Operating System Support for Multimedia Applications". *International Conference on Multimedia Computing and Systems*. 1994; 90-99.
- Meyer, R. A., Seawright, L. H. "A Virtual Machine Time-Sharing System". *IBM Systems Journal*. 1970; 9(3):199-218.
- Microsoft MS-DOS User's Reference and Microsoft MS-DOS Programmer's Reference. Microsoft Press, 1986.
- Microsoft Windows NT Workstation Resource Kit. Microsoft Press, 1996.
- Microsoft Developer Network Development Library. Microsoft Press, 2000.
- Microsoft Windows 2000 Server Resource Kit. Microsoft Press, 2000.
- Milenkovic, M. Operating Systems: Concepts and Design. McGraw-Hill, 1987.
- Miller, E. L., Katz, R. H. "An Analysis of File Migration in a UNIX Supercomputing Environment". *Proceedings of the Winter USENIX Conference*. 1993; 421-434.
- Milojicic, D. S., Douglass, F., Paindaveine, Y., Wheeler, R., Zhou, S. "Process Migration". *ACM Computing Survey*. 2000; 32(3):241-299.
- Mockapetris, P. "Domain Names - Concepts and Facilities". *Network Working Group, Request for Comments: 1034* 1987;
- Mohan, C., Lindsay, B. "Efficient Commit Protocols for the Tree of Processes Model of Distributed Transactions". *Proceedings of the ACM Symposium on Principles of Database Systems*. 1983.
- Mok, A. K. "Fundamental Design Problems of Distributed Systems for the Hard Real-Time Environment". tese de Ph.D. Massachusetts Institute of Technology, MA, 1983.
- Morris, J. H. "Protection in Programming Languages". *Communications of the ACM*. 1973; 16(1):15-21.
- Morris, R., Thompson, K. "Password Security: A Case History". *Communications of the ACM*. 1979; 22(11):594-597.
- Morris, J. H., Satyanarayanan, M., Conner, M. H., Howard, J. H., Rosenthal, D. S. H., Smith, F. D. "Andrew: A Distributed Personal Computing Environment". *Communications of the ACM*. 1986; 29(3):184-201.
- Morshedian, D. "How to Fight Password Pirates". *Computer*. 19(1), 1986.

- PowerPC 601 RISC Microprocessor User's Manual. Motorola Inc., 1993.
- Mullender, S. Distributed Systems, Terceira Edição. Addison-Wesley, 1993.
- Myers, B., Beigl, M. "Handheld Computing". *Computer*. 2003; 36:27-29. [Número 9].
- Navarro, J., Lyer, S., Druschel, P., Cox, A. "Practical, Transparent Operating System Support for Superpages". *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*. 2002.
- Needham, R. M., Walker, R. D.H. "The Cambridge CAP Computer and Its Protection System". *Proceedings of the Sixth Symposium on Operating System Principles*. 1977; 1-10.
- Nelson, M., Welch, B., Ousterhout, J. K. "Caching in the Sprite Network File System". *ACM Transactions on Computer Systems*. 1988; 6(1):134-154.
- Nian-Min, Y., Ming-Yang, Z., Jiu-Bin, J. "Pipeline: A New Architecture of High Performance Servers". *Operating Systems Review*. 36(4), 2002.
- Norton, P., Wilton, R. The New Peter Norton Programmer's Guide to the IBM PC & PS/2. Microsoft Press, 1988.
- Nutt, G. Operating Systems: A Modern Perspective, Segunda Edição. Addison-Wesley, 2004.
- Oaks, S., Wong, H. Java Threads, Terceira Edição. O'Reilly & Associates, 2004.
- Obermarck, R. "Distributed Deadlock Detection Algorithm". *ACM Transactions on Database Systems*. 1982; 7(2):187-208.
- O'Leary, B. T., Kitts, D. L. "Optical Device for a Mass Storage System". *Computer*. 18(7), 1985.
- Olsen, R. P., Kenley, G. "Virtual Optical Disks Solve the On-Line Storage Crunch". *Computer Design*. 1989; 28(1):93-96.
- Organick, E.I. The Multics System: An Examination of Its Structure. MIT Press, 1972.
- Ortix, S. "Embedded OSs Gain the Inside Track". *Computer*. 34(11), 2001.
- Ousterhout, J. "The Role of Distributed State". In: Rashid R.F., ed. *CMU Computer Science: a 25th Anniversary Commemorative*. Addison-Wesley; 1991:199-217. [(1991)].
- Ousterhout, J. K., Costa, H. D., Harrison, D., Kunze, J. A., Kupfer, M., Thompson, J. G. "A Trace-Driven Analysis of the UNIX 4.2 BSD File System". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1985; 15-24.
- Ousterhout, J. K., Cherenson, A. R., Douglis, F., Nelson, M. N., Welch, B. B. "The Sprite Network-Operating System". *Computer*. 1988; 21(2):23-36.
- Parameswaran, M., Susarla, A., Whinston, A. B. "P2P Networking: An Information-Sharing Alternative". *Computer*. 34(7), 2001.
- Parmelee, R. P., Peterson, T. I., Tillman, C. C., Hatfield, D., "Virtual Storage and Virtual Machine Concepts". *IBM Systems Journal*, 1972;11(2):99-130
- Parnas, D. L. "On a Solution to the Cigarette Smokers' Problem Without Conditional Statements". *Communications of the ACM*. 1975; 18(3):181-183.
- Patil, S. "Limitations and Capabilities of Dijkstra's Semaphore Primitives for Coordination Among Processes". Technical Report. Massachusetts Institute of Technology, 1971.
- Patterson, D. A., Gibson, G., Katz, R. H. "A Case for Redundant Arrays of Inexpensive Disks (RAID)". *Proceedings of the ACM SIGMOD International Conference on the Management of Data*. 1988.
- Pease, M., Shostak, R., Lamport, L. "Reaching Agreement in the Presence of Faults". *Communications of the ACM*. 1980; 27(2):228-234.
- Pechura, M. A., Schoeffler, J. D. "Estimating File Access Time of Floppy Disks". *Communications of the ACM*. 1983; 26(10):754-763.
- Perlman, R. Network Layer Protocols with Byzantine Robustness. Tese de Ph.D., MIT, 1988.
- Peterson, G. L. "Myths About the Mutual Exclusion Problem". *Information Processing Letters*. 12(3), 1981.
- Peterson, J. L., Norman, T. A. "Buddy Systems". *Communications of the ACM*. 1977; 20(6):421-431.
- Pfleeger, C., Pfleeger, S. Security in Computing. Terceira Edição, Prentice Hall, 2003.
- Philbin, J., Edler, J., Anshus, O. J., Doublas, C. C., Li, K. "Thread Scheduling for Cache Locality". *Architectural Support for Programming Languages and Operating Systems*. 1996; 60-71.
- Pinilla, R., Gill, M. "JVM: Platform Independent vs. Performance Dependent". *Operating System Review*. 37(2), 2003.
- Popek, G. J. "Protection Structures". *Computer*. 1974; 7(6):22-33.
- Popek G., Walker B., eds. The LOCUS Distributed System Architecture. MIT Press, 1985.
- Prieve, B. G., Fabry, R. S. "VMIN-An Optimal Variable Space Page-Replacement Algorithm". *Communications of the ACM*. 1976; 19(5):295-297.
- Psaltis, D., Mok, F. "Holographic Memories". *Scientific American*. 1995; 273(5):70-76.
- Purdin, T. D.M., Schlichting, R. D., Andrews, G. R. "A File Replication Facility for Berkeley UNIX". *Software - Practice and Experience*. 1987; 17:923-940.
- Purdom, P. W., Stigler, S. M. "Statistical Properties of the Buddy system". *J. ACM*. 1970; 17(4):683-697.
- Quinlan, S. "A Cached WORM File System". *Software - Practice and Experience*. 1991; 21(12):1289-1299.
- Rago, S. UNIX System V Network Programming. Addison-Wesley, 1993.
- Rashid, R. F. "From RIG to Accent to Mach: The Evolution of a Network Operating System". *Proceedings of the ACM/IEEE Computer Society, Fall Joint Computer Conference*. 1986.
- Rashid, R., Robertson, G. "Accent: A Communication-Oriented Network Operating System Kernel". *Proceedings of the ACM Symposium on Operating System Principles*. 1981.
- Raymond, E.S. The Cathedral & the Bazaar. O'Reilly & Associates, 1999.
- Raynal, M. Algorithms for Mutual Exclusion. MIT Press, 1986.
- Raynal, M. "A Simple Taxonomy for Distributed Mutual Exclusion Algorithms". *Operating Systems Review*. 1991; 25(1):47-50.
- Raynal, M., Singhal, M. "Logical Time: Capturing Causality in Distributed Systems". *Computer*. 1996; 29(2):49-56.
- Reddy, A. L.N., Wyllie, J. C., "I/O Issues in a Multimedia System". *Computer*, 1994;27(3):69-74
- Redell, D. D., Fabry, R. S. "Selective Revocation of Capabilities". *Proceedings of the IRIA International Workshop on Protection in Operating Systems*. 1974; 197-210.
- Redell, D. D., Dalal, Y. K., Horsley, T. R., Lauer, H. C., Lynch, W. C., McJones, P. R., Murray, H. G., Purcell, S. P. "Pilot: An Operating System for a Personal Computer". *Communications of the ACM*. 1980; 23(2):81-92.
- Reed, D. P. "Implementing Atomic Actions on Decentralized Data". *ACM Transactions on Computer Systems*. 1983; 1(1):3-23.
- Reed, D. P., Kanodia, R. K. "Synchronization with Eventcounts and Sequences". *Communications of the ACM*. 1979; 22(2):115-123.
- Regehr, J., Jones, M.B., Stankovic, J.A. "Operating System Support for Multimedia: The Programming Model Matters". Technical Report. Microsoft Research, 2000.
- Reid, B. "Reflections on Some Recent Widespread Computer Break-Ins". *Communications of the ACM*. 1987; 30(2):103-105.
- Ricart, G., Agrawala, A. K. "An Optimal Algorithm for Mutual Exclusion in Computer Networks". *Communications of the ACM*. 1981; 24(1):9-17.
- Richards, A. E. "A File System Approach for Integrating Removable Media Devices and Jukeboxes". *Optical Information Systems*. 1990; 10(5):270-274.
- Richter, J. Advanced Windows. Microsoft Press, 1997.
- Riedel, E., Gibson, G. A., Faloutsos, C. "Active Storage for Large-Scale Data Mining and Multimedia". *Proceedings of 24th*

- International Conference on Very Large Data Bases*. 1998; 62-73.
- Ripeanu, M., Imnitchi, A., Foster, I. "Mapping the Gnutella Network". *IEEE Internet Computing*. 6(1), 2002.
- Rivest, R. L., Shamir, A., Adleman, L. "On Digital Signatures and Public Key Cryptosystems". *Communications of the ACM*. 1978; 21(2):120-126.
- Roberson, J. "ULE: A Modern Scheduler for FreeBSD". *2003 USENIX BSDCon*. 2003.
- Rodeheffer, T. L., Schroeder, M. D. "Automatic Reconfiguration in Autonet". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1991; 97-183.
- Rojas, R., Hashagen, U. *The First Computers - History and Architectures*. MIT Press, 2000.
- Rosenblum, M., Ousterhout, J. K. "The Design and Implementation of a Log-Structured File System". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1991; 1-15.
- Rosenkrantz, D. J., Stearns, R. E., Lewis, P. M. "System Level Concurrency Control for Distributed Database Systems". *ACM Transactions on Database Systems*. 1978; 3(2):178-198.
- Ruemmler, C., Wilkes, J. "Disk Shuffling". Technical report, 1991.
- Ruemmler, C., Wilkes, J. "Unix Disk Access Patterns". *Proceedings of the Winter USENIX Conference*. 1993; 405-420.
- Ruemmler, C., Wilkes, J. "An Introduction to Disk Drive Modeling". *Computer*. 1994; 27(3):17-29.
- Rushby, J. M. "Design and Verification of Secure Systems". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1981; 12-21.
- Rushby, J., Randell, B. "A Distributed Secure System". *Computer*. 1983; 16(7):55-67.
- Russell, D., Gangemi, G.T. *Computer Security Basics*. O'Reilly & Associates, 1991.
- Russinovich, M.E., Solomon, D.A. *Microsoft Windows Internals*, Quarta Edição. Microsoft Press, 2005.
- Saltzer, J. H., Schroeder, M. D. "The Protection of Information in Computer Systems". *Proceedings of the IEEE* 1975; 1278-1308
- Sandberg, R. *The Sun Network File System: Design, Implementation and Experience*. Sun Microsystems, 1987.
- Sandberg, R., Goldberg, D., Kleiman, S., Walsh, D., Lyon, B. "Design and Implementation of the Sun Network Filesystem". *Proceedings of the Summer USENIX Conference*. 1985; 119-130.
- Sargent, M., Shoemaker, R. *The Personal Computer from the Inside Out*. Terceira Edição, Addison-Wesley, 1995.
- Sarisky, L. "Will Removable Hard Disks Replace the Floppy?". *Byte*. 1983; 110-117.
- Satyanarayanan, M. "Scalable, Secure and Highly Available Distributed File Access". *Computer*. 1990; 23(5):9-21.
- Savage, S., Wetherall, D., Karlin, A. R., Anderson, T. "Practical Network Support for IP Traceback". *Proceedings of ACM SigCOMM Conference on Data Communication*. 2000; 295-306.
- Schell, R. R. "A Security Kernel for a Multiprocessor Microcomputer". *Computer*. 1983; 47-53.
- Schindler, J., Gregory, G. "Automated Disk Drive Characterization". *Technical Report*. 1999.
- Schlichting, R. D., Schneider, F. B. "Understanding and Using Asynchronous Message Passing Primitives". *Proceedings of the Symposium on Principles of Distributed Computing*. 1982; 141-147.
- Schneider, F. B. "Synchronization in Distributed Programs". *ACM Transactions on Programming Languages and Systems*. 1982; 4(2):125-148.
- Schneier, B. *Applied Cryptography*. Segunda Edição, John Wiley and Sons, 1996.
- Schrage, L. E. "The Queue M/G/I with Feedback to Lower Priority Queues". *Management Science*. 1967; 13:466-474.
- Schwarz, R., Mattern, F. "Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail". *Distributed Computing*. 1994; 7(3):149-174.
- Seely, D. "Password Cracking: A Game of Wits". *Communications of the ACM*. 1989; 32(6):700-704.
- Seltzer, M., Chen, P., Ousterhout, J. "Disk Scheduling Revisited". *Proceedings of the Winter USENIX Conference*. 1990; 313-323.
- Seltzer, M. I., Bostic, K., McKusick, M. K., Staelin, C. "An Implementation of a Log-Structured File System for UNIX". *USENIX Winter*. 1993; 307-326.
- Seltzer, M. I., Smith, K. A., Balakrishnan, H., Chang, J., McMains, S., Padmanabhan, V. N. "File System Loggin Versus Clusterin: A Performance Comparison". *USENIX Winter*. 1995; 249-264.
- Shrivastava, S. K., Panzieri, F. "The Design of a Reliable Remote Procedure Call Mechanism". *IEEE Transactions on Computers*. 1982; C-31(7):692-697.
- Siddha, S., Pallipadi, V., Mallick, A. "Process Scheduling Challenges in the Era of Multi-Core Processors". *Intel Technology Journal*. 11, 2007.
- Silberschatz, A., Korth, H.F., Sudarshan, S. *Database System Concepts*. Quinta Edição, McGraw-Hill, 2006.
- Silverman, J. M. "Reflections on the Verification of the Security of an Operating System Kernel". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1983; 143-154.
- Silvers, C. "UBC: An Efficient Unified I/O and Memory Caching Subsystem for NetBSD". *USENIX Annual Technical Conference - FREEENIX Track*. 2000.
- Simmons, G. J. "Symmetric and Asymmetric Encryption". *Computing Surveys*. 1979; 11(4):304-330.
- Sincerbox G.T., ed. *Selected Papers on Holographic Storage*. Optical Engineering Press, 1994.
- Singh, A. *Mac OS X Internals: A Systems Approach*. Addison-Wesley, 2007
- Singhal, M. "Deadlock Detection in Distributed Systems". *Computer*. 1989; 22(11):37-48.
- Sirer, E. G., Grimm, R., Gregory, A. J., Bershad, B. N. "Design and Implementation of a Distributed Virtual Machine for Networked Computers". *Symposium on Operating Systems Principles*. 1999; 202-216.
- Smith, A. J. "Cache Memories". *ACM Computing Surveys*. 1982; 14(3):473-530.
- Smith, A. J. "Disk Cache-Miss Ratio Analysis and Design Considerations". *ACM Transactions on Computer Systems*. 1985; 3(3):161-203.
- Sobti, S., Garg, N., Zheng, F., Lai, J., Shao, Y., Zhang, C., Ziskind, E., Krishnamurthy, A., Wang, R. "Segank: A Distributed Mobile Storage System". *Proceedings of the Third USENIX Conference on File and Storage Technologies*. 2004.
- Solomon, D.A. *Inside Windows NT*, Segunda Edição. Microsoft Press, 1998.
- Solomon, D.A., Russinovich, M.E. *Inside Microsoft Windows 2000*. Terceira Edição, Microsoft Press, 2000.
- Spafford, E. H. "The Internet Worm: Crisis and Aftermath". *Communications of the ACM*. 1989; 32(6):678-687.
- Spector, A. Z., Schwarz, P. M. "Transactions: A Construct for Reliable Distributed Computing". *ACM SIGOPS Operating Systems Review*. 1983; 17(2):18-35.
- Stallings, W. *Local and Metropolitan Area Networks*. Prentice Hall, 2000.
- Stallings, W. *Operating Systems*. Quarta Edição, Prentice Hall, 2000.
- Stallings, W. *Cryptography and Network Security: Principles and Practice*. Terceira Edição, Prentice Hall, 2003.
- Stankovic, J. S. "Software Communication Mechanisms: Procedure Calls Versus Messages". *Computer*. 15(4), 1982.
- Stankovic, J. A. "Strategic Directions in Real-Time and Embedded Systems". *ACM Computing Surveys*. 1996; 28(4):751-763.
- Staunstrup, J. "Message Passing Communication Versus Procedure Call Communication". *Software - Practice and Experience*. 1982; 12(3):223-234.
- Steinmetz, R. "Analyzing the Multimedia Operating System". *IEEE MultiMedia*. 1995; 2(1):68-84.

- Stephenson, C. J. "Fast Fits: A New Method for Dynamic Storage Allocation". *Proceedings of the Ninth Symposium on Operating Systems Principles*. 1983; 30-32.
- Stevens, R. Advanced Programming in the UNIX Environment. Addison-Wesley, 1992.
- Stevens, R. TCP/IP Illustrated Volume 1: The Protocols. Addison-Wesley, 1994.
- Stevens, R. TCP/IP Illustrated, Volume 2: The Implementation. Addison-Wesley, 1995.
- Stevens, W.R. UNIX Network Programming - Volume I. Prentice Hall, 1997.
- Stevens, W.R. UNIX Network Programming; Volume II. Prentice Hall, 1998.
- Stevens, W.R. UNIX Network Programming Interprocess Communications; Volume 2. Prentice Hall, 1999.
- Stoica, I., Abdel-Wahab, H., Jeffay, K., Baruah, S., Gehrke, J., Plaxton, G. "A Proportional Share Resource Allocation Algorithm for Real-Time, Time-Shared Systems". *IEEE Real-Time Systems Symposium*. 1996.
- Stokes, J. Inside the Machine. No Starch Press, 2007.
- Su, Z. "A Distributed System for Internet Name Service". *Network Working Group, Request for Comments: 830*. 1982.
- Sugerman, J., Venkitachalam, G., Lim, B. "Virtualizing I/O Devices on VMware Workstation's Hosted Virtual Machine Monitor". *2001 USENIX Annual Technical Conference*. 2001.
- Network Programming Guide. Sun Microsystems, 1990
- Svobodova, L. "File Servers for Network-Based Distributed Systems". *ACM Computing Survey*. 1984; 16(4):353-398.
- Talluri, M., Hill, M. D., Khalidi, Y. A. "A New Page Table for 64-bit Address Spaces". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1995.
- Tamches, A., Miller, B. P. "Fine-Grained Dynamic Instrumentation of Commodity Operating System Kernels". *USENIX Symposium on Operating Systems Design and Implementation*. 1999.
- Tanenbaum, A.S. Structured Computer Organization. Terceira Edição, Prentice Hall, 1990.
- Tanenbaum, A.S. Modern Operating Systems. Prentice Hall, 2001.
- Tanenbaum, A.S. Computer Networks. Quarta Edição, Prentice Hall, 2003.
- Tanenbaum, A. S., Van Renesse, R. "Distributed Operating Systems". *ACM Computing Surveys*. 1985; 17(4):419-470.
- Tanenbaum, A., van Steen, M. Distributed Systems: Principles and Paradigms. Prentice Hall, 2002.
- Tanenbaum, A.S., Woodhull, A.S. Operating System Design and Implementation. Segunda Edição, Prentice Hall, 1997.
- Tate, S. Windows 2000 Essential Reference. New Riders, 2000.
- Tay, B. H., Ananda, A. L. "A Survey of Remote Procedure Calls". *Operating Systems Review*. 1990; 24(3):68-79.
- Teorey, T. J., Pinkerton, T. B. "A Comparative Analysis of Disk Scheduling Policies". *Communications of the ACM*. 1972; 15(3):177-184.
- Tevanian, A., Jr., Rashid, R. F., Golub, D. B., Black, D. L., Cooper, E., Young, M. W. "Mach Threads and the Unix Kernel: The Battle for Control". *Proceedings of the Summer USENIX Conference*. 1987.
- Tevanian, A., Jr., Rashid, R.F., Young, M.W., Golub, D.B., Thompson, M.R., Bolosky, W., Sanzi, R. "A UNIX Interface for Shared Memory and Memory Mapped Files Under Mach". Technical Report. Carnegie-Mellon University, 1987.
- Tevanian, A., Jr., Smith, B. "Mach: The Model for Future Unix". *Byte*. 1989.
- Thekkath, C. A., Mann, T., Lee, E. K. "Frangipani: A Scalable Distributed File System". *Symposium on Operating Systems Principles*. 1997; 224-237.
- Thompson, K. "Reflections on Trusting Trust". *Communications of ACM*. 1984; 27(8):761-763.
- Thorn, T. "Programming Languages for Mobile Code". *ACM Computing Surveys*. 1997; 29(3):213-239.
- Toigo, J. "Avoiding a Data Crunch". *Scientific American*. 2000; 282(5):58-74.
- Traiger, I. L., Gray, J. N., Galtieri, C. A., Lindsay, B. G. "Transactions and Consistency in Distributed Database Management Systems". *ACM Transactions on Database Systems*. 1982; 7(3):323-342.
- Tudor, P.N. "MPEG-2 Video Compression Tutorial". IEEE Colloquium on MPEG-2 - What It Is and What It Isn't. IEEE, 1995.
- Vahalia, U. Unix Internals: The New Frontiers. Prentice Hall, 1996.
- Vee, V., Hsu, W. "Locality-Preserving Load-Balancing Mechanisms for Synchronous Simulations on Shared-Memory Multiprocessors". *Proceedings of the Fourteenth Workshop on Parallel and Distributed Simulation*. 2000; 131-138.
- Venners, B. Inside the Java Virtual Machine. McGraw-Hill, 1998.
- Wah, B. W. "File Placement on Distributed Computer Systems". *Computer*. 1984; 17(1):23-32.
- Wahbe, R., Lucco, S., Anderson, T. E., Graham, S. L. "Efficient Software-Based Fault Isolation". *ACM SIGOPS Operating Systems Review*. 1993; 27(5):203-216.
- Wallach, D. S., Balfanz, D., Dean, D., Felten, E. W., "Extensible Security Architectures for Java". *Proceedings of the ACM Symposium on Operating Systems Principles* 1997;
- Welsh, M., Culler, D., Brewer, E. "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services". *Proceedings of the ACM Symposium on Operating Systems Principles*. 2001.
- Wilkes, J., Golding, R., Staelin, C., Sullivan, T. "The HP AutoRAID Hierarchical Storage System". *ACM Transactions on Computer Systems*. 1996; 14(1):108-136.
- Williams, R. Computer Systems Architecture - A Networking Approach. Addison-Wesley, 2001.
- Williams, N. "An Implementation of Scheduler Activations on the NetBSD Operating System". *2002 USENIX Annual Technical Conference, FREEENIX Track*. 2002.
- Wilson, P. R., Johnstone, M. S., Neely, M., Boles, D. "Dynamic Storage Allocation: A Survey and Critical Review". *Proceedings of the International Workshop on Memory Management*. 1995; 1-116.
- Wolf, W. "A Decade of Hardware/Software Codesign". *Computer*. 2003; 36(4):38-43.
- Wood, P., Kochan, S. LZMX System Security. Hayden, 1985.
- Woodside, C. "Controllability of Computer Performance Tradeoffs Obtained Using Controlled-Share Queue Schedulers". *IEEE Transactions on Software Engineering*. 1986; SE-12(10):1041-1048.
- Worthington, B. L., Ganger, G. R., Patt, Y. N. "Scheduling Algorithms for Modern Disk Drives". *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*. 1994; 241-251.
- Worthington, B. L., Ganger, G. R., Patt, Y. N., Wilkes, J. "On-line Extraction of SCSI Disk Drive Parameters". *Proceedings of the ACM Sigmetrics Conference on Measurement and Modeling of Computer Systems*. 1995; 146-156.
- Wulf, W. A. "Performance Monitors for Multiprogramming Systems". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1969; 175-181.
- Wulf, W.A., Levin, R., Harbison, S.P.Hydra/C.mmp: An Experimental Computer System. McGraw-Hill, 1981.
- Yeong, W., Howes, T., Kille, S. "Lightweight Directory Access Protocol". Network Working Group, 1995. [Request for Comments: 1777].
- Young, M., Tevanian, A., Rashid, R., Golub, D., Eppinger, J. "The Duality of Memory and Communication in the Implementation of a Multiprocessor Operating System". *Proceedings of the ACM Symposium on Operating Systems Principles*. 1987; 63-76.
- Yu, X., Gum, B., Chen, Y., Wang, R. Y., Ki, K., Krishnamurthy, A., Anderson, T. E. "Trading Capacity for Performance in a Disk Array". *Proceedings of the ACM Symposium on Operating Systems Design and Implementation*. 2000; 243-258.
- Zabatta, F., Young, K. "A Thread Performance Comparison: Windows NT and Solaris on a Symmetric Multiprocessor". *Proceedings of the 2nd USENIX Windows NT Symposium*. 1998.

Zapata, M., Asokan, N. "Securing Ad Hoc Routing Protocols". *Proc. 2002 ACM Workshop on Wireless Security*. 2002.

Índice

- .NET Framework, , 65
10BaseT Ethernet, , 536
100BaseT Ethernet, , 536
2PC, protocolo, *See commit de duas fases (2PC), protocolo*
- A**
- acesso:**
anônimo, 349
arquivo, *See acesso a arquivo*
controlado, 353
- acesso a arquivo**, , 332, , 353-354
- acesso a arquivo remoto (sistemas de arquivo distribuídos)**, , 563-566
caching, 566 *See also serviço remoto*,
consistência, 565-566
esquema básico, 563
local de cache, 563-564
política de atualização de cache, 564-565
- acesso aleatório**, , 616
acesso anônimo, 349
acesso controlado, 353
acesso direto (arquivos), 336-337
acesso não uniforme à memória (NUMA), 13, 300
- acesso sequencial (arquivos)**, , 336
- acesso uniforme à memória (UMA)**, , 13
- ACL (lista de controle de acesso)**, , 353
- Active Directory (Windows XP), , 702
- adaptador de host**, , 441
- AES (padrão de codificação avançado)**, , 506
- afinidade de processador**, , 159
- afinidade flexível**, , 159
- afinidade rígida**, , 159
- AFS, *See also Andrew, sistema de arquivos*
- agrupamento**, , 377
ajuste de desempenho, 66
- algoritmo bully (valentão)**, , 590-591
- algoritmo de alocação de quadro**, , 290
- algoritmo de anel**, , 591
- algoritmo de assinatura digital**, , 509
algoritmo de bits de referência adicionais, 294, 295
algoritmo de detecção de deadlock totalmente distribuído, 588-589
algoritmo de escalonamento da tarefa mais curta primeiro (SJF), 150-151
algoritmo de escalonamento de fila de feedback multinível, 155-156
algoritmo de escalonamento de fila multinível, 154-155
algoritmo de escalonamento monotônico de taxa, 606-608
algoritmo de escalonamento por prioridade, 151-152
- algoritmo de escalonamento round-robin (RR)**, , 152-154
- algoritmo de escalonamento SJF**, *See also algoritmo de escalonamento da tarefa mais curta primeiro*
algoritmo de política (Linux), 649
- algoritmo de relógio**, *See also algoritmo de substituição de página pela segunda chance*
- algoritmo de requisição de recursos**, , 234-235
- algoritmo de segurança**, , 234
- algoritmo de substituição de local (algoritmo de substituição de prioridade)**, , 301
- algoritmo de substituição de página**, , 290

algoritmo de substituição de página baseada em contagem, , 296-297
algoritmo de substituição de página ideal, , 292-293
algoritmo de substituição de página pela segunda chance (algoritmo de relógio), , 295-295
algoritmo de substituição de página por aproximação LRU, , 294-295
algoritmo de substituição de página usado mais frequentemente (MFU), , 296-297
algoritmo de substituição de prioridade, 301
algoritmo do banqueiro, , 234, , 242-243
algoritmo do elevador, *See also SCAN, algoritmo de escalonamento*
algoritmo do gráfico de alocação de recursos, , 233
algoritmo twofish, , 506
algoritmos de alcance, , 591-593
algoritmos de buffer de página, , 297
algoritmos de controle de admissão, , 606
algoritmos de controle de concorrência, , 210
algoritmos de eleição, , 589-591
algoritmos de pilha, , 294
alocação:
 espaço de disco, 368-375
 alocação contígua, 368-370
 alocação indexada, 372-374
 alocação vinculada, 370-372
 e desempenho, 374-375
 igual, 298
 problema, 337
 proporcional, 299
 sistema buddy, 308
 slab, 308-310
alocação de espaço contíguo em disco, , 368-370
alocação de memória, , 254-256
alocação de memória contígua, , 254
alocação de memória do kernel, , 308-310
alocação de quadro, , 297-300
 acesso à memória não uniforme, 300
 alocação igual, 298
 alocação proporcional, 299
 global, 299-300 *See also local*,
alocação de recursos (serviço do sistema operacional), , 38
alocação de sistema buddy, , 308
alocação de slab, , 308-310, , 647
alocação igual, , 298
alocação indexada de espaço em disco, , 372-374
alocação proporcional, , 299
alocação vinculada de espaço em disco, , 370-372
alocador de página (Linux), , 645-646
alocador de potência de 2, 308
alocador de recursos, sistema operacional como, , 4
alta disponibilidade, , 13-14
alto desempenho, , 669
ambiente Windows de 16 bits, , 689
ambiente Windows de 32 bits, , 690
ambientes computadorizados, , 27-29
 computação baseada na Web, 29
 computação cliente-servidor, 28-29
 computação peer-to-peer, 28-29
 tradicionais, 27
ameaças, , 491. *See also ameaças ao programa*
ameaças ao programa, , 494-500
 ataques de estouro de pilha ou buffer, 495-498
 bombas lógicas, 495
 cavalos de Troia, 494-495
 trap doors, 495
 vírus, 498-500
amplificação de direitos (Hydra), , 482

análise de falha, , 65
análise de rede de enfileiramento, , 172
Andrew, sistema de arquivos (AFS), , 569-573
 espaço de nomes compartilhado, 570-571
 implementação, 572-573
 operações de arquivo, 571-572
anel de chaves, , 509
anomalia de Belady, , 291, , 292
APCs, *See also chamadas de procedimento assíncronas*
API, *See also interface de programa de aplicação*
API Win32, , 667-668, , 690
aplicações thread-safe, , 195
Apple Computers, , 40
AppleTalk, protocolo, , 698
applets, , 61-62
Application Domain, , 65
área de logging, , 693
área de reinício, , 693
armazenamento conectado à rede, 402-403
armazenamento conectado ao host, 402
armazenamento de apoio, 252
armazenamento estável, 208, 420-421
armazenamento holográfico, 423
armazenamento não volátil, 9, 207-208
armazenamento quase linear, , 423
armazenamento secundário, , 9, , 361. *See also discos*
armazenamento terciário, , 421-429
 discos removíveis, 421-423
 fitas, 422-423
 questões de desempenho, 426-429
 suporte do sistema operacional, 423-424
 tecnologia futura, 423
armazenamento utilitário, , 418
armazenamento volátil, , 9, , 207
armazenamento. *See also estrutura de armazenamento em massa*
 estável, 208
 holográfico, 423
 não volátil, 9, 208
 secundário, 9, 361
 terciário, 21
 utilitário, 418
 volátil, 9, 207
ARP (protocolo de resolução de endereço), , 551
arquitetura de barramento, , 11
arquitetura de switch, , 10
arquitetura(s), , 10-15
 do Windows XP, 670
 sistemas de único processador, 10-11
 sistemas em clusters, 13-15
 sistemas multiprocessados, 11-13
arquivado em fita, , 423
arquivo de busca, , 330
arquivos, , 20, , 329-330. *See also diretórios*
 abrindo/fechando, 331
 acessando informações, 336-338
 acesso direto, 336-337
 acesso sequencial, 336
 atributos, 330
 batch, 334
 bloqueando, 332
 definição, 329
 estrutura de armazenamento, 339-340
 estrutura interna, 336

executáveis, 79
extensões, 334-335
operações sobre, 330-334
protegendo, 353-356
 via acesso a arquivo, 353-354
 via senhas/permissões, 356, 356
recuperação, 381-383

arquivos batch, , 334

arquivos codificados, , 617

arquivos compartilhados imutáveis, , 352

arquivos de dados, , 330

arquivos de log, , 65

arquivos de mapeamento de memória, , 706

arquivos de paginação (Windows XP), , 677

arquivos de programa, , 330

arquivos de texto, , 330

arquivos do sistema, , 342

arquivos executáveis, , 79, , 330

arquivos mapeados na memória, , 306-307, , 678

arquivos MPEG, 618
arquivos objeto, 330

arquivos zipados, , 618

arquivos-fonte, , 330

array ativo (Linux), , 642

array de armazenamento, , 413

array expirado (Linux), , 642

arrays, , 279

arrays redundantes de discos pouco dispendiosos, *See also RAID*

árvore B+ (NTFS), , 692

árvores de domínio, , 701

ASIDs, *See also identificadores de espaço de endereço*

assinaturas, , 518

assinaturas digitais, , 509

assistentes digitais pessoais (PDAs), 9, 26, 27

assunto do servidor (Windows XP), , 524

assunto simples (Windows XP), , 524

ataque do homem no meio, , 492

ataques, , 491. *See also ataques de negação de serviço*
 dia zero, 518
 homem no meio, 492
 reprodução, 492

ataques de DOS, *See also ataques de negação de serviço*

ataques de estouro de buffer, , 495-498

ataques de estouro de pilha, , 495-498

ataques de negação de serviço (DOS), , 492, , 504

ataques de negação de serviço distribuídos (DDOS), , 504

ataques de replay, , 492

ataques do dia zero, , 518

ativação do escalonador, , 133-134

Atlas, sistema operacional, , 717-718

atomicidade, , 579-581

atraso, , 619

atributos, , 692

atributos não residentes, 692

atributos residentes, , 692

autenticação:
 codificação, 508-509
 falha de, 492
 fator dois, 515
 Linux, 662
 Windows, 691

autenticação de fator dois, , 515

autenticação do usuário, , 512-516

com biometria, **515-516**
com senhas, **512-516**
autenticação única e segura, , **351**
automapa de tabela de página, , **677**
autoridades de certificação, , **510**
autorização, , **524**
avaliação analítica, , **171**
avaliação de risco, , **516**

B

back door, , **449**
backup completo, , **383**
backup incremental, , **383**
backups, , **383**
balanceadores de carga, , **29**
balanceamento de carga, , **159-160**
banco de dados de localização de volume (NFS V4), , **570**
banco de dados de quadro de página, , **681**
barramento:
 definição, **440**
 expansão, **440**
 PCI, **440**
barramento (E/S), , **400**
barramento de expansão, , **440**
barramento PCI, **440**
barramentos ATA, , **400**
barramentos seriais universais (USBs), , **400**
barrier (sincronização de thread), , **141**, , **219**
base de computadores de confiança (TCB), , **523**
base do segmento, , **270**
bibliotecas:
 compartilhadas, **251-252**, , **281**
 sistema Linux, **635**, **636**
bibliotecas compartilhadas, , **251-252**, , **281**
bibliotecas de link dinâmico (DLLs), , **251-252**, , **670**
bibliotecas de threads, , **121-122**
 Pthreads, **122-123**
 sobre, **121-122**
 threads Java, **124-127**
 threads Win32, **123**
bibliotecas do sistema (Linux), , **635**, , **636**
biometria, , **515-516**
bit de modo, , **17**
bit de válido/inválido, , **262**
bit(s), , **5**
 modificação (sujo), **290**
 modo, **17**
 referência, **294**
 válido-inválido, **262**
bits de modificação (bits sujos), , **290**
bits de referência, , **294**
bits sujos (modificação), , **290**
bloco de boot, , **69**, , **363**, , **408-409**
bloco de controle de arquivo (FCB), , **362**
bloco de controle de boot, , **363**
bloco de controle de endereço virtual (VACB), , **685-686**
bloco de controle de volume, , **363**
bloco de índice, , **372**
bloco de índice de esquema combinado, , **374**
bloco de índice de esquema interligado, , **374**
bloco de mensagem do servidor (SMB), , **697**
bloco(s), , **44**, , **255**, , **336**
boot, **69**, , **408-409**

controle de arquivo, 362
controle de boot, 363
controle de volume, 363
defeituosos, 409-410
definição, 657
diretos, 374
índice para, 337-338
índice, 372
indiretos, 374
lógicos, 401
blocos de controle de processo (PCBs, blocos de controle de tarefa), , 80-81
blocos de controle de tarefa, See also **blocos de controle de processo**
blocos de dados, , 394
blocos defeituosos, , 409-410
blocos diretos, 374, 394
blocos indiretos, 374
blocos indiretos duplos, 374
blocos indiretos triplos, 374
blocos indiretos únicos, 374
blocos lógicos, 401
bloqueio (dados compartilhados), 47
bloqueio compartilhado, 332
bloqueio de E/S, 451-452
bloqueio indefinido (starvation), 152, 187
boot do sistema, 69-71
booting, 69-71, 688
borda de atribuição, 225
borda de requisição, 225, 233
braço de disco, 399
broadcasting, 551, 623
BSD UNIX, 31
buddy heap (Linux), 646
buffer, , 657
 circular, 382
 definição, 453
buffer duplo, , 453, , 625-626
buffer ilimitado, , 92
buffering, , 96, , 453-454, , 625-626
burst de CPU, , 145-146
bus-mastering, placas de E/S, , 445
byte, , 5

C

C, biblioteca, , 47
C-LOOK, algoritmo de escalonamento, , 406
C-SCAN, algoritmo de escalonamento, , 406

cabeçalhos

 de requisição, 554
 de resposta, 556
 gerais, 554, 556
 entidade, 556

cache:

 acesso a arquivo remoto:
 e consistência, 565-566
 local do cache, 563-564
 política de atualização, 564-565
 buffer unificado, 379
 como buffer de memória, 248
 definição, 454
 melhoria de desempenho, 378-380
 no Linux, 647
 no Windows XP, 685-686
 página, 379

RAM não volátil, 413
slabs, 308-310

cache de página, , 379, , 647

cachefs, sistema de arquivos, , 564

caching, , 21-23, , 454

- duplo, 379
- no cliente, 701
- serviço remoto, 566
- write-back, 564

caching duplo, , 379

caching write-back, , 564

cadeia de bloco cifrado, , 506

cadeia de margaridas, , 440

cadeias de firewall, , 661

caixa de brita (sistema de arquivos Tripwire), , 519-520

caixas de correio (portas), , 95

callbacks, , 571

camada de abstração de hardware (HAL), , 670

camada de aplicação, , 545

- camada de apresentação, 545
- camada de calço, 690
- camada de conversão, 669
- camada de enlace de dados, 545
- camada de rede, 545
- camada de sessão, 545
- camada de sistemas, 618
- camada de thunking, 669
- camada de transporte, 545
- camada física, 544
- camadas (de protocolos de rede), 511

Cambridge CAP, sistema, 483

caminho de busca, 342

canais cobertos, 495

canal de E/S, 462-463

cancelamento de thread, 129

cancelamento de thread adiado, 129

cancelamento de threads assíncrono, 129

capacidade(s), 478, 483

- capacidade de dados, 483
- capacidade de software, 483
- capacidade limitada (da fila), 96
- capacidade não limitada (da fila), 96
- capacidade zero (da fila), 96

carga e execução do programa, 50

carregador, 715

carregador de classe, 62

carregamento:

- dinâmico, 251
- no Linux, 650-651

Carrier Sense with Multiple Access (CSMA), , 544

cartões de controle, , 45, , 714-715

CAV (Constant Angular Velocity), , 401

cavalos de Troia, , 494-495

CD, See also detecção de colisão

certificação, , 523

certificados digitais, , 509-510

chamadas de monitor, See also chamadas do sistema

chamadas de procedimento assíncronas (APCs), , 130-131, , 672-673

chamadas de procedimento locais (LPCs), , 669, , 683-684

chamadas de procedimento remoto (RPCs), , 699-700

chamadas de sistema (chamadas de monitor), , 7, , 41-43

- API, 42-43
- exemplos, 46

funcionamento, 41-43
para comunicação, 49-50
para controle de processo, 45-48
para gerenciamento de arquivos, 48
para gerenciamento de dispositivo, 48-49
para manutenção de informações, 49
para proteção, 50

chamadas de sistema de controle de processo, , 45-48

chamadas de sistema para gerenciamento de arquivos, , 48

chamadas de sistema para gerenciamento de dispositivos, , 48-49

chamadas de sistema para manutenção de informações, , 49

chamadas do sistema de comunicação, , 49-50

chave de sensibilidade, , 455

chave mestra, , 481

chaves, , 479, , 481, , 505
privadas, 507-508
públicas, 507-508
chaves privadas, 507-508

chaves públicas, , 507-508

ciclo de burst de CPU-E/S, , 145-146

ciclo de execução de instrução, , 247

ciclo ler-modificar-escrever, , 416

ciclos:
burst CPU-E/S, 145-146
no CineBlitz, 625

cifras de bloco, , 506

cifras de stream, , 506-507

CIFS (Common Internet File System), , 351

CineBlitz, , 625

classe de tempo real, , 164

classe variável, , 164

classes (Java), , 486

classes imutáveis, , 124

Client-Side Caching (CSC), , 701

cliente(s):
definição, 560
sem disco, 561
SSL, 511

clientes de tempo real, , 625
clientes não de tempo real, 625
clientes sem disco, 561

clock de CPU, 247
clock lógico, 576
clones, 389

close(), operação, , 331

clustering:
assimétrico, 14
no Windows XP, 314

clusters, , 408, , 691-692

clusters (blocos), , 371-372

clusters de alta disponibilidade, , 550

clusters de computação, , 550

CLV (Constant Linear Velocity), , 401

codificação, , 505-510
assimétrica, 507-508
autenticação, 508-509
distribuição de chaves, 509-510
simétrica, 506-507
Windows XP, 696-697

codificação assimétrica, , 507-508

codificação simétrica, , 506-507

código:
absoluto, 250

reentrante, 263
código C/C++ nativo, 44
código de autenticação de mensagem (MAC), 509
código de byte, 62
código de correção de erro (ECC), 408, 415
código de sensibilidade adicional, 455
código de status, 556
código independente de posição (PIC), 652
código puro, , 263
coerência de cache, , 23
coleta de lixo, , 62-63, , 347
colisões (de nomes de arquivo), , 368
colocador de vírus, , 499
COM, *See also Component Object Model*
commit de duas fases (2PC), protocolo, , 579-581
Common Internet File System (CIFS), , 351
compactação, , 256, , 369
 em sistemas multimídia, 617-657
 no Windows XP, 696-697
compactação com perdas, , 617-618
compactação off-line do espaço, 370
compactação on-line do espaço, 370
compactação sem perdas, , 617
compartilhamento:
 carga, 157, , 532
 paginação, 262-264
 recursos, 531
 tempo, 16-17
compartilhamento de arquivos, , 348-352
 com múltiplos usuários, 348-349
 com redes, 349-351
 modelo cliente-servidor, 349-350
 modos de falha, 351
 sistemas de informação distribuídos, 350-351
 semântica de consistência, 351-352
compartilhamento de carga, , 157, , 532
compartilhamento de informações, , 91
compartilhamento de recursos, , 531
compartilhamento do processador, , 153
compartilhamentos (CPU), , 163
compiladores just-in-time, , 63
complexidade administrativa, 562
Component Object Model (COM), 700
computação baseada na Web, 29
computação de alto desempenho, 14
computação peer-to-peer, 28-29
computação segura, 519
computação tradicional, 27
computadores, mini, 4
computadores em rede, 28
comunicação direta, , 94
comunicação entre processos (IPC), , 91-96
 sistemas cliente-servidor, 99-107
 chamadas de procedimento remoto, 101-103
 invocação de método remoto, 104-106
 sockets, 100-102
 no Linux, 632, 659-660
 Mach, exemplo do, 97-98
 sistemas de memória compartilhada, 92-94
 sistemas de passagem de mensagens, 94
 Windows XP, exemplo do, 98-100
comunicação indireta, , 95
comunicações:

diretas, **94**
em sistemas operacionais distribuídos, **533**
entre processos, *See comunicação entre processos*
indiretas, **95**
não confiáveis, **592**
programas do sistema para, **50-51**
comunicações (serviço do sistema operacional), , **38**
comunicações não confiáveis, , **592**
comutação:
circuito, **543**
domínio, **472**
mensagem, **543**
pacotes, **543**
comutação de circuitos, , **543**
concentradores de terminais, , **462**
condição de corrida (race), , **180**, , **196**
condição de espera circular (deadlocks), , **230-231**
condição de não preempção (deadlocks), , **230**
condição manter e esperar (deadlocks), , **229-230**
condições de erro, , **279**
conexão, , **250**
conexão de tecnologia avançada (ATA), barramentos, , **400**
confiabilidade, , **543**
sistemas multimídia, **619**
sistemas operacionais distribuídos, **531-532**
Windows XP, **668**
confidencialidade, quebra de, , **491**
conjunto de entidades, , **196**
conjunto de espera, , **197**
conjunto de trabalho máximo (Windows XP), , **314**
conjunto de trabalho mínimo (Windows XP), , **314**
conjunto de volumes, , **693-694**
conjunto duplex, , **696**
conjunto espelho, , **696**
conjuntos de caixa de correio, , **98**
conjuntos de trabalho, , **302**, , **304**
consistência, , **565-566**
consolidação, , **58**
Constant Angular Velocity (CAV), , **401**
Constant Linear Velocity (CLV), , **401**
contabilidade (serviço do sistema operacional), , **39**
contadores de programa, , **19**, , **79**, , **81**
contagem, , **377**
contas do usuário, , **524**
contêineres (zonas), , **61**
contenção, , **239**
contexto (de processo), , **85**
contexto de segurança (Windows XP), , **524**
controlador de acesso direto à memória (DMA), , **445**
controlador de disco, , **400**
controlador de DMA, *See also controlador de acesso direto à memória*
controlador de host, , **400**
controladores, , **400**, , **439-441**
acesso direto à memória, **445**
definição, **439-441**
disco, **400**
host, **400**
controladores de dispositivos, , **6**, , **457**. *See also sistemas de E/S*
controle de acesso baseado em papel (RBAC), , **480**
controle de acesso no Linux, , **662-663**
controle de admissão, , **619**, , **625-626**
controle de concorrência, , **581-584**
com estampa de tempo, **583-584**

com protocolos de bloqueio, 581-583
e sincronismo, 201-204

controle de fluxo, , 460

convenção universal de nomes (UNC), , 699

conveniência, , 3

conveniência do usuário, , 92

coordenação distribuída:

- algoritmos de alcance, **591-593**
- algoritmos de eleição, **589-591**
- atomicidade, **579-581**
- controle de concorrência, **581-584**
- deadlocks, **584-589**
 - detecção, **585-589**
 - prevenção/impedimento, **584-585**
- exclusão mútua, **577-578**
- ordenação de eventos, **575-577**

coordenador de detecção de impasse, , 586

coordenador de tradução, , 579

cópia na escrita, técnica, , 287-288

cópias de sombra de volume, , 697

core dump, , 65

corte automático do conjunto de trabalho (Windows XP), , 314

CP/M, , 722

CPU (Central Processing Unit), , 3, , 247-248

crackers, , 492

crash dump, , 65

criação:

- de arquivos, **330**
- processo, **86-89**

criptografia, , 504-512

- codificação, **505-510**
- exemplo de SSL, **511-512**
- implementação, **510-511**

CSC (Client-Side Caching), , 701

CSMA, See also Carrier Sense with Multiple Access

CTSS, sistema operacional, , 719-720

D

d (deslocamento de página), , 257

dados:

- específicos da thread, **132-134**
- multimídia, **26**
- recuperação, **381-383**

dados de mídia contínua, , 616

dados de multimídia, , 26, , 615-616

dados específicos da thread, , 132-134

Darwin, , 31

Data-Encryption Standard (DES), , 506

datagramas, , 543

DCOM, , 700

DDOS, ataques, , 504

deadlocks, , 186-187, , 584-589

- condições necessárias, **224**
- definição, **223**
- detecção, **235-237, 585-589**
 - única instância de cada tipo de recurso, **235-236**
 - uso do algoritmo, **237**
 - várias instâncias de um tipo de recurso, **236-237**
- em Java, **196, 226-229**
- gráficos de alocação de recurso do sistema para descrever, **225**
- impedimento, **228, 232-235**
 - com algoritmo de estado seguro, **232-233**
 - com algoritmo de gráfico de alocação de recursos, **233**

com algoritmo do banqueiro, 234
modelo do sistema, 223-224
prevenção, 227-228
 condição de espera circular, 230-231
 condição de exclusão mútua, 229
 condição manter e esperar, 229-230
 condição sem preempção, 230
prevenção/impedimento, 584-585
recuperação, 237-238
 por preempção de recurso, 238
 por término do processo, 238

Deferred Procedure Calls (DPCs), , 672-673

degradação controlada, , 12

delegação (NFS V4), , 569

densidade de área, , 385

depuração, , 64-68
 ajuste de desempenho, 66
 análise de falhas, 65
 com DTrace, 66-68
 definição, 64
 objetivo, 64, 65

depuradores, , 45, , 65

DES (Data-Encryption Standard), , 506

desafio (senhas), , 514

descritor de arquivo (fd), , 364, , 396

descritor de segurança (Windows XP), , 525

descritores (definição), , 677

descritores (kernel), , 674

descritores de endereço virtual (VADs), , 682

descritores de instância, , 704

desempenho:
 alocação de espaço em disco, 374-375
 com armazenamento terciário, 426-429
 confiabilidade, 427
 custo, 427-429
 velocidade, 426
 sistema de E/S, 460-463
 Windows XP, 669

desinfecção de programa, , 519-521

desktop, , 40

deslizamento de setor, , 410

deslocamento de página (d), , 257

despachante, , 148

despachante de exceção, , 673

detecção baseada em assinatura, , 517-518

detecção de anomalia, , 518

detecção de colisão (CD), , 544

detecção de erro, , 38

detecção de intrusão, , 517-519

detecção e eliminação de órfãos, , 568

DFS, *See also Distributed File System*

DFS sem estado, , 351

diagrama de enfileiramento, , 83

diário, , 655-656

diário de mudança (Windows XP), , 697

Digital Rights Management (DRM), , 30

Direct Memory Access (DMA), , 10, , 445-446

Direct Virtual Memory Access (DVMA), , 446

direitos:
 auxiliares (Hydra), 481-4
 de acesso, 471, 480-481
 de grupo (Linux), 662
 de owner (Linux), 662

do usuário (Linux), 662
globais (Linux), 662

diretório atual, , 343

diretório de arquivo do usuário (UFD), , 341

diretório de dispositivo, , 339. *See also diretórios*

diretório de página, , 678

diretórios, , 339

- dois níveis, 341-342
- estruturados em árvore, 342-344
- gráfico acíclico, 344-346
- gráfico geral, 346-347
- implementação, 368
- recuperação, 381-383
- único nível, 340-341

disciplina de linha, , 658-659

disco bruto, , 297, , 365

disco de boot (disco do sistema), , 69, , 378

disco do sistema, *See also disco de boot*

disco eletrônico, , 9

discos, , 399-400. *See also estrutura de armazenamento em massa*

- algoritmos de escalonamento, 403-412
 - C-SCAN, 406
 - FCFS, 404
 - LOOK, 406
 - SCAN, 405
 - selecionando, 406-407
 - SSTF, 404-405
- alocação de espaço, 368-375
 - alocação contígua, 368-370
 - alocação indexada, 372-374
 - alocação vinculada, 370-372
 - e desempenho, 374-375
- bloco de boot, 408-409
- blocos defeituosos, 409-410
- boot, 69, 409
- brutos, 297
- conectados à rede, 402-403
- conectados ao host, 402
- eletrônicos, 9
- estado sólido, 22
- estrutura, 401
- flexíveis, 399-400
- formatação, 408
- formatados em baixo nível, 401
- gerenciamento do espaço livre, 375-377
- hot spare, 418
- independentes, 413
- leitura e escrita, 422
- magnéticos, 8, 399-400
- melhoria de desempenho, 378-381
- mudança de fase, 422
- óptico-magnéticos, 422
- ópticos, 422
- rede de armazenamento, 403
- removíveis, 421-423
- sistema, 409
- somente de leitura, 422
- uso eficiente, 377-378
- WORM, 422

dispositivo do sistema, 688

dispositivos:

- assíncronos, 448
- compartilháveis, 448, 449

de acesso aleatório, 448, 449, 716
de acesso sequencial, 716
de armazenamento terciário, 21
de bloco, 449, 657-658
de caractere (Linux), 657, 658-659
de leitura-escrita, 448, 449
de stream de caracteres, 449
dedicados, 448, 449
em rede, 450, 657
sequenciais, 448, 449
síncronos, 448, 449
somente de escrita, 448, 449
somente de leitura, 448, 449
disputa, 543-544
disquetes, 399-400
distribuição de chaves, 509-510
distribuições, , 31
Distributed File System (DFS), , 349
 sem estado, 351
 Windows XP, 701
DLLs, *See also bibliotecas de link dinâmico*
DLM (Distributed Lock Manager), , 15
DMA, *See also acesso direto à memória*
DMZ (zona desmilitarizada), , 521
domínio de proteção, , 471
domínio público, , 633
domínios, , 64, , 308, , 701-702
domínios de confiança, , 64
domínios de segurança, , 521
download progressivo, , 616
downsizing, , 533
DPCs (Deferred Procedure Calls), , 672-673
DRAM, *See also Dynamic Random-Access Memory*
driver de miniporta, , 685
 driver de porta, 685
 drivers de dispositivo, 10, 361, 440, 457, 715
drivers de filtro, , 684-685
DRM (Digital Rights Management), , 30
DTrace, , 66-68
dumpster diving, , 492
DVMA (Direct Virtual Memory Access), , 446
Dynamic Random-Access Memory (DRAM), , 8

E

E/S (entrada/saída), , 3, , 10-11
 bruta, 449
 direta, 449
 mapeada na memória, 307, 441
 programada, 307, 445
 sem bloqueio, 451-452
 sobreposta, 715-717
ECBs (Enabling Control Blocks), , 67
ECC, *See also código de correção de erro*
EDF, escalonamento, *See also escalonamento do prazo mais curto primeiro*
EEPROM (Electrically Erasable Programmable Read-Only Memory), , 8
efeito comboio, , 149
eficiência, , 3, , 377-378
EIDE, barramentos, , 400
Electrically Erasable Programmable Read-Only Memory (EEPROM), , 8
eleição, , 544
empacotamento, , 336
Enabling Control Blocks (ECBs), , 67
encadeamento de interrupção, , 443

encaminhamento, , 410

encapsulamento (Java), , 487

endereço(s):

definição, 443

esparços, 267, 281

físico, 250

Internet, 541

linear, 271

lógico, 250

virtual, 250

engenharia:

engenharia social, 492

reversa, 30

Enhanced Integrated Drive Electronics (EIDE), barramentos, , 400

entrada/saída, *See also E/S*

entradas do diretório de página (PDEs), , 678-679

envelhecimento, , 152, , 552

envio, , 96

EPROM (Erasable Programmable Read-Only Memory), , 70

Erasable Programmable Read-Only Memory (EPROM), , 70

erros, , 455

de hardware, 410

flexíveis, 408

rígidos, 410

escalabilidade, , 550

escalonadores, , 84-85

curto prazo, 84, , 146

de jobs, 84

de processos, 82

longo prazo, 84

médio prazo, 85

escalonamento, , 145

algoritmos de escalonamento de disco, 403-412

C-SCAN, 406

CineBlitz, 625

em sistemas multimídia, 621-622

FCFS, 404

LOOK, 406

SCAN, 405

selecionando, 406-407

SSTF, 404-405

baseado em prioridade, 603-604

compartilhamento proporcional, 610

cooperativo, 147

CPU, 16, 148-162, 171-173, 620-621

E/S, 452

Java, 167-171

prioridades de thread, 167-168

sistemas Solaris, 168-171

Linux, 642-645

multiprocessamento simétrico, 645

processo, 642-643

sincronização do kernel, 643-645

monotônico de taxa, 606-608

não preemptivo, 147

prazo mais antigo primeiro, 608

preemptivo, 147

Pthread, 610

tarefa, 16

thread, 145, 157

Windows XP, 671-672, 705-706

modelos, 171-173

análise de rede em fila, 172

implementação, 173
modelagem determinística, 171-172
simulações, 172-173
sistemas de tempo real, 606-610
de multiprocessador, , 157-162
 escalonamento do prazo mais curto primeiro, 608
 escalonamento monotônico de taxa, 606-608
 escalonamento por fatia proporcional, 610
 escalonamento Pthread, 610
 afinidade de processador, 159
 balanceamento de carga, 159-160
 exemplos:
 Linux, 166-167
 Solaris, 162-163
 Windows XP, 163-166
processadores multicore, 160-161
técnicas, 157
virtualização, 161-162

escalonamento de processos:

do prazo mais curto primeiro (EDF), , 608, , 621
do tempo restante mais curto primeiro, , 151
 escalonamento de threads, 145
 não preemptivo, 147
 não serial, 211
 preemptivo, 147
 Pthread, 610
 serial, 210-211
escalonamentos (serialização), 210-211
escape (sistemas operacionais), , 449
escopo de disputa, , 156-157
escritas assíncronas, , 380
escritas síncronas, , 380
escritores, , 189-191
espaço de disco, , 338, , 339
espaço de endereço:
 lógico, 250 *See also* **físico**,
 virtual, 280-281, 649
espaço de nomes:
 compartilhado, 570
 de nomes local, 570
espaço de sessão, 677
espaço de swap, 284
especificação de interface de dispositivo de rede (NDIS), 697
especificação de tempo real para Java (RTSJ), 26
espelhamento, 413, 695-696
espera:
 circular (condição de deadlock), 224
 ocupada, 185, 195-196, 442
esqueleto, 104-106
esquema:
 conservador de ordenação por estampa de tempo, 584
 de chave de lock, 479
 de partição fixa, 255
 de partição variável, 255
 esquema wound-wait, 585
estações de trabalho, 4
estado:
 de espera, 80, , 671
 de execução, 80
 de thread, 126-127, 671
 do processo, 81
 não sinalizado, 205
 novo, 80

pronto, 80

sinalizado, 205

suspenso, 705

terminado, 80

estágio (fita magnética), , 422

estampa de tempo, , 576, , 583-584

estratégia:

do melhor ajuste, 255-256

do pior ajuste, 255-256

do primeiro ajuste, 255

estrutura de anel, , 578

estrutura de armazenamento em massa, , 399-401

algoritmos de escalonamento de disco, 403-412

C-SCAN, 406

FCFS, 404

LOOK, 406

SCAN, 405

selecionando, 406-407

SSTF, 404-405

armazenamento terciário, 421-429

discos removíveis, 421-423

fitas magnéticas, 422-423

questões de desempenho, 426-429

suporte do sistema operacional, 423-424

tecnologia do futuro, 423

conexão de disco:

conectado à rede, 402-403

conectado ao host, 402

rede de armazenamento, 403

discos magnéticos, 399-400

estrutura de disco, 401

estrutura RAID, 412-420

melhoria de confiabilidade, 412-413

melhoria de desempenho, 413

níveis de RAID, 413-418

problemas, 419-420

extensões, 419

fitas magnéticas, 401

gerenciamento de disco:

bloco de boot, 408-409

blocos defeituosos, 409-410

formatação de discos, 408

gerenciamento de espaço de swap, 410-412

implementação de armazenamento estável, 420-421

estrutura de dados da fila de execução, , 166-167, , 642

estrutura simples do sistema operacional, , 53-54

esvaziamento, , 262

eventos, , 205

exceções (com interrupções), , 443

exclusão de arquivo, , 330

exclusão mútua, , 224, , 577-578

técnica centralizada, 577

técnica de passagem de tokens, 578

técnica totalmente distribuída, 577-578

exec(), chamada do sistema, , 127

execução:

de programas do usuário, 650-652

do programa (serviço do sistema

operacional), , 37

ext2fs, See also segundo sistema de arquivos estendido

extensão (espaço contíguo), , 57, , 370, , 692

extremidade de driver (STREAM), , 459-460

F**facilidade de uso**, , 3, , 668**falha**, , 255

detecção de, 548

durante escrita de bloco, 421

recuperação de, 549

tempo médio, 412

falsos:

negativos, 518

positivos, 518

falta de confiabilidade, , 543**fase de conflito (da latência de despacho)**, , 605**FAT (tabela de alocação de arquivo)**, , 372**fatias**, , 339**FC, barramentos**, , 400, , 402**FCB (bloco de controle de arquivo)**, , 362**FCFS, algoritmo de escalonamento**, See also **algoritmo de escalonamento primeiro a entrar, primeiro a ser atendido****fd**, See also **descriptor de arquivo****fiber channel (FC)**, , 400, , 402**fibras**, , 705-706**fids (NFS V4)**, , 570**FIFO, algoritmo de substituição de página**, , 291-292**filas**, , 83-84

capacidade, 96

dispositivos, 83

entrada, 250

escrita, 658

espera, 659

jobs, 83

leitura, 658

mensagem, 719

ordenada, 658

prontos, 83, 253

File System Hierarchy Standard, documento, , 633**File Transfer Protocol (FTP)**, , 533-534**filhos**, , 86-87**firewalls**, , 28, , 521-522**FireWire**, 401**firmware**, 6, 70

First-Come, First-Served (FCFS), algoritmo de escalonamento, 149, 404

fitas:

de rastreio, 172-173

magnéticas, 400, 422-423

fixação, 685

florestas, 701

fork:

dados, 336

memória virtual, 288

recursos, 335-336

fork():

chamada do sistema, 127

e exec(), modelo de processo (Linux), 639-641

formatação:

em baixo nível (discos), 408

física, 408

lógica, 408

fórmula de Little, , 172**fragmentação**, , 256-257

externa, 256-257, , 369

interna, 256, 336

fragmentos de pacote, , 661**frase de motivo**, , 556**Free Software Foundation (FSF)**, , 30

frequência de falta de página (PFF), , 303-304
ftp, , 349
FTP, *See also protocolo de transferência de arquivos*
funções de hash, , 509

G

gancho de atracação, , 501
ganho de velocidade de computação, , 91, , 532
Gantt, diagrama, , 149
gargalos, , 64, , 65
Gates, Bill, , 722
gateways, , 542
GB (gigabyte), , 5
gcc (GNU C Compiler), , 633
GDT (Global Descriptor Table), , 271
geração do sistema (SYSGEN), , 68-69
gerência de armazenamento, , 20-23
 caching, 21-23
 gerenciamento de armazenamento em massa, 21
 hierárquico (HSM), 425
 sistemas de E/S, 23

gerenciador:

 E/S, **684-685**
 lock distribuído (DLM), 15
 memória virtual (VM), 677-682
 processos (Windows XP), 682-683
 recursos, 620
 requisição, 658
 VM, *See gerenciador de memória virtual*

gerenciamento:

 arquivos, **50**
 cache, 22
 espaço de swap, 410-412
 espaço livre (discos), 375-377
 firewall, 661
 memória, 20, **645-652**, 706-708
 processos, 19-21, 425, 622-624, 639-645, 693-696
 gigabyte (GB), 5
 Global Descriptor Table (GDT), 271

GNU:

 C Compiler (gcc), **633**
 Linux, 30
 Manifesto, 30
 Portable Threads, 120
 Public License (GPL), 30

GPL (GNU Public License), , 30

gráfico

 acíclico, **344**
 alocação de recursos do sistema, 225-227

grau de multiprogramação, , 84-85

gravando arquivos, , 330

Green, threads, , 120

grupos:

 blocos, **654**
 cilindros, 654

GUIs, *See also interfaces gráficas com o usuário*

H

HAL, *See also camada de abstração de hardware*
hands-on, sistemas computadorizados, *See also sistemas computadorizados interativos*
handshaking, , 442, , 457, , 548
hardware, , 3
 de controlador de interrupção, 443

para armazenar tabelas de página, 260-262
sincronização, 182-184
sistemas de E/S, 439-447
acesso direto à memória, 445-446
interrupções, 442-445
polling, 442
Hardware Transactional Memory (HTM), , 208
heaps, , 79, , 707-708
hierárquico (HSM), , 425
hiperespaço, , 677
história, , 114
hive, , 688
homogeneidade, , 157
HSM (gerência de armazenamento hierárquico), , 425
HTM (memória transacional em hardware), , 208
Hydra, , 481-483

I

IBM OS/360, , 720-721
ícones, , 40
identidade de processo (Linux), , 639-640
identificação de cliente falsa, , 349-350
identificador de processo (pid), , 86-87
identificadores:
 arquivo, 330, , 364, , 563
 de espaço de endereço (ASIDs), 261-262
 grupo, 24
 usuário, 24
idioma de lock duplamente verificado, , 217
IDPs (sistemas de prevenção de intrusão), , 517
IDSs, *See also* **sistemas de detecção de intrusão**
IKE, protocolo, , 511
ILM (gerenciamento do ciclo de vida da informação), , 425
implementação:
 algoritmos de escalonamento de CPU, 173
 máquinas virtuais, 59
 sistemas operacionais de tempo real, 603-606
 escalonamento baseado em prioridade, 603-604
 kernels preemptivos, 603
 minimizando a latência, 604-606
 sistemas operacionais, 52-53
 técnicas de nomes transparentes, 562-563
imposição baseada no compilador, , 483-486
independência de local, , 560
índice, , 337-338, , 374
informação:
 contábil, 81
 de escalonamento de CPU, 81
 de estado, 351
 de gerenciamento de memória, 81
 de status, 50, 81
inicialização lazy, , 217
início de stream, , 459-460
InServ, array de armazenamento, 418
inspeção de pilha, 487
instruções privilegiadas, 18
integridade, quebra de, , 492
Intel Pentium, processador, , 269-273
intellimirror, , 702
interface com o usuário (UI), , 37-41
Interface:
 da linha de comandos (CLI), 39
 de aplicação (sistemas de E/S), 447-452

de bloco e caractere, 449
dispositivos de rede, 450
E/S com e sem bloqueio, 451-452
relógios e temporizadores, 450
de batch, 37
de chamada de sistema, 43
de cliente, 560
de driver de transporte (TDI), 697
de programa de aplicação (API), 42-43
de shell, criando, 112-114
entre máquinas, 560
gráficas com o usuário (GUIs), 39-41
redes Windows XP, 697
shell, 112-114
socket, 450

interlock de E/S, , 313-314

Internet Protocol (IP), , 511

interpretador

de cartão de controle, 715
de comandos, 39-40

interrupções, , 6, , 442-445

de software (traps), 445
definição, 442
mascaráveis, 443
não mascarável, 443
no Linux, 644-645

InterruptedException, 201

intervalo de tempo real (escalonadores Linux), 642

intrusos, 492

inversão de prioridade, 187-188, 606

invocação de método remoto (RMI), , 104-106

IP, See also Internet Protocol

IPC, See also comunicação entre processos

IPSec, , 511

IRP (pacote de requisição de E/S), , 684

iSCSI, , 403

ISO:

modelo de referência, 511
pilha de protocolos, 545

J

janelas pop-up do navegador, 495

Java:

ambiente de desenvolvimento, 63
API, 43, 62
arquivos mapeados na memória, 306-307
bloqueio de arquivos, 332
componentes, 61-64
Development Kit (JDK), 63
escalonamento, 167-171
estados de thread, 126-127
linguagem de programação, 61-62
passagem de mensagem, 96
processos, 89-90
proteção baseada em linguagem, 486-487
sincronização, *See sincronização*
sistemas operacionais, 63-64
threads, 124-127
tratamento de deadlock, 196, 226-229
Virtual Machine (JVM), 62-63, 126

JDK (Java development kit), , 63

jitter, , 619

jobs, processos, 79

jukebox robótico, , 425

K

KB (kilobyte), , 5

Kerberos, , 691

kernel(s), , 5, , 452-457

buffering, 453-454

caching, 454

desenvolvimento (Linux), 632

e subsistemas de E/S, 457

escalonamento de E/S, 452

estruturas de dados, 456-457

Linux, 635, 636

não preemptivos, 181

preemptivos, 181, 603

produção (Linux), 632

proteção, 455-456

sincronização de tarefa (no Linux), 643-645

sistemas de multimídia, 618-620

spooling e reserva de dispositivo, 454

tempo real, 601-602

tratamento de erro, 454-455

Windows XP, 670-674, 703

Kerr, efeito, , 422

keystreams, , 507

kilobyte (KB), , 5

Kindall, Gary, , 722

L

LANs, *See also* **redes locais**

largura de banda:

disco, 404

efetiva, 426

sustentada, 426

latência:

acesso, 426

despacho, 148, 605

evento, 604

interrupção, 604

rotação (discos), 399, 404

sistemas de tempo real, 604-606

lazy swapper, , 282

LCNs (números de cluster lógicos), , 692

LDAP, *See also* **Lightweight Directory-Access Protocol**

LDT (Local Descriptor Table), , 271

Least-Frequently Used (LFU), algoritmo de substituição de página, , 296-297

Least-Recently-Used (LRU), algoritmo de substituição de página, , 293-294

Lei de Kernighan, , 66

leitores, , 189-191

lendo arquivos, , 330

LFU, algoritmo de substituição de página, , 296-297

lgroub, entidade, , 300

licenças de software, , 218

Lightweight Directory-Access Protocol (LDAP), , 351, , 702

limite de segmento, , 270

linha:

requisição, 442, , 554

status, 555

links:

comunicação, 94

definição, 345

de endereço, 250

dinâmico, 651

dinâmico, 251-252, 651 *See also* estático,

estático, 251-252, 651

resolvendo, 345

rígidos, 345

simbólicos, 675

Linux, , 30-31, , 631-664

componentes do sistema, 631, , 635-636

comunicação entre processos, 659-660

distribuições, 631, 633

em sistemas Pentium, 272-273

escalonamento, 642-645

 multiprocessamento simétrico, 645

 processo, 642-643

 sincronização de kernel, 643-645

estrutura de rede, 660-661

exemplo de escalonamento, 166-167

exemplo de threads, 135-136

gerenciamento de memória, 645-652

 execução e carga de programas do usuário, 650-652

 memória física, 645-647

 memória virtual, 648-650

gerenciamento de processo, 639-645

 fork() e exec(), modelo de processo, 639-641

 processos e threads, 641-642

gerenciamento do espaço de swap, 412

história, 631-634

 descrição do sistema, 633

 distribuições, 633

 licença, 633-634

 primeiro kernel, 632-633

incluindo chamada de sistema ao kernel do Linux (projeto), 96-98

kernel, 632-633

modelo de segurança, 661-663

 autenticação, 661

 controle de acesso, 662-663

 módulos do kernel, 637-639

 princípios de projeto, 634-636

 representação de processo, 82

 sincronismo, 206

 sistema de E/S, 657-659

 dispositivos de bloco, 657-658

 dispositivos de caractere, 658-659

 sistemas de arquivos, 652-657

 diário, 655-656

 ext2fs, 653-655

 processo, 656-657

 virtuais, 652-653

 tempo real, 612

Lista, , 279

 ativa, 591

 controle de acesso (ACL), 353

 de acesso (NFS V4), 570

 de capacidade, 478

 espaço livre, 375

 exportação, 385

 interligadas, 376

 lineares (arquivos), 368

Live:

 CD, 31

 DVD, 31

livelock, , 195-196

livros-código, , 515

Local Descriptor Table (LDT), , 271

localidade de referência, , [284](#)

localização, arquivo, , [330](#)

lock(s), , [182-183](#), , [479](#)

compartilhados, [332](#)

consultivos, [334](#)

de arquivo, [332-334](#)

exclusivos, [332](#)

leitor-escritor, [189-191](#)

mutex, [184](#)

obrigatórios, [334](#)

logging, write-ahead, , [208-209](#)

login:

de rede, [351](#)

remoto, [533](#)

LOOK, algoritmo de escalonamento, , [406](#)

loop arbitrado (FC-AL), , [402](#)

loopback, , [101](#)

LPCs, *See also* [chamadas de procedimento local](#)

M

- MAC (Message-Authentication Code), , 509**
- Mach, sistema operacional, , 55, , 97-98, , 723-724**
- Macintosh, sistema operacional, , 335-336, , 723**
- mailslots, , 699**
- mainframes, , 4**
- manipulação do sistema de arquivos (serviço do sistema operacional), , 38**
- MANs (Metropolitan-Area Networks), , 25**
- mapa:**
 - de swap, **412**
 - de espaço, **377**
- mapeamento de memória, , 254, , 304-307**
 - definição, **304**
 - E/S mapeada na memória, **307**
 - mecanismo básico, **304-306**
 - no Linux, **650-652**
- máquinas virtuais, , 6, , 57-61**
 - alternativas, **59-61**
 - benefícios, **57-59**
 - ideia básica, **57**
 - implementação, **59**
 - VMware como exemplo, **59**
- Mars Pathfinder, , 187**
- marshalling, , 700**
- máscara de proteção (Linux), , 662**
- mascaramento, , 492**
- Master File Directory (MFD), , 341**
- matchmakers, , 103**
- matriz de acesso, , 474**
 - definição, **474-477**
 - e controle de acesso, **479-480**
 - e revogação de direitos de acesso, **480-481**
 - implementação de, **477-479**
- MB (megabyte), , 5**
 - MBR (Master Boot Record), **409**
 - MCP, sistema operacional, **724-725**
 - mecanismo de E/S rápida, **685**
 - mecanismo de paginação (Linux), **649**
 - mecanismo de pipe, **659**
 - mecanismo de serviço remoto, **563**
 - mecanismos, **51-52**
 - mecanismos de processamento distribuído, **699-700**
 - mecanismos de proteção retroajustados, **356**
 - mecanismos obrigatórios de lock de arquivo, **334**
 - média exponencial, **150**

Medium Access Control (MAC) endereço, 551-552

megabyte (MB), 5

melhoria de desempenho, 378-381, 413

memória:

acesso direto à memória virtual, 446

acesso direto à memória, 10

alocação a mais, 288

anônima, 411

compartilhada, 92, 281

física, 17

lógica, 17, 280

memória virtual unificada, 379

núcleo, 717

principal, *See memória principal*

secundária, 284

semicondutora, 9

transacional, 208

virtual, *See memória virtual*

memória anônima, , 411

memória compartilhada, , 92, , 281

memória de acesso aleatório (RAM), , 8

memória de demanda zero, , 648

memória de semicondutor, , 8

memória física, 17, 279-280, 645-647

memória lógica, , 17, , 280. *See also memória virtual*

memória principal, , 8, , 717

alocação contígua, 254

fragmentação, 256-257

mapeamento, 254

métodos, 254-256

proteção, 254

e carga dinâmica, 251

e espaço de endereço lógico, 250 *See also físico,*

e hardware, 247-249

e swapping, 252-254

e vínculo de endereço, 249-251

e vínculo dinâmico, 251-252

Intel Pentium, exemplo:

com Linux, 272-273

paginação, 271-272

segmentação, 270-273

paginação para gerenciamento de, 256-260

e páginas compartilhadas, 262-264

hardware, 260-262

Intel Pentium, exemplo, 271-272

método básico, 257-260

paginação hierárquica, 264-266

proteção, 262
tabelas de página em hash, 237
tabelas de página invertidas, 267-268
segmentação para gerenciamento de, 268-270
hardware, 269
Intel Pentium, exemplo, 269-273
método básico, 268-269

memória secundária, , 284

memória somente de leitura (ROM), , 69, , 409

memória transacional, , 208

memória transacional por software (STM), , 208

memória virtual, , 17, , 279-281

- acesso direto à memória virtual, 446
- alocação de memória do kernel,
 - alocação de quadro, 297-300
 - acesso de memória não uniforme, 300
 - alocação global, 299-300 *See also* local,
 - alocação igual, 298
 - alocação proporcional, 299
- kernel, 650
- Linux, 648-650
- mapeamento de memória, 304-307
 - E/S mapeada na memória, 307
 - mecanismo básico, 304-306
- paginação por demanda para conservar, 282-287
 - alcance de TLB, 311
 - com tabelas de página invertidas, 312
 - desempenho, 285-287
 - estrutura de programa, 312-313
 - instruções para reinício, 284-285
 - interlock de E/S, 313-314
 - mecanismo básico, 282-285
 - paginação por demanda pura, 284
 - pré-paginação, 310
 - tamanho de página, 310-311
- rede, 563
- separação entre memória lógica e memória física, 280
- Solaris, 315-316
- substituição de página para economizar, 288-297
 - algoritmos de buffer de página, 297
 - desempenho da aplicação, 297
 - mecanismo básico, 288-291
 - substituição de página baseada em contagem, 296-297
 - substituição de página FIFO, 291-292
 - substituição de página ideal, 292-293
 - substituição de página LRU, 293-294
 - substituição de página por aproximação LRU, 294-296

tamanho, 279
técnica de cópia na escrita, 287-288
thrashing, 300-304
causa, 300-301
estratégia de frequência de falta de página, 303-304
modelo de conjunto de trabalho, 302-303
unificada, 379
Windows XP, 314
memória virtual em rede, 563

memória virtual unificada, , 379

MEMS (Micro-Electronic Mechanical Systems), , 423

mensagens:
de resposta, 555
em sistemas operacionais distribuídos, 533
sem conexão, 543

metadados, , 351, , 692

metafiles, , 623

método:
de partição múltipla, 255
desaprovados, 129-130
Java, 486

MFD (Master File Directory), , 341

MFU, algoritmo de substituição de página, , 296-297

microkernels, , 55-57

Microsoft:
Interface Definition Language, 700
Windows, ver sob Windows

mídia de armazenamento removível:
discos, 421-423
discos magnéticos, 399-400
fitas magnéticas, 400, 422-423
gerência de armazenamento hierárquico, 425
interface de aplicação com, 424
nomes de arquivos, 424-425

migração:
arquivo, 560
computação, 535
dados, 534-535
processo, 535
pull, 159-160
push, 159-160, 561
recursos, 711

minicomputadores, , 4

minidiscos, , 339

mistura de processos, , 85

MMU, See also unidade de gerenciamento de memória

mobilidade do usuário, , 385

modo supervisor, *See also modo kernel*

modelagem determinística, , 171-172

modelo:

cliente-servidor, 349-350

conjunto de trabalho, 302-303

localidade, 301

memória compartilhada, 50, 92-94

multithreading muitos para muitos,
120-121

multithreading muitos para um, 120

multithreading um para um, 120

passagem de mensagem, 49, 94

referência ISO, 511

Modo:

bloqueio compartilhado, 581

de falha (diretórios), 351

do sistema, ver modo kernel

endereçamento real, 602

hot-standby, 14

kernel, 17, 635

lock exclusivo, 581

modo privilegiado, ver modo kernel

simétrico, 14

usuário, 17

modificação:

arquivos, 50

mensagem, 492

modularidade, 91

módulo, , 56-57, , 459-460

de autenticação conectáveis (PAM), 661

de stream, 459-460

do kernel, 637-639

gerenciamento, 637-638

organização de arquivo, 362

registro de driver (Linux), 638

registro de driver, 638

resolução de conflito (Linux), 638-639

resolução de conflito, 638-639

monitores, , 192-193

de referência de segurança (SRM), 686

implementação usando semáforos, 194-195

residente, 714

solução do jantar dos filósofos usando, 194-195

utilização, 193-194

monocultura, , 500

monotônico, , 576

montagem, , 364-366

Morris, Robert, , 501-503
MS-DOS, , 689, , 722-723
multicasting, , 623
MULTICS, sistema operacional, , 473-474, , 720
multimídia, , 615-615

 como termo, 615-615
 questões de sistema operacional com, 617

múltiplas notificações e sincronização, , 198-199

multiprocessamento:

 assimétrico, 12, , 159
 simétrico, 12, 159, 645
multiprogramação, , 15-16, , 84
multitarefa, *See also* **compartilhamento de tempo**
multithreading:

 ativações do escalonador, 133-134
 benefícios, 118
 cancelamento de thread, 129
 chamada do sistema exec(), 127
 chamada do sistema fork(), 127
 coarse-grained, 161
 dados específicos da thread, 132-134
 fine-grained, 161
 modelos, 120
 pools de threads, 131-132
 tratamento de sinais, 130-131

multithreading fine-grained, , 161

MUP (provedor múltiplo de convenção de nome universal), , 701

mutex:

 adaptativo, 204-205
 Windows XP, 672

mutual-exclusion condition (deadlocks), , 229

N

named pipes, , 699
não preempção (condição para deadlock), 224
não repúdio, 509
NDIS (Network Device Interface Specification), 697
negociação, 619
NetBEUI (NetBIOS Extended User Interface), 698
NetBIOS (Network Basic Input/Output System), 697, 699
NetBIOS Extended User Interface (NetBEUI), 698
Network Basic Input/Output System, *See also* **NetBIOS**
NFS, protocolo, , 386-387
NFS, *See also* **sistemas de arquivos de rede**
NFS V4, , 569
NIS (Network Information Service), , 350
níveis, 618

níveis de prioridade de interrupção, 443
NLS (National-Language-Support), API, 670
nomeação, , 94-96, , 350-351
comunicação da rede, 540-541
de arquivos, 330
definição, 560
Lightweight Directory-Access Protocol, 351
sistema de nome de domínio, 350

nomes:

resolução, 540-542, , 702
Windows XP, 674-675

nomes de caminho, , 342

absolutos, 343-344
relativos, 343-344

Novell NetWare, protocolos, 698

NTFS, 691-692

núcleos, 13

NUMA, *See also* **acesso não uniforme à memória**

número:

de bloco relativo, 337
de cluster lógicos (LCNs), 692
de identificação pessoal (PIN), 515
de página (p), 256
de vnode (NFS V4), 570
mágico (arquivos), 335

NVRAM:

RAM não volátil, 9
RAM não volátil, cache, 413

O

Object Linking and Embedding (OLE), 700

objeto:

arquivo, 367, , 652
contêiner (Windows XP), 525
de sessão, 678
dentry, 367, 652
despachantes, 205, 672, 674
diretório (Windows XP), 675
em cache, 308-310
evento (Windows XP), 672
hardware, 470
hardware versus software, 470
inode, 367, 652
job, 682-683
link simbólico, 675
listas de acesso, 478
livres, 308, 647

locais (não remotos), 106
não contêiner (Windows XP), 525
não remotos (locais), 106
no Linux, 647
no Windows XP, 674-677
processo (Windows XP), 672
remotos, 106-107
seção, 99
software, 470
superbloco, 367, 652
usados, 308, 647

OLE, See also Object Linking and Embedding

OLPC (One-Laptop per Child), , 723

One-Laptop per Child (OLPC), , 723

one-time pad, , 515

open (termo), , 500

open(), operação, , 331

operações de E/S (serviço do sistema operacional), 37

operações em conflito, 211

operações remotas, 388

ordenação de eventos, 575-577

ordenação global, 576

organização de correção de erro no estilo da memória, 415

organização de paridade intercalada por bit, 415

organização de paridade intercalada por bloco, 416

OS/2, sistema operacional, 667

P

p (número de página), 256

P + Q, esquema de redundância, 416

pacote de requisição de E/S (IRP), , 684

pacotes, , 543, , 660-661

padrão de codificação avançado (AES), , 506

padrão de projeto Singleton, , 217

pageout (Solaris), , 315

pager (termo), , 282

paginação, , 256-264

exemplo do Intel Pentium, 271-272

hierárquica, 264-266

invertida, 267-268

método básico, 257-260

no Linux, 649-650

páginas compartilhadas, 262-264

prioridade, 316

proteção de memória, 262

suporte do hardware, 260-262

swapping, 411

tabelas de página em hash, 266

paginação hierárquica, , 264-266

paginação por demanda, , 282-287

- alcance do TLB, 311
- com tabelas de página invertidas, 312
- definição, 282
- desempenho, 285-287
- estrutura do programa, 312-313
- interlock de E/S, 313-314
- mecanismo básico, 282-285
- pré-paginação, 310
- pura, 284
- reiniciando instruções, 284-285
- tamanho de página, 310-311

paginação por demanda pura, , 284

paginação por prioridade, 316

páginas:

- compartilhadas, 262-264
- definição, 256

páginas residentes na memória, , 283

PAM (Pluggable Authentication Modules), , 662

papéis, , 480

parallelismo, , 14

paravirtualização, , 61

parcelas, , 104-106

paridade distribuída intercalada por bloco, , 416

partição de boot, , 409

particionamento de disco, , 408

partições, , 254-255, , 338, , 339, , 364-365

- boot, 409
- brutas, 411
- raiz, 364-365

partições brutas, , 411

partições raiz, , 364-365

passagem de mensagem, , 92, , 96

- assíncrona (sem bloqueio), 96
- por bloqueio (síncrona), 96
- sem bloqueio (assíncrona), 96
- síncrona, 96

passagem de tokens, 544, 578

pastas, 40

PC, sistemas, 3

PCBs, See also blocos de controle de processo

PCS (Process-Contention Scope), , 156-157

PDAs, See also assistentes digitais pessoais

PDEs (Page-Directory Entries), , 678-679

pegada, 600

perda de dados, tempo médio para, 413
perfis, 618
perfis roaming, 701
 períodos, , 618
 permissões (sistema Unix), , 356
 persistência de visão, , 616
 PFF, *See also frequência de falta de página*
 phishing, , 492
 PIC (Position-Independent Code), , 652
 pid (Process Identifier), , 85-87
pilha, 44, 79
 PIN (Personal Identification Number), , 515
 PIO, *See also E/S programada*
placa (discos), 399
players de mídia, 623-624
 plug-and-play (PnP) e gerenciadores, , 686-688
 PnP, gerenciadores, *See also plug-and-play (PnP) e gerenciadores*
 política, , 51-52
escrita atrasada, 564, , 565
grupo, 702
pageout (Linux), 649
segurança, 516
write-through, 564
polling, 442
 ponteiro:
arquivo, 332, , 396
da posição atual no arquivo, 330
disco, 394
quadro, 496
 pontos:
cancelamento, 129
montagem, 347, 697
preempção, 604
verificação, 209
 pools:
páginas livres, 288
RAID, 420
threads, 131-132, 706
portabilidade, 670
portais, 28, 64
portas, 307, 440
portas (caixas de correio), 95
portas de E/S, 307
POSIX, 667, 669, 690
posse (de capacidade), 478
PPTP (Point-to-Point Tunneling Protocol), 698
pré-paginação, 310

preempção de recursos, recuperação de deadlock, 238
princípio do menor privilégio, 470
prioridade dinâmica, 621
prioridade estática, , 621
privilégios, escalando, , 24
problema da seção crítica, , 180-181
e semáforos, 184-185
deadlocks, 186-187
implementação, 185-186
starvation, 187
uso, 184-185
hardware de sincronização, 182-184
solução de Peterson, 181-182
problema de alocação dinâmica de armazenamento, , 255, , 369
problema de buffer limitado, , 92, , 188-189, , 195-198
e condição de corrida, 196
e deadlock, 196
e espera ocupada, 195-196
e métodos de espera/notificação, 197-198
problema de confinamento, , 477
problema de consistência de cache, , 563
problema do barbeiro dorminhoco, , 219
problema do jantar dos filósofos, , 191-192, , 194-195
problema dos generais bizantinos, , 591
problema dos leitores-escritores, , 189-191, , 199-200
processadores de comunicação, , 537
processadores de front-end, , 462
processadores multicore, , 160-161
processo daemon, , 473
processo despachado, , 83
processo do sistema (Windows XP), , 688
processo pai, , 85-87, , 676
processos, , 16
ambiente, 640
componentes, 79
comunicação entre, *See comunicação entre processos*
contexto, 85, 640-641
cooperação, 91-92
definição, 79
escalonamento, 82-85
estado, 80
falha, 592-593
independentes, 91
Java, 13-14
Linux, 641-642
multithreaded, *See multithreading*
operações, 85-91

criação, 85-89
término, 90-91
pesados, 117
primeiro plano, 154
programas, 20, 79, 80
segundo plano, 154
tarefa, 79
termo, 79
threads executadas por, 82
trocas de contexto, 85
única thread, 117
voltados para E/S, 85 *See also* voltados para CPU,
Windows XP, 703

processos de primeiro plano, , 154

processos de única thread, , 117

processos em cooperação, , 91-92

processos em segundo plano, 154

processos filhos, 676-677

processos independentes, 91

processos periódicos, 618

processos pesados, , 117

processos voltados para CPU, , 85

processos voltados para E/S, , 85

produto de matrizes, , 141

profiling, , 66

programação multicore, , 119-120

programas, processos, 79, , 80. *See also* **versus**, Ver também **programas de aplicação**

programas aplicativos, , 3, , 51

desinfecção, 519-521

processamento em múltiplas etapas, 249, 250

processes, 20

utilitários do sistema, 50

programas armazenados, , 711-712

programas de bootstrap, , 408-409, , 501

programas de bootstrap (carregadores de bootstrap), , 6, , 69

programas de computador, *See also* **programas de aplicação**

programas de controle, , 4

programas de vetor, , 501

programas do sistema, , 49-51

programas do usuário (tarefas do usuário), , 79, , 650

projeto de sistemas operacionais:

- Linux, 634-636
- mecanismos e políticas, 51-52
- objetivos, 51
- sistemas operacionais distribuídos, 550-551
- Windows XP, 668-670

proteção, , 469

ambiente paginado, [262](#)
arquivo, [330](#)
chamadas do sistema, [50](#)
contra cópia, [30](#)
contra vírus, [519–521](#)
controle de acesso, [353–354](#)
de memória, [254](#)
dinâmica, [471](#)
domínio, [470–474](#)
 estrutura, [471–472](#)
 exemplo do MULTICS, [473–474](#)
 exemplo do UNIX, [472–473](#)
E/S, [455–456](#)
estática, [471](#)
estática, [471](#) *See also* dinâmica,
matriz de acesso como modelo, [474–477](#)
 controle de acesso, [479–480](#)
 implementação, [477–479](#)
objetivos, [469](#)
permissões, [356](#)
princípio do menor privilégio, [470](#)
retroajustada, [356](#)
revogação de direitos de acesso, [480–481](#)
segurança, [491](#)
serviço do sistema operacional, [39](#)
sistemas baseados em capacidade, [481–483](#)
 Cambridge CAP, sistema, [483](#)
 Hydra, [481–483](#)
sistemas baseados em linguagem, [483–487](#)
 imposição baseada em compilador, [483–486](#)
 Java, [486–487](#)
sistemas computadorizados, [23–24](#)
sistemas de arquivo, [353–356](#)
tratamento de erros, [454–455](#)

protelar, , 247

protocolo:

baseados em estampa de tempo, [212–213](#)
camada de rede, [511](#)
camada de transporte (TCP), [511](#)
commit, [579](#)
controle de transmissão (TCP), [546](#)
datagrama do usuário (UDP), [546](#)
de lock, [211, 581–583](#)
herança de prioridade, [187, 205, 606](#)
lock em duas fases, [211](#)
maioria, [582](#)
montagem, [385–386](#)

parcial, 582-583
redes Windows XP, 697-698
resolução de endereço (ARP), 551
roteamento, 543
sem estado, 624
transporte em tempo real (RTP), 622
tunelamento ponto a ponto (PPTP), 698
provedor múltiplo de convenção de nome universal (MUP), , 701
provedores, , 67
provisão a mais, , 618
PTBR (Page-Table Base Register), , 261
Pthreads, , 122-123
 escalonamento, 157
 sincronização, 206
PTLR (Page-Table Length Register), , 262

Q

quadro(s), , 257, , 543, , 616
 de página, 678-679
 pilha, 496
 vítima, 288
qualificador do código de sensibilidade adicional, , 455
quantum, , 671
 de tempo, 152
quebra, , 65
 confidencialidade, 491
 disponibilidade, 492
 integridade, 492
 quebras, 65

R

R-timestamp, , 212-213
RAID (Redundant Arrays of Inexpensive Disks), , 412-420
 array, 413
 estruturação, 413
 melhoria de confiabilidade, 412-413
 melhoria de desempenho, 413
 níveis, 413-418
 problemas com, 419-420
raiz de índice, , 692
rajada de E/S, , 145-146
RAM (Random-Access Memory), , 8
 não volátil (NVRAM), 91, , 413
razão de compactação, 617
RBAC (Role-Based Access Control), 480
RC 4000 sistema operacional, 719
recepção por bloqueio, 96

recepção sem bloqueio, 96

reconfiguração, , 548-549

recuperação:

backup e restauração, **383**

de arquivos e diretórios, **381-383**

de deadlock, **237-238**

pela preempção de recurso, **238**

pelo término de processo, **238**

de falhas, **549**

verificação de consistência, **381**

Windows XP, **693**

recurso de montagem automática, , 562

rede(s). See also redes locais (LANs); redes remotas (WANs)

área metropolitana (MANs), **25**

área pequena, **25**

definição, **25**

estrutura de comunicação, **540-544**

disputa, **543-544**

estratégias de conexão, **543**

estratégias de pacote, **543**

estratégias de roteamento, **542-543**

resolução de nomeação/nomes, **540-542**

exemplo, **551-552**

Linux, **660-661**

parâmetros de comunicação, **544-547**

questões de projeto, **550-551**

robustez, **548-549**

segurança, **492-494**

sem fio, **27**

threats to, **500-501**

tipos, **536**

topologia, **538-540**

Windows XP, **697-702**

Active Directory, **702**

domínios, **701-702**

interfaces, **697**

mecanismos de processamento distribuído, **699-700**

protocolos, **697-698**

redirecionadores e servidores, **700-701**

resolução de nomes, **702**

redes:

área pequena, **25**

armazenamento (SANs), **15, 402, 403**

locais (LANs), **14, 25, 536-537**

metropolitanas (MANs), **25**

parcialmente conectadas, **539**

privativas virtuais (VPNs), **511, 698**

remotas (WANs), 14, 25, 537–539
sem fio (WiFi), 28, 536, 537
WiFi (sem fio), 28, 536, 537

redirecionadores, , 700–701

redundância, , 413. *See also RAID*

Reed-Solomon, códigos, , 416

referência de arquivo, , 692

regiões da memória virtual, , 648

registrador de controle, , 441

registrador de entrada de dados, , 441

registrador de instrução, , 9

registrador de saída de dados, , 441

registrador de status, , 441

registradores, , 44

- base da tabela de página, 261
- base, 247, 248
- CPU, 81
- endereço de memória, 250
- limite, 247, 248
- para tabelas de página, 260–261
- relocação, 251
- saída de dados, 441
- status, 441
- tamanho da tabela de página, 262
- toque de tecla, 500

registro, , 50, , 688

registro base de arquivo, , 692

registros:

- lógicos, 336
- mestre de boot, 409

regra dos, 50, , 256

regras de escalonamento, , 705

relação ocorreu antes, , 575–577

relacionamento de confiança:

- enlace cruzado, 701–702
- transitivo, 701
- unidirecional, 701

relógio, , 450

- lógico, 576

remapeamento de cluster, 696

remessa de chave fora de faixa, 509

rendezvous, 96

reparo, tempo médio para, 413

replicação, 389, 418, 419

replicação de arquivo (sistemas de arquivos distribuídos), 568–569

reposicionamento (em arquivos), , 330

representação de dados externos (XDR), , 103

representação do processo (Linux), , 82
reprodução local, , 616
requisição (modelo do sistema), , 223
reserva de dispositivo, , 454
reserva de setor, , 410, , 696
reservas de recursos, , 620
resolução:
 nome, **540-542**
 tamanho de página, 311
resolvendo links, , 345
restauração:
 dados, **383**
 estado, 85
restauração do sistema, , 688
revogação de direitos de acesso, , 480-481
Rich Text Format (RTF), , 519
RMI, *See also* **Remote Method Invocation**
robustez, , 548-549
roll out, roll in, , 253
ROM, *See also* **memória somente de leitura**
root uid (Linux), , 662
roteamento:
 comunicação em rede, **542-543**
 em redes parcialmente conectadas, 539
roteamento dinâmico, , 542
roteamento fixo, , 542
roteamento virtual, , 542
rotinas de serviço de interrupção da metade inferior, , 645
rotinas de serviço de interrupção da metade superior, , 645
rotinas stub, , 699
roubo de ciclo, , 446
roubo de serviço, , 492
RPCs (Remote Procedure Calls), , 699-700
RR, algoritmo de escalonamento, *See also* **algoritmo de escalonamento round-robin**
RTF (Rich Text Format), , 519
RTP (Real-Time Transport Protocol), , 622
RTSJ (Real-Time Specification for Java), , 26
RW (Read-Write), formato, , 21

S

salvamento de estado, , 85

SANs, *See also* **redes de armazenamento**

SATA, barramentos, , 400

SCAN (elevador), algoritmo de escalonamento, , 405, , 621-622

SCAN circular (C-SCAN), algoritmo de escalonamento, , 406

SCOPE, sistema operacional, , 725

script de shell, , 335

script kiddies, , 498

SCS (System-Contention Scope), , 156-157

SCSI, alvos, , 402

SCSI, barramentos, , 400

SCSI, iniciador, , 402

SCSI (Small Computer-Systems Interface), , 10

seção:

dados (do processo), 79

críticas, 180

entrada, 180

restante, 180

saída, 180

texto (do processo), 79

segmentação, , 268-269

definição, 268

exemplo do Intel Pentium, 269-273

hardware, 269

método básico, 268-269

segredo mestre (SSL), 511

segredo pré-master (SSL), 511

segundo sistema de arquivos estendido (ext2fs), 653-655

segurança de tipo (Java), 487

segurança física, 492

segurança humana, 492

segurança. *See also* **acesso a arquivo; ameaças ao programa; proteção; autenticação do usuário**

ameaças ao sistema/rede, 500-503

negação de serviço, 503-504

varredura de porta, 503

vermes, 501-503

classificações, 522-523

com protocolos de comunicação, 546, 547

como serviço do sistema operacional, 38-39

em sistemas de computação, 24

firewalling, 521-522

implementação, 516-521

auditoria, 521

avaliação de vulnerabilidade, 516-517

contabilidade, 521
detecção de intrusão, 517-519
logging, 521
política de segurança, 516
proteção contra vírus, 519-521
Linux, 661-663
autenticação, 661
controle de acesso, 662-663
níveis, 492-493
no Windows XP, 523-525, 668
problema, 491-494
proteção, 491
uso de criptografia, 504-512
codificação, 505-510
exemplo do SSL, 511-512
implementação, 510-511
via autenticação do usuário, 512-516
biometria, 515-516
senhas, 512-516
Windows XP, 693
seguro por padrão, 500

semáforos, , 184-185

binários, 184
contando, 184
deadlocks, 186-187
definição, 184
implementação de monitores usando, 194-195
implementação, 185-186
inversão de prioridade, 187-188
starvation, 187
utilização, 184
Windows XP, 672

semântica:

arquivos compartilhados imutáveis, 352
consistência, 351-352
cópia, 453-454
sessão, 352

sementes, , 515

senhas, , 512-516

criptografadas, 514
emparelhadas, 514
única vez, 514-515
vulnerabilidades, 513-514

senhas emparelhadas, , 514

sequência automática de tarefas, 714
sequência segura, 232
sequestro de sessão, 492

serial ATA (SATA), barramentos, 400

serialização, 210-211

serialização de objetos, 106

serviço:

de arquivo

com estado, 567

de log, 693

sem estado, 567

de informação de rede (NIS), 350

de nomes distribuídos, *See sistemas de informação distribuídos*

do sistema operacional, 37-39

servidor web (projeto), , 554-557

servidores, , 4

cluster, 570

de lâmina, 13

definição, 560

SSL, 511

sessão de arquivo, 351

sessões de comunicações, 543

setor:

de boot, 409

da partição, 363

de disco, 399

shells, 39

shoulder surfing, 513

simulações, , 60, , 172-173

sinais:

Linux, 659

UNIX, 130-131

sinalizar e continuar, , 194

sinalizar e esperar, , 194

sincronismo de bloco, , 200-201

sincronismo de processo:

exemplos:

Linux, 206

Pthreads, 206

Solaris, 204-205

Windows XP, 205-206

monitores, 192-193

semáforos, implementação usando, 194-195

solução do jantar dos filósofos, 194-195

utilização, 193-194

problema de buffer limitado, 188-189

problema de leitores-escritores, 189-191, 199-200

problema de seção crítica, 180-181

solução de Peterson, 181-182

solução do hardware, 182-184

problema do jantar dos filósofos, 191-192, 194-195

semáforos, 184-185

sobre, 179-180

transações atômicas, 207-213

modelo de sistema, 207-208

pontos de verificação, 209-210

recuperação baseada em log, 208-209

transações concorrentes, 210-211

sincronização, , 95, , 195-204. *See also sincronização de processo*

bloco, 200-201

InterruptedException, 201

notificações múltiplas, 198-199

problema de buffer limitado, 195-198

problema de escritores-leitores, 189-191, 199-200

recursos de concorrência em Java, 201-204

regras, 200-201

síntese de mensagem (valor de hash), , 509

sistema buddy (Linux), , 646

sistema de arquivos estendido, , 362, , 653

sistema de arquivos lógico, , 362

sistema de arquivos virtual (VFS), , 366-367, , 652-653

sistema de nome de domínio (DNS), , 350, , 540

sistema em execução, , 71

sistema no chip (SOC), estratégia, , 600, , 601

sistemas clientes, , 28

sistemas críticos à segurança, , 600

sistemas de arquivo de diário, *See also sistemas de arquivos orientados a transação baseados em log*

sistemas de arquivo de processo (Linux), , 656-657

sistemas de arquivo remoto, , 349

sistemas de arquivos, , 329, , 361-362

aspectos básicos, 361-362

criação, 340

distribuídos, 362 *See also sistemas de arquivos distribuídos*,

implementação, 363-368

montagem, 364-366

partições, 364-365

sistemas virtuais, 366-367

Linux, 651-657

lógicos, 362

montagem, 347-348

níveis, 361

orientados a transação baseados em log, 381-382

problemas de projeto, 361

rede, 383-388

remotos, 349

WAFL, 388-390

sistemas de arquivos básicos, , [361-362](#)

sistemas de arquivos de rede (NFS), , [383-388](#)

operações remotas, [388](#)

protocolo de montagem, [385-386](#)

protocolo NFS, [386-387](#)

tradução de nome de caminho, [387-388](#)

sistemas de arquivos distribuídos (DFSs), , [559-560](#)

acesso remoto a arquivo, [563-566](#)

caching, [566](#) *See also* **serviço remoto**,

consistência, [565-566](#)

esquema básico, [563](#)

local de cache, [563-564](#)

política de atualização de cache, [564-565](#)

definição, [559](#)

exemplo do AFS, [569-573](#)

espaço de nomes compartilhado, [570-571](#)

implementação, [572-573](#)

operações com arquivos, [571-572](#)

nomeação, [560-563](#)

replicação de arquivos, [568-569](#)

serviço com e sem estado, [566-568](#)

sistemas de arquivos estruturados em log, , [381-382](#)

sistemas de banco de dados, , [207](#)

sistemas de boot dual, , [365](#)

sistemas de computação:

ameaças, [500-501](#)

armazenamento, [8-10](#)

arquitetura:

sistemas de único processador, [10-11](#)

sistemas em clusters, [13-15](#)

sistemas multiprocessados, [11-13](#)

estrutura de E/S em, [10-11](#)

gerência de armazenamento, [20-23](#)

caching, [21-23](#)

gerenciamento de armazenamento em massa, [21](#)

sistemas de E/S, [23](#)

gerência de processos, [19-20](#)

gerenciamento de memória, [20](#)

gerenciamento do sistema de arquivos em, [20-21](#)

operação de, [6-8](#)

proteção, [23-24](#)

segurança, [24](#)

seguros, [491](#)

sistema operacional visto por, [4](#)

sistemas de uso específico, [25-27](#)

sistemas de multimídia, [26](#)

sistemas de tempo real embutidos, [25-26](#)

- sistemas portáteis, 26-27
- sistemas distribuídos, 24-25
- sistemas de computação interativos (hands-on), , 16**
- sistemas de computador de uso específico, , 25-27**
 - sistemas de tempo real embutidos, 25-26
 - sistemas multimídia, 26
 - sistemas portáteis, 27
- sistemas de computador pessoal (PC), 3
- sistemas de detecção de intrusão (IDSs), , 517-519**
- sistemas de E/S, , 439**
 - e desempenho do sistema, 460-463
 - hardware, 439-447
 - acesso direto à memória, 445-446
 - interrupções, 442-445
 - polling, 442
 - interface de aplicação, 447-452
 - clocks e temporizadores, 450
 - dispositivos de bloco e caractere, 449
 - dispositivos de rede, 450
 - E/S por bloqueio e sem bloqueio, 451-452
 - kernels, 452-457
 - buffering, 453-454
 - cache, 454
 - e subsistemas de E/S, 457
 - escalonamento de E/S, 452
 - estruturas de dados, 456-457
 - proteção, 455-456
 - spooling e reserva de dispositivo, 454
 - tratamento de erros, 454-455
 - Linux, 657-659
 - dispositivos de bloco, 657-658
 - dispositivos de caractere, 658-659
 - STREAMS, mecanismo, 459-460
 - transformação de requisições em operações de hardware, 457-459
- sistemas de informação distribuídos (serviços de nomes distribuídos), , 350**
- sistemas de multimídia, , 26, , 615**
 - características, 617
 - CineBlitz, exemplo, 625-626
 - compactação, 617-657
 - escalonamento de CPU, 620-621
 - escalonamento de disco, 621-622
 - gerenciamento de rede, 622-624
 - kernels, 618-620
- sistemas de prevenção de intrusão (IDPs), , 517**
- sistemas de proteção baseados em capacidade, , 481-483**
 - Cambridge CAP sistema, 483
 - Hydra, 481-483

sistemas de proteção baseados em linguagem, , [483-487](#)

imposição baseada em compilador, [483-486](#)

Java, [486-487](#)

sistemas de servidor de arquivos, , [28](#)

sistemas de tempo real, , [25-26](#), , [599-600](#)

características, [600-601](#)

definição, [599](#)

escalonamento de CPU, [606-610](#)

exemplo do VxWorks, [610-611](#)

flexíveis, [600](#), [620](#)

implementação, [603-606](#)

escalonamento baseado em prioridade, [603-604](#)

kernels preemptivos, [604](#)

reduzindo a latência, [604-606](#)

pegada, [600](#)

recursos não necessários, [601-602](#)

rígidos, [600](#), [620](#)

tradução de endereço em, [602](#)

sistemas de tempo real flexíveis, , [600](#), , [620](#)

sistemas de tempo real rígidos, , [600](#), , [620](#)

sistemas de único processador, , [10-11](#), , [145](#)

sistemas distribuídos, , [24-25](#)

benefícios, [531-533](#)

definição, [531](#)

sistemas operacionais de rede como, [533-534](#)

sistemas operacionais distribuídos como, [534-535](#)

sistemas em clusters, , [13-15](#)

sistemas embutidos, , [600](#)

sistemas extensíveis baseados em linguagem, , [63](#)

sistemas fortemente acoplados, *See also sistemas multiprocessados*

sistemas mecânicos microeletrônicos (MEMS), , [423](#)

sistemas multiprocessados (sistemas paralelos, sistemas fortemente acoplados), , [11-13](#)

sistemas operacionais, , [3](#)

alocador de recursos, [4](#)

controlados por interrupção, [17](#)

definição, [3](#), [4-5](#)

estrutura, [15-17](#), [53-57](#)

estrutura simples, [53-54](#)

microkernels, [55-57](#)

módulos, [56-57](#)

técnica em camadas, [54-55](#)

funcionamento, [3-5](#)

guest, [59](#)

implementação, [52-53](#)

interface com o usuário, [3-4](#), [39-41](#)

mechanismos, [51-52](#)

migração de recursos, [711](#)

objetivos de projeto, 51
operações:
 modos, 17-19
 timer, 19
políticas, 51-52
primeiros, 711-717
 E/S sobreposta, 715-717
sistemas computadorizados compartilhados, 713-715
sistemas de computador dedicados, 712-713
recursos, 3
rede, 25
segurança, 492-494
serviços fornecidos, 37-39
tempo real, 25-26
visão do sistema, 4

sistemas operacionais controlados por interrupção, , 17

sistemas operacionais de código-fonte aberto, , 6, , 29-32

- benefícios, 29-30
- BSD UNIX, 31
- definição, 29
- história, 30
- Linux, 30-31
- Solaris, 31

sistemas operacionais de código-fonte fechado, , 29

- sistemas operacionais de rede, 25, 533-534

sistemas operacionais de tempo real, , 25-26

sistemas operacionais distribuídos, , 534-535

sistemas operacionais guest, , 59

sistemas paralelos, *See also* sistemas multiprocessados

sistemas portáteis, , 27

sistemas seguros, , 491

sistemas tolerantes a falhas, , 549

sistemas zumbis, , 503

slots de página, , 412

Small Computer Systems Interface, *See also* sob SCSI

SMB, *See also* bloco de mensagem do servidor

SMP, *See also* multiprocessamento simétrico

snapshots, , 382, , 389-390

sniffing, , 513

sobrealocação (de memória), , 288

SOC, estratégia, *See also* system-on-chip, estratégia

sockets, , 100-102

- orientados a conexão (TCP), 100
- sem conexão (UDP), 100

Solaris, , 31, , 61

- escalonamento de threads Java, 168-170
- exemplo de escalonamento, 162-164

gerenciamento de espaço de swap, 411
interpretador de comandos, 40
memória virtual, 315-316
sincronismo, 204-205
solução de Peterson, 181-182
soma de verificação, 552
somas de verificação (RAID), 419
sondagens automáticas, 638
sondas, 67
spinlock, , 185
spoofing, , 522
spool, , 454
spooling, , 454, , 716-717
spyware, , 495
SRM, *See also monitor de referência de segurança*
SSDs, *See also discos em estado sólido*
SSL 3.0, , 511-512
SSTF, algoritmo de escalonamento, *See also tempo de busca mais curto primeiro (SSTF), algoritmo de escalonamento*
stall de memória, , 160
Stallman, Richard, , 30
starvation, *See also bloqueio indefinido*
status de interrupção, , 129-130, , 201
STM (Software Transactional Memory), , 208
streaming, , 615-616, , 623-624
STREAMS, mecanismo, , 459-460
string, referência, , 290
string de referência, , 290
stripe set, , 694-695
striping:
dados, 413
disco, 695
em nível de bit, 413
em nível de bloco, 413
stubs, , 104-106, , 251
subsistemas:
ambientais, , 669
E/S, 23
kernels em, 5, 452-457
procedimentos supervisionados por, 457
proteção (Windows XP), 670
substituição de local, , 299-300
substituição de página, , 288. *See also alocação de quadro*
algoritmos de buffer de página, 297
desempenho da aplicação, 297
global, 299-300 *See also local*,
mecanismo básico, 288-291

substituição de página baseada em contagem, 296-297
substituição de página FIFO, 291-292
substituição de página ideal, 292-293
substituição de página LRU, 293-294
substituição de página por aproximação LRU, 294-296

substituição global, , 299-300

sumário de volume, , 339

Sun Cluster, , 550

SunOS, , 31

superbloco, , 363, , 394

superfície de ataque, , 500

suporte:

à linguagem de programação, 50
ao idioma nacional (NLS), API, 670

swapper (termo), , 282

swapping, , 17, , 85, , 252-254, , 282

Linux, 649
paginação, 411

SYSGEN, *See also* **geração do sistema**

T

tabela de alocação de arquivo (FAT), , 372

tabela de arquivo, , 396

tabela de arquivo-mestre, , 363

tabela de arquivos abertos, 331-332
tabela de arquivos abertos no sistema, 363
tabela de arquivos abertos por processo, 364
tabela de descritor de arquivo, 396
tabela de despacho de interrupção (Windows XP), 674

tabela de montagem, 457
tabela de objetos, 676

tabela de roteamento, , 542

tabelas, , 279

alocação de arquivos, 372
arquivo-mestre, 363
arquivos abertos no sistema, 363
arquivos abertos por processo, 364
arquivos abertos, 331
hash, 368
montagem, 365, 457
objeto, 676
página, 284, 678-679
roteamento, 542
segmento, 270

tabelas de hash, , 368

tabelas de página, , 256-258, , 284, , 678-679

em clusters, 237

em hash, 266
hardware para armazenar, 260-262
invertidas, 267-268, 312
mapeadas para a frente, 265

tabelas de página em clusters, , 237

tabelas de página em hash, , 237

tabelas de página invertidas, , 267-268, , 312

tabelas de página mapeadas para a frente, , 265

tabelas de segmento, , 270

tags, , 478

tamanho de página, , 310-311

tarefas:

- Linux, 641-642
- VxWorks, 610

tarefas expiradas (Linux), , 642

taxa de acerto, , 262, , 311

taxa de falta de página, , 286

taxa de transferência (discos), , 399, , 400

TCB (Trusted Computer Base), , 523

TCP, sockets, , 100

TCP/IP, *See also Transmission Control Protocol/Internet Protocol*

TDI (Transport Driver Interface), , 697

técnica de coordenador múltiplo (controle de concorrência), , 582

técnica de leitura antecipada, , 381

técnica de preenchimento de zero por demanda, , 288

técnica de segurança pela obscuridade, , 517

técnica em camadas (estrutura do sistema operacional), , 54-55

técnica free-behind, , 380-381

técnicas de codificação rígidas, , 95

telnet, , 533

tempo:

- acesso efetivo à memória, 262
- acesso efetivo, 285-286
- carga, 250
- compilação, 250
- criação/uso de arquivo, 330
- espera, 148
- execução, 250
- resposta, 16, 148-149
- turnaround, 148

tempo compartilhado (multitarefa), , 16-17

tempo de acesso (discos), , 399

tempo de acesso efetivo, , 285-286

tempo de busca (discos), , 399, , 404

tempo de busca mais curto primeiro (SSTF), algoritmo de escalonamento, , 404-405

tempo de carga, , 250

tempo de compilação, , 250

tempo de desenvolvimento de sistema, , **58**

tempo de espera, , **148**

tempo de execução, , **250**

tempo de paralisação, , **370**

tempo de posicionamento (discos), **399**

tempo de resposta, , **16**, , **148-149**

tempo de turnaround, , **148**

tempo efetivo de acesso à memória, , **262**

tempo médio para falha, , **412**

tempo médio para perda de dados, , **413**

tempo médio para reparo, , **413**

temporizador variável, , **19**

teorema de Bayes, , **519**

término:

casca, **91**

processo, **90-91**, **237-238**

teste de penetração, **516**

THE, sistema operacional, , **719**

thrashing, , **300-304**

causa, **300-301**

definição, **300**

estratégia de frequência de falta de página, **303-304**

modelo de conjunto de trabalho, **302-303**

thread de destino, , **129**

thread primária, **703**

threads do kernel, , **120**

threads do usuário, , **120**

threads ociosas, , **164**

threads. *See also multithreading*

cancelamento de thread, **129**

componentes, **117**

destino, **129**

escalonamento, **157**

kernel, **120**

Linux, **135-136**, **641-642**

modelo de processo, **82**

ociosas, **164**

pools de threads, **131-132**

usos, **117-118**

usuário, **120**

Windows XP, **134-135**, **671-672**, **703-706**

throughput, , **148**, , **619**

thunking, , **690**

time-out, esquemas, , **548**, , **592**

timer, , **450**

intervalo programável, **450**

objetos, **672**

variável, 19

tipo de dados abstrato, , 330

tipos de objetos, 367, 676

TLB, alcance, , 311

TLB, perda, , 261

TLB, *See also Translation Look-aside Buffer*

tokens, , 544, , 578

tokens de acesso de segurança (Windows XP), , 524

tolerância a falhas, , 12, , 549-550, , 693-696

topologia de rede, , 539-540

TOPS-20, , 721-722

Torvalds, Linus, , 30, , 631

tradução de nome de caminho, , 387-388

transações, , 207. *See also transações atômicas*

definição, 655

Linux, 655-656

sistemas de arquivo estruturados em log, 381-382

transações abortadas, , 207

transações atômicas, , 183, , 207-213

concorrentes, 210-211

protocolos baseados em estampa de tempo, 212-213

protocolos de lock, 211

serialização, 210-211

logging write-ahead, 208

modelo de sistema, 207-208

pontos de verificação, 209-210

transações confirmadas, , 207

transações de memória, , 208

transações revertidas, , 207

Transarc DFS, , 569-570

transferência de arquivo, , 533-534

transferência de arquivo remoto, , 533-534

transferência de dados controlada por interrupção, , 307

transformações de caixa preta, , 506

Translation Look-aside Buffer (TLB), 261, 679-680

Transmission Control Protocol/Internet Protocol (TCP/IP), 698

transparência, 550, 560, 560

trap doors, 495

traps, 17, 283-284, 445

traps de falta de página, 283-284

tratador:

upcall, 134

de sinal, 130-131

de sinais definidos pelo usuário, 130

de sinais padrão, 130

tratamento de falha (protocolo 2PC), 580-581

trilhas de disco, 399

triple DES, 506
Tripwire, sistema de arquivos, 520

troca:

contexto, 85, , 461
domínio, 472
mensagens, 543
pacotes, 543
turnstile, 205

U

Ubuntu Linux, , 31

UDP, sockets, , 100

UDP (User Datagram Protocol), , 546

UFD (User File Directory), , 341

UFS (UNIX File System), , 362

UI, *See also interface com o usuário*

UID efetivo, , 24

UMA (Uniform Memory Access), , 13

UNC (Uniform Naming Convention), , 699

unicasting, , 622

UNICODE, , 670

unidade:

central de processamento, *See sob CPU*
compactação, 696-697
componentes, 560
execução de instrução, 689
gerenciamento de memória (MMU), 251, 678-679
lógicas, 402

UNIX, sistema de arquivos (UFS), , 362

UNIX, sistema operacional:

criação de processos, 87-88
Linux, 631
permissões, 356
semântica de consistência, 351-352
sinais, 130-131
swapping, 254
troca de domínio, 472-473

UnixBSD, , 31

upcalls, , 134-134

USBs, *See also barramentos seriais universais*

uso (modo do sistema), , 223

uso internacional, , 670

uso multinacional, , 670

usuários, , 3-4

usuários (compartilhamento de arquivos), , 348-349

utilitários do sistema, , 50, , 635-636

utilização, , 713

utilização de recursos, , 4

V

VACB, *See also bloco de controle de endereço virtual*

VADs (Virtual Address Descriptors), 682

valor de hash (síntese de mensagem), 509

valor de tempo real (Linux), 166

valor nice (Linux), 166, 642

variáveis automáticas, 496

variável de condição, 193

varredura de porta, 503

varreduras de vulnerabilidade, , 516-517

VDM, *See also Virtual DOS Machine*

velocidade de operações (dispositivos de E/S), , 448, , 449

velocidade relativa, , 181

verificação de consistência, , 381

Veritas Cluster, , 550

vermes, , 501-503

versão (modelo do sistema), , 223

vetor:

ambiente, 640

argumentos, 640

bits (mapa de bits), 375-376

interrupção, 7-8, 254, 443

vfork() (fork de memória virtual), , 288

VFS, *See also sistema de arquivo virtual*

Virtual DOS Machine (VDM), , 689

virtualização, , 161-162

vírus, , 498-500, , 519-521

arquivo, 499

blindados, 500

boot, 499

codificados, 500

código-fonte, 499

furtivos, 500

macro, 499

multipartes, 500

polimórficos, 499

tunelamento, 500

visões, 678

VMware, exemplos de, 6, 59

VMware Workstation, 59

vnode, 366

volumes, 339, 570

von Neumann, arquitetura, 8

VPNs, *See also redes privativas virtuais*

VxWorks, , 610-611

W

W-timestamp, , [212-213](#)

WAFL, sistema de arquivos, , [388-390](#)

wait-die, esquema, , [585](#)

WANs, *See also* [redes remotas](#)

watchdog (UNIX), , [526](#)

web clipping, , [27](#)

Web Distributed Authoring and Versioning (WebDAV), , [698](#)

Win32, biblioteca de threads, , [123](#)

Windows, , [723](#)

criação de processo, [88-89](#)

swapping no, [254](#)

Windows 2000, [668](#), , [670](#)

Windows NT, , [667-668](#)

Windows XP, , [667-708](#)

compatibilidade de aplicação, [669](#)

componentes do sistema, [670-688](#)

camada de abstração de hardware, [670](#)

executivo, *See* [Windows XP, executivo](#)

kernel, [670-674](#)

confiabilidade, [668](#)

desempenho, [669](#)

exemplo de comunicação entre processos, [98-100](#)

exemplo de escalonamento, [163-166](#)

extensibilidade, [669-670](#)

histórico, [667-668](#)

interface de programador, [702-708](#)

acesso a objeto do kernel, [703](#)

compartilhando objetos entre processos, [703](#)

comunicação entre processos, [706-706](#)

gerência de memória, [706-708](#)

gerência de processos, [703-706](#)

memória virtual, [315](#)

portabilidade, [670](#)

princípios de projeto, [668-670](#)

redes, [697-702](#)

Active Directory, [702](#)

domínios, [701-702](#)

interfaces, [697](#)

mecanismos de processamento distribuído, [699-700](#)

protocolos, [697-698](#)

redirecionadores e servidores, [700-701](#)

resolução de nome, [702](#)

segurança, [668, 690-691](#)

sincronização, [205-206](#)

sistemas de arquivos, [691-697](#)

compactação e codificação, [696-697](#)

cópias de sombra de volume, 697
diário de mudança, 697
gerenciamento de volume e tolerância a falhas, 693–696
NTFS, árvore B+, 692
NTFS, layout interno, 691–692
NTFS, metadados, 692
pontos de montagem, 697
recuperação, 693
segurança, 693
subsistemas ambientais, 689–691
logon, 690–691
MS-DOS, 689
POSIX, 690
segurança, 690–691
Win32, 690–690
Windows de 16 bits, 689
Windows de 32 bits, 690
threads, exemplo, 134–135
versões de desktop, 668

Windows XP, executivo, , 674–688

booting, 688
gerenciador de cache, 685–686
gerenciador de E/S, 684–685
facilidade de chamada de procedimento local, 683–684
gerenciador de objetos, 674–677
plug-and-play e gerenciadores de energia, 686–688
gerenciador de processos, 682–683
registry, 688
monitor de referência de segurança, 686
gerenciador de memória virtual, 677–682

Winsock, , 699

Witness, , 231

word, , 5

World Wide Web, , 349

WORM, discos, *See also discos write-once, read-many-times*

WORM (write-once, read-many), formato, , 21

write-once, read-many-times (WORM), discos, , 422

X

XDR (eXternal Data Representation), , 103

XDS-940, sistema operacional, , 718

Xerox, , 40

XML, firewalls, , 522

Z

zona desmilitarizada (DMZ), 521

zonas, 61, 645