Tradução da Sexta Edição

# SILBERSCHATZ · GALVIN · GAGNE

# SISTEMAS OPERACIONAIS

## com JAVA


Tradução

**Daniel Vieira**

*Presidente da Multinet Informática*
*Programador e tradutor especializado em Informática*

Revisão Técnica

**Sergio Guedes de Sousa**

*Pesquisador – Núcleo de Computação Eletrônica*
*NCE – UFRJ*
*Professor Colaborador – Departamento de Ciência da Computação,*
*Instituto de Matemática – DCC/IM – UFRJ*





ELSEVIER

EDITORA CAMPUS

Do Original:
*Operating System Concepts with Java*
Tradução autorizada do idioma inglês da edição publicada por John Wiley & Sons
Copyright © 2004 by John Wiley & Sons, Inc.



*Editoração Eletrônica*
Estúdio Castellani

*Preparação de Originais*
Lígia Paixão

*Revisão Gráfica*
Marco Antônio Côrrea

*Projeto Gráfico*
Elsevier Editora
A Qualidade da Informação.
Rua Sete de Setembro, 111 – 16º andar
20050-006 Rio de Janeiro RJ Brasil
Telefone: (21) 3970-9300 FAX: (21) 2507-1991
E-mail: *info@elsevier.com.br*
*Escritório São Paulo:*
Rua Elvira Ferraz, 198
04552-040 Vila Olímpia São Paulo SP
Tel.: (11) 3841-8555

ISBN 85-352-1485-2
Edição original: ISBN 0-471-48905-0

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

S576s
Silberschatz, Abraham
Sistemas operacionais : conceitos e aplicações /
Abraham Silberschatz, Peter Baer Galvin, Greg Gagme ;
tradução de Daniel Vieira. – Rio de Janeiro : Elsevier, 2004

Tradução de: Operating systems concepts with Java, 6th ed
Inclui bibliografia
ISBN 85-352-1485-2

1. Sistemas operacionais (Computadores). 2. Java
(Linguagem de programação de computadores). I. Galvin,
Peter B. II. Gagne, Greg. III. Título.

04-2066. CDD – 005.43
CDU – 004.451

04 05 06 07 5 4 3 2 1

# Sumário

## PARTE UM
## VISÃO GERAL

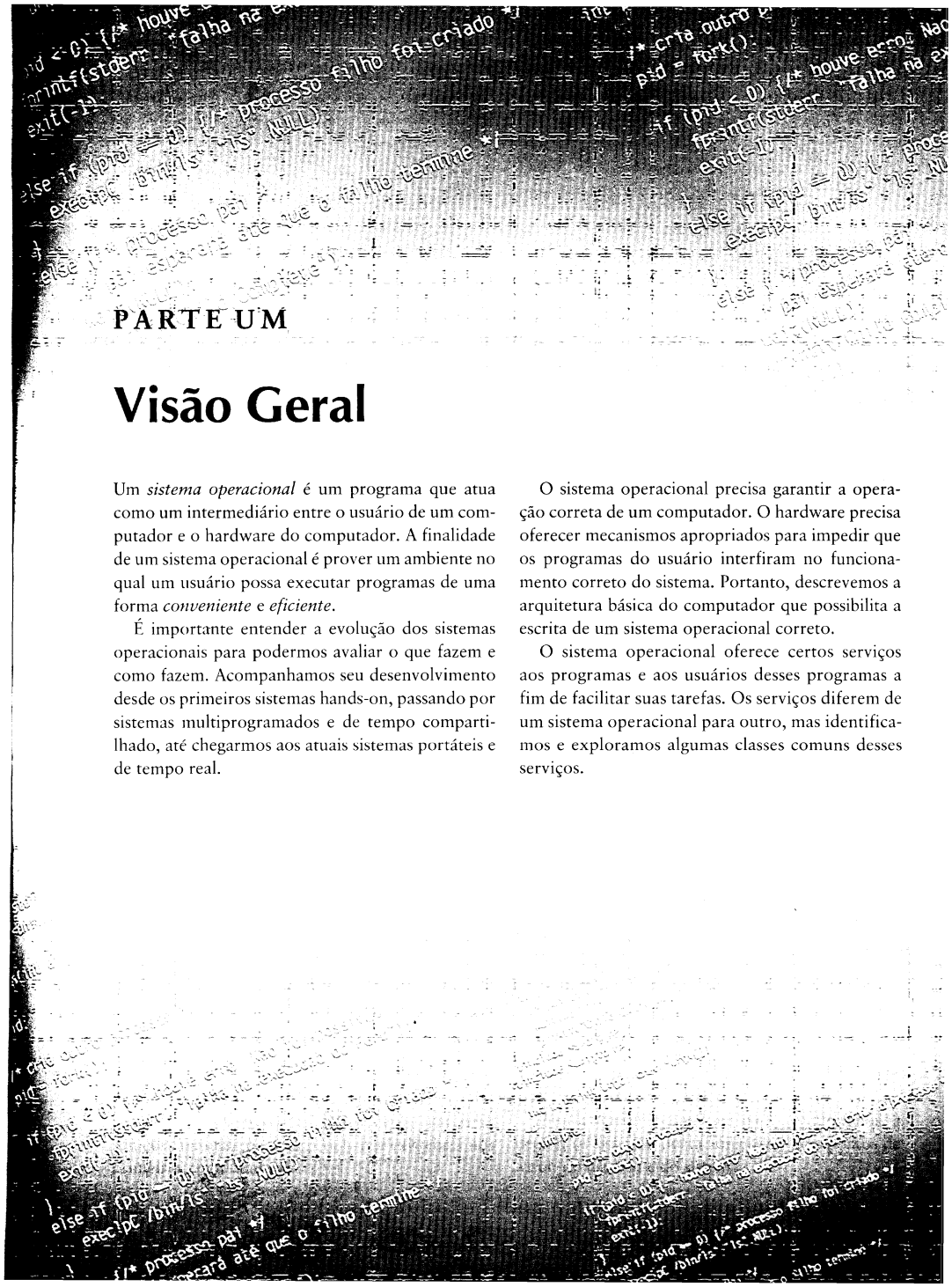
# PARTE TRÊS
# GERÊNCIA DE ARMAZENAMENTO

## PARTE QUATRO
## SISTEMAS DE E/S



## PARTE CINCO
## SISTEMAS DISTRIBUÍDOS

## PARTE SETE
## ESTUDOS DE CASO



PARTE UM

# Visão Geral

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

É importante entender a evolução dos sistemas operacionais para podermos avaliar o que fazem e como fazem. Acompanhamos seu desenvolvimento desde os primeiros sistemas hands-on, passando por sistemas multiprogramados e de tempo compartilhado, até chegarmos aos atuais sistemas portáteis e de tempo real.

O sistema operacional precisa garantir a operação correta de um computador. O hardware precisa oferecer mecanismos apropriados para impedir que os programas do usuário interfiram no funcionamento correto do sistema. Portanto, descrevemos a arquitetura básica do computador que possibilita a escrita de um sistema operacional correto.

O sistema operacional oferece certos serviços aos programas e aos usuários desses programas a fim de facilitar suas tarefas. Os serviços diferem de um sistema operacional para outro, mas identificamos e exploramos algumas classes comuns desses serviços.

CAPÍTULO 1

# Introdução

Um **sistema operacional** é um programa que gerencia o hardware do computador. Ele também oferece uma base para os programas aplicativos e atua como um 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 oferecer um ambiente em que um 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.

Para entender realmente o que são os sistemas operacionais, primeiro temos de entender como se desenvolveram. Neste capítulo, depois de oferecer uma descrição geral do que fazem os sistemas operacionais, acompanhamos seu desenvolvimento desde os primeiros sistemas hands-on até os multiprogramados e de tempo compartilhado para PCs e computadores portáteis. Também discutimos as variações de sistema operacional, como sistemas paralelos, de tempo real e embutidos. Enquanto prosseguimos pelos diversos estágios, vemos como os componentes dos sistemas operacionais evoluíram como soluções naturais para os problemas existentes nos primeiros computadores.

## 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).

O **hardware** – a **unidade central de processamento (CPU – Central Processing Unit)**, a **memória** e os **dispositivos de entrada/saída (E/S)** – oferece os recursos de computador 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 de computação 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 oferece 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 simplesmente oferece 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.

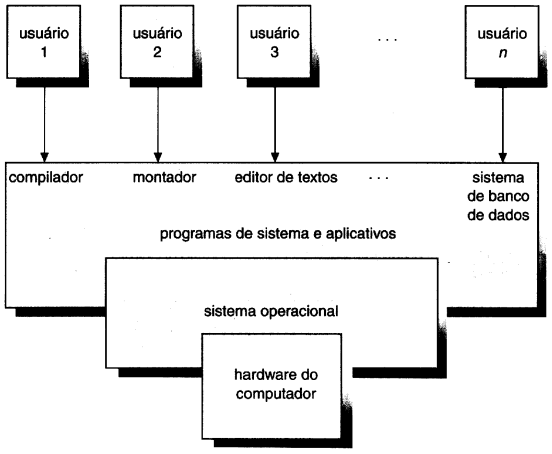



FIGURA 1.1 *Visão abstrata dos componentes de um computador.*

### 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 um monitor, teclado, mouse e unidade do sistema. Esse sistema foi projetado para um 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 as demandas colocadas sobre o sistema por um único usuário são muito pequenas para que a utilização de recursos se torne um problema. 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 disponíveis 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 a rede e servidores – 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. Esses dispositivos são unidades independentes, usadas isoladamente por usuários individuais. Alguns estão conectados a redes, seja diretamente por fio ou (com mais freqüência) por modems sem fio. Devido a limitações de potência 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 podem 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 – de hardware e de software – 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.

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 oferecem 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 certas 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. (Um kilobyte, ou KB, contém 1.024 bytes; um megabyte, ou MB, contém $1.024^2$ bytes; e um gigabyte, ou GB, contém $1.024^3$ 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 definição mais comum é que o sistema operacional é o único programa que executa o tempo todo no computador (normalmente chamado de **kernel**), com todos os outros sendo programas do sistema e programas aplicativos. Essa última definição é a que normalmente utilizamos.

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. Como resultado, a Microsoft foi considerada culpada por usar seu monopólio de sistema operacional para limitar a concorrência.

### 1.1.4 Objetivos do sistema

É mais fácil definir um sistema operacional pelo que ele *faz* do que pelo que ele *é*, mas até mesmo isso pode não ser muito fácil. O objetivo principal de alguns sistemas operacionais é a *conveniência para o usuário*. Eles existem porque a computação com eles é supostamente mais fácil do que a computação sem eles. Como vimos, essa visão é bem clara quando você examina os sistemas operacionais para PCs pequenos. O objetivo principal de outros sistemas

operacionais é a operação *eficiente* do sistema computadorizado. Esse é o caso para sistemas grandes, compartilhados, multiusuário. Esses sistemas são caros e, por isso, é desejável torná-los o mais eficiente possível. Às vezes, esses dois objetivos – conveniência e eficiência – são contraditórios. No passado, a eficiência normalmente era mais importante do que a conveniência (Seção 1.2.1). Assim, grande parte da teoria dos sistemas operacionais se concentra no uso ideal dos recursos de computação.

Os sistemas operacionais também evoluíram com o tempo afetando os objetivos do sistema. Por exemplo, o UNIX começou com um teclado e uma impressora como sua interface, limitando sua conveniência para os usuários. Com o passar do tempo, o hardware mudou, e o UNIX foi transportado para o novo hardware, com interfaces mais amigáveis para o usuário. Muitas **interfaces gráficas para usuário (GUIs)** foram acrescentadas, permitindo que o UNIX fosse mais conveniente de usar e ainda voltado para a eficiência.

O projeto de qualquer sistema operacional é uma tarefa complexa. Os projetistas encaram muitas opções, e muitas pessoas estão envolvidas não apenas em fazer o sistema operacional funcionar, mas também em revisá-lo e atualizá-lo constantemente. O fato de um determinado sistema operacional cumprir seus objetivos de projeto bem ou mal está aberto a debates e envolve julgamentos subjetivos por parte de diferentes usuários.

Os sistemas operacionais e a arquitetura do computador causaram grandes modificações um no outro. Para facilitar o uso do hardware, os pesquisadores desenvolveram sistemas operacionais. Os usuários dos sistemas operacionais propuseram, então, mudanças no projeto do hardware para simplificá-los. Nessa curta revisão histórica, observe como a identificação de problemas com o sistema operacional levou à introdução de novos recursos de hardware.

## 1.2 Sistemas de grande porte

Os **computadores de grande porte (mainframes)** foram os primeiros utilizados para trabalhar com muitas aplicações comerciais e científicas. Nesta seção, acompanhamos o crescimento dos sistemas de mainframe desde simples **sistemas em lote (ou batch)**, em que o computador executa uma e somente uma aplicação, até **sistemas de tempo compartilhado (time-shared systems)**, que permitem a interação do usuário com o computador.

### 1.2.1 Sistemas Batch (em lote)

Os primeiros computadores eram máquinas enormes fisicamente, executadas a partir de consoles. Os dispositivos de entrada comuns eram leitoras de cartões e unidades de fita. Já os de saída eram as impressoras de linhas, unidades de fita e perfuradoras de cartões. O usuário não interagia diretamente com o sistema computadorizado. Em vez disso, ele preparava uma tarefa (job) – que consistia em um programa, os dados e algumas informações de controle sobre a natureza da tarefa (cartões de controle) – e a submetia ao operador do computador. Em geral, a tarefa era realizada na forma de cartões perfurados. Algum tempo depois (após minutos, horas ou dias), a saída aparecia. A saída consistia no resultado do programa, bem como uma listagem do conteúdo final da memória e dos registradores, para fins de depuração.

O sistema operacional nesses primeiros computadores era bem simples. Sua tarefa principal era transferir automaticamente o controle de uma tarefa para a seguinte. O sistema operacional sempre estava residente na memória (Figura 1.2).

Para agilizar o processamento, os operadores agrupavam (**Batched**) tarefas com necessidades se-

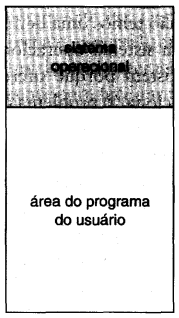



FIGURA 1.2 *Diagrama de memória para um sistema Batch (em lote) simples.*

melhantes e as executavam no computador como um grupo. Assim, os programadores deixavam seus programas com o operador. O operador classificaria os programas em lotes com requisitos semelhantes e, quando o computador estivesse disponível, executaria cada lote. A saída de cada lote seria enviada para o respectivo programador.

Nesse ambiente de execução, a CPU está sempre ociosa, pois trabalha muito mais rapidamente do que os dispositivos de E/S mecânicos. Até mesmo uma CPU lenta trabalha na faixa dos microssegundos, executando milhares de instruções por segundo. Uma leitora de cartões rápida, ao contrário, podia ler 1.200 cartões por minuto (ou 20 cartões por segundo). Assim, a diferença de velocidade entre a CPU e seus dispositivos de E/S pode ser de três ordens de grandeza ou mais. Com o tempo, é claro, as melhorias tecnológicas e a introdução dos discos resultaram em dispositivos de E/S muito mais rápidos. Porém, as velocidades de CPU aumentaram ainda mais, de modo que o problema não apenas não foi resolvido, mas também foi exacerbado.

A introdução da tecnologia de disco permitiu que o sistema operacional mantivesse todas as tarefas em um disco, em vez de uma leitora de cartões serial. Com o acesso direto a várias tarefas, o sistema operacional poderia realizar o **escalonamento de tarefas**, a fim de utilizar os recursos e realizar as tarefas de formas mais eficientes. Aqui, discutiremos alguns aspectos importantes do escalonamento de tarefas e de CPU; o Capítulo 6 contém uma explicação mais detalhada.

### 1.2.2 Sistemas multiprogramados

O aspecto mais importante do escalonamento de tarefas é a capacidade de multiprogramar. Geralmente, um único usuário não pode manter a CPU ou os dispositivos de E/S ocupados o tempo todo. A **multiprogramação** aumenta a utilização de CPU, organizando as tarefas de modo que a CPU sempre tenha uma para executar.

Vamos explicar melhor essa idéia. O sistema operacional mantém várias tarefas na memória simultaneamente (Figura 1.3). Esse conjunto de tarefas é um subconjunto das tarefas mantidas no banco de tarefas – que contém todas as tarefas que entram no sistema –, pois o número de tarefas que podem ser

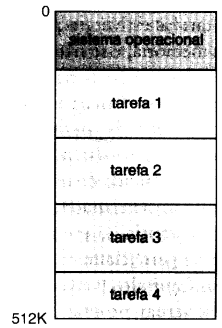



FIGURA 1.3 *Diagrama de memória para um sistema de multiprogramação.*

mantidas simultaneamente na memória costuma ser muito menor do que a quantidade de tarefas que podem ser 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 simplesmente 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 idéia 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, nunca ficará ocioso por falta de trabalho. (Os advogados ociosos costumam se tornar políticos, de modo que existe um certo valor social em mantê-los ocupados.)

A multiprogramação representa o primeiro caso em que o sistema operacional precisou tomar decisões para os usuários. Os sistemas operacionais multiprogramados, portanto, são bastante sofisticados. Como já mencionado, todas as tarefas que entram no sistema são mantidas no banco de tarefas. Esse banco consiste em todos os processos que residem no disco e aguardam a alocação de memória princi-

pal. Se várias tarefas estiverem prontas para serem levadas para a memória, e se não houver espaço suficiente para todas, então o sistema precisa fazer uma escolha entre elas. Tomar essa decisão é denominado *escalonamento de tarefa*, que discutiremos no Capítulo 6. Quando o sistema operacional seleciona uma tarefa do banco de tarefas, ele a carrega na memória para ser executada. Vários programas carregados na memória ao mesmo tempo exigem alguma forma de gerenciamento de memória, que explicamos nos Capítulos 9 e 10. Além disso, se várias tarefas estiverem prontas para execução ao mesmo tempo, o sistema precisa escolher entre elas. Essa tomada de decisão é chamada de *escalonamento de CPU*, que também é explicado no Capítulo 6. Finalmente, a execução de várias tarefas simultaneamente exige que sua capacidade de afetar umas às outras seja limitada em todas as fases do sistema operacional, incluindo o escalonamento de processos, armazenamento em disco e gerência de memória. Essas considerações serão discutidas no decorrer do texto.

### 1.2.3 Sistemas de tempo compartilhado

Os sistemas batch multiprogramados ofereciam um ambiente em que os diversos recursos do sistema (por exemplo, CPU, memória, dispositivos periféricos) eram utilizados com eficiência, mas não providenciavam interação do usuário com o sistema. **Tempo compartilhado (Time sharing)**, ou **multitarefa**, 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 freqüência que os usuários podem interagir com cada programa enquanto ele está sendo executado.

O Time sharing 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. De acordo com isso, o **tempo de resposta (response time)** 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 curta, somente um pouco de tempo de CPU é necessário para cada usuário. Enquanto 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 oferecer a cada usuário uma pequena parte de um computador com o 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.

Os sistemas operacionais de tempo compartilhado são ainda mais complexos dos que os sistemas operacionais multiprogramados. Em ambos, várias tarefas precisam ser mantidas simultaneamente na memória, de modo que o sistema precisa ter gerenciamento e proteção de memória (Capítulo 9). Para obter um tempo de resposta razoável, o sistema pode ter de trocar tarefas entre a memória principal e o disco, que agora serve como um local de armazenamento de apoio para a memória principal. Um método comum para conseguir esse objetivo é a **memória virtual**, uma técnica que permite a execução de uma tarefa que não está completamente na memória (Capítulo 10). 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 seqüência de armazena-

mento 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 oferecem um sistema de arquivos (Capítulos 11 e 12). O sistema de arquivos reside em uma coleção de discos; logo, é preciso que haja gerenciamento de discos (Capítulo 14). Os sistemas de tempo compartilhado também oferecem um mecanismo para execução simultânea, exigindo sofisticados esquemas de escalonamento de CPU (Capítulo 6). Para assegurar a execução em ordem, o sistema também oferece mecanismos para sincronismo de tarefas e comunicação entre elas (Capítulo 7) e pode garantir que as tarefas nunca ficarão atadas por um deadlock, aguardando umas pelas outras eternamente (Capítulo 8).

A idéia de compartilhamento de tempo já tinha sido demonstrada em 1960; mas, como os sistemas de tempo compartilhado são difíceis e caros de construir, eles não se tornaram comuns antes do início da década de 1970. Embora ainda exista algum processamento batch, a maioria dos sistemas de hoje, incluindo computadores pessoais, utiliza o compartilhamento de tempo. Conseqüentemente, a multiprogramação e o compartilhamento de tempo são os temas centrais dos sistemas operacionais modernos e, portanto, também são os temas centrais deste livro.

## 1.3 Sistemas desktop

Os computadores pessoais (PCs) apareceram na década de 1970. Os sistemas operacionais para PC não eram multiusuário nem multitarefa. Nos dias atuais, as CPUs incluídas nos PCs normalmente oferecem recursos para proteger um sistema operacional contra os programas do usuário. Porém, diferente dos sistemas multiprogramados e de tempo compartilhado, cujos objetivos principais incluem a maximização da utilização de CPU e periféricos, os sistemas operacionais para PC também precisam maximizar a conveniência e a reação às ações do usuário. Esses sistemas incluem PCs executando o Microsoft Windows e o Apple Macintosh. O sistema operacional MS-DOS da Microsoft foi substituído por diversas variações do Microsoft Windows. O sistema operacional Apple Macintosh foi portado para um hardware mais avançado e agora inclui recursos como memória virtual e multitarefa. Com o lançamento do Mac OS X, o kernel do sistema operacional agora é baseado no Mach e FreeBSD UNIX para obter maior escalabilidade, desempenho e recursos, porém, mantendo a mesma GUI rica. O Linux, um sistema operacional UNIX de código-fonte aberto (open-source), disponível para PCs, possui todos esses recursos e também se tornou muito popular.

Os sistemas operacionais para esses computadores se beneficiaram de várias formas com o desenvolvimento dos sistemas operacionais para mainframes. Naturalmente, algumas das decisões de projeto feitas para sistemas operacionais de mainframe não são apropriadas para sistemas menores. Como os custos de hardware para os microcomputadores são relativamente baixos, em geral, um indivíduo possui o uso exclusivo de um microcomputador. Assim, enquanto a utilização eficiente da CPU é uma preocupação importante para os mainframes, isso não é mais tão importante para os sistemas que dão suporte a um único usuário por vez. Porém, outras decisões de projeto ainda se aplicam. Por exemplo, a proteção dos arquivos, a princípio, não era necessária em máquinas pessoais. Contudo, esses computadores agora costumam estar conectados a outros computadores por meio de redes locais ou por conexões com a Internet. Quando outros computadores e outros usuários podem acessar os arquivos em um PC, a proteção dos arquivos novamente se torna um recurso necessário do sistema operacional. Na realidade, a falta dessa proteção facilitou a destruição de dados em sistemas indefesos, como o MS-DOS, por parte de programas maliciosos. Esses programas podem estar se replicando automaticamente e podem se espalhar com rapidez por mecanismos de worm ou **vírus**, quebrando empresas inteiras ou até mesmo redes de alcance mundial. Recursos avançados de compartilhamento de tempo, como memória protegida e permissões de arquivo, não são suficientes, por si sós, para proteger um sistema contra ataques. Brechas de segurança recentes têm mostrado isso de tempos em tempos. Esses tópicos são discutidos nos Capítulos 18 e 19.

## 1.4 Sistemas multiprocessados

A maioria dos sistemas atuais é de processador único; ou seja, eles só possuem uma CPU principal. Entretanto, os **sistemas multiprocessados (Multiprocessor systems)**, também conhecidos como **sistemas paralelos** 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, o relógio e, às vezes, 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 certo custo adicional é contraído para fazer com que todas as partes funcionem 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. Os servidores blade são um desenvolvimento recente em que várias placas de processador, placas de E/S e placas de rede podem ser colocadas no mesmo gabinete. A diferença entre eles e os sistemas multiprocessados tradicionais é que cada placa de processador blade é inicializada independentemente e executa seu próprio sistema operacional.
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. Essa capacidade de continuar oferecendo serviço proporcional em nível do hardware sobrevivente é chamada de **degradação controlada.** Os sistemas designados para essa degradação também são considerados **tolerantes a falhas (fault tolerant).**

Observe que a terceira vantagem, a operação contínua na presença de falhas, requer um mecanismo para permitir que a falha seja detectada, diagnosticada e, se possível, corrigida. O sistema da Tandem utiliza duplicação de hardware e software para garantir a operação contínua apesar das falhas. O sistema consiste em dois processadores idênticos, cada qual com sua própria memória local. Os processadores estão conectados por um barramento comum. Um processador é o principal, e o outro é o de reserva. Duas cópias são mantidas para cada processo: uma no processador principal e a outra no reserva. Em pontos de verificação fixos durante a execução do sistema, as informações de estado de cada tarefa – incluindo uma cópia da imagem da memória – são copiadas da máquina principal para a reserva. Se for detectada uma falha, a cópia de reserva é ativada e reiniciada a partir do ponto de verificação mais recente. Essa solução é cara, pois envolve uma considerável duplicação de hardware.

Dois tipos de sistemas multiprocessados estão em uso atualmente. O mais comum usa o **multiprocessamento simétrico (SMP)**, no qual cada processador executa uma cópia do sistema operacional e essas cópias se comunicam entre si de acordo com a necessidade. 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.

SMP significa que todos os processadores são iguais; não existe um relacionamento mestre-escravo entre os processadores. Cada um deles, como mencionado, executa simultaneamente uma cópia do sistema operacional. A Figura 1.4 ilustra uma arquitetura SMP típica. 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 cópias do UNIX. O benefício desse modelo é que muitos processos podem ser executados simultaneamente – $N$ processos podem estar em execução se houver $N$ CPUs –, sem causar uma 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 certas 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 7. Quase todos os sistemas operacionais modernos – incluindo Windows 2000, Windows XP, Mac OS X e Linux – agora oferecem suporte para SMP.

A diferença entre o multiprocessamento simétrico e 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 Sun-OS versão 4 da Sun oferece multiprocessamento assimétrico, enquanto a Versão 5 (Solaris) é simétrica no mesmo hardware.

À medida que os microprocessadores se tornam mais caros e mais poderosos, algumas funções adicionais do sistema operacional são deixadas para processadores escravos (ou **back-ends**). Por exemplo, é muito fácil acrescentar um microprocessador com sua própria memória para gerenciar um sistema de disco. O microprocessador poderia receber uma seqüência de requisições da CPU principal e implementar sua própria fila de disco e algoritmo de escalonamento. Esse arranjo retira da CPU principal o trabalho adicional de escalonamento de disco. Os PCs contêm um microprocessador no teclado, para converter as teclas digitadas em códigos para serem enviados à CPU. Na verdade, esse uso dos microprocessadores tornou-se tão comum que isso não é mais considerado multiprocessamento.

## 1.5 Sistemas distribuídos

O crescimento das redes de computador – especialmente a Internet e a World Wide Web (WWW) – teve uma profunda influência no desenvolvimento recente dos sistemas operacionais. Quando os PCs foram introduzidos na década de 1970, eles foram projetados para uso "pessoal" e, em geral, eram considerados computadores isolados. Com o início do uso público generalizado da Internet, na década de 1980, para correio eletrônico, ftp e gopher, muitos PCs foram conectados a redes de computadores. Com a introdução da Web em meados da década de 1990, a conectividade da rede tornou-se um comportamento essencial de um sistema computadorizado.

Praticamente todos os PCs e estações de trabalho modernas são capazes de executar um navegador Web para acessar documentos **hipertexto** na Web. Os sistemas operacionais (como Windows, Mac OS X e UNIX) agora também incluem o software do sistema (como TCP/IP e PPP) que permite a um com-

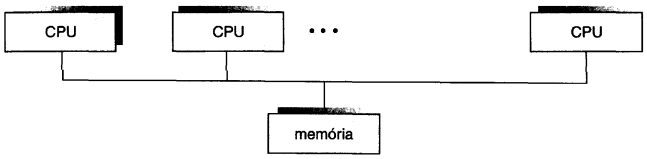



FIGURA 1.4 *Arquitetura de multiprocessamento simétrico.*

putador acessar a Internet por meio de uma rede local ou de uma conexão por telefone. Vários deles incluem o próprio navegador Web, além de clientes e servidores de correio eletrônico, login remoto e transferência de arquivos.

Ao contrário dos sistemas fortemente acoplados, discutidos na Seção 1.4, as redes de computadores usadas nessas aplicações consistem em uma coleção de processadores que não compartilham memória ou um relógio. Em vez disso, cada processador possui sua própria memória local. Os processadores se comunicam entre si por meio de diversas linhas de comunicação, como barramentos de alta velocidade ou linhas telefônicas. Esses sistemas normalmente são chamados de **sistemas distribuídos** ou **sistemas fracamente acoplados** (loosely coupled systems).

Por serem capazes de se comunicar, os sistemas distribuídos podem compartilhar tarefas de computação e oferecer um rico conjunto de recursos aos usuários. Nesta seção, examinamos as redes de computadores e como elas podem ser estruturadas para formar sistemas distribuídos. Também exploramos as estratégias cliente-servidor e peer-to-peer para oferecer serviços em um sistema distribuído. Finalmente, examinamos como os sistemas distribuídos afetaram o projeto dos sistemas operacionais.

### 1.5.1 Redes de computadores

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. O TCP/IP é o protocolo de rede mais comum, embora o ATM e outros protocolos sejam bastante utilizados. Da mesma forma, o suporte aos protocolos pelo sistema operacional também varia. A maioria dos sistemas operacionais oferece suporte ao TCP/IP, incluindo os sistemas Windows e UNIX. Alguns sistemas oferecem suporte a protocolos próprios, de acordo com suas necessidades. Para um sistema operacional, um protocolo de rede precisa simplesmente de um dispositivo de interface – um adaptador de rede, por exemplo – com um device driver 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 (LAN – Local Area Network)** conecta computadores dentro de uma sala, um pavimento ou um prédio. Uma **rede de longa distância (WAN – Wide Area Network)** 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 (MAN – Metropolitan Area Network)** poderia conectar prédios dentro de uma cidade. Dispositivos BlueTooth e 802.11b utilizam a tecnologia sem fio para a comunicação por uma distância de vários metros, criando uma **rede doméstica**, como uma 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 e rádio. Quando dispositivos de computação são conectados a telefones celulares, eles criam uma rede. Até mesmo a comunicação de infravermelho de curta duração pode ser usada para as redes. Em um nível rudimentar, sempre que os computadores se comunicam, eles utilizam ou criam uma rede. Essas redes também variam em seu desempenho e confiabilidade.

### 1.5.2 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. 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 1.5.

Os sistemas servidores podem ser categorizados de modo geral como servidores de computação e servidores de arquivos:

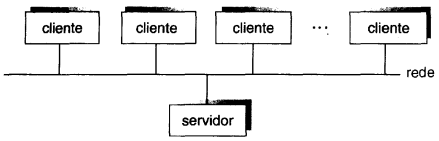



FIGURA 1.5 *Estrutura geral de um sistema cliente-servidor.*

- **O sistema servidor de processamento (compute-server)** oferece uma interface para a qual um cliente pode enviar requisições para realizar uma ação; 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)** oferece 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 oferece arquivos aos clientes que utilizam navegadores Web.

### 1.5.3 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 se ele está solicitando ou fornecendo um serviço. Os sistemas peer-to-peer oferecem 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 (e solicitar serviços de) outros nós na rede. 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 oferece 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 oferece o serviço desejado, enviando um broadcast de requisição de serviço a todos os outros nós na rede. O nó (ou nós) que está oferecendo 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 acima; 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 de música continua incerto, devido a 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.

### 1.5.4 Sistemas operacionais distribuídos

Alguns sistemas operacionais têm levado 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 oferece recursos por meio da rede, como compartilhamento de arquivo, e que inclui um esquema de comunicação para permitir que diferentes processos em diversos computadores troquem mensagens. Um computador que esteja executando um sistema operacional

de rede atua de forma autônoma de todos os outros computadores na rede, embora esteja ciente da rede e seja capaz de se comunicar com outros computadores em rede. Um sistema operacional distribuído oferece um ambiente menos autônomo: os diferentes sistemas operacionais se comunicam o suficiente para oferecer a ilusão de que um único sistema operacional controla a rede. Explicamos redes de computadores e sistemas distribuídos nos Capítulos de 15 a 17.

## 1.6 Sistemas em clusters

Outro desenvolvimento em sistemas operacionais envolve os sistemas em clusters. Como os sistemas paralelos, os **sistemas em clusters** reúnem diversas CPUs para realizar trabalho de computação. Entretanto, os sistemas em clusters diferem dos sistemas paralelos porque são compostos de dois ou mais sistemas individuais e 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.

O agrupamento em clusters costuma ser usado para oferecer um serviço de **alta disponibilidade**, ou seja, um serviço que continuará a ser fornecido 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**, dois ou mais hosts estão executando aplicações, e eles estão monitorando um ao outro. Esse modo, obviamente, é mais eficiente, pois utiliza todo o hardware disponível. Ele exige que mais de uma aplicação esteja disponível para ser executada.

Outras formas de clusters incluem clusters paralelos e uso de clusters por uma WAN. Os clusters 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 Parallel Server 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, para assegurar que não haverá operações em conflito. Essa função, normalmente conhecida como **gerenciador de lock distribuído (DLM – Distributed Lock Manager)**, está incluída em algumas tecnologias de cluster.

A tecnologia de cluster está mudando rapidamente. Algumas tendências futuras incluem clusters de dezenas de nós e clusters globais, em que as máquinas poderiam estar em qualquer lugar do mundo (ou em qualquer lugar que uma WAN alcance). Esses projetos ainda são assunto de pesquisa e desenvolvimento. Muitas dessas melhorias se tornaram possíveis por meio de **Sistemas de Armazenamento em Rede**, conhecidos como SANs (Storage Area Networks), conforme descreveremos na Seção "Rede de área de armazenamento", no Capítulo 14, permitindo que muitos sistemas se conectem a um conjunto de armazenamento. Os SANs permitem a inclusão fácil de vários computadores a diversas unidades de armazenamento. Ao contrário, os clusters atuais normalmente são limitados a dois ou quatro computadores, devido à complexidade de conectá-los ao armazenamento compartilhado.

## 1.7 Sistemas de tempo real

Outra forma de sistema operacional de uso especial é o **sistema 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 então 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, que pode não ter qualquer restrição de tempo.

Sistemas de tempo real podem ser de dois tipos: rígido e flexível. Um **sistema de tempo real rígido** garante que as tarefas críticas serão concluídas a tempo. Esse objetivo requer que todos os atrasos no sistema sejam evitados, desde a recuperação dos dados armazenados até o tempo gasto pelo sistema operacional para realizar qualquer requisição feita. Essas restrições de tempo ditam as facilidades disponíveis nos sistemas de tempo real rígidos. O armazenamento secundário de qualquer tipo normalmente é limitado ou ausente, com os dados sendo armazenados na memória de curta duração ou na memória somente de leitura (ROM). A ROM está localizada em dispositivos de armazenamento não voláteis, que retêm seu conteúdo mesmo no caso de falta de energia; a maioria dos outros tipos de memória é volátil. Recursos mais avançados do sistema operacional também estão ausentes, pois costumam separar o usuário do hardware, e essa separação resulta na incerteza sobre o tempo que uma operação levará. Por exemplo, a memória virtual (Capítulo 10) quase nunca é encontrada em sistemas de tempo real. Portanto, os sistemas de tempo real rígidos entram em conflito com a operação dos sistemas de tempo compartilhado, e os dois não se misturam. Como nenhum dos sistemas operacionais de uso geral existentes admite a funcionalidade de tempo real rígido, não nos preocuparemos com esse tipo de sistema neste texto.

Um tipo de sistema de tempo real menos restritivo é um **sistema de tempo real flexível**, no qual uma tarefa crítica de tempo real tem prioridade sobre outras tarefas e retém essa prioridade até que sua execução seja completada. Assim como nos sistemas de tempo real rígidos, os atrasos no kernel do sistema operacional precisam ser evitados: uma tarefa de tempo real não pode ser mantida aguardando indefinidamente.

O tempo real flexível é um objetivo que pode ser alcançado, e que pode ser combinado com outros tipos de sistemas. No entanto, os sistemas de tempo real flexível possuem utilidade mais limitada do que os sistemas de tempo real rígido. Dada sua falta de suporte para deadlock, eles são arriscados de usar em controle industrial e robótica. Contudo, são úteis em diversas áreas, incluindo multimídia, realidade virtual e projetos científicos avançados – como exploração submarina e aventuras planetárias. Esses sistemas precisam de recursos avançados do sistema operacional, que não podem ser admitidos por sistemas de tempo real rígido. Devido aos usos expandidos para a funcionalidade do tempo real flexível, ele está encontrando seu caminho nos sistemas operacionais mais atuais, incluindo as principais versões do UNIX.

No Capítulo 6, consideraremos a facilidade de escalonamento necessária para implementar a funcionalidade de tempo real flexível em um sistema operacional. No Capítulo 10, descreveremos o projeto de gerência de memória para a computação em tempo real. Finalmente, no Capítulo 21, descreveremos os componentes de tempo real do sistema operacional Windows 2000.

## 1.8 Sistemas portáteis

**Sistemas portáteis** incluem **assistentes digitais pessoais (PDAs – Personal Digital Assistants)**, como

*Palm*, Pocket-PCs ou telefones celulares. Os desenvolvedores de sistemas portáteis e suas aplicações encaram muitos desafios, sendo a maioria relacionada ao tamanho limitado desses dispositivos. Por exemplo, um PDA tem, em geral, cerca de 12,5cm de altura e 7,5cm de largura, e pesa em torno de 200g. Devido ao seu tamanho, a maioria dos dispositivos portáteis possui uma pequena quantidade de memória, processadores lentos e telas de vídeo pequenas. Vejamos cada uma dessas limitações.

Muitos dispositivos portáteis possuem entre 512 KB e 128MB 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 10, 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 freqüência. Para reduzir o tamanho da maioria dos dispositivos portáteis, usam-se processadores menores e mais lentos, que também 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 são as pequenas telas que normalmente estão disponíveis. Enquanto um monitor para um computador doméstico pode medir até 23 polegadas, a tela de um dispositivo portátil não costuma ter mais do que 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 (consulte a Seção "Sistemas distribuídos", anteriormente neste capítulo), 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 oferecem 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 expandir à medida que as conexões da rede se tornam mais disponíveis e outras opções, como câmeras digitais e players MP3, expandem sua utilidade.

## 1.9 Migração de recursos

Em geral, um exame dos sistemas operacionais para mainframes e microcomputadores mostra que os recursos uma vez disponíveis apenas em mainframes foram adotados para microcomputadores. Os mesmos conceitos de sistema operacional são apropriados para as diversas classes de computadores: mainframes, minicomputadores, microcomputadores e portáteis. Muitos dos conceitos e recursos representados na Figura 1.6 serão abordados mais adiante neste livro. Contudo, para começar a entender os sistemas operacionais modernos, você precisa reconhecer o tema da migração de recursos e a história longa de muitos recursos do sistema operacional.

Um bom exemplo desse movimento ocorreu 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 funcionava em um mainframe

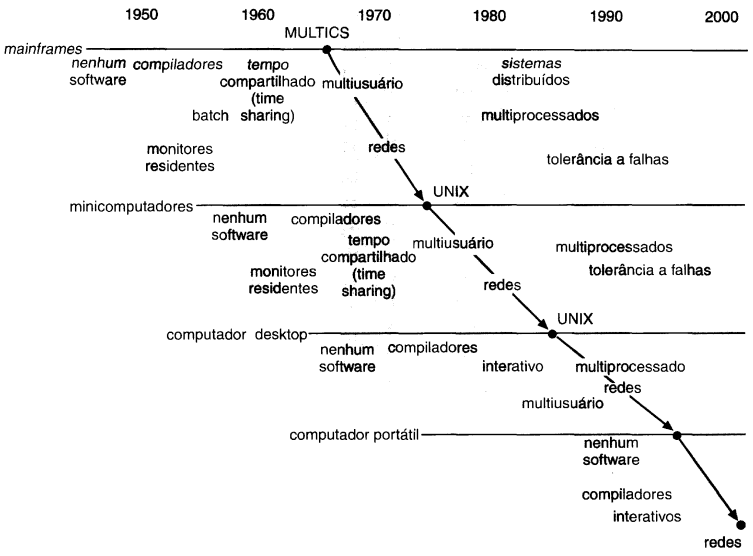



FIGURA 1.6 *Migração de conceitos e recursos do sistema operacional.*

grande e complexo (o GE 645). Muitas das idéias desenvolvidas para o MULTICS foram usadas mais tarde na Bell Laboratories (um dos parceiros originais no desenvolvimento do MULTICS) no projeto do UNIX. O sistema operacional UNIX foi projetado em torno de 1970 para um minicomputador PDP-11. Por volta de 1980, os recursos do UNIX se tornaram a base para sistemas operacionais tipo UNIX em sistemas em microcomputadores; e esses recursos estão sendo incluídos em sistemas operacionais mais recentes, como Microsoft Windows 2000 e Windows XP, e no sistema operacional Mac OS X. Assim, os recursos desenvolvidos para um grande mainframe foram levados para os microcomputadores com o passar do tempo. O Linux inclui alguns desses mesmos recursos e agora já pode ser encontrado nos PDAs.

## 1.10 Ambientes de computação

Acompanhamos o desenvolvimento dos sistemas operacionais desde os primeiros sistemas na prática, passando pelos sistemas multiprogramados e de tempo compartilhado, até chegar aos PCs e computadores portáteis. Concluímos com uma rápida visão geral de como esses sistemas são usados em uma série de ambientes de computação.

### 1.10.1 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 oferecendo 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 prevalentes em muitas empresas, com ainda menos opções de acesso remoto e portabilidade.

A tendência atual é oferecer mais formas de acessar esses ambientes. As tecnologias Web estão alargando as fronteiras da computação tradicional. As empresas estabelecem **portais**, que oferecem acessibilidade da Web a seus servidores internos. Os Net-

**work 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 cliente 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 há uma década.

### 1.10.2 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 oferecendo 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 XP, 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.

### 1.10.3 Computação embutida

Computadores embutidos, que executam sistemas operacionais de tempo real embutidos, são a forma prevalente de computadores atualmente. Esses dispositivos são encontrados em toda a parte, desde motores de carro e robôs de manufatura até videocassetes e fornos de microondas. Eles costumam ter tarefas muito específicas. Os sistemas em que eles executam normalmente são primitivos, e assim os sistemas operacionais oferecem recursos limitados. Por exemplo, 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.

Como um exemplo, considere os firewalls e os balanceadores de carga que mencionamos. 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, oferecendo apenas a funcionalidade desejada.

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 em casa. Algum dia, o refrigerador poderá ligar para o supermercado quando notar que o leite está acabando.

## 1.11 Resumo

Os sistemas operacionais têm sido desenvolvidos há mais de 5 anos para duas finalidades principais. Primeiro, o sistema operacional tenta escalonar as atividades de computação para garantir o bom desempenho do sistema de computação. Segundo, ele oferece um ambiente conveniente para o desenvolvimento e a execução de programas.

Inicialmente, os sistemas computadorizados eram usados a partir do console frontal. Software

como montadores, carregadores, linkers e compiladores melhoraram a conveniência da programação do sistema, mas também exigiram um tempo de preparação substancial.

Os sistemas batch permitiram a seqüência automática de tarefas por um sistema operacional residente e melhoraram bastante a utilização geral do computador. O computador não precisava mais esperar pela operação humana. Porém, a utilização da CPU ainda era baixa, devido à baixa velocidade dos dispositivos de E/S em relação à da CPU.

Para melhorar o desempenho geral do sistema computadorizado, os desenvolvedores introduziram o conceito de multiprogramação, de modo que várias tarefas pudessem ser mantidas na memória ao mesmo tempo. A CPU é alternada entre elas para aumentar a utilização de CPU e para diminuir o tempo total necessário para executar as tarefas.

A multiprogramação não apenas melhora o desempenho, mas também permite o compartilhamento de tempo. Sistemas operacionais de tempo compartilhado permitem que muitos usuários (de um a várias centenas) utilizem um sistema computadorizado interativamente ao mesmo tempo.

PCs são microcomputadores consideravelmente menores e mais baratos do que os mainframes. Os sistemas operacionais para esses computadores se beneficiaram do desenvolvimento de sistemas operacionais para mainframes de diversas maneiras. Como um indivíduo utiliza o computador exclusivamente, a utilização da CPU não é um problema primordial. Logo, algumas das decisões de projeto, tomadas para os sistemas operacionais de mainframe, podem não ser apropriadas para esses sistemas menores. Outras decisões de projeto, como as de segurança, são apropriadas para sistemas grandes e pequenos, pois os PCs agora estão conectados a outros computadores e usuários por meio de redes e da Web.

Sistemas multiprocessados, ou sistemas paralelos, são compostos de duas ou mais CPUs em íntima comunicação. As CPUs compartilham o barramento do computador e às vezes compartilham memória e dispositivos periféricos. Esses sistemas podem oferecer maior throughput e mais confiabilidade.

Sistemas distribuídos permitem que os usuários compartilhem recursos em computadores geograficamente dispersos, por meio de uma rede de computadores. Os serviços podem ser fornecidos por meio do modelo cliente-servidor ou pelo modelo peer-to-peer. Em um sistema em clusters, várias máquinas podem realizar computações sobre dados que residem no armazenamento compartilhado, e a computação pode continuar mesmo quando algum subconjunto dos membros agrupados falhar.

Um sistema de tempo real rígido normalmente é usado como um dispositivo de controle em uma aplicação dedicada. Um sistema operacional de tempo real rígido possui restrições de tempo fixas e bem definidas. O processamento *precisa* ser feito dentro de restrições definidas, ou então o sistema falhará. Os sistemas de tempo real flexível possuem restrições de tempo menos rigorosas e não admitem o escalonamento de prazos.

Os sistemas portáteis são cada vez mais populares e apresentam vários desafios para desenvolvedores de aplicação, devido a menos memória e processadores mais lentos do que os PCs desktop.

Recentemente, a influência da Internet e da World Wide Web incentivou o desenvolvimento dos sistemas operacionais modernos, que incluem navegadores Web, redes e software de comunicação como recursos integrais.

Mostramos a progressão lógica do desenvolvimento do sistema operacional, controlada pela inclusão de recursos no hardware da CPU necessário para uma funcionalidade mais avançada. Essa tendência pode ser vista hoje na evolução dos PCs, com um hardware pouco dispendioso, sendo melhorado suficientemente para permitir, por sua vez, características aperfeiçoadas.

## Exercícios

**1.1** Quais são os três propósitos principais de um sistema operacional?

**1.2** Relacione os quatro passos necessários para executar um programa em uma máquina completamente dedicada.

**1.3** Qual é a principal vantagem da multiprogramação?

**1.4** Quais são as principais diferenças entre o sistema operacional para mainframes e PCs?

**1.5** Em um ambiente de multiprogramação e tempo compartilhado, vários usuários compartilham o sistema

simultaneamente. Essa situação pode resultar em diversos problemas de segurança.

**a.** Cite dois desses problemas?

**b.** Podemos garantir o mesmo grau de segurança em uma máquina de tempo compartilhado como temos em uma máquina dedicada? Explique sua resposta.

**1.6** Defina as propriedades essenciais dos seguintes tipos de sistemas operacionais:

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

**1.7** Enfatizamos a necessidade de que um sistema operacional faça uso eficaz do hardware de computação. Quando é apropriado que um sistema operacional abandone esse princípio e "desperdice" recursos? Por que esse sistema não é realmente desperdiçador?

**1.8** Sob quais circunstâncias seria melhor para um usuário usar um sistema de tempo compartilhado ao invés de um PC ou estação de trabalho monousuário?

**1.9** Descreva as diferenças entre o multiprocessamento simétrico e assimétrico. Cite três vantagens e uma desvantagem dos sistemas multiprocessados.

**1.10** Qual é a principal dificuldade que um programador precisa contornar na escrita de um sistema operacional para um ambiente de tempo real?

**1.11** Faça a distinção entre os modelos cliente-servidor e peer-to-peer dos sistemas distribuídos.

**1.12** Considere as diversas definições do *sistema operacional*. Considere se o sistema operacional deverá incluir aplicações como navegadores Web e programas de correio. Argumente as posições pró e contra, explicando suas respostas.

**1.13** Quais são as escolhas inerentes aos computadores portáteis?

**1.14** 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.

## Notas bibliográficas

Os sistemas de tempo compartilhado foram propostos inicialmente por Strachey [1959]. Os primeiros sistemas de tempo compartilhado foram o Compatible Time-Sharing System (CTSS), desenvolvido no MIT (Corbato e outros [1962]), e o sistema SDC Q-32, montado pela System Development Corporation (Schwartz e outros [1964], Schwartz e Weissman [1967]). Outros sistemas antigos, porém menos sofisticados, incluem o sistema MULTiplexed Information and Computing Services (MULTICS), desenvolvido no MIT (Corbato e Vyssotsky [1965]), o sistema XDS-940, desenvolvido na Universidade da Califórnia em Berkeley (Lichtenberger e Pirtle [1965]) e o sistema IBM TSS/360 (Lett e Konigsford [1968]).

Uma visão geral do sistema operacional Linux pode ser encontrada em Bovet e Cesati [2002]. Solomon e Russinovich [2000] oferecem uma visão geral do Microsoft Windows 2000 e detalhes técnicos consideráveis sobre o interior e os componentes do sistema. Mauro e McDougall [2001] abordam o sistema operacional Solaris. O Mac OS X é apresentado em *http://www.apple.com/macosx*.

A cobertura sobre os sistemas peer-to-peer inclui Parameswaran e outros [2001], Gong [2002] e Ripeanu e outros [2002]. Uma boa cobertura da computação de clusters é apresentada por Buyya [1999]. Avanços recentes na computação em cluster são descritos por Ahmed [2000].

Discussões com relação a dispositivos portáteis são oferecidas por Murray [1998] e Rhodes e McKeehan [1999].

Muitos livros-texto gerais abordam sistemas operacionais, incluindo Stallings [2000b], Nutt [2000] e Tanenbaum [2001].

CAPÍTULO 2

# Estruturas do computador

Antes de explorar os detalhes da operação do sistema computadorizado, precisamos conhecer a estrutura do sistema. Neste capítulo, veremos várias partes dessa estrutura. O capítulo trata principalmente da arquitetura do sistema computadorizado. Portanto, se você já compreende bem os conceitos, pode passar por ele superficialmente ou então pulá-lo.

Começamos discutindo as funções básicas da inicialização do sistema, E/S e armazenamento. Mais adiante no capítulo, descrevemos a arquitetura básica do computador, que possibilita a escrita de um sistema operacional funcional. Concluímos com uma visão geral da arquitetura de rede.

## 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, oferecendo acesso à memória compartilhada (Figura 2.1). Cada controle de dispositivo está encarregado de um tipo específico de dispositivo (por exemplo, unidades de disco, dispositivos de entrada, barramento de E/S 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. É claro que existem diversas variações dessa estrutura básica, incluindo várias CPUs e barramentos dedicados à CPU-memória, mas os conceitos apresentados neste capítulo também se aplicam independente da estrutura.

Para um 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, ele é armazenado na memória somente de leitura (ROM), como o firmware ou EEPROM 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

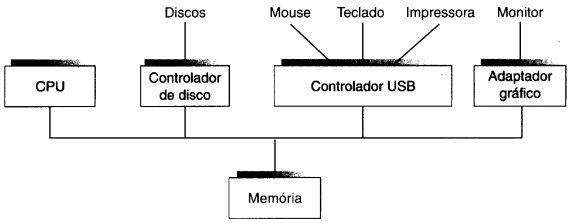



FIGURA 2.1 *Um sistema computadorizado moderno.*

software pode disparar uma interrupção executando uma operação especial, denominada **system call (chamada de sistema)** ou **monitor call (chamada ao monitor)**.

Os sistemas operacionais modernos são **controlados por interrupção (interrupt driven)**. 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 que ação deve ser realizada, e a rotina de serviço de interrupção (**Interrupt Service Routine – ISR**) é fornecida para tratar dessa interrupção.

Quando a CPU é interrompida, ela pára 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 2.2.

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. Em primeiro lugar, 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 e depois chamar o tratador específico da interrupção. Contudo, como só pode haver uma quantidade predefinida de interrupções, uma tabela de ponteiros para rotinas de interrupção pode ser usada no lugar desse enfoque. A rotina de interrupção, então, é

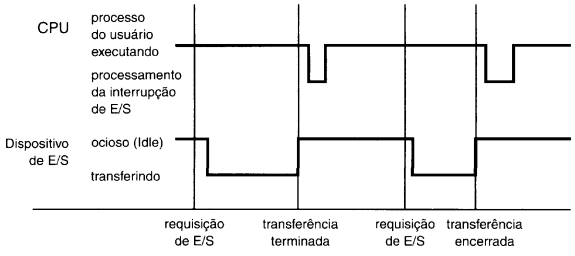



FIGURA 2.2 *Linha de tempo da interrupção para um único processo efetuando uma saída.*

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 fornecer o endereço da rotina de atendimento de interrupção para o dispositivo que está interrompendo. Sistemas operacionais tão diferentes quanto MS-DOS e UNIX despacham interrupções dessa forma.

A arquitetura de interrupção também precisa salvar o endereço da instrução interrompida. Muitos projetos antigos simplesmente 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 precisa 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.

Uma chamada de sistema é invocada de diversas maneiras, dependendo da funcionalidade oferecida pelo processador em uso. Em todas as formas, esse é o método usado por um processo para solicitar a ação do 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 de trap genérica, embora alguns sistemas (como a família MIPS R2000) tenham uma instrução `syscall`* específica.

## 2.2 Estrutura de E/S

Como mencionado na seção anterior, 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. O tamanho do buffer local dentro de um controlador de dispositivo varia de um controlador para outro, dependendo do dispositivo controlado. Por exemplo, o tamanho do buffer de um controlador de disco é igual ou um múltiplo do tamanho da menor parte endereçável de um disco, chamada **setor**, que costuma ser de 512 bytes. Atualmente, são comuns buffers de controlador de disco de 2 a 8MB.

### 2.2.1 Interrupções de E/S

Para iniciar uma operação de E/S, a CPU 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. Por exemplo, se ele encontrar uma requisição de leitura, o controlador começará a transferir dados do dispositivo para o seu buffer local. Quando a transferência de dados estiver concluída, o controlador de dispositivo informará à CPU que terminou a operação. Ele realiza essa comunicação disparando uma interrupção.

Em geral, essa situação ocorrerá como resultado de um processo do usuário solicitando E/S. Quando a E/S for iniciada, duas ações são possíveis. No caso mais simples, a E/S é iniciada; depois, ao término da E/S, o controle é retornado ao processo do usuário. Esse caso é conhecido como E/S **síncrona**. A outra possibilidade, chamada E/S **assíncrona**, retorna o controle ao programa do usuário sem esperar o término da E/S. A E/S, então, pode continuar enquanto ocorrem outras operações do sistema (Figura 2.3).

Qualquer que seja o método utilizado, a espera pelo término da E/S pode ser feita de duas formas. Alguns computadores possuem uma instrução de es-

* *Nota do revisor técnico:* Syscall é a contração de system call.

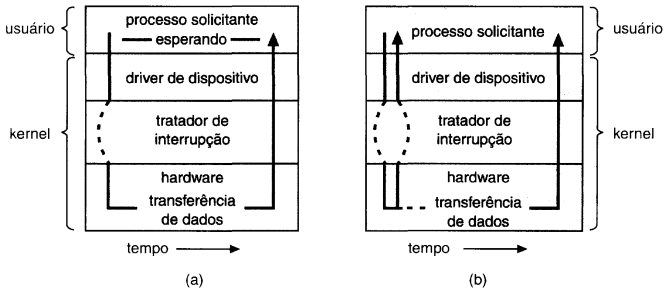



FIGURA 2.3 *Dois métodos de E/S: (a) síncrona e (b) assíncrona.*

pera especial, que deixa a CPU ociosa até a próxima interrupção. As máquinas que não possuem tal instrução podem ter um loop de espera:

*Loop:* `jmp` *Loop*

Esse loop continua até que haja uma interrupção, transferindo o controle para outra parte do sistema operacional. Um loop desse tipo também pode ser necessário em dispositivos de E/S que não trabalham com a estrutura de interrupção, mas que, em vez disso, simplesmente definem um sinalizador (flag) em um de seus registradores e esperam que o sistema operacional observe esse sinalizador.

Se a CPU sempre esperar pelo término da E/S, como acontece com a técnica síncrona, somente uma requisição de E/S poderá estar pendente de cada vez. Assim, sempre que ocorre uma interrupção de E/S, o sistema operacional sabe exatamente qual dispositivo está interrompendo. Entretanto, essa técnica impossibilita a operação de vários dispositivos de E/S ao mesmo tempo ou a realização de alguma computação útil enquanto a E/S é realizada.

Uma alternativa melhor é iniciar a E/S e depois continuar processando outro código do sistema operacional ou do programa do usuário – a técnica assíncrona. Portanto, é necessária uma chamada de sistema para permitir que o programa do usuário espere o término da E/S, se desejar. Se nenhum programa do usuário estiver pronto para execução e o sistema operacional não tiver outro trabalho para realizar, ainda precisamos da instrução `wait` ou o loop ocioso (idle loop), como antes. Também precisamos registrar muitas requisições de E/S ao mesmo tempo. Para essa finalidade, o sistema operacional utiliza uma tabela contendo uma entrada para cada dispositivo de E/S: a **tabela de status de dispositivo** (Figura 2.4). Cada entrada na tabela indica o tipo do dispositivo, o endereço e o 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. Como é possível que outros processos emitam requisições ao mesmo dispositivo, o sistema operacional também manterá uma **fila de espera** – uma lista de requisições aguardando – para cada dispositivo de E/S.

Um dispositivo de E/S interrompe quando precisa de atendimento. Quando ocorre uma interrupção, o sistema operacional primeiro determina qual dispositivo de E/S causou a interrupção. Depois, examina a tabela de dispositivos de E/S para determinar o status desse dispositivo e modifica a entrada da tabela para refletir a ocorrência da interrupção. Para a maioria dos dispositivos, uma interrupção sinaliza o término de uma requisição de E/S. Se houver requisições adicionais aguardando na fila, o sistema operacional inicia o processamento da próxima requisição.

Finalmente, o controle retorna da interrupção de E/S. Se um processo estiver esperando o término de sua requisição (conforme registrado na tabela de status do dispositivo), agora podemos retornar o controle para a execução do programa do usuário ou para o loop de espera. Em um sistema de tempo compartilhado, o sistema operacional poderia passar para outro processo pronto para execução.

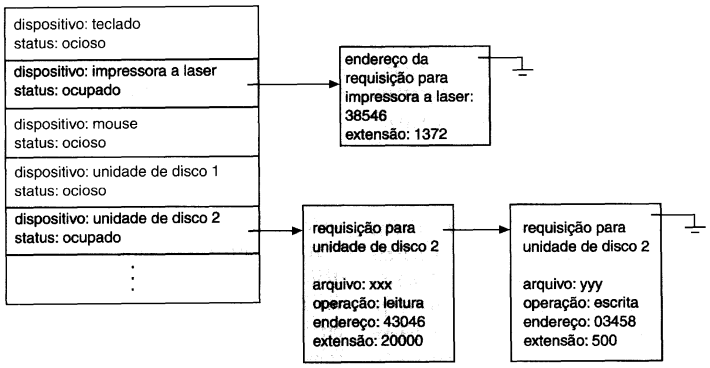



FIGURA 2.4 *Tabela de status de dispositivo.*

Os esquemas utilizados por dispositivos de entrada específicos podem variar desse descrito. Muitos sistemas interativos permitem que os usuários digitem antecipadamente – incluam dados antes que eles sejam solicitados – no teclado. Nesse caso, as interrupções podem ocorrer, sinalizando a chegada de caracteres do terminal, enquanto o bloco de status do dispositivo indique que nenhum programa requisitou a entrada desse dispositivo. Se a digitação antecipada for permitida, então é preciso haver um buffer para armazenar os caracteres digitados até que algum programa os solicite. Em geral, mantemos um buffer no kernel para cada dispositivo, assim como cada dispositivo possui seu próprio buffer.

Em um driver simples de entrada de terminal, quando uma linha tiver de ser lida do terminal, o primeiro caractere digitado é enviado ao computador. Quando esse caractere é recebido, o dispositivo de comunicação assíncrona (ou porta serial) ao qual a linha do terminal está conectada interrompe a CPU. Quando a requisição de interrupção do terminal chegar, a CPU está pronta para executar alguma instrução. (Se a CPU estiver no meio da execução de uma instrução, a interrupção costuma ser mantida em estado pendente do término da execução da instrução.) O endereço dessa instrução interrompida é salvo, e o controle é transferido para a rotina de atendimento de interrupção para o dispositivo apropriado.

A rotina de atendimento de interrupção salva o conteúdo de quaisquer registradores da CPU que ela precisará usar. Ela verificará quaisquer condições de erro que possam ter resultado da operação de entrada mais recente. Depois, apanha o caractere do dispositivo e o armazena em um buffer. A rotina de interrupção também precisa ajustar variáveis ponteiro e contadores, para ter certeza de que o próximo caractere inserido será armazenado no próximo local no buffer. Em seguida, a rotina de interrupção marca (colocando um valor predefinido) um sinalizador na memória, indicando às outras partes do sistema operacional que uma nova entrada foi recebida. As outras partes são responsáveis por processar os dados no buffer e por transferir os caracteres para o programa que está requisitando entrada. Depois, a rotina de atendimento de interrupção restaura o conteúdo de quaisquer registradores salvos e transfere o controle de volta para a instrução interrompida.

Deve ter ficado claro que a principal vantagem da E/S assíncrona é a maior eficiência do sistema. Enquanto a E/S está ocorrendo, a CPU do sistema pode ser usada para processar ou iniciar operações de E/S para outros dispositivos. Como a E/S pode ser lenta em comparação com a velocidade do processador, o sistema utiliza suas facilidades de forma eficiente. Na próxima seção, descrevemos outro mecanismo utilizado para melhorar o desempenho do sistema.

### 2.2.2 Transferências de DMA

Considere os caracteres digitados em um teclado. O teclado pode aceitar e transferir um caractere aproximadamente a cada 1 milissegundo, ou 1.000 microssegundos. Uma rotina de atendimento de interrupção bem escrita para a entrada de caracteres em um buffer pode exigir 2 microssegundos por caractere, deixando 998 microssegundos de cada 1.000 para computação da CPU (e para atender a outras interrupções). Devido a essa disparidade, a E/S assíncrona normalmente recebe uma prioridade de interrupção baixa, permitindo que outras interrupções mais importantes sejam processadas primeiro, ou até mesmo que assumam o lugar da interrupção atual. Entretanto, um dispositivo de alta velocidade – como unidade de fita, disco ou rede de comunicações – pode ser capaz de transmitir informações em velocidades próximas às velocidades da memória; se a CPU precisa de 2 microssegundos para responder a cada interrupção e as interrupções chegam, digamos, a cada 4 microssegundos, o que não deixa muito tempo para a execução do processo.

Para resolver esse problema, o **acesso direto à memória (DMA – Direct Memory Access)** é utilizado para dispositivos de E/S de alta velocidade. 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, ao invés de uma interrupção por byte, gerada para os dispositivos de baixa velocidade.

A operação básica da CPU é a mesma. Um programa do usuário, ou o próprio sistema operacional, pode requisitar a transferência de dados. O sistema operacional encontra um buffer (um buffer vazio para entrada ou um buffer cheio para saída) a partir de um banco de buffers para a transferência. (Os buffers em um banco de buffers podem ter um tamanho fixo ou podem variar de acordo com o tipo da E/S realizada.) Em seguida, uma parte do sistema operacional, chamada **driver de dispositivo (device driver)**, marca (colocando valores predefinidos) os registradores do controlador de DMA para utilizarem endereços de origem e destino apropriados, além da extensão da transferência. O controlador de DMA, em seguida, é instruído a iniciar a operação de E/S. Enquanto o controlador de DMA está realizando a transferência de dados, a CPU está livre para realizar outras tarefas. Como a memória pode transferir apenas uma word de cada vez, o controlador de DMA "rouba" ciclos de memória da CPU. Esse roubo de ciclos pode atrasar a execução da CPU enquanto uma transferência de DMA está em andamento.

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, pois não é preciso haver roubo de ciclos durante as transferências de E/S para a memória. O controlador de DMA interrompe a CPU quando a transferência tiver terminado. A CPU pode, então, liberar o buffer de saída ou avisar ao processo que espera pela entrada do buffer de que existe um E/S aguardando.

## 2.3 Estrutura de armazenamento

Os programas de computador precisam estar na memória principal (também chamada **memória de acesso aleatório** ou **RAM – Random Access Memory**) para serem executados. A memória principal é a única grande área de armazenamento (de milhões a bilhões de bytes) que o processador pode acessar diretamente. Ela é implementada em uma tecnologia de semicondutores chamada **memória de acesso aleatório dinâmica (DRAM – Dynamic Random Access Memory)**, que forma um conjunto de words de memória. Cada word possui seu próprio endereço. A interação é obtida por meio de uma seqüência de instruções para carregar ou armazenar, para especificar endereços de memória específicos. A instrução para carregar move uma word da memória principal para um registrador interno dentro da CPU, enquanto a instrução para armazenar 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.

Um ciclo típico de execução de instrução, conforme executado em um sistema com uma arquitetura

**von Neumann**, primeiro apanhará uma instrução da memória e armazenará essa instrução no **registrador de instruções**. A instrução é, então, decodificada e pode fazer com que operandos sejam 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 e assim por diante) ou para que servem (instruções ou dados). Como conseqüência, podemos ignorar *como* um endereço de memória é gerado por um programa. Só estamos interessados na seqüência 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 oferece 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 oferece um espaço para armazenamento para programas e dados. A maioria dos programas (navegadores Web, compiladores, processadores de textos, planilhas e assim por diante) 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, o gerenciamento apropriado do armazenamento em disco é de grande importância para um sistema computadorizado, conforme discutiremos no Capítulo 14.

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 oferece as funções básicas de armazenamento de um dado e manutenção desse dado até que seja recuperado. As principais diferenças entre os diversos sistemas de armazenamento estão na velocidade, custo, tamanho e volatilidade. Nas Seções 2.3.1 a 2.3.3, descrevemos a memória principal, os discos magnéticos e as fitas magnéticas, pois ilustram as propriedades gerais de todos os dispositivos de armazenamento disponíveis comercialmente. No Capítulo 14, discutiremos as propriedades de muitos dispositivos específicos, como disquetes, discos rígidos, CD-ROMs e DVDs.

### 2.3.1 Memória principal

Conforme sugerido anteriormente, 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.

No caso da E/S, como mencionado na Seção 2.1, cada controlador de E/S inclui registradores para manter comandos e dados que estão sendo transferidos. Normalmente, instruções de E/S especiais permitem transferência de dados entre esses registradores e a memória do sistema. Para oferecer acesso mais conveniente aos dispositivos de E/S, muitas arquiteturas de computador utilizam **E/S mapeada na memória**. Nesse caso, faixas de endereços de memória são reservadas e mapeadas para os registradores dos dispositivos. As leituras e escritas nesses endereços de memória fazem com que os dados sejam transferidos de e para os registradores de dispositivos. Esse método é apropriado para dispositivos rápidos, pois possuem tempos de resposta pequenos, como controladores de vídeo. No IBM PC, cada ponto da tela é mapeado em uma locação de memó-

ria. A exibição de texto na tela é quase tão fácil quanto escrever texto em locações apropriadas da memória mapeada.

A E/S mapeada em memória também é conveniente para outros dispositivos, como as portas seriais e paralelas usadas para a conexão de modems e impressoras a um computador. A CPU transfere dados por meio desses tipos de dispositivos lendo e escrevendo em alguns registradores de dispositivos, chamados **portas** de E/S. Para enviar uma longa seqüência de bytes por uma porta serial mapeada em memória, a CPU escreve um byte de dados no registrador de dados, depois marca um bit (coloca em 1, por exemplo) 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. Em seguida, a CPU pode transferir o próximo byte. A CPU pode usar a técnica de polling para consultar e observar o bit de controle, verificando repetidamente se o dispositivo está pronto; esse método de operação é denominado **E/S programada** (ou **PIO – Programmed I/O**). Se a CPU não consultar 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 **controlada por interrupção (interrput driven)**.

Os registradores internos a 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, quando o processador normalmente precisa **protelar (stall)** suas operações, pois não tem os dados necessários para completar a instrução que está executando. Como o acesso à memória é feito com muita freqüência, essa situação se torna intolerável. A solução é acrescentar uma memória de acesso rápida entre a CPU e a memória principal. Esse buffer de memória usado para alojar um diferencial de velocidade, chamado **cache**, é descrito na Seção 2.4.1.

* *Nota do revisor técnico*: Um clock tick é não mais do que um pulso de relógio (uma batida) utilizado pelo sistema operacional em diversas operações.

## 2.3.2 Discos magnéticos

Os **discos magnéticos** são responsáveis pela maior parte do armazenamento secundário nos sistemas computadorizados modernos. Conceitualmente, os discos são simples (Figura 2.5). Cada **prato** do disco possui uma forma circular plana, como um CD. Os diâmetros comuns dos pratos variam desde 1,8 até 5,25 polegadas. As duas superfícies de um prato são revestidas por um material magnético. Armazenamos informações gravando-as magneticamente sobre os pratos.

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**. Uma unidade de disco pode ter 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 250 vezes por segundo. A velocidade do disco possui

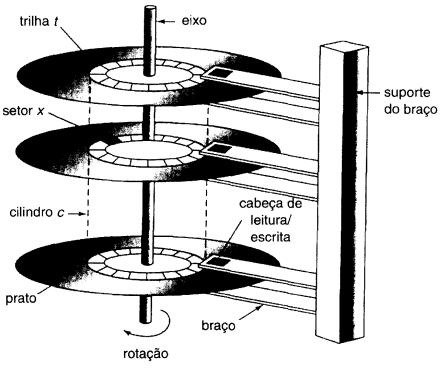



FIGURA 2.5 *Mecanismo de disco com cabeça móvel.*

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 aleatório (random-access time)**, consiste no tempo necessário para mover o braço do disco até o cilindro desejado, chamado **tempo de busca (seek time)**, mais o tempo necessário 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 mícrons), a cabeça pode fazer contato com a superfície do disco. Quando isso acontece, embora os pratos de disco sejam revestidos por uma fina camada protetora, 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.

Os discos podem ser **removíveis**, 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 protegê-lo contra danos enquanto não está na unidade de disco. Um **disquete** é um disco magnético removível de baixo custo, consistindo em um prato flexível dentro de um compartimento de plástico. 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 1MB. Porém, existem discos removíveis que trabalham de forma muito semelhante a discos rígidos, e possuem capacidades medidas em gigabytes. Outras mídias removíveis incluem CD-ROMs e DVD-ROMs, que são lidos por raio laser para detectar sulcos no meio de armazenamento. CD-ROMs e DVD-ROMs são dispositivos apenas de leitura, mas também existem versões para leitura e escrita. Quase todos os meios removíveis são mais lentos do que os meios fixos.

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), FC (fibre 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)**, ou adaptador de barramento do host (HBA – Host Bus Adapter), 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 2.3.1. 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.

### 2.3.3 Fitas magnéticas

A **fita magnética** foi utilizada a princípio como um meio de armazenamento secundário. Embora sendo relativamente permanente e podendo armazenar grandes quantidades de dados, seu tempo de acesso é lento em comparação com o da memória principal. Além disso, o acesso aleatório à fita magnética é cerca de mil vezes mais lento do que o acesso aleatório ao disco magnético. Devido a esses problemas de velocidade, as fitas não são muito úteis para armazenamento secundário. São usadas principalmente para backup, para o armazenamento de informações usadas com pouca freqüência e como um meio para transferir informações de um sistema para outro.

Uma fita é mantida em um spool 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 posicionadas, podem gravar dados em velocidades comparáveis às das unidades de disco. As fitas e seus drivers normalmente estão categorizados por largura, incluindo 4, 8 e 19 milímetros, e 1/4 e 1/2 polegada. As capacidades da

fita variam muito, dependendo do tipo da unidade de fita. Hoje, em termos de capacidade, fitas e discos geralmente são semelhantes.

## 2.4 Hierarquia de armazenamento

A grande variedade de sistemas de armazenamento disponíveis para computadores pode ser organizada em uma hierarquia (Figura 2.6), 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 –, então 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 2.6 podem ser construídos com a utilização de memória de semicondutores.

Além de ter diferentes velocidades e custos, os diversos sistemas de armazenamento são voláteis ou não voláteis. O **armazenamento volátil** perde seu conteúdo quando o poder do dispositivo é removido. 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 2.6, 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.

O projeto de um sistema de memória completo precisa equilibrar todos esses fatos: ele utiliza apenas a quantidade de memória cara necessária, enquanto oferece 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.

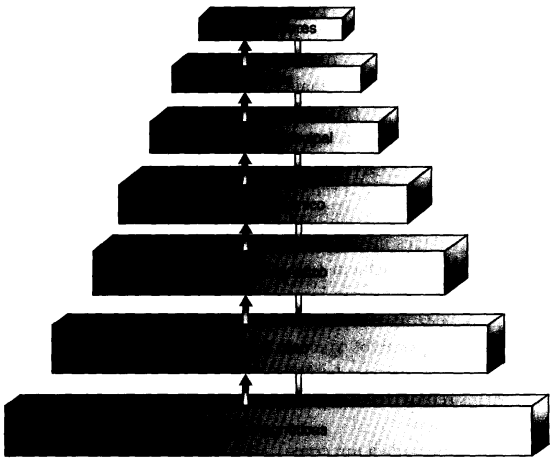


FIGURA 2.6 *Hierarquia de dispositivo de armazenamento.*

### 2.4.1 Caching

O **caching**, já citado várias vezes neste capítulo, 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, lemos 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, oferecem 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 instrução 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 sistema em que 80 a 99% de todos os acessos ocorrem no cache, aumentando bastante o desempenho. Diversos algoritmos de substituição para caches controlados por software são discutidos no Capítulo 10.

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 14).

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. Ao contrário disso, a transferência de dados do disco para a memória normalmente é controlada pelo sistema operacional.

### 2.4.2 Coerência e consistência

Em uma estrutura de armazenamento hierárquica, os mesmos dados podem aparecer em diferentes níveis. Por exemplo, suponha que o inteiro A esteja localizado no arquivo B, que reside em disco magnético. O inteiro A deverá ser incrementado em 1. 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. Em seguida, A é 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 (ver Figura 2.7). Quando o incremento ocorre no registrador interno, os valores de A nos diversos sistemas de

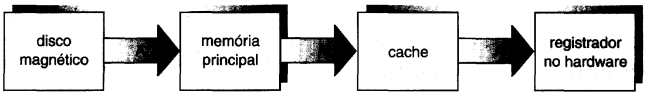



FIGURA 2.7 *Migração do inteiro A do disco para o registrador.*

armazenamento diferem. Os valores só se tornam iguais depois que o novo valor de A for escrito do registrador interno para o disco magnético.

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, então, 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, 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 diferentes computadores, distribuídos no espaço. Como as diversas réplicas podem ser acessadas e atualizadas ao mesmo tempo, temos de garantir 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 16.

## 2.5 Proteção do hardware

Como já vimos, os primeiros sistemas computadorizados eram sistemas monousuários operados pelo programador. Quando os programadores operavam o computador pelo console, eles tinham controle completo do sistema. Todavia, com o desenvolvimento dos sistemas operacionais, esse controle mudou. Os primeiros sistemas operacionais eram chamados de **monitores residentes**; começando com os monitores residentes, os sistemas operacionais começaram a realizar muitas das funções, especialmente E/S, pelas quais o programador era responsável anteriormente.

Além disso, para melhorar a utilização do sistema, o sistema operacional começou a *compartilhar* recursos do sistema entre diversos programas simultaneamente. Com o *spooling*, por exemplo, um programa poderia estar sendo executado enquanto a E/S ocorria para outros processos; o disco manteria dados simultâneos para muitos processos. Com a multiprogramação, vários programas poderiam estar na memória ao mesmo tempo.

Esse compartilhamento melhorou a utilização, mas aumentou os problemas. Quando o sistema era executado sem compartilhamento, um erro em um programa só poderia causar problemas para o único programa em execução. Com o compartilhamento, muitos processos poderiam ser prejudicados por um erro em um programa.

Por exemplo, considere um sistema operacional batch simples (Seção 1.2.1), que não oferece nada além de seqüência automática de tarefas. Se um programa ficar preso em um loop, lendo a entrada de cartões, o programa lerá todos os dados e, a menos que algo evite isto, continuará lendo os cartões da próxima tarefa, e da seguinte, e assim por diante. Esse loop poderia impedir a operação correta de muitas tarefas.

Erros mais sutis podem ocorrer em um sistema de multiprogramação, no qual um programa com erro poderia modificar outro programa, os dados de outro programa ou até mesmo o próprio monitor residente. MS-DOS e o Macintosh OS permitem esse tipo de erro.

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 com que outros programas sejam executados de forma incorreta.

### 2.5.1 Operação no modo dual

Para garantir uma operação apropriada, temos de proteger o sistema operacional e todos os outros programas e seus dados contra qualquer programa que não esteja funcionando. A proteção é necessária para qualquer recurso compartilhado. A técnica utilizada por muitos sistemas operacionais oferece suporte de hardware que permite diferenciar entre diversos modos de execução.

No mínimo, precisamos de dois **modos** de operação separados: modo usuário e modo monitor (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: monitor (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. Como veremos, essa melhoria arquitetônica também é útil para muitos outros aspectos do sistema operacional.

No momento do boot do sistema, o hardware começa no modo monitor.O sistema operacional é, então, carregado e inicia os processos do usuário no modo usuário. Sempre que ocorre um trap ou uma interrupção, o hardware passa do modo usuário para o modo monitor (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 monitor. 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 oferece 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 monitor. Se houver uma tentativa de executar uma instrução privilegiada no modo usuário, o hardware não executa a instrução, a trata como ilegal e chama o sistema operacional.

O conceito de instruções privilegiadas também oferece meios para um programa do usuário solicitar que o sistema operacional realize tarefas reservadas ao sistema operacional em nome do programa do usuário. Cada requisição desse tipo é chamada pelo usuário executando uma instrução privilegiada. Ela é conhecida como *chamada de sistema* (também conhecida como *chamada de monitor* ou *chamada de função do sistema operacional*) – conforme descrevemos na Seção 2.1.

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 monitor. A rotina de serviço da chamada de sistema é uma parte do sistema operacional. O monitor 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 sendo passados nos registradores). O monitor verifica se os parâmetros são corretos e válidos, executa a requisição e retorna o controle à instrução após a chamada de sistema.

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 um 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 escrever em um dispositivo ao mesmo tempo, possivelmente com resultados desastrosos. As versões mais recentes e mais avançadas da CPU Intel, como o Pentium, oferecem o modo de operação dual. Por conseguinte, sistemas operacionais mais recentes, como Microsoft Windows 2000 e XP, IBM OS/2 e Linux e Solaris para sistemas x86, aproveitam esse

recurso e oferecem 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 pode ser gerado um dump da memória. Esse dump é gravado em um arquivo, para que o usuário ou programador possa examiná-lo, talvez corrigi-lo, e reiniciar o programa.

### 2.5.2 Proteção da E/S

Um programa do usuário pode atrapalhar a operação normal do sistema emitindo instruções de E/S ilegais, acessando locais da memória dentro do próprio sistema operacional ou recusando-se a abrir mão da CPU. 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, por meio de uma chamada de sistema (Figura 2.8). O sistema operacional, executando no modo monitor, verifica e certifica-se de que a requisição é válida e (se for válida) realiza a operação de E/S requisitada. O sistema operacional, em seguida, retorna ao usuário.

Para que a proteção da E/S seja completa, temos de garantir que um programa do usuário nunca possa obter o controle do computador no modo monitor. Se pudesse, a proteção da E/S poderia ser comprometida. Considere um computador executando no modo usuário. Ele passará para o modo monitor sempre que houver uma interrupção ou trap, saltando para o endereço determinado pelo vetor de interrupção. Se um programa do usuário, como parte de sua execução, armazenar um novo endereço no vetor de interrupção, esse novo endereço poderia substituir o endereço anterior por um endereço no programa do usuário. Depois, quando o trap ou in-

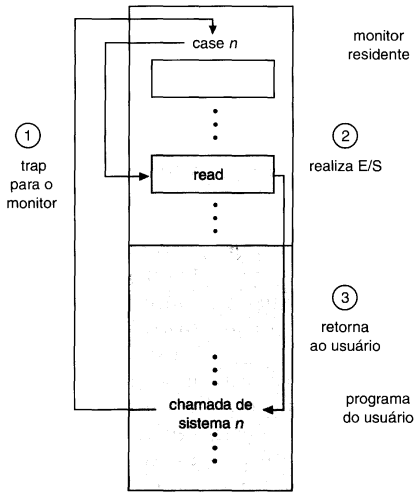



FIGURA 2.8 *Uso de uma chamada de sistema para realizar E/S.*

terrupção correspondente acontecesse, o hardware passaria para o modo monitor e transferiria o controle por meio do vetor de interrupção (modificado) para o programa do usuário. O programa do usuário também poderia obter o controle do computador no modo monitor de muitas outras maneiras. Além disso, todos os dias são descobertos novos erros que podem ser explorados para vencer as proteções do sistema. Esses tópicos são discutidos nos Capítulos 18 e 19.

### 2.5.3 Proteção da memória

Para garantir a operação correta, temos de proteger o vetor de interrupção contra modificações por um programa do usuário. Além disso, precisamos proteger as rotinas de atendimento de interrupção no sistema operacional contra modificações. Mesmo que o usuário não obtivesse controle não-autorizado do computador, a modificação das rotinas de atendimento de interrupção provavelmente acabaria com a operação apropriada do computador e de suas operações com spool e buffer.

Vemos, então, que é preciso oferecer proteção à memória pelo menos para o vetor de interrupção e as rotinas de atendimento de interrupção do sistema operacional. Em geral, queremos proteger o sistema operacional contra acesso pelos programas do usuário e, além disso, proteger os programas do usuário um do outro. Essa proteção precisa ser fornecida pelo hardware. Ela pode ser implementada de várias maneiras, conforme descrevemos no Capítulo 9. Aqui, esboçamos uma implementação possível.

Para separar o espaço de memória de cada programa do espaço dos outros, precisamos da capacidade de determinar o intervalo de endereços válidos que o programa pode acessar e proteger a memória fora desse espaço. Podemos oferecer esse tipo de proteção usando dois registradores, normalmente uma base e um limite, conforme ilustramos na Figura 2.9. 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 420940, inclusive.

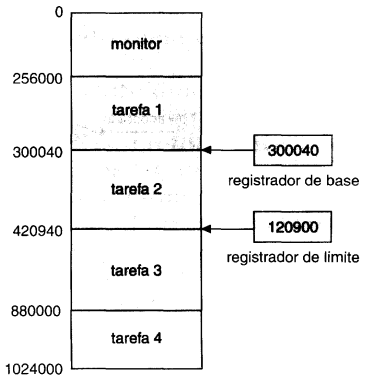



FIGURA 2.9 *Um registrador de base e um registrador de limite definem um espaço de endereço lógico.*

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 2.10). Esse esquema impede que o programa do usuário (acidental ou deliberadamente) modifique o código ou as estruturas de dados do sistema operacional ou de outros usuários.

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 monitor 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 monitor, é dado acesso irrestrito à memória do monitor e dos usuários. 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.

### 2.5.4 Proteção da CPU

Além de proteger E/S e memória, temos de garantir que o sistema operacional mantenha o controle.

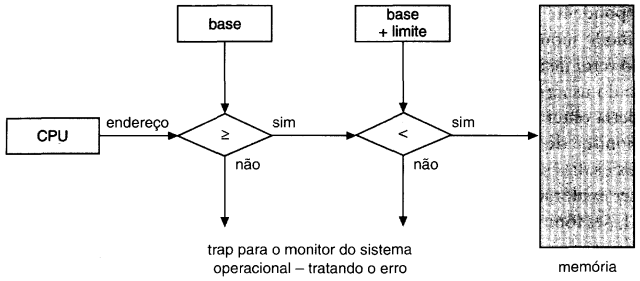



FIGURA 2.10 *Proteção de endereços de hardware com registradores de base e limite.*

Precisamos impedir que um programa do usuário, por exemplo, fique preso em um loop infinito 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 pode 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.

Um uso mais comum de um temporizador é para implementar o compartilhamento de tempo. No caso mais simples, o temporizador poderia ser definido para interromper a cada $N$ milissegundos, onde $N$ é a **fatia de tempo (time slice)** durante a qual cada usuário tem permissão para executar antes de o próximo usuário obter o controle da CPU. O sistema operacional é chamado ao final de cada fatia de tempo para realizar diversas tarefas de manutenção, como acrescentar o valor $N$ ao registro que especifica (para fins de contagem) o período durante o qual o programa do usuário executou até aqui. O sistema operacional também salva registradores, variáveis internas e buffers e altera diversos outros parâmetros para se preparar para a execução do próximo programa. Esse procedimento é conhecido como **troca de contexto (context switch)**; explicado no Capítulo 4. Após uma troca de contexto, o próximo programa continua com sua execução a partir do ponto em que parou (quando a fatia de tempo anterior foi esgotada).

Outro uso do temporizador é para calcular a hora atual. Uma interrupção do temporizador sinaliza a passagem de algum período, permitindo que o sistema operacional calcule a hora atual em referência a alguma hora inicial. Se tivermos interrupções a cada 1 segundo, por exemplo, e tivermos tido 1.427 interrupções desde que fomos informados que era 1:00 da tarde, então podemos calcular que a hora atual é 1:23:47 da tarde. Alguns computadores de-

terminam a hora atual dessa maneira, mas os cálculos precisam ser feitos com cuidado para que o tempo seja mantido com precisão, pois o tempo de processamento da interrupção (e outros tempos, quando as interrupções estão desativadas) costuma atrasar o relógio de software. A maioria dos computadores possui um relógio de hardware separado, para definir a hora do dia, que é independente do sistema operacional.

## 2.6 Estrutura de rede

Nossa discussão até agora abordou computadores isolados. Entretanto, conectando dois ou mais computadores a uma rede, pode-se formar um sistema distribuído. Tal sistema pode oferecer um rico conjunto de serviços aos usuários, conforme explicado na Seção 1.5. Nesta seção, vamos explorar duas técnicas para a construção de redes de computador.

Conforme discutimos no Capítulo 1, existem basicamente dois tipos de redes: redes locais (LAN – Local Area Network) e redes de longa distância (WAN – Wide Area Network). 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 da União). Essas diferenças implicam grandes variações na velocidade e na confiabilidade da rede de comunicações e são refletidas no projeto do sistema operacional distribuído.

### 2.6.1 Redes locais

As redes locais surgiram no início da década de 1970, como um substituto 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.

Em geral, as LANs são usadas em um ambiente de escritório (e até mesmo em um ambiente residencial). 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. Em geral, os cabos de LAN são usados 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 (cobre) 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 rede de rádio local Bluetooth, até 10 gigabits por segundo, para rede Ethernet a 10 gigabits. Dez megabits por segundo é muito comum e é a velocidade da **10BaseT Ethernet**. 100BaseT Ethernet, que trabalha a 100 megabits por segundo, 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. O uso da gigabit Ethernet (1000BaseT) para servidores de enlace dentro de um centro de dados está se espalhando, assim como o uso do acesso sem fio, que não envolve um cabeamento físico. Os dois esquemas mais comuns para os sistemas sem fio são 802.11b e 802.11g, que trabalham a 11 Mbps e 54 Mbps, respectivamente.

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 oferecem acesso a outras redes (Figura 2.11). 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.

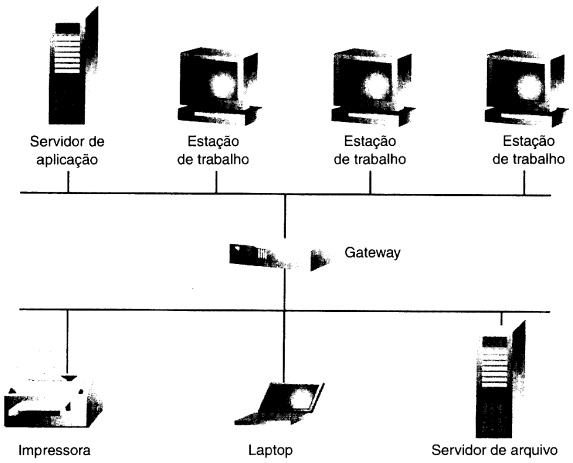



FIGURA 2.11 *Uma rede local.*

### 2.6.2 Redes remotas

As redes remotas surgiram no final da década de 1960, principalmente como um projeto acadêmico de pesquisa para oferecer comunicação eficiente entre as instalações, permitindo que o hardware e o software sejam 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 (as taxas de transmissão variam atualmente de 56K bits por segundo a mais de 1 megabit por segundo) e, às vezes, pouco confiáveis. Os enlaces típicos são linhas telefônicas, linhas de dados dedicadas, enlaces de microondas e canais de satélite. Esses enlaces de comunicação são controlados por **processadores de comunicação** especiais (Figura 2.12), responsáveis por definir a interface por meio da qual as instalações se comunicam pela rede, bem como transferir informações entre as diversas instalações.

Considere a Internet como um exemplo. Os computadores host que se comunicam por meio da Internet 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 15.3.2) para formar uma rede mundial. As conexões entre as redes utilizam um serviço do sistema telefônico chamado T1, que oferece 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 T1s, que atuam em paralelo, para oferecer 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

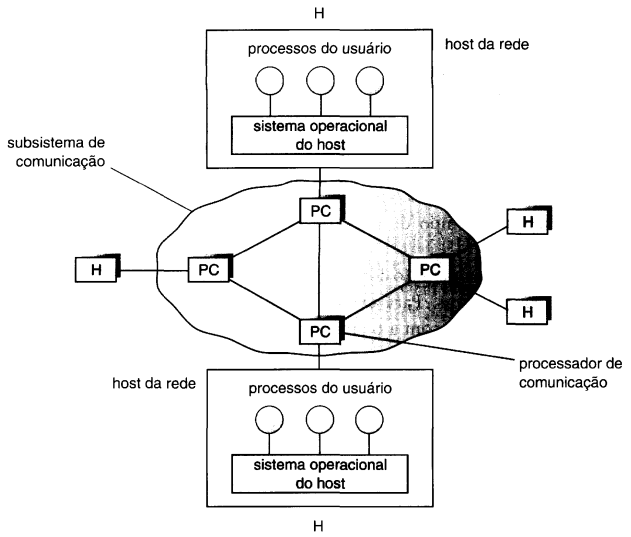



FIGURA 2.12 *Processadores de comunicação em uma rede remota.*

lado 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 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 seu destino (mensagens privadas). UUCP foi substituído por PPP, o Point-to-Point Protocol. PPP funciona por meio de conexões de modem, permitindo que computadores domésticos sejam totalmente conectados à Internet. Existem muitos outros métodos de conexão, incluindo DSL e modem a cabo. O acesso varia, dependendo do local físico e das ofertas do provedor de serviços, custo e velocidade.

## 2.7 Resumo

Multiprogramação e sistemas de tempo compartilhado (time-sharing) melhoram o desempenho sobrepondo operações de CPU e E/S em uma única máquina. Essa sobreposição exige que a transferência de dados entre a CPU e um dispositivo de E/S seja tratada por consulta (polling) ou controlado por interrupção (interrupt-driven) a uma porta de E/S, ou por uma transferência de dados com DMA.

Para um computador executar programas, os programas precisam estar na memória principal. A memória principal é a única área de armazenamento grande que o processador pode acessar diretamente. Ela é um conjunto de words ou bytes, variando em tamanho desde milhões até bilhões. Cada word possui seu próprio endereço. A memória principal é um dispositivo de armazenamento volátil, que perde seu conteúdo quando a energia é desligada ou perdida. A maioria dos computadores oferece armazenamento secundário como uma extensão da memória principal. O principal requisito do armazenamento secundário é a capacidade de manter grandes quantidades de dados permanentemente. O dispositivo de armazenamento secundário mais comum é o disco magnético, que oferece armazenamento de programas e dados. Um disco magnético é um dispositivo de armazenamento não volátil, que também oferece acesso aleatório. As fitas magnéticas são usadas principalmente para backup, para armazenamento

de informações usadas com pouca freqüência e como um meio de transferir informações de um sistema para outro.

A grande variedade de sistemas de armazenamento em um computador pode ser organizada em uma hierarquia, de acordo com sua velocidade e custo. Os níveis mais altos são caros, mas são rápidos. À medida que descemos na hierarquia, o custo por bit geralmente diminui, enquanto o tempo de acesso aumenta.

O sistema operacional garante a operação correta do computador de diversas maneiras. Para evitar que programas do usuário interfiram com a operação correta do sistema, o hardware possui dois modos: modo usuário e modo monitor. Diversas instruções, como instruções de E/S e instruções de parada (halt), são privilegiadas e só podem ser executadas no modo monitor. A memória em que o sistema operacional reside é protegida contra modificações pelo usuário. Um temporizador evita loops infinitos. Essas facilidades – modo dual, instruções privilegiadas, proteção de memória e interrupção com temporizador – são blocos de montagem básicos, utilizados pelo sistema operacional para conseguir a operação correta. O Capítulo 3 continua essa discussão com os detalhes das facilidades oferecidas pelos sistemas operacionais.

LANs e WANs são os dois tipos básicos de redes. As LANs, normalmente conectadas por cabeamento caro de par trançado ou fibra óptica, permitem a comunicação entre os processadores distribuídos por uma área geográfica pequena. As WANs, conectadas por linhas telefônicas, linhas alugadas, enlaces de microondas ou canais de satélite, permitem a comunicação entre processadores distribuídos por uma área geográfica maior. As LANs normalmente transmitem mais de 100 megabits por segundo, enquanto as WANs mais lentas transmitem de 56K bits por segundo a mais de 1 megabit por segundo.

## Exercícios

**2.1** *Prefetching* é um método de sobreposição da E/S de uma tarefa com a própria computação dessa tarefa. A idéia é simples. Depois que uma operação de leitura terminar e a tarefa estiver para começar a operar os dados, o dispositivo de entrada é instruído a iniciar a próxima leitura imediatamente. A CPU e o dispositivo de saída ficam ocupados. Com sorte, quando a tarefa estiver pronta para o próximo item de dados, o dispositivo de entrada terá terminado a leitura desse item de dados. A CPU pode, então, começar a processar os dados lidos, enquanto o dispositivo de entrada começa a ler os dados seguintes. Uma idéia semelhante pode ser usada para a saída. Nesse caso, a tarefa cria dados que são colocados em um buffer até que um dispositivo de saída possa aceitá-los.

Compare o esquema de prefetching com o esquema *spooling*, no qual a CPU sobrepõe a entrada de uma tarefa com a computação e a saída de outras tarefas.

**2.2** Como a distinção entre modo monitor e modo usuário funciona como uma forma rudimentar de sistema de proteção (segurança)?

**2.3** Quais são as diferenças entre um trap e uma interrupção? Para que é usada cada função?

**2.4** Para que tipos de operações o DMA é útil? Explique sua resposta.

**2.5** Quais das seguintes instruções deverão ser privilegiadas?

**a.** Definir o valor do temporizador.
**b.** Ler o relógio.
**c.** Apagar a memória.
**d.** Desativar interrupções.
**e.** Passar do modo usuário para o modo monitor.

**2.6** Alguns sistemas de computador não provêem um modo de operação privilegiado no hardware. É possível construir um sistema operacional seguro para esses computadores? Dê argumentos mostrando que isso é e não é possível.

**2.7** Alguns computadores antigos protegiam o sistema operacional colocando-o em uma partição da memória que não poderia ser modificada pela tarefa do usuário ou pelo próprio sistema operacional. Descreva duas dificuldades que poderiam surgir com esse esquema.

**2.8** A proteção do sistema operacional é crucial para garantir que o sistema computadorizado opere corretamente. A provisão dessa proteção é o motivo para a operação no modo dual, a proteção de memória e o temporizador. Porém, para permitir o máximo de flexibilidade, você também deverá colocar o mínimo de restrição sobre o usuário.

A seguir, vemos uma lista de instruções que normalmente estão protegidas. Qual é o conjunto *mínimo* de instruções que precisam ser protegidas?

**a.** Passar para o modo usuário.
**b.** Passar para o modo monitor.
**c.** Ler da memória do monitor.
**d.** Escrever na memória do monitor.

**e.** Buscar uma instrução da memória do monitor.

**f.** Ativar a interrupção por temporizador.

**g.** Desativar a interrupção por temporizador.

**2.9** Indique dois motivos pelos quais os caches são úteis. Que problemas eles resolvem? Que problemas causam? Se um cache puder ser tão grande quanto o dispositivo para o qual está dando suporte (por exemplo, um cache tão grande quanto um disco), por que não mantê-lo com esse tamanho e eliminar o dispositivo?

**2.10** Escrever um sistema operacional que possa operar sem interferência de programas de usuário maliciosos ou não depurados exige assistência do hardware. Cite três recursos do hardware para a escrita de um sistema operacional e descreva como poderiam ser usados em conjunto para proteger o sistema operacional.

**2.11** Algumas CPUs oferecem mais de dois modos de operação. Quais são dois usos possíveis para esses múltiplos modos?

**2.12** Quais são as principais diferenças entre uma WAN e uma LAN?

**2.13** Que configuração de rede seria mais adequada para os seguintes ambiente?

**a.** Um pavimento de dormitório.

**b.** Um campus universitário.

**c.** Um estado.

**d.** Uma nação.

## Notas bibliográficas

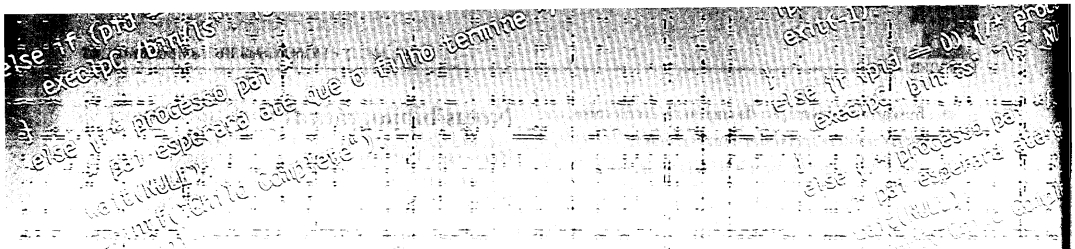
Hennessy e Patterson [2002] provêem uma cobertura sobre sistemas de E/S e barramentos, além de arquitetura de sistemas em geral. Tanenbaum [1990] descreve a arquitetura dos microcomputadores, começando em um nível detalhado de hardware.

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]. Discussões sobre disquetes podem ser encontradas em Pechura e Schoeffler [1983] e em Sarisky [1983].

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

Discussões gerais com relação à tecnologia de armazenamento de massa são oferecidas por Chi [1982] e por Hoagland [1985].

Tanenbaum [2003] e Halsall [1992] apresentam introduções gerais às redes de computadores. Fortier [1989] apresenta uma discussão detalhada sobre hardware e software de rede.



CAPÍTULO 3

# Estruturas do sistema operacional

Um sistema operacional oferece 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. É importante que os objetivos do sistema sejam muito bem definidos antes que o projeto seja iniciado. Esses objetivos formam a base das opções entre diversos algoritmos e estratégias.

Podemos ver um sistema operacional de diversos pontos de vista. Um deles focaliza os serviços oferecidos 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. Consideramos quais serviços são oferecidos por um sistema operacional, como são fornecidos 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.

## 3.1 Componentes do sistema

Só podemos criar um sistema tão grande e complexo quanto um sistema operacional dividindo-o em partes menores. Cada uma dessas partes deverá ser um pedaço do sistema bem delineado, com entradas, saídas e funções cuidadosamente definidas. É claro que nem todos os sistemas possuem a mesma estrutura. No entanto, muitos sistemas modernos compartilham o objetivo de dar suporte aos comportamentos do sistema esboçados nas próximas oito seções.

### 3.1.1 Gerência de processos

Um programa não faz nada a menos que suas instruções sejam executadas por uma CPU. Um programa em execução pode ser considerado 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 é um processo. Uma tarefa do sistema, como o envio de saída para uma impressora, também é um processo. 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. Conforme veremos no Capítulo 4, é possível fornecer chamadas de sistema que possibilitam aos processos criar subprocessos para execução simultânea.

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 é 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.

Enfatizamos 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 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 5.) A execução de tal processo precisa ser seqüencial. 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 seqüências de execução separadas. É comum ter um programa que gera muitos processos enquanto é executado. 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:

- 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
- Prover mecanismos para o tratamento de deadlock

Discutiremos as técnicas de gerenciamento de processos nos Capítulos de 4 a 7.

### 3.1.2 Gerência da memória principal

Conforme discutimos no Capítulo 1, 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). As operações de E/S implementadas por DMA também lêem e escrevem dados na memória principal. 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 maneira, 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, temos de manter vários programas na memória, criando a necessidade de um gerenciamento de memó-

ria diferentes são utilizados. Esses esquemas refletem diversas técnicas, e a eficácia dos diferentes algoritmos depende da situação em particular. A seleção de um esquema de gerenciamento de memória para um sistema específico depende de 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 à gerência de memória:

- Registrar quais partes da memória estão sendo usadas atualmente e por quem
- Decidir quais processos devem ser carregados para a memória quando o espaço de memória se tornar disponível
- Alocar e desalocar espaço de memória conforme a necessidade

As técnicas de gerenciamento de memória serão discutidas nos Capítulos 9 e 10.

### 3.1.3 Gerência de arquivos

A gerência de arquivos é um dos comportamentos 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 (seqüencial ou aleatório).

Para o uso conveniente do sistema computadorizado, o sistema operacional oferece uma visão lógica 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.

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 um 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 para facilitar o uso. Finalmente, quando vários usuários têm acesso a arquivos, pode ser desejável controlar por quem e de que maneiras (por exemplo, leitura, escrita, inserção) eles podem ser acessados.

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

- Criação e remoção de arquivos
- Criação e remoção de diretórios
- 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 gerenciamento de arquivos serão discutidas nos Capítulos 11 e 12.

### 3.1.4 Gerência do sistema 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:

- Um componente de gerenciamento 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.

No Capítulo 2, 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.

### 3.1.5 Gerência do armazenamento secundário

A finalidade principal de um computador é executar programas. Esses programas, com os dados que acessam, precisam estar na memória principal, ou **armazenamento primário**, durante a execução. Como a memória principal é muito pequena para acomodar todos os dados e programas, e como os dados que mantém 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, o gerenciamento correto do armazenamento em disco é de importância vital para um computador. O sistema operacional é responsável pelas seguintes atividades relacionadas ao gerenciamento de disco:

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

Como o armazenamento secundário é usado com muita freqüência, 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. As técnicas para o gerenciamento do armazenamento secundário serão discutidas no Capítulo 14.

### 3.1.6 Redes

Um **sistema distribuído** é uma coleção de processadores que não compartilham memória, dispositivos periféricos ou um relógio. Em vez disso, cada processador possui sua própria memória local e relógio, e os processadores se comunicam um com o outro por meio de diversas linhas de comunicação, como barramentos de alta velocidade ou redes. 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 diferentes. A rede pode estar conectada total ou parcialmente. O projeto da rede de comunicação precisa considerar o roteamento de mensagens e estratégias de conexão, além de problemas de contenção e segurança.

Um sistema distribuído coleta sistemas fisicamente dispersos, talvez heterogêneos, em um sistema coerente, oferecendo 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. Os 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. Os protocolos que criam um sistema distribuído podem afetar bastante a utilidade e a popularidade desse sistema. A inovação da World Wide Web foi criar um novo método de acesso para o compartilhamento de informações. Ela melhorou o protocolo de transferência de arquivos (FTP) e o protocolo do sistema de arquivos de rede (NFS) existentes, evitando a necessidade de um usuário efetuar o login antes de obter acesso a um recurso remoto. Ela definiu um novo protocolo, **http**, para ser usado na comunicação entre um servidor Web e um navegador Web. Um navegador Web só precisa enviar uma requisição de informação para o servidor Web de uma máquina remota, e a informação (texto, gráficos, links para outras informações) é retornada. Esse aumento na conveniência favoreceu o imenso crescimento no uso de http e da Web em geral.

Discutiremos as redes e sistemas distribuídos, com o foco principal na computação distribuída usando Java, nos Capítulos 15 e 16.

### 3.1.7 Sistema de proteção

Se um computador possui diversos usuários e permite a execução simultânea dos diversos processos, então esses processos precisam ser protegidos um do outro. Para esse propósito, os mecanismos garantem que os arquivos, segmentos de memória, CPU e outros recursos possam ser operados somente pelos processos que tiveram autorização própria 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 programas, processos ou usuários aos recursos definidos por um sistema computadorizado. Esse mecanismo precisa oferecer 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. 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 a proteção oferece meios para distinguir entre uso autorizado e não-autorizado, conforme discutiremos no Capítulo 18.

### 3.1.8 Sistema interpretador de comandos

Um dos programas do sistema mais importantes para um sistema operacional é o **interpretador de comandos**, que é a interface entre o usuário e o sistema operacional. Alguns sistemas operacionais incluem o interpretador de comandos no kernel. Outros, como MS-DOS e UNIX, tratam o interpretador de comandos como um programa especial que está executando quando uma tarefa é iniciada ou quando um usuário efetua o logon inicialmente (em sistemas de tempo compartilhado).

Muitos comandos são dados ao sistema operacional pelas **instruções de controle**. Quando uma nova tarefa é iniciada em um sistema batch, ou quando um usuário se conecta a um sistema de tempo compartilhado, um programa que lê e interpreta instruções de controle é executado automaticamente. Esse programa às vezes é chamado de **interpretador de cartão de controle**, ou **shell**. Sua função é simples: apanhar a próxima instrução de comando e executá-la.

Os sistemas operacionais são diferenciados na área do shell, com um interpretador de comandos amigável tornando o sistema mais conveniente para alguns usuários. Um estilo de interface amigável é a janela baseada em mouse e o sistema de menus, utilizados no Macintosh e no Microsoft Windows. O mouse é movido para posicionar o ponteiro do mouse sobre imagens ou **ícones** na tela, que representam programas, arquivos e funções do sistema. Dependendo do local do ponteiro do mouse, o clique de um botão do mouse pode executar um programa, selecionar um arquivo ou diretório – conhecido como uma **pasta** – ou descer um menu que contém comandos. Shells mais poderosos, mais complexos e mais difíceis de aprender são apreciados por outros usuários. Em alguns desses shells, os comandos são digitados em um teclado e exibidos em uma tela ou terminal de impressão, com a tecla Enter (ou Return) sinalizando que um comando está completo e pronto para ser executado. Os shells do MS-DOS e do UNIX operam dessa maneira.

As instruções de comando, propriamente ditas, tratam da criação e gerenciamento de processos, tratamento de E/S, gerenciamento do armazenamento secundário, gerenciamento da memória principal, acesso ao sistema de arquivos, proteção e uso da rede.

## 3.2 Serviços do sistema operacional

Um sistema operacional oferece um ambiente para a execução de programas. Ele oferece certos serviços aos programas e aos usuários desses programas. Os serviços específicos oferecidos diferem de um sistema operacional para outro, mas podemos identificar classes comuns. Esses serviços do sistema operacional são fornecidos para a conveniência do programador, para facilitar a tarefa de programação.

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

- *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 rebobinar uma unidade de fita 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 oferecer 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 do mesmo.
- *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.

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 uma unidade de fita para ser usada por uma tarefa. Uma rotina desse tipo localiza uma unidade de fita não usada e marca uma tabela interna para registrar o novo usuário da unidade. Outra rotina é usada para limpar essa tabela. Essas rotinas também podem alocar impressoras, modems e outros dispositivos periféricos.
- *Contabilidade:* Queremos registrar quais usuários utilizam quanto e que tipos de recursos do computador. Essa manutenção de registro pode ser usada para contabilidade (para que os usuários possam ser cobrados) ou simplesmente para acumular estatísticas de uso. As estatísticas de uso podem ser uma ferramenta valiosa para pesquisadores que queiram reconfigurar o sistema, a fim de aprimorar os serviços de computação.
- *Proteção e segurança:* Os proprietários das informações armazenadas em um sistema computadorizado multiusuário 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 *se-*

*gurança* 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.

## 3.3 Chamadas de sistema

As **chamadas de sistema (System Calls)** oferecem a interface entre um processo e o sistema operacional. Em geral, essas chamadas estão disponíveis como instruções em linguagem assembly e estão listadas nos manuais usados por programadores em linguagem assembly.

Determinados sistemas permitem que as chamadas de sistema sejam feitas diretamente por um programa em linguagem de mais alto nível, quando as chamadas têm a forma de chamadas de função ou sub-rotina predefinidas. Elas podem gerar uma chamada a uma rotina especial em tempo de execução (runtime), que faz a chamada de sistema, ou então a chamada de sistema pode ser gerada diretamente em linha.

Diversas linguagens – por exemplo, C e C++ – foram definidas para substituir a linguagem assembly para a programação de sistemas. Essas linguagens permitem que as chamadas de sistema sejam feitas diretamente. Por exemplo, em UNIX podem ser feitas diretamente por um programa em C ou C++. Já chamadas de sistema para plataformas modernas do Microsoft Windows, fazem parte da API Win32, que está disponível para todos os computadores escritos para Microsoft Windows.

Java não permite que chamadas de sistema sejam feitas diretamente, pois uma chamada de sistema é específica a um sistema operacional e resulta em código específico da plataforma. Contudo, se uma aplicação exige recursos específicos do sistema, um programa Java pode chamar um método escrito em outra linguagem – normalmente C ou C++ –, que pode fazer a chamada de sistema. Esses métodos são conhecidos como métodos "nativos".

Como um exemplo de como as chamadas de sistema são usadas, considere a escrita de 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 seqüência 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 seqüência 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 seqüência 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 seqüência 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, final da fita, 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). Como podemos ver, os programas podem utilizar muito o sistema operacional. Freqüentemente, os sistemas executam milhares de chamadas de sistema por segundo.

Todavia, a maioria dos programadores nunca vê esse nível de detalhe. O sistema de suporte em tempo de execução (o conjunto de funções embutidas nas bibliotecas de um compilador) para a maioria das linguagens de programação oferece uma interface muito mais simples. Por exemplo, a instrução `cout( )` em C++ provavelmente é compilada para uma chamada a uma rotina de suporte em tempo de execução, que emite as chamadas de sistema necessárias, verifica erros e, por fim, retorna ao programa do usuário. Assim, a maioria dos detalhes da interface do sistema operacional fica escondida do programador pelo compilador e pelo pacote de suporte em tempo de execução.

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 3.1). Essa é a técnica utilizada pelo Linux. 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.

As chamadas de sistema podem ser agrupadas de modo geral em cinco categorias principais: **controle de processos, manipulação de arquivos, manipulação de dispositivos, manutenção de informações** e **comunicações**. Nas próximas seções, discutiremos os tipos de chamadas de sistema oferecidas por um sistema operacional. A maioria delas admite (ou são

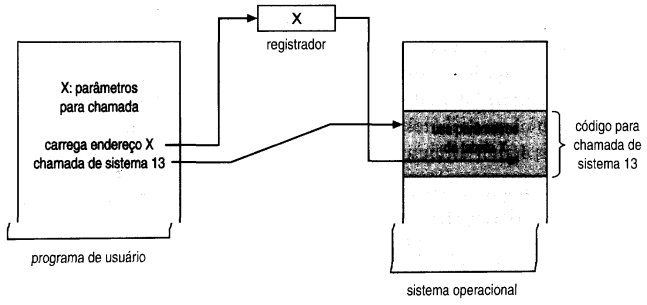



FIGURA 3.1 *Passagem de parâmetros como uma tabela.*

admitidas por) conceitos e funções discutidos em outros capítulos. A Figura 3.2 resume os tipos de chamadas de sistema fornecidos por um sistema operacional.

### 3.3.1 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 batch, o interpretador de comandos termina a tarefa e continua com a seguinte. Alguns sistemas permitem que cartões de controle indiquem ações de recuperação especiais caso

- 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 3.2 *Tipos de chamadas de sistema.*

ocorra um erro. 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 severos 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 e executar 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 simultaneamente 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 simultaneamente, 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 permissível 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 certo 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`). As chamadas de sistema desse tipo, lidando com a coordenação de processos simultâneos, serão discutidas em detalhe no Capítulo 7.

Outro conjunto de chamadas de sistema é útil na depuração de um programa. Muitos sistemas oferecem chamadas de sistema para dump de memória. Essa provisão é útil para depuração. Um rastreamento do programa lista cada instrução à medida que é executada; isso é oferecido por poucos sistemas. Até mesmo os microprocessadores oferecem um modo da CPU conhecido como *passo a passo* (single step), em que 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. Ele indica a quantidade de tempo que o programa é executado em determinada locação ou conjunto de locações. Um perfil de tempo exige uma facilidade de rastreamento ou interrupções de tempo regulares. A cada ocorrência da interrupção do temporizador, o valor do contador de programa é registrado. Com interrupções de temporizador suficientemente freqüentes, uma imagem estatística do tempo gasto nas diversas partes do programa poderá ser obtida.

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, que possui um interpretador de comandos chamado quando o computador é iniciado (Figura 3.3(a)). 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 3.3(b)). 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 ta-

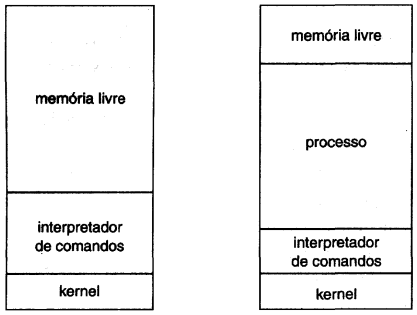



FIGURA 3.3 *Execução do MS-DOS: (a) no boot do sistema; (b) executando um programa.*

refa 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.

Embora o sistema operacional MS-DOS não tenha capacidade para multitarefa, ele oferece um método para a execução simultânea limitada. Um programa TSR é um programa que "prende uma interrupção" e depois sai com a chamada de sistema `terminate and stay resident`. Por exemplo, ele pode prender a interrupção de relógio colocando o endereço de uma de suas sub-rotinas na lista de rotinas de interrupção a serem chamadas quando o temporizador do sistema for disparado. Desse modo, a rotina TSR será executada várias vezes por segundo, a cada batida do relógio. A chamada de sistema `terminate and stay resident` faz com que o MS-DOS reserve o espaço ocupado pelo TSR, de modo que não seja apagado quando o interpretador de comandos for recarregado.

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 solicitados pelo usuário. No entanto, como o FreeBSD é um sistema multitarefa, o interpretador de comandos pode continuar executando enquanto outro programa é executado (Figura 3.4). 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 solicita 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 do mouse e janelas. Nesse ínterim, o usuário está livre para pedir ao shell para executar 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 4, com um exemplo de programa usando as chamadas de sistema `fork( )` e `exec( )`.

### 3.3.2 Gerência de arquivos

O sistema de arquivos será discutido com mais detalhes nos Capítulos 11 e 12. 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 atribu-

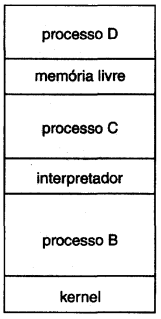



FIGURA 3.4 *FreeBSD executando vários programas.*

tos 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 oferecem muitas outras chamadas.

### 3.3.3 Gerência de dispositivos

Um programa em execução pode precisar de recursos adicionais para prosseguir – mais memória, unidades de fita, acesso a arquivos, e assim por diante. Se os recursos estiverem disponíveis, eles poderão ser concedidos, e o controle pode ser retornado ao programa do usuário. Caso contrário, o programa terá de esperar até haver recursos suficientes. Os arquivos podem ser imaginados como dispositivos abstratos ou virtuais. Assim, muitas das chamadas de sistema para arquivos também são necessárias para os dispositivos. Entretanto, se houver vários usuários no sistema, primeiro precisamos requisitar o dispositivo, para garantir seu uso exclusivo. Depois de terminar de usar o dispositivo, temos de liberá-lo. Essas funções são semelhantes às chamadas de sistema `open` e `close` para arquivos.

Quando o dispositivo tiver sido solicitado (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 e MS-DOS, 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.

### 3.3.4 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 data atuais. Outras chamadas de sistema podem retornar informações sobre o sistema, como o número de usuários atuais, o número de versão do sistema operacional, a quantidade livre de memória ou espaço em disco, e assim por diante.

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 4.1.3, discutiremos quais informações são mantidas.

### 3.3.5 Comunicações

Existem dois modelos de comunicação: o modelo de troca de mensagens e o modelo de memória compartilhada. No **modelo de troca de mensagens**, as informações são trocadas por meio de uma facilidade de comunicação entre processos, fornecida pelo sistema operacional. 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 host*, como um nome IP, 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 map memory para 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 e o local dos dados são determinados por esses processos e não estão 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 7. No Capítulo 5, 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. Os dois modelos de comunicação são comparados na Figura 3.5. No Capítulo 4, veremos uma implementação Java de cada modelo.

## 3.4 Programas do sistema

Outro aspecto de um sistema moderno é a coleção de programas do sistema. 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 oferecem um ambiente conveniente para desenvolvimento e execução de programas. Alguns deles são simplesmente interfaces do usuário para chamadas de sistema; outros são bem mais complexos. Eles podem ser divididos nestas categorias:

- *Gerenciamento de arquivos:* Esses programas criam, removem, copiam, renomeiam, imprimem, fazem dump, listam e geralmente manipulam arquivos e diretórios.

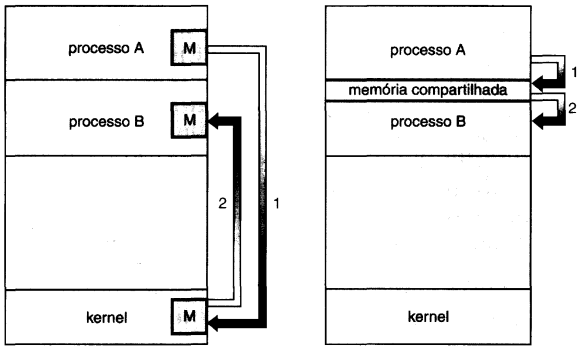



FIGURA 3.5 *Modelos de comunicação: (a) troca de mensagens; (b) memória compartilhada.*

- *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. Essas informações são formatadas e impressas no terminal ou outro dispositivo de saída ou arquivo.
- *Modificação de arquivos:* Diversos editores de textos podem estar disponíveis para criar e modificar o conteúdo dos arquivos armazenados em disco ou fita.
- *Suporte para linguagem de programação:* Compiladores, montadores (assemblers) 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, embora alguns desses programas agora sejam fornecidos e vendidos separados.
- *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 oferecem 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 estão navegadores Web, processadores e formatadores de textos, planilhas, sistemas de banco de dados, compiladores, pacotes gráficos e de análise estatística, e jogos. Esses programas são conhecidos como **utilitários do sistema, programas de aplicação** ou **programas aplicativos.**

Talvez o programa do sistema mais importante para um sistema operacional seja o **interpretador de comandos.** Sua função principal é apanhar e executar o próximo comando especificado pelo usuário. Muitos dos comandos dados nesse nível manipulam arquivos: `create`, `delete`, `list`, `print`, `copy`, `execute` etc. Existem duas maneiras gerais de implementar esses comandos.

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 desviar para uma seção do código que prepara 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 exige 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 simplesmente usa o comando para identificar um arquivo a ser carregado na memória e executado. Assim, o comando do UNIX para excluir um arquivo

`rm` *G*

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

Embora essa segunda técnica para o projeto do interpretador de comandos ofereça vantagens, ela também possui problemas. Observe primeiro que, como o código para executar um comando é um programa do sistema separado, o sistema operacional precisa oferecer um mecanismo para passar parâmetros do interpretador de comandos para o programa do sistema. Essa tarefa pode ser confusa, pois o interpretador de comandos e o programa do sistema podem não estar na memória ao mesmo tempo, e a lista de parâmetros pode ser grande. Além disso, é mais lento carregar um programa e executá-lo do

que desviar para outra seção do código dentro do programa atual.

Outro problema é que a interpretação dos parâmetros é deixada para o programador do programa do sistema. Assim, os parâmetros podem ser fornecidos de forma incoerente entre os programas que parecem ser semelhantes ao usuário, mas que foram escritos em ocasiões diferentes por programadores diferentes.

A visão do sistema operacional vista pela maioria dos usuários, portanto, é definida pelos programas do sistema, e não pelas chamadas de sistema reais. Considere os PCs. Quando seu computador está executando o sistema operacional Microsoft Windows, um usuário pode ver um shell do MS-DOS na linha de comandos ou a interface gráfica com mouse e janelas. Ambos utilizam o mesmo conjunto de chamadas de sistema, mas as chamadas aparecem de forma diferente e atuam de maneiras diferentes. Como conseqüência, essa visão do usuário pode estar distante da estrutura real do sistema. O projeto de uma interface útil e amigável com o usuário, portanto, não é uma função direta do sistema operacional. Neste livro, vamos nos concentrar nos problemas fundamentais relacionados à dificuldade de oferecer um serviço adequado aos programas do usuário. Do ponto de vista do sistema operacional, não distinguimos entre os programas do usuário e os programas do sistema.

## 3.5 Estrutura do sistema

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 particionar 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 (Seção 3.1). Nesta seção, discutimos como esses componentes são interconectados e ligados ao kernel.

### 3.5.1 Estrutura simples

Muitos sistemas 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. MS-DOS é um exemplo desse sistema. Ele foi projetado e implementado originalmente por algumas pessoas que não tinham idéia de que ele se tornaria tão popular. Ele foi escrito para fornecer o máximo de funcionalidade no menor espaço possível (devido ao hardware limitado em que era executado), de modo que não foi dividido com cuidado em módulos. A Figura 3.6 mostra sua estrutura.

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 oferece um modo dual e nenhuma proteção do hardware, os projetistas do MS-DOS não tinham outra escolha além de deixar o hardware básico acessível.

Outro exemplo de estruturação limitada é o sistema operacional UNIX original. O UNIX é outro sistema que, 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

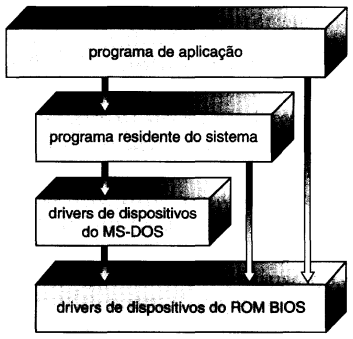



FIGURA 3.6 *Estrutura de camadas do MS-DOS.*

evoluía. Podemos ver o sistema operacional UNIX tradicional como sendo em camadas, como mostra a Figura 3.7. Tudo abaixo da interface de chamada de sistema e acima do hardware físico é o kernel. O kernel oferece o sistema de arquivos, escalonamento de CPU, 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. Os programas do sistema utilizam as chamadas de sistema que possuem suporte do kernel para oferecer funções úteis, como compilação e manipulação de arquivos.

As chamadas de sistema definem a **interface de programa de aplicação** (API – Application Program Interface) para o UNIX; o conjunto de programas do sistema disponíveis define a **interface com o usuário.** As interfaces do programa e do usuário definem o contexto ao qual o kernel precisa dar suporte.

As novas versões do UNIX são projetadas para utilizar um hardware mais avançado. 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.

### 3.5.2 Enfoque em camadas

Um sistema pode ser criado 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.

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* – é representada na Figura 3.8. Ela 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 **modularidade.** As camadas são selecionadas de modo que cada uma utilize funções (operações) e serviços apenas de camadas de nível mais baixo. Essa téc-

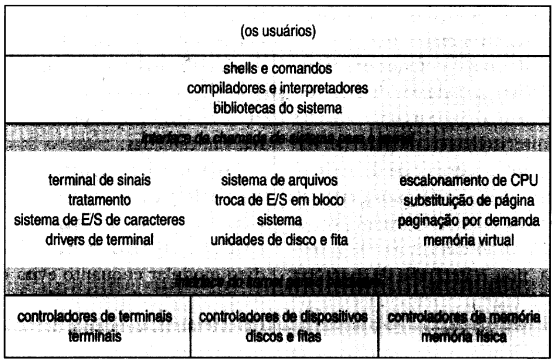



FIGURA 3.7 *Estrutura do sistema UNIX.*

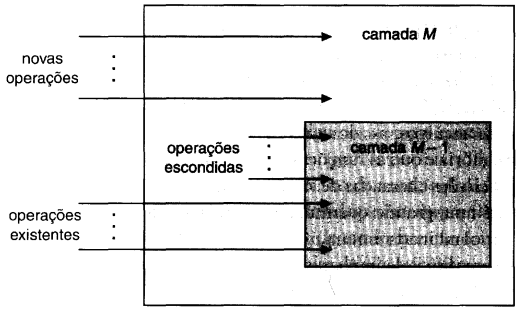



FIGURA 3.8 *Uma camada do sistema operacional.*

nica 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 (que é 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 quando o sistema está dividido em camadas.

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 gerenciamento de memória, pois o 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, oferecendo a maioria das vantagens do código modular enquanto evita os problemas difíceis da definição e interação da camada. Por exemplo, o OS/2 é um descendente do MS-DOS que acrescenta multitarefa e operação no modo dual, além de outros recursos. Devido a essa complexidade adicional

e ao hardware mais poderoso para o qual o OS/2 foi projetado, o sistema foi implementado de uma maneira mais em camadas. Compare a estrutura do MS-DOS com a que mostramos na Figura 3.9; pelos pontos de vista do projeto do sistema e da implementação, o OS/2 leva vantagem. Por exemplo, o acesso direto do usuário a facilidades de baixo nível não é permitido, oferecendo ao sistema operacional mais controle sobre o hardware e mais conhecimento de quais recursos cada programa do usuário está utilizando.

### 3.5.3 Microkernels

Conforme 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 oferecem gerenciamento mínimo de processo e memória, além de uma facilidade de comunicação.

A função principal do microkernel é oferecer uma 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 3.3.5. 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.

Os benefícios da técnica de microkernel incluem a facilidade de extensão do sistema operacional. Todos os novos serviços são acrescentados ao espaço do usuário e, como conseqüência, não exigem modificação do kernel. Quando o kernel tiver 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 oferece mais segurança e confiabilidade, pois a maioria dos serviços está executando como processos do usuário – em vez do kernel. Se um serviço falhar, o restante do sistema operacional permanece intocável.

Vários sistemas operacionais contemporâneos usaram a técnica de microkernel. O Tru64 UNIX

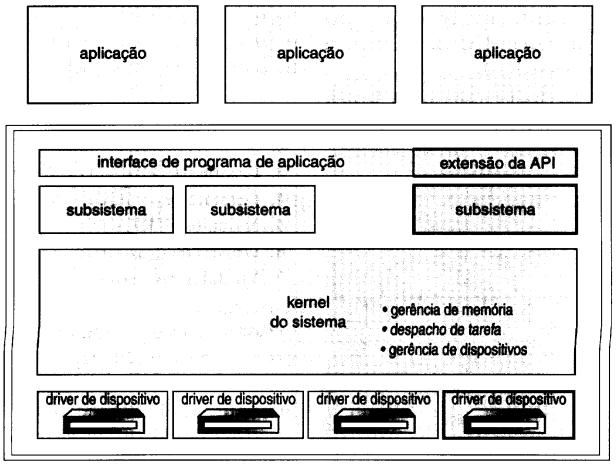



FIGURA 3.9 · *Estrutura em camadas do OS/2.*

(originalmente Digital UNIX) oferece 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.

QNX é um sistema operacional de tempo real baseado no projeto do microkernel. O microkernel do QNX oferece 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 ofereceu um desempenho baixo em comparação com o do 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.

O sistema operacional OS X do Apple Macintosh utiliza uma estrutura híbrida. O OS X (também conhecido como *Darwin*) estrutura o sistema operacional usando uma técnica de camadas, na qual uma camada consiste no microkernel Mach. A estrutura do OS X aparece na Figura 3.10.

As camadas superiores incluem ambientes de aplicação e um conjunto de serviços oferecendo uma 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 oferece gerência de memória; suporte para remote procedures calls (RPCs) e comunicação entre processos (IPC – Interprocess communication), incluindo troca de mensagens e escalonamento de threads. O componente BSD oferece 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 oferece um kit de E/S para o desenvolvimento de drivers de dispositivos e módulos carregáveis dinamicamente (os 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.

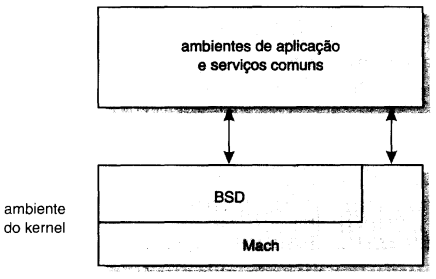



FIGURA 3.10 *A estrutura do OS X.*

## 3.5.4 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 dinamicamente 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 3.11, é organizada em torno de um kernel básico com sete tipos de módulos de kernel carregáveis:

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

Esse projeto permite que o kernel ofereça serviços básicos e também permite que certos 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 carre-

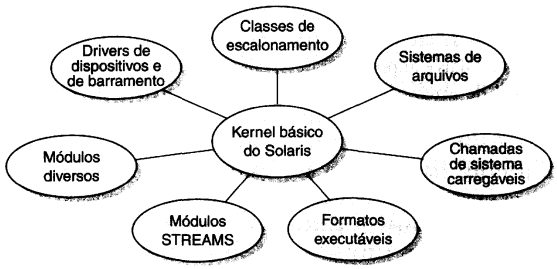



FIGURA 3.11 *Módulos carregáveis do Solaris.*

gáveis. 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.

## 3.6 Máquinas virtuais

A técnica de camadas é levada à sua conclusão lógica no conceito de uma **máquina virtual**. O sistema operacional VM para sistemas IBM é o melhor exemplo do conceito de máquina virtual, pois a IBM realizou um trabalho pioneiro nessa área.

Usando técnicas de escalonamento de CPU (Capítulo 6) e de memória virtual (Capítulo 10), um sistema operacional pode criar a ilusão de que um processo possui seu próprio processador com sua própria memória (virtual). Normalmente, um processo possui recursos adicionais, como chamadas de sistema e um sistema de arquivos, que não são fornecidos pelo hardware puro. A técnica de máquina virtual não oferece tal funcionalidade adicional, mas oferece uma interface *idêntica* à do hardware básico. Cada processo recebe uma cópia (virtual) do computador básico (Figura 3.12).

O computador físico compartilha recursos para criar as máquinas virtuais. O escalonamento de CPU pode compartilhar a CPU para criar a aparência de

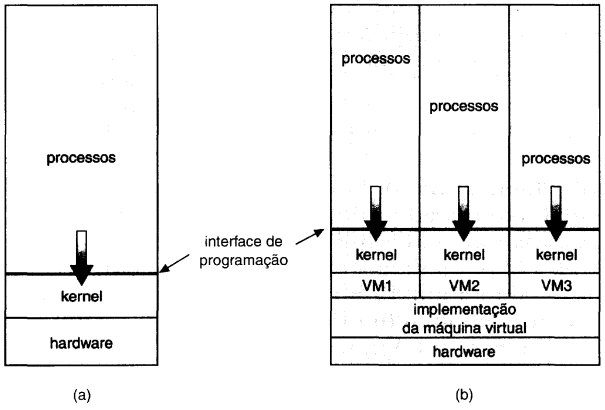



FIGURA 3.12 *Modelos de sistema: (a) máquina não-virtual; (b) máquina virtual.*

que os usuários possuem seus próprios processadores. O spooling e um sistema de arquivos podem oferecer leitoras de cartões virtuais e impressoras de linha virtuais. Um terminal de usuário normal, de tempo compartilhado, oferece a função do console do operador da máquina virtual.

Uma dificuldade importante com a técnica de máquina virtual envolve os sistemas de disco. Suponha que a máquina tenha três unidades de disco, mas queira dar suporte a sete máquinas virtuais. Logicamente, ela não pode alocar uma unidade de disco a cada máquina virtual, pois o próprio software da máquina virtual precisará de um espaço de disco substancial para oferecer memória virtual e spooling. A solução é oferecer discos virtuais – chamados de *minidiscos* no sistema operacional VM da IBM –, que são idênticos em todos os aspectos, exceto o tamanho. O sistema implementa cada minidisco alocando tantas trilhas nos discos físicos quanto o minidisco precisar. Obviamente, a soma dos tamanhos de todos os minidiscos terá de ser menor do que o tamanho do espaço disponível no disco físico.

Assim, os usuários recebem suas próprias máquinas virtuais. Eles podem, então, executar qualquer um dos sistemas operacionais ou pacotes de software que estiverem disponíveis na máquina. Para o sistema VM da IBM, um usuário normalmente executa o CMS – um sistema operacional interativo monousuário. O software da máquina virtual trata da multiprogramação de várias máquinas virtuais em uma máquina física, mas não precisa considerar qualquer software de suporte ao usuário. Esse arranjo pode oferecer um modo útil de dividir o problema de projetar um sistema interativo multiusuário em duas partes menores.

### 3.6.1 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 monitor. O software da máquina virtual pode ser executado no modo monitor, 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 conseqüência, precisamos ter um modo usuário virtual e um modo monitor virtual, ambos executando em um modo de usuário físico. As ações que causam uma transferência do modo usuário para o modo monitor 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 monitor 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 monitor virtual. Se a máquina virtual tentar, por exemplo, ler de sua leitora de cartões virtual, ela executará uma instrução de E/S privilegiada. Como a máquina virtual está executando no modo usuário físico, essa instrução será interceptada para o monitor da máquina virtual. O monitor da máquina virtual precisará então simular o efeito da instrução de E/S. Primeiro, ele encontra o arquivo de spool que implementa a leitora de cartões virtual. Depois, ele traduz a leitura da leitora de cartões virtual em uma leitura no arquivo de disco em spool e transfere a próxima "imagem de cartão" virtual para a memória virtual da máquina virtual. Finalmente, ele pode reiniciar a máquina virtual, cujo estado foi modificado como se a instrução de E/S tivesse sido executada com uma leitora de cartões real para uma máquina real, executando no modo monitor real.

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 oferecer uma máquina virtual verdadeira. 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.

### 3.6.2 Benefícios

O conceito de máquina virtual possui diversas vantagens. Observe que, nesse ambiente, existe proteção completa dos diversos recursos do sistema. Cada máquina virtual é isolada de todas as outras máquinas virtuais, de modo que não existem problemas de proteção. Todavia, ao mesmo tempo, não existe compartilhamento direto de recursos. Duas técnicas para oferecer compartilhamento foram implementadas. Primeiro, é possível compartilhar um minidisco e, portanto, compartilhar arquivos. Esse esquema é modelado em um disco físico compartilhado, mas é implementado pelo software. Segundo, é possível definir uma rede de máquinas virtuais, cada uma podendo enviar informações pela rede de comunicações virtual. Novamente, a rede é modelada nas redes de comunicação físicas, mas é implementada no software.

Esse 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 monitor, uma mudança errada em um ponteiro poderia causar um erro que destruiria o sistema de arquivos inteiro. Assim, é preciso testar todas as mudanças ao 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.

### 3.6.3 Exemplos

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. Por exemplo, existem milhares de programas disponíveis para Microsoft Windows em sistemas baseados em CPU Intel. Fornecedores de computadores, como a Sun Microsystems, utilizam outros processadores, mas gostariam que seus clientes pudessem executar essas aplicações para Windows. A solução é criar uma máquina Intel virtual em cima do processador nativo. Um programa para Windows é executado nesse ambiente, e suas instruções Intel são traduzidas para o conjunto de instruções nativo. O Microsoft Windows também é executado nessa máquina virtual, de modo que o programa pode fazer suas chamadas de sistema normalmente. O resultado final é um programa que parece estar sendo executado em um sistema baseado na Intel, mas na realidade está executando em um processador diferente. Se o processador for suficientemente rápido, o programa do Windows será executado com rapidez, embora cada instrução esteja sendo traduzida para diversas instruções nativas, para fins de execução. De modo semelhante, o Apple Macintosh baseado no PowerPC inclui uma máquina virtual Motorola 68000 para permitir a execução de códigos binários escritos para o Macintosh mais antigo, baseado no processador 68000. Infelizmente, quanto mais complexa for a máquina simulada, mais difícil será montar uma máquina virtual precisa e mais lentamente ela será executada.

Um exemplo mais recente surgiu com o crescimento do sistema operacional Linux. Agora existem máquinas virtuais que permitem que aplicações Windows executem em computadores baseados em Li-

nux. A máquina virtual executa a aplicação Windows e o sistema operacional Windows.

Como veremos em seguida, a plataforma Java é executada em uma máquina virtual, em cima de um sistema operacional de qualquer um dos tipos de projeto discutidos anteriormente. Assim, os métodos de projeto de sistema operacional – simples, em camadas, microkernel, módulos e máquinas virtuais – não são mutuamente exclusivos.

## 3.7 Java

Java é uma tecnologia introduzida pela Sun Microsystems em 1995. Vamos nos referir a ela como uma *tecnologia*, em vez de uma linguagem de programação, porque ela oferece mais do que uma linguagem de programação convencional. A tecnologia Java consiste em três componentes essenciais:

1. Especificação da linguagem de programação
2. Interface de programa de aplicação (API)
3. Especificação de máquina virtual

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

### 3.7.1 Linguagem de programação

A linguagem Java pode ser mais bem caracterizada como uma linguagem de programação orientada a objeto, independente de arquitetura, distribuída e multithread. 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). 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. Java também oferece suporte de alto nível para objetos em rede e distribuídos. Essa é uma linguagem multithread, significando que um programa Java pode ter várias threads distintas. Abordamos os objetos distribuídos usando a chamada de método remoto (RMI) da Java no Capítulo 4, e discutiremos os programas Java dotados de múltiplas threads, ou fluxos, de controle no Capítulo 5.

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 19. Java 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.

### 3.7.2 API

A API Java consiste em três subconjuntos:

1. Uma API padrão para projetar aplicações desktop e applets com suporte básico da linguagem para gráficos, E/S, utilitários e redes
2. Uma API empresarial para o projeto de aplicações de servidor, com suporte para bancos de dados e aplicações no servidor (conhecidas como **servlets**)
3. Uma API para dispositivos pequenos, como computadores portáteis, pagers e telefones celulares

Neste texto, vamos nos concentrar na API padrão.

### 3.7.3 Máquina virtual

A 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, conforme diagramado na Figura 3.13. O carregador de classes carrega arquivos `.class` do programa Java e da API Java para execução pelo interpretador Java. O interpretador Java pode ser um interpretador de software que interpreta os códigos de byte um de cada vez, ou então pode ser um compilador **just-in-time** (JIT), que transforma os códigos de bytes independentes de arquitetura em linguagem de máquina nativa ao computador host. Em outros casos, o interpretador pode ser implementado em um chip de hardware que executa códigos de bytes Java em formato nativo.

A **plataforma Java** consiste na JVM e na API Java; ela aparece na Figura 3.14. A plataforma Java pode ser implementada em cima de um sistema operacio-

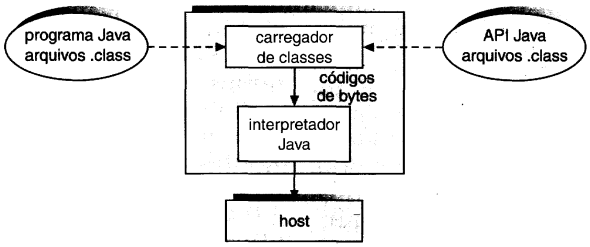



FIGURA 3.13 *A máquina virtual Java.*

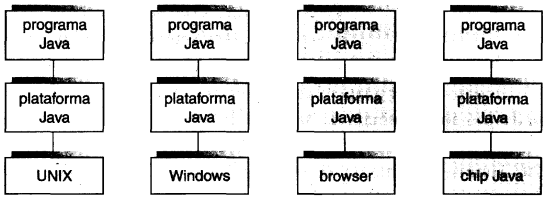



FIGURA 3.14 *A plataforma Java.*

nal host, como UNIX ou Windows; como parte de um navegador Web; ou no hardware.

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.

É a plataforma Java que possibilita o desenvolvimento de programas independentes da arquitetura e portáveis. A implementação da plataforma é específica do sistema, e ela separa o sistema de uma maneira padrão para o programa Java, oferecendo uma interface limpa e independente da arquitetura. Essa interface permite que um arquivo .class seja executado em qualquer sistema que tenha implementado a JVM e API; isso é mostrado na Figura 3.15. A implementação da plataforma Java consiste no desenvolvimento de uma JVM e API Java para um sistema específico (como Windows ou UNIX), de acordo com a especificação para a JVM.

Usamos a JVM no decorrer deste livro para ilustrar os conceitos do sistema operacional. Vamos nos referir à especificação da JVM, em vez de qualquer implementação em particular.

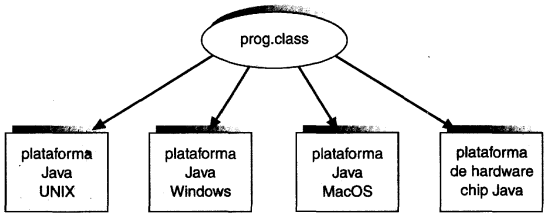



FIGURA 3.15 *Arquivo* `.class` *da Java em diferentes plataformas.*

### 3.7.4 Ambiente de desenvolvimento Java

O ambiente de desenvolvimento Java consiste em um ambiente em tempo de compilação e um ambiente em tempo de execução. O ambiente em tempo de compilação transforma um arquivo fonte Java em um código de bytes (arquivo `.class`). O arquivo fonte Java pode ser um programa Java ou um applet. O ambiente em tempo de execução é a plataforma Java para o host. O ambiente de desenvolvimento é representado na Figura 3.16.

### 3.7.5 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 a 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 oferecer proteção à memória (Seção 2.5.3). 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.

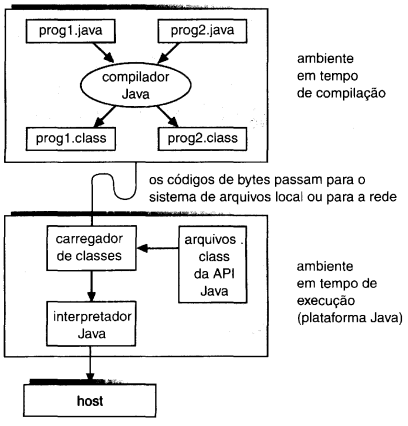



FIGURA 3.16 *Ambiente de desenvolvimento Java.*

O sistema operacional JX foi escrito quase totalmente em Java e também oferece um sistema em tempo de execução para aplicações Java. O JX organiza seu sistema de acordo com **domínios**. Cada domínio contém seu próprio conjunto de classes e threads. Além do mais, cada domínio é responsável por alocar memória para a criação de objeto e threads dentro de si mesmo, além da coleta de lixo. O domínio zero é um microkernel (Seção 3.5.3) responsável pelos detalhes de baixo nível como inicialização do sistema e salvamento e restauração do estado da CPU. 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**.

## 3.8 Projeto e implementação do sistema

Nesta seção, discutimos os problemas encarados quando se projeta e implementa um sistema operacional. Naturalmente, não existem soluções completas para os problemas de projeto, mas existem técnicas bem-sucedidas.

### 3.8.1 Objetivos do 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 opção de hardware e tipo do sistema: batch, tempo compartilhado (time-sharing), monousuário, multiusuário, distribuído, tempo real ou uso geral.

Além desse nível de projeto mais alto, os requisitos podem ser muito mais difíceis de implementar. Contudo, 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 precisa ser conveniente no uso, fácil de aprender e usar, confiável, seguro e veloz. Naturalmente, essas especificações não são úteis no projeto do sistema, pois não existe um acordo geral sobre como alcançá-las.

Um conjunto de requisitos semelhante pode ser definido por aquelas pessoas que precisam projetar, criar, manter e operar o sistema: o sistema precisa ser fácil de projetar, implementar e manter; ele deve ser flexível, confiável, livre de erros e eficiente. Mais uma vez, esses requisitos são vagos e podem ser interpretados de diversas maneiras.

Resumindo, não existe uma solução única para o problema de definir os requisitos para um sistema operacional. A grande variedade de sistemas existente mostra que diferentes requisitos podem resultar em uma grande variedade de soluções para diferentes ambientes. Por exemplo, os requisitos para VxWorks, um sistema operacional de tempo real para sistemas embutidos, precisam ser diferentes daqueles para MVS, o grande sistema operacional multiusuário e multiacesso para mainframes IBM.

A especificação e o projeto de um sistema operacional é uma tarefa bastante criativa. Embora nenhum livro texto possa dizer como fazê-lo, foram desenvolvidos princípios gerais no campo da **engenharia de software** e passaremos agora a uma discussão sobre alguns desses princípios.

### 3.8.2 Mecanismos e políticas

Um princípio importante é a separação entre **política** e **mecanismo**. Os mecanismos determinam *como* fazer alguma coisa; políticas determinam o *que* será feito. Por exemplo, a construção do temporizador (ver Seção 2.5) é um mecanismo para garantir a proteção da CPU, mas decidir por quanto tempo o temporizador deve ser definido para determinado usuário é uma decisão da política.

A separação entre política e mecanismo é importante por flexibilidade. As políticas provavelmente mudarão conforme o lugar e o tempo. No pior caso, cada mudança na política exigiria uma mudança no mecanismo básico. Um mecanismo geral insensível a mudanças na política seria mais desejável. Uma mudança na política exigiria a redefinição somente 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 estiver separado e independente da política, ele poderá ser usado para dar suporte a uma decisão da política de que os programas com uso intenso de E/S devem ter prioridade em relação aos que utilizam CPU intensamente, ou para dar suporte à política oposta.

Os sistemas operacionais baseados em microkernel levam à separação entre mecanismo e política a um extremo, implementando um conjunto básico de blocos de montagem primitivos. Esses blocos são quase livres de política, permitindo que mecanismos e políticas mais avançados sejam acrescentados por meio de módulos do kernel, criados pelo usuário ou pelos próprios programas do usuário. Como um exemplo, considere a história do UNIX. A princípio, 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 em batch, tempo real, fatia justa ou qualquer combinação deles. Fazer com que o mecanismo de escalonamento seja de uso geral permite que grandes mudanças na política sejam feitas com um único comando `load-new-table`. No outro extremo está um sistema como o Windows, em que tanto o mecanismo quanto a política são codificados no sistema para impor um estilo global. Todas as aplicações possuem interfaces semelhantes, pois a própria interface está embutida no kernel do sistema.

As decisões da política são importantes para toda a alocação de recursos. Sempre que for necessário decidir se um recurso deve ser alocado ou não, uma decisão da política precisa ser tomada. Sempre que a questão é *como* em vez de *que*, esse é um mecanismo que precisa ser determinado.

### 3.8.3 Implementação

Quando um sistema operacional é projetado, ele precisa ser implementado. Tradicionalmente, os sis-

temas operacionais têm sido escritos em linguagem assembly. Agora, porém, eles geralmente são escritos em linguagens de mais alto nível, como C ou C++. Na Seção 3.7.5, vimos esforços recentes para a escrita de sistemas em Java.

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

As vantagens do uso de uma linguagem de alto nível ou, pelo menos, uma linguagem de implementação de sistemas, para implementar os sistemas operacionais, são as mesmas daquelas resultantes de quando a linguagem é usada para programas de aplicação: o código pode ser escrito mais rapidamente, ele é mais compacto e mais fácil de entender e depurar. Além disso, as melhorias na tecnologia de contrução de compiladores melhorarão o código gerado para o sistema operacional inteiro, pela simples recompilação. Por fim, um sistema operacional é muito mais fácil de *portar* – passar para algum outro hardware – se for escrito em uma linguagem de alto nível. Por exemplo, o MS-DOS foi escrito em linguagem assembly do processador Intel 8088. Como conseqüência, ele está disponível apenas na família de CPUs Intel. O sistema operacional Linux, ao contrário, que foi escrito principalmente em C, está disponível para diversas CPUs diferentes, incluindo Intel 80X86, Motorola 680X0, SPART e MIPS RX000.

As principais desvantagens existentes para a implementação de um sistema operacional em uma linguagem de alto nível são a menor velocidade e o maior requisito de armazenamento. Embora um programador especializado em linguagem assembly possa produzir pequenas rotinas eficientes, para programas grandes, um compilador moderno pode realizar análise complexa e aplicar otimizações sofisticadas, que produzem um código excelente. Os processadores modernos possuem pipelining profundo e múltiplas unidades funcionais, que podem lidar com dependências complexas, que seriam demais para a capacidade limitada da mente humana de acompanhar todos os detalhes.

Como acontece em outros sistemas, as maiores melhorias de desempenho nos sistemas operacionais provavelmente são o resultado de melhores estruturas de dados e algoritmos do que de um código excelente na linguagem assembly. Além disso, embora os sistemas operacionais sejam grandes, somente uma pequena quantidade do código é crítica para o alto desempenho; é provável que o gerenciador de memória e o escalonador de CPU sejam as rotinas mais críticas. Depois que o sistema é escrito e está funcionando corretamente, as rotinas que formam um gargalo podem ser identificadas e substituídas por equivalentes em linguagem assembly.

Para identificar gargalos (Bottlenecks), temos de ser capazes de monitorar o desempenho do sistema. O código precisa ser acrescentado para calcular e exibir medidas do comportamento do sistema. Em diversos sistemas, o sistema operacional realiza essa tarefa produzindo listagens de rastreamento do comportamento do sistema. Todos os eventos interessantes são registrados com a hora em que ocorreram e com parâmetros importantes, sendo gravados em um arquivo. Mais adiante, um programa de análise pode processar o arquivo de log para determinar o desempenho do sistema e identificar gargalos e ineficiências. Esses mesmos rastreamentos podem ser executados como entrada para a simulação de um sistema melhorado sugerido. Os rastreamentos também podem ajudar as pessoas a localizar erros no comportamento do sistema operacional.

Uma alternativa é calcular e exibir as medidas de desempenho em tempo real. Por exemplo, um temporizador pode disparar uma rotina para armazenar o valor atual do ponteiro de instrução. O resultado é uma imagem estatísticas das locações usadas mais freqüentemente dentro do programa. Essa técnica pode permitir que operadores do sistema se familiarizem com o comportamento do sistema e modifiquem a operação do sistema em tempo real.

## 3.9 Geração do sistema

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. 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.

- Que 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.
- Quanta memória está disponível? Alguns sistemas determinarão esse valor por si mesmos, referenciando 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 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 simplesmente 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 freqüência (ou falta de freqüência) de tais mudanças.

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, armazenada na ROM, conhecida como *programa de boot* ou *bootstrap loader*. Esse código é capaz de localizar o kernel, carregá-lo para a memória principal e iniciar 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. O bootstrap de um sistema é discutido na Seção 14.3.2 e no Apêndice A.

## 3.10 Boot do sistema

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 das tarefas 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 freqüência, 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 14.3.1) é chamado de **disco de boot** ou **disco do sistema**.

Agora que o programa de boot completo foi carregado, ele sabe como 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).

## 3.11 Resumo

Os sistemas operacionais oferecem 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 oferece 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 oferecer 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 seqüência de cha-

madas 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 seqüência 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.

Qualquer sistema operacional que tenha implementado a JVM é capaz de executar todos os programas em Java, pois a JVM extrai o sistema básico para o programa Java, oferecendo uma interface independente da arquitetura.

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 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 seqüência 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

**3.1** Quais são as cinco principais atividades de um sistema operacional em relação à gerência de processos?

**3.2** Quais são as três principais atividades de um sistema operacional em relação à gerência de memória?

**3.3** Quais são as três principais atividades de um sistema operacional em relação à gerência do armazenamento secundário?

**3.4** Quais são as cinco principais atividades de um sistema operacional em relação à gerência de arquivos?

**3.5** Qual é a finalidade do interpretador de comandos? Por que ele normalmente é separado do kernel?

**3.6** Relacione cinco serviços fornecidos por um sistema operacional. Explique como cada um provê conveniência aos usuários. Em que casos seria impossível que os programas no nível do usuário provessem esses serviços? Explique.

**3.7** Qual é a finalidade das chamadas de sistema?

**3.8** Usando chamadas de sistema, escreva um programa em C ou C++ que leia dados de um arquivo e os copie para outro arquivo. Esse programa foi descrito na Seção 3.3.

**3.9** Por que Java provê a capacidade de chamar, de um programa Java, métodos nativos escritos, digamos, em C ou C++? Dê um exemplo onde um método nativo é útil.

**3.10** Qual é a finalidade dos programas de sistema?

**3.11** Qual é a principal vantagem da técnica em camadas para o projeto de sistema?

**3.12** Qual é a principal vantagem da técnica de microkernel para o projeto de sistema?

**3.13** 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?

**3.14** Por que um compilador just-in-time é útil para a execução de programas Java?

**3.15** Por que a separação do mecanismo e da política é desejável?

**3.16** O sistema operacional experimental Synthesis possui um montador (assembler) incorporado dentro do kernel. Para otimizar o desempenho da chamada de sis-

tema, 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.

**3.17** Por que alguns sistemas armazenam o sistema operacional em firmware e outros em disco?

**3.18** Como um sistema poderia ser projetado para permitir uma escolha dos sistemas operacionais que darão boot? O que o programa de boot precisaria fazer?

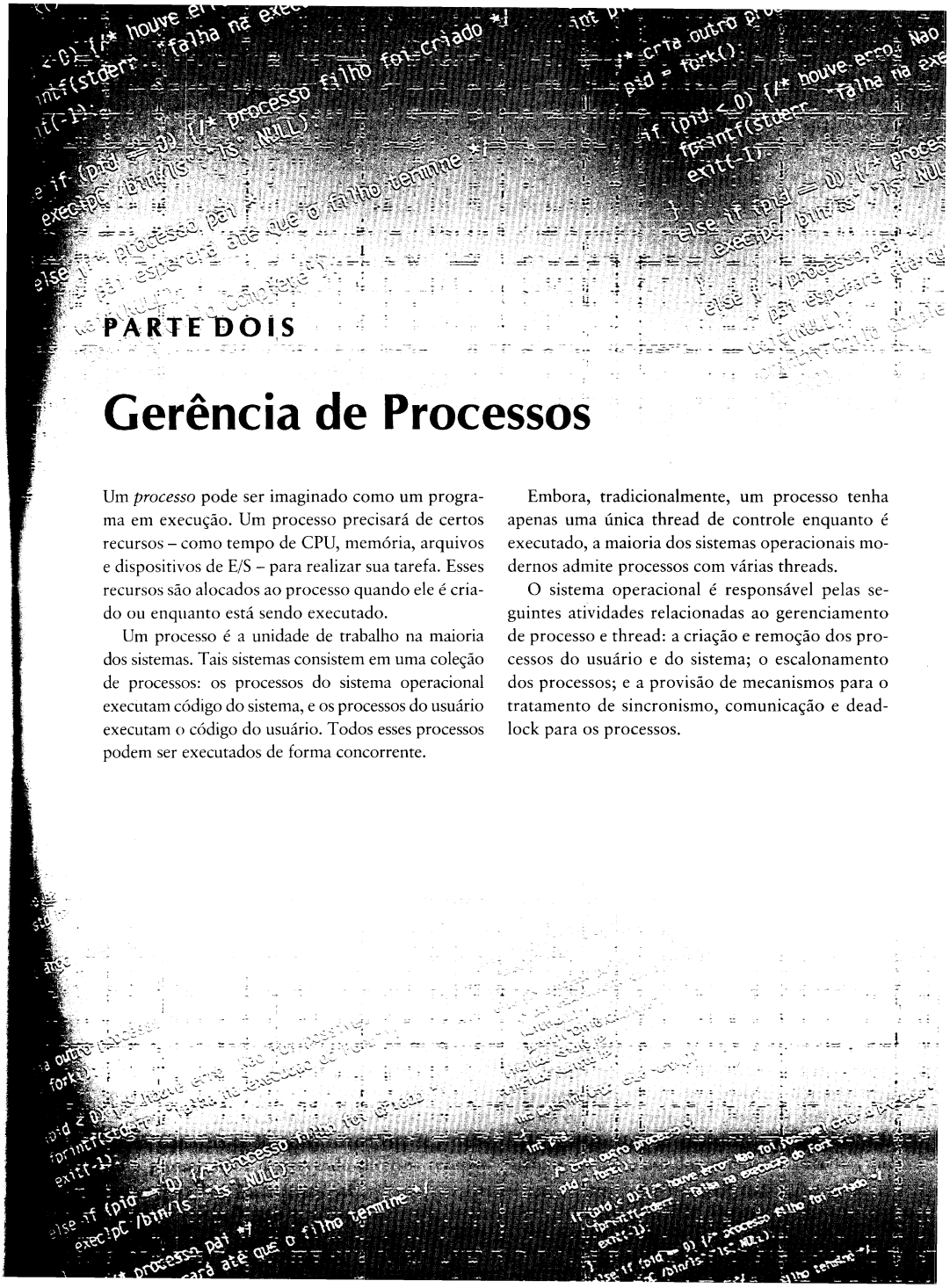
## Notas bibliográficas

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.

O primeiro sistema operacional a prover 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. 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]. BSD UNIX é descrito em McKusick e outros [1996]. Uma boa descrição do OS/2 é dada por Iacobucci [1988]. Bovet e Cesati [2002] abordam o kernel do Linux com detalhes. Diversos sistemas UNIX, incluindo Mach, são abordados com detalhes em Vahalia [1996]. Mac OS X é apresentado em *http://www.apple.com/macosx*. O sistema operacional experimental Synthesis é discutido por Massalin e Pu [1989]. Solaris é descrito por completo em Mauro e McDougall [2001].

A especificação para a linguagem Java e a máquina virtual Java são apresentadas por Gosling e outros [1996] 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://www.javasoft.com*.



PARTE DOIS

# 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. Tais sistemas consistem em uma coleção de processos: os processos do sistema operacional executam 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 várias threads.

O sistema operacional é responsável pelas seguintes atividades relacionadas ao gerenciamento 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 4

# Processos

Os primeiros sistemas computadorizados permitiam que apenas um programa fosse executado de cada vez. Esse programa tinha controle total do sistema e tinha acesso a todos os recursos do sistema. Em contraste, os sistemas computadorizados dos dias atuais 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 um **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 que ele faça 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, examinamos os processos mais de perto. Começamos discutindo o conceito de processo e prosseguimos descrevendo diversos recursos dos processos, incluindo escalonamento, criação e término, e comunicação. Terminamos o capítulo descrevendo a comunicação nos sistemas cliente-servidor.

## 4.1 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, 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*) simplesmente porque *processo* substituiu *tarefa*.

### 4.1.1 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 (PC – program counter)** 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.

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; 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.

Embora um programa possa ser associado a dois processos, mesmo assim eles são considerados duas seqüências de execução separadas. Por exemplo, diversos usuários podem estar executando diferentes cópias do programa de correio, ou então o mesmo usuário pode invocar muitas cópias do programa editor. Cada um desses é um processo separado; e embora as seções de texto sejam equivalentes, as seções de dados variam. Também é comum ter um processo que cria muitos processos enquanto é executado. Discutiremos essas questões na Seção 4.4.

### 4.1.2 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 4.1.


FIGURA 4.1 *Diagrama de estado do processo.*

### 4.1.3 Bloco de controle de processo

Cada processo é representado no sistema operacional por um **bloco de controle de processo** (**PCB – Process Control Block**) – também conhecido como *bloco de controle de tarefa*. Um PCB é representado na Figura 4.2. Ele contém muitas informações associadas a um processo específico, incluindo estas:

- *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 4.3).
- *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 6 descreve o escalonamento de processos.)

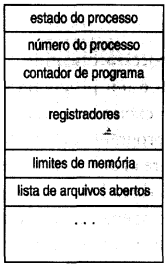



FIGURA 4.2 *Bloco de controle de processo (PCB).*

- *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 9).
- *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 simplesmente serve como o repositório para quaisquer informações que possam variar de um processo para outro.

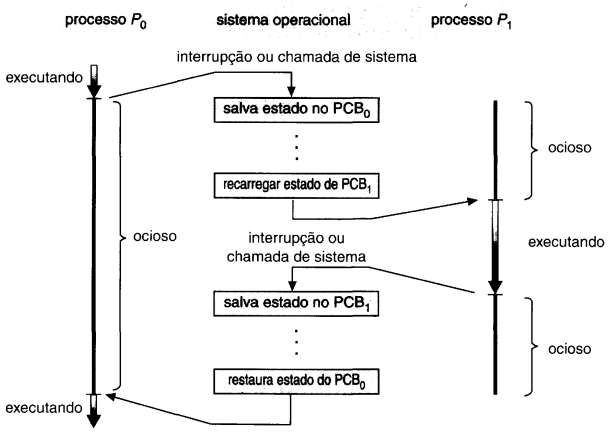



FIGURA 4.3 *Diagrama mostrando a troca de CPU de um processo para outro.*

### 4.1.4 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. O Capítulo 5 explora os detalhes dos processos dotados de múltiplas threads (multithreaded).

## 4.2 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 freqüência 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 monoprocessado, 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.

### 4.2.1 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 PCBs 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 um determinado dispositivo de E/S é denominada **fila de dispositivo (device queue)**. Cada dispositivo possui sua própria fila de dispositivo (Figura 4.4).

Uma representação comum para uma discussão sobre escalonamento de processos é um **diagrama de fila**, como o que aparece na Figura 4.5. 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.

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

- O processo poderia emitir uma requisação de E/S e depois ser colocado em uma fila de E/S.
- O processo poderia criar um novo subprocesso e esperar pelo término do subprocesso.
- O processo poderia 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 será removido de todas as filas e perderá a alocação do seu PCB e de seus recursos.

### 4.2.2 Escalonadores

Um processo migra entre diversas filas de escalonamento no decorrer do seu tempo de vida. O sistema

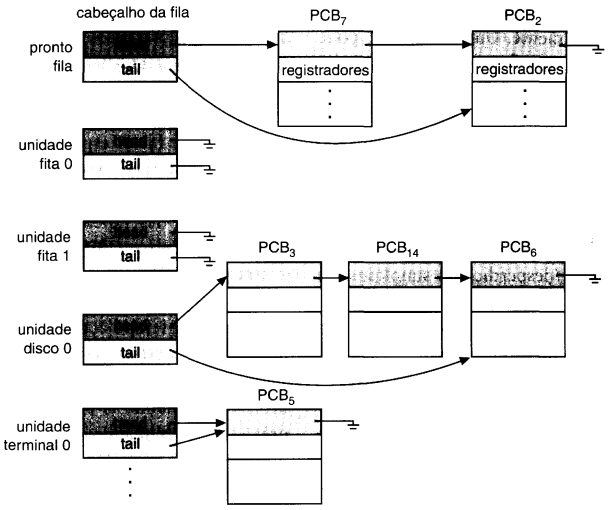



FIGURA 4.4 *A fila de prontos e diversas filas de dispositivo de E/S.*

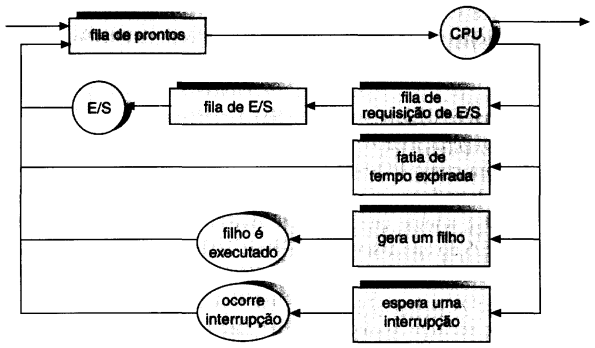



FIGURA 4.5 *A representação do diagrama de fila do escalonamento de processos.*

operacional precisa selecionar, para fins de escalonamento, processos vindos dessas filas de alguma maneira. 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 freqüência de execução. O escalonador

de curto prazo precisa selecionar um novo processo para a CPU freqüentemente. 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 por cento da CPU sendo usada (desperdiçada) simplesmente para o escalonamento do trabalho.

O escalonador de longo prazo é executado com muito menos freqüência; 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, então a taxa média de criação do processo precisa 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 freqüência, usando mais do seu tempo realizando cálculos. É importante que o escalonador de longo prazo selecione uma 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 auto-ajuste dos usuários humanos. Se o desempenho diminuir até chegar a níveis inaceitáveis em um sistema multiusuário, alguns usuários simplesmente sairão.

Alguns sistemas operacionais, como os sistemas de tempo compartilhado, podem introduzir um nível de escalonamento intermediário adicional. Esse **escalonador de médio prazo (médium-term scheduler)** é representado na Figura 4.6. A idéia 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 re-introduzido 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 esca-

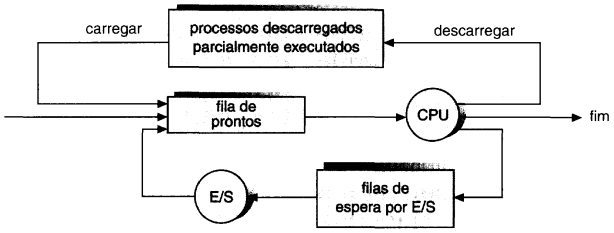



FIGURA 4.6 *Acréscimo do escalonamento de meio termo para o diagrama de enfileiramento.*

lonador de médio prazo. A troca pode ser necessária para melhorar 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 9.

### 4.2.3 Troca de contexto

A passagem da CPU para outro processo exige salvar o estado do processo antigo e carregar o estado salvo do novo processo. Essa tarefa é conhecida como **troca de contexto (context switch)**. O **contexto** de um processo é representado no PCB do processo; ele inclui o valor dos registradores da CPU, o estado do processo (ver Figura 4.1) e informações de gerenciamento de memória. 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 programado para execução. 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). As velocidades típicas são inferior a 10 milissegundos.

Os tempos de troca de contexto dependem bastante do suporte do hardware. Por exemplo, alguns processadores (como o Sun UltraSPARC) oferecem 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 for o sistema operacional, mais trabalho deverá ser feito durante uma troca de contexto. Conforme veremos no Capítulo 9, 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, isso depende do método de gerência de memória do sistema operacional. Conforme veremos no Capítulo 5, a troca de contexto tornou-se um gargalo tão grande para o desempenho que os programadores estão usando estruturas alternativas (threads) para agilizar a troca – e possivelmente evitá-la – sempre que possível.

## 4.3 Operações sobre processos

Os processos na maioria dos sistemas podem ser executados de forma concorrente e podem ser criados e removidos dinamicamente. Assim, esses sistemas operacionais precisam oferecer um mecanismo para a criação e o término de processo.

### 4.3.1 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 (Figura 4.7).

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 então 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 obtidos por um processo quando ele é criado, dados de inicialização (entrada) podem ser passados pelo processo pai para o processo filho. Por exemplo, considere um processo cuja função seja exibir o conteúdo de um arquivo – digamos, *A1* – na tela de um terminal. Quando for criado, ele apanhará, como entrada do seu processo pai, o nome do arquivo *A1*, e usará esse nome de arquivo, abrirá o arquivo e escreverá o

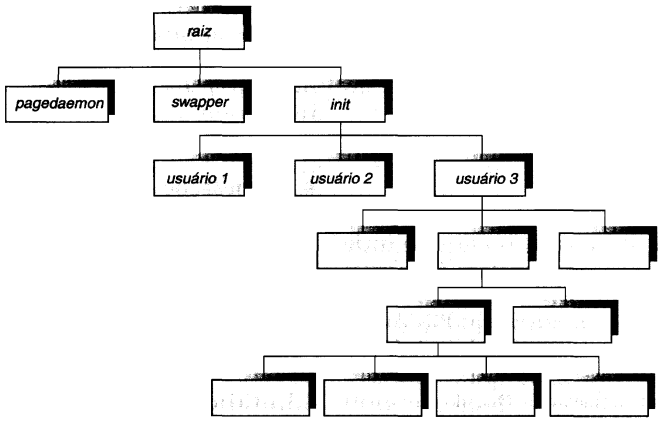



FIGURA 4.7 *Uma árvore de processos em um sistema UNIX.*

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, *A1* 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.

Para ilustrar essas diferenças, vamos considerar o sistema operacional UNIX. No UNIX, 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 (sysall) `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 4.8 ilustra as chamadas de sistema do UNIX descritas anteriormente. Agora temos dois processos diferentes executando uma cópia do mesmo programa. O valor do *pid* para o processo filho é zero; o pai possui um valor inteiro maior do que zero. O processo filho sobrepõe seu espaço de endereços com o comando do UNIX `/bin/ls` (usado para obter uma listagem do

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[ ])
{
int pid;

   /* cria outro processo */
   pid = fork( );

   if (pid < 0) { /* houve erro -- Não foi possível criar o processo */
      fprintf(stderr, "falha na execucao do Fork");
      exit(-1);
   }
   else if (pid == 0) { /* processo filho foi criado */
      execlpC /bin/ls","ls",NULL);
   }
   else { /* processo pai */
      /* pai esperará até que o filho termine */
      wait(NULL);
      printf("Child Complete");
      exit(0);
   }
}
```

FIGURA 4.8 *Exemplo de Programa em C para criar um novo processo.*

diretório) usando a chamada de sistema (syscall) `execlp( )` (execlp( ) é uma versão da chamada de sistema `exec( )`)). O pai espera até que o processo filho termine com a chamada de sistema (syscall) `wait( )`. Quando o processo filho termina, o processo pai retorna da chamada `wait( )` e termina usando a chamada de sistema `exit( )`.

O sistema operacional VMS do DEC, ao contrário, cria um novo processo, carrega um programa especificado nesse processo e inicia sua execução. O sistema operacional Microsoft Windows NT admite dois modelos: o espaço de endereços do pai pode ser duplicado, ou o pai pode especificar o nome de um programa para o sistema operacional carregar no espaço de endereços do novo processo.

### 4.3.2 Término de processo

Um processo termina quando termina 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 desalocados 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, `abort( )`. 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 um do outro 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 estes:

- 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.

Muitos 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.

Para ilustrar a execução e o término do processo, consideramos que, 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 possivelmente muitos 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.

## 4.4 Processos cooperativos

Os processos concorrentes em execução 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 (temporários ou persistentes) 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 oferecer 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 oferecer um ambiente para permitir o acesso concorrente a esses tipos de recursos.
- *Agilidade na computação:* Se queremos que determinada tarefa seja executada mais rapidamente, temos de dividi-la em subtarefas, cada um 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 3.
- *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.

A execução simultânea que exige cooperação entre os processos exige mecanismos para permitir que os processos se comuniquem entre si (Seção 4.5) e sincronizem suas ações (Capítulo 7).

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 assembler (ou montador). O assembler, por sua vez, pode produzir módulos objeto, que são consumidos pelo carregador. O problema produtor-consumidor também oferece 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 de arquivos produz (ou seja, oferece) um arquivo que é consumido (ou seja, lido) pelo cliente solicitando o arquivo.

Para permitir que processos produtor e consumidor sejam executados simultaneamente, precisamos ter à disposição um buffer de itens que possa ser preenchido pelo produtor e esvaziado pelo 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. Nessa situação, o consumidor precisa esperar até que um item seja produzido.

As soluções para o problema produtor-consumidor podem implementar a interface `Buffer`, mostrada na Figura 4.9. O processo produtor chama o método `insert( )` quando deseja inserir um item no

```
public interface Buffer
{
   // produtores chamam este método
   public abstract void insert(Object item);

   // consumidores chamam este método
   public abstract Object remove( );
}
```

FIGURA 4.9 *Interface para implementações de buffer.*

buffer, e o consumidor chama o método `remove( )` quando quer consumir um item do buffer.

A natureza do buffer – ilimitado ou limitado – fornece um meio de descrever o problema produtor-consumidor. O problema produtor-consumidor com **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 problema produtor-consumidor com **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.

O buffer pode ser fornecido pelo sistema operacional por meio do uso de uma facilidade de comunicação entre processos (IPC – Interprocess Communication) (Seção 4.5) ou codificado explicitamente pelo programador de aplicação com o uso de memória compartilhada. Embora a linguagem Java não ofereça 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 4.10), 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.

```
import java.util.*;

public class BoundedBuffer implements Buffer
{
   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 Object[ ] buffer;

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

      buffer = new Object[BUFFER_SIZE];
   }
   // produtores chamam esse método
   public void insert(Object item) {
      // Figura 4.11
   }

   // consumidores chamam esse método
   public Object remove( ) {
      // Figura 4.12
   }
}
```

FIGURA 4.10 *Solução de memória compartilhada para o problema produtor-consumidor.*

```
public void insert(Object item) {
   while (count == BUFFER_SIZE)
      ; // não faz nada -- nenhum buffer livre

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

FIGURA 4.11 *O método insert( ).*

```
public Object remove( ) {
   Object item;

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

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

   return item;
}
```

FIGURA 4.12 *O método remove( ).*

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. `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 produto quanto o consumidor serão bloqueados no loop `while` se o buffer não puder ser utilizado por eles. No Capítulo 7, discutiremos como o sincronismo entre os processos cooperativos pode ser implementado de forma eficiente em um ambiente de memória compartilhada.

## 4.5 Comunicação entre processos

Na Seção anterior, mostramos como os processos cooperativos podem se comunicar em um ambiente de memória compartilhada. O esquema exige que esses processos compartilhem um banco de buffers comum e que o código para a implementação do buffer seja escrito explicitamente pelo programador da aplicação. Outra maneira de conseguir o mesmo efeito é deixar que o sistema operacional ofereça os meios para que os processos cooperativos se comuniquem entre si por meio de uma facilidade de **comunicação entre processos (IPC – InterProcess Communication)**.

IPC oferece um mecanismo para permitir que os processos se comuniquem e sincronizem suas ações sem compartilhar o mesmo espaço de endereços. IPC é útil em um ambiente distribuído, no qual os processos em comunicação podem residir em diferentes computadores, conectados com uma rede. Um exemplo é um programa de **bate-papo** usado na World Wide Web.

IPC é fornecido melhor por um sistema de troca de mensagens, e os sistemas de mensagens podem ser definidos de muitas maneiras diferentes. A seguir, examinaremos os diferentes aspectos de projeto, apresentando uma solução Java para o problema produtor-consumidor que utiliza a troca de mensagens.

### 4.5.1 Sistema de troca de mensagens

A função de um sistema de mensagens é permitir que os processos se comuniquem entre si sem a necessidade de lançar mão dos dados compartilhados. Um recurso de IPC oferece 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. Ao contrário, 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 15), 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.

## 4.5.2 Nomeação

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

### 4.5.2.1 Comunicação direta

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 definidos como:

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

Um enlace de comunicação nesse esquema possui as seguintes propriedades:

- Um enlace é estabelecido automaticamente entre cada par de processos que desejam se comunicar. Os processos precisam saber apenas a identidade um do outro 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 como o nome do processo com o qual ocorreu 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 rígidas, nas quais os identificadores precisam ser indicados explicitamente, são menos desejáveis do que aquelas envolvendo um nível de indireção, conforme descrevemos a seguir.

### 4.5.2.2 Comunicação indireta

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. 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 processo 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). 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 da caixa de correio (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.

Ao contrário, 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 oferecer um mecanismo permitindo que um processo faça o seguinte:

- Crie uma nova caixa de correio
- Envie e receba mensagens por meio da caixa de correio
- Exclua 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 poderia resultar em diversos receptores para cada caixa de correio.

### 4.5.3 Sincronismo

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 (non-blocking)** – 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.

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

### 4.5.4 Buffers

Seja a comunicação direta ou indireta, as mensagens trocadas pelos processos em comunicação residem em uma fila temporária. Basicamente, existem três maneiras de implementar tal fila:

- **Capacidade zero:** a fila tem o tamanho máximo de 0; 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 podem residir nela. Se a fila não estiver cheia quando uma nova mensagem for enviada, a mais recente 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, o enlace possui capacidade 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 buffer automático.

### 4.5.5 Exemplo de produtor-consumidor

Agora, podemos apresentar uma solução para o problema produtor-consumidor utilizando a passagem de mensagens. Nossa solução implementará a interface `Channel` mostrada na Figura 4.13. O produtor e o consumidor se comunicarão indiretamente usando a caixa de correio compartilhada, apresentada na Figura 4.14.

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 4.15.

```
public interface Channel
{
   // Envia uma mensagem ao canal
   public abstract void send(Object item);

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

FIGURA 4.13 *Interface para a troca de mensagens.*

```
import java.util.Vector;

public class MessageQueue implements Channel
{
   private Vector queue;

   public MessageQueue( ) {
      queue = new Vector( );
   }

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

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

FIGURA 4.14 *Caixa de correio para troca de mensagens.*

```
Channel mailBox;

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

FIGURA 4.15 *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 4.16.

O Capítulo 5 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.

### 4.5.6 Um exemplo: Mach

Como um exemplo de um sistema operacional baseado em mensagens, consideraremos a seguir o sis-

```
Channel mailBox;

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

FIGURA 4.16 *O processo consumidor.*

tema operacional Mach, desenvolvido na Universidade Carnegie Mellon. Apresentamos o Mach no Capítulo 3 como parte do sistema operacional Mac OS X. O kernel do Mach admite 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, RPC modela um procedimento típico de chamada à sub-rotina, mas pode atuar entre sistemas.

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 acesso de recebimento para a 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 inclui o tamanho da mensagem e dois nomes de caixa de correio. Quando uma mensagem é enviada, 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; o nome da caixa de correio do emissor é passado para a tarefa que recebe, que poderá usá-lo como um "endereço de retorno", para enviar mensagens de volta.

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 mensa-

gem 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 15 a 17, mas o Mach também é adequado para sistemas monoprocessado. 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 gerenciamento de memória virtual (Capítulo 10). 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 oferece um grande aumento de desempenho, mas funciona somente para mensagens dentro do sistema. O sistema operacional Mach é discutido no capítulo extra que está incluído em nosso Web site.

### 4.5.7 Um exemplo: Windows XP

O sistema operacional Windows XP 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 oferece suporte a vários ambientes operacionais, ou *subsistemas*, com os quais os programas de aplicação se comunicam por meio de um mecanismo de passagem de mensagens. Os programas de aplicação podem ser considerados clientes do subsistema servidor do Windows XP.

A facilidade de troca de mensagens no Windows XP é denominada **camada de procedimento local (LPC – Local Procedure Call)**. 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 um canal de comunicação (Capítulo 21). 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 ou-

tro. Sob esse método, podem ser enviadas mensagens de até 256 bytes.

Se um cliente precisa enviar uma mensagem maior, ele passará a mensagem por um objeto seção (ou 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 puderem responder imediatamente a uma requisição. O mecanismo de callback permite que realizem o tratamento assíncrono de mensagens.

## 4.6 Comunicação em sistemas cliente-servidor

Nas Seções 4.4 e 4.5, 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 (1.5.2). 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.

### 4.6.1 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*; 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 4.17. Os pacotes trafegando entre os hosts são entregues ao processo apropriado, com base no número da porta de destino.

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. Java oferece uma interface fácil para a programação com sockets e possui uma rica biblioteca de utilitários de rede adicionais. Quem estiver

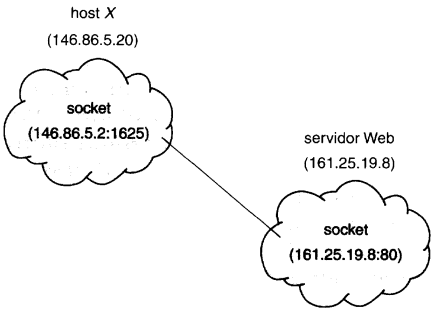



FIGURA 4.17 *Comunicação usando sockets.*

interessado na programação com sockets em C ou C++ deverá consultar as Notas Bibliográficas deste capítulo.

Java oferece 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`. Finalmente, 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 solicitar 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 listado na Figura 4.18. 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.

Os detalhes ilustrando como o servidor se comunica com o socket são os seguintes. 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 fecha o socket com o cliente e continua 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

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

public class DateServer
{
   public static void main(String[ ] args) throws IOException {
      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 4.18 *Servidor de data.*

programa Java que aparece na Figura 4.19. O cliente cria um Socket e solicita 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 fluxo 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.

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).

### 4.6.2 Remote Procedure Calls

Uma das formas mais comuns de serviço remoto é o paradigma da RPC, que discutimos rapidamente na Seção 4.5.4. 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 4.5, 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 oferecer o serviço remoto. Ao contrário da facilidade de IPC, as mensagens troca-

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

public class DateClient
{
   public static void main(String[ ] args) throws IOException {
      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);
      }
   }
}
```

FIGURA 4.19 *Cliente de data.*

das 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 quisesse permitir que outros sistemas sejam capazes de listar seus usuários atuais, ele teria um daemon dando suporte a tal RPC conectado 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, oferecendo 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 passagem 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 usam o endereço de memória alto para armazenar o byte mais significativo (algo conhecido como *big-endian*), enquanto outros sistemas armazenam o byte menos significativo no endereço de memória mais alto (algo conhecido como *little-endian*). Para resolver diferenças como essa, muitos sistemas de RPC definem uma representação de dados independente da máquina. Uma representação desse tipo é conhecida como **representação de dados externos (XDR – eXternal Data Representation)**. 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)*, em vez de *no máximo uma vez (at most once)*. A maior parte das chamadas de procedimento locais possui essa funcionalidade, mas isso é mais difícil de implementar.

Primeiro, consideramos "no máximo uma vez". Essa semântica pode ser garantida anexando-se uma estampa de hora a cada mensagem. O servidor precisa manter um histórico de todas as estampas de hora das mensagens já processadas ou um histórico grande o suficiente para garantir que mensagens repetidas sejam detectadas. As mensagens que chegam e que possuem uma estampa de hora 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 hora é discutida na Seção 17.1.)

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, além de 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 RCP 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 9), 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 são comuns. Primeiro, a informação de associação pode ser predeterminada, na forma de endereços de porta fixos. No momento da compilação, uma chamada por RCP 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. Segundo, a associação pode ser feita de forma dinâmica por um mecanismo de encontro. Normalmente, um sistema operacional oferece 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 trave). Esse método exige o custo adicional da requisição inicial, mas é mais flexível do que a primeira técnica. A Figura 4.20 mostra um exemplo de interação.

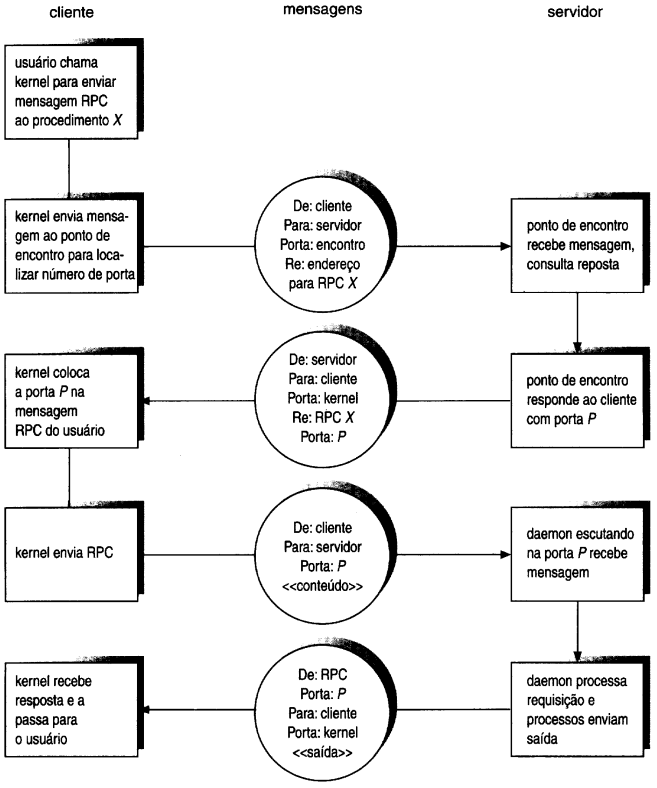



FIGURA 4.20 *Execução de uma remote procedure call (RPC).*

O esquema RPC é útil na implementação de um sistema de arquivos distribuído (Capítulo 16). 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 a 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.

### 4.6.3 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 4.21.

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. Ao contrário, 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 RMI, é possível passar objetos como parâmetros aos métodos remotos. Permitindo que um programa Java invoque métodos em objetos remotos, RMI permite que os usuários desenvolvam aplicações Java distribuídas por uma rede.

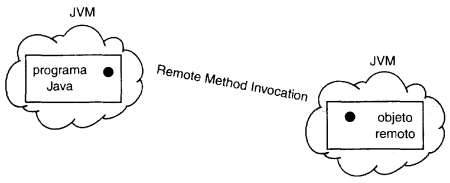



FIGURA 4.21 *Remote Method Invocation.*

Para tornar os métodos remotos transparentes ao cliente e ao servidor, 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 `algumMétodo(Objeto, Objeto)`, que retorna um valor `boolean`. O cliente executa a instrução

```
boolean val = server.algumMétodo(A, B);
```

A chamada a `algumMétodo( )` 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 `algumMétodo( )`. A implementação real de `algumMétodo( )` reside no servidor. Quando o método é completado, o esqueleto empacota o valor `boolean` retornado de `algumMétodo( )` 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 na Figura 4.22.

Felizmente, o nível de abstração que a RMI oferece 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.

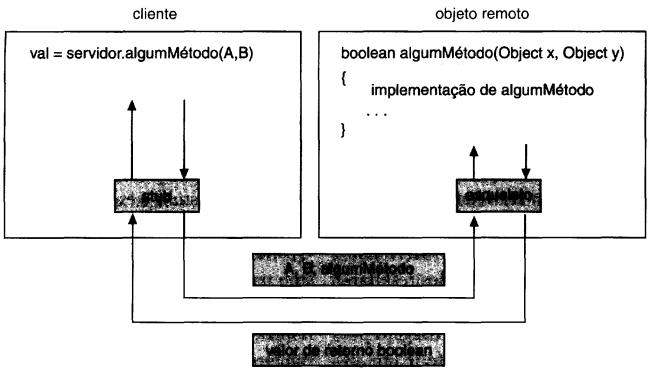



FIGURA 4.22 *Desempacotando parâmetros.*

- Se os parâmetros empacotados forem objetos **locais** (ou **não-remotos**), eles são passados por cópia, usando uma técnica conhecida como **serialização de objetos**. Entretanto, se os parâmetros também forem objetos remotos, eles são passados por referência. Em nosso exemplo, se *A* é um objeto local e *B* é um objeto remoto, *A* é serializado e passado por cópia, enquanto *B* é passado por referência. Isso, por sua vez, permite ao servidor invocar os métodos sobre *B* remotamente.
- Se os objetos locais tiverem de ser passados como parâmetros para objetos remotos, eles terão de implementar a interface *java.io.Serializable.* Muitos objetos na API Java básica implementam *Serializable,* permitindo que sejam usados com RMI. A serialização de objetos permite que o estado de um objeto seja escrito em um fluxo de bytes.

Em seguida, usando RMI, montamos uma aplicação semelhante ao programa baseado em sockets, mostrado na Subeção "Sockets", anteriormente neste capítulo , que retorna a data e hora atuais.

### 4.6.3.1 Objetos remotos

A montagem de uma aplicação distribuída requer inicialmente a definição dos objetos remotos necessários. Definimos os objetos remotos primeiro declarando uma interface que especifique os métodos que podem ser invocados remotamente. Nesse exemplo de um servidor de data, o método remoto será chamado `getDate( )`, e retornará um tipo `Date` contendo a data atual. Para providenciar objetos remotos, essa interface também precisa estender a interface `java.rmi.Remote`, que identifica os objetos implementando essa interface como sendo remotos. Além disso, cada método declarado na interface precisa lançar uma exceção `java.rmi.RemoteException`. Para os objetos remotos, oferecemos a interface `RemoteDate`, mostrada na Figura 4.23.

A classe que define o objeto remoto precisa implementar a interface `RemoteDate` (Figura 4.24). Além de definir o método `getDate( )`, a classe também precisa estender `java.rmi.server.UnicastRemoteObject`. A extensão de `UnicastRemoteObject`

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

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

FIGURA 4.23 *A interface RemoteDate.*

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 `"DateServer"`. 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.

#### 4.6.3.2 Acesso ao objeto remoto

Quando um objeto é registrado no servidor, um cliente (mostrado na Figura 4.25) pode receber uma referência do proxy a esse objeto remoto a partir do registro da RMI sendo executado no servidor, usando o método estático `lookup( )` na classe `Naming`. A RMI oferece um esquema de pesquisa baseado em URL, usando a forma *`rmi://host/nomeObjeto`*, na qual o `host` é o nome IP (ou endereço) do servidor no qual o objeto remoto `nomeObjeto` reside. `nomeObjeto` é o nome do objeto remoto especificado pelo servidor no método `rebind( )` (neste caso, `DateServer`). Quando o cliente tiver a referência do substituto para o objeto remoto, ele chamará o método remoto `getDate( )`, que retorna a data atual. Como os métodos remotos – bem como o método `Naming.lookup( )` – podem lançar exceções, eles precisam ser colocados em blocos `try-catch`.

#### 4.6.3.3 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, 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.*
2. *Gere o stub e o esqueleto.* O usuário gera o stub e o esqueleto usando a ferramenta `rmic`, digitando

   ```
   rmic RemoteDateImpl
   ```

```
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 "DateServer"
            Naming.rebind("DateServer", dateServer);
        }
        catch (Exception e) {
            System.err.println(e);
        }
    }
}
```

FIGURA 4.24 *Implementação da interface* `RemoteDate`.

```
import java.rmi.*;

public class RMIClient
{
   public static void main(String args[ ]) {
      try {
        String host = "rmi://127.0.0.1/DateServer";

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

FIGURA 4.25 *O cliente de RMI.*

na linha de comandos; isso cria os arquivos `RemoteDateImpl_Skel.class` e `RemoteDateImpl_Stub.class`. (Se você estiver executando esse exemplo em dois computadores diferentes, certifique-se de que todos os arquivos de classe – incluindo as classes stub – estejam disponíveis em cada comunicação. É possível carregar as classes dinamicamente usando RMI, um assunto fora do escopo deste texto, mas abordado nos textos mencionados nas Notas Bibliográficas.)

3. *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 `DateServer`.

4. *Referencie o objeto remoto.* A instrução

   ```
   java RMIClient
   ```

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

#### 4.6.3.4 RMI *versus* Sockets

Compare o programa cliente baseado em socket, mostrado na Figura 4.19, com o cliente usando RMI, mostrado na Figura 4.25. 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 permite invocar o método remoto `getDate( )` da mesma forma como invocaria um método local comum.

Isso ilustra a atração por técnicas como RPCs e RMI: elas oferecem aos desenvolvedores de sistemas distribuídos um mecanismo de comunicação permitindo o projeto de programas distribuídos sem incorrer o custo adicional do gerenciamento de sockets.

## 4.7 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 o gerenciamento de memória. O escalonamento de curto prazo (CPU) é a seleção de um processo a partir da fila de prontos.

Os processos na maioria dos sistemas podem ser executados simultaneamente. Existem vários motivos para permitir a execução simultânea: compartilhamento de informações, agilidade da computação, modularidade e conveniência. A execução concorrente exige um mecanismo para a criação e a exclusão de processos.

Os processos em execução no sistema operacional podem ser processos independentes ou processos cooperativos. Os processos cooperativos precisam ter meios de comunicação entre si. A comunicação é obtida por meio de dois esquemas complementares: memória compartilhada e sistemas de mensagem. 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 oferecer a memória compartilhada. O método do sistema 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.

A comunicação nos sistemas cliente-servidor pode utilizar (1) sockets, (2) remote procedure calls (RPCs) ou (3) o 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. RPCs são outra forma de comunicação distribuída. Uma RPC ocorre quando um processo (ou thread) chama um procedimento em uma aplicação remota. RMI é a versão Java de uma RPC. 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, em 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

**4.1** O sistema operacional Palm não oferece uma forma de processamento concorrente. Discuta três complicações importantes que o processamento concorrente acrescenta a um sistema operacional.

**4.2** Descreva as diferenças entre o escalonamento de curto, médio e longo prazo.

**4.3** O processador UltraSPARC da Sun possui vários conjuntos de registradores. Descreva as ações de uma troca de contexto se o novo contexto já estiver carregado em um dos conjuntos de registradores. O que mais deverá acontecer se o novo contexto estiver na memória, e não em um conjunto de registradores, e todos os conjuntos estiverem em uso?

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

**4.5** Quais são os benefícios e as desvantagens de cada um dos itens a seguir? Considere os níveis do sistema e do programador.

- **a.** Comunicação síncrona e assíncrona
- **b.** Buffer automático e explícito
- **c.** Enviar por cópia e enviar por referência
- **d.** Mensagens de tamanho fixo e de tamanho variável

**4.6** Considere o mecanismo de RPC. Descreva as circunstâncias 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.

**4.7** Novamente considerando o mecanismo de RPC, considere a semântica "exatamente uma vez". O algoritmo para implementar essa semântica é executado correta-

mente mesmo se a mensagem "ACK" de volta ao cliente for perdida devido a um problema na rede? Descreva a seqüência de mensagens e se o esquema "exatamente uma vez" ainda é preservado.

**4.8** Modifique o servidor de data, mostrado na Figura 4.18, de modo que ele ofereça números de sorte aleatórios em uma linha, em vez da data atual.

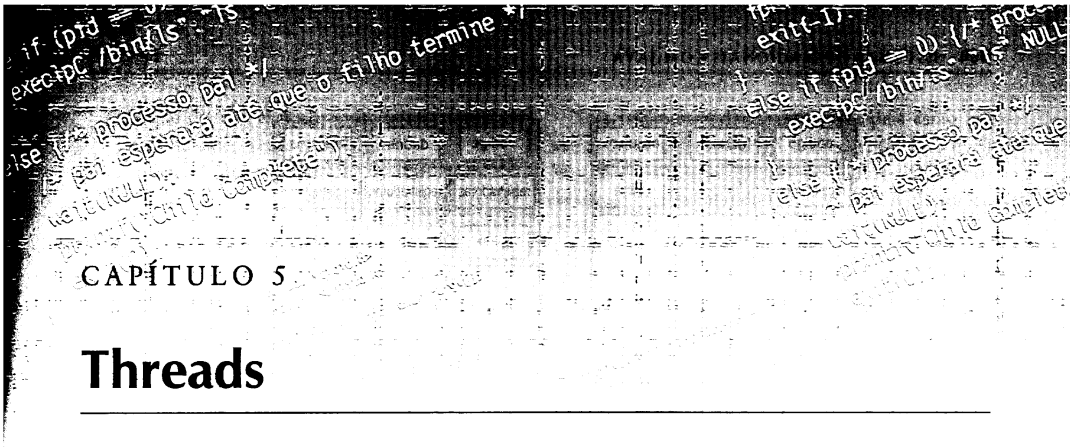
**4.9** Modifique o servidor de data RMI, mostrado na Figura 4.24, para oferecer números de sorte aleatórios em uma linha, em vez da data atual.

## Notas bibliográficas

O assunto de comunicação entre processos foi discutido por Brinch-Hansen [1970] com relação ao sistema RC 4000. Schlichting e Schneider [1982] discutiram sobre as primitivas de troca de mensagem assíncrona. A facilidade de IPC implementada no nível do usuário foi descrita por Bershad e outros [1990].

Os detalhes da comunicação entre processos nos sistemas UNIX foram apresentados por Gray [1997]. Barrera [1991] e Vahalia [1996] apresentaram a comunicação entre processos no sistema Mach. Solomon e Russinovich [2000] e Stevens [1999] esboçam a comunicação entre processos no Windows 2000 e no UNIX, respectivamente.

Discussões referentes à implementação de RPCs foram apresentadas por Birrell e Nelson [1984]. Um projeto de um mecanismo de RPC confiável foi apresentado por Shrivastava e Panzieri [1982]. Um estudo sobre RPCs foi apresentado por Tay e Ananda [1990]. Stankovic [1982] e Staunstrup [1982] discutiram a respeito da comunicação por chamadas de procedimento *versus* troca de mensagens. Grosso [2002] discute a RMI com detalhes significativos. Calvert e Donahoo [2001] oferecem uma explicação sobre programação com sockets em Java.



CAPÍTULO 5

# Threads

O modelo de processo apresentado no Capítulo 4 considerava que um processo era um programa em execução com uma única thread de controle. Muitos sistemas operacionais modernos agora oferecem recursos permitindo que um processo tenha diversas threads de controle. Este capítulo apresenta muitos conceitos associados aos sistemas computadorizados dotados de múltiplas threads (multithreaded), incluindo uma discussão da API Pthreads e threads em Java. Examinamos muitas questões relacionadas à programação multithreads e como ela afeta o projeto dos sistemas operacionais. Finalmente, exploramos como os diversos sistemas operacionais modernos admitem threads no nível do kernel.

## 5.1 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 5.1 ilustra a diferença entre um processo tradicional dotado de uma única thread (single-threaded) e um processo dotado de múltiplas threads (multithreaded).

### 5.1.1 Motivação

Muitos pacotes de software executados nos PCs desktop modernos são **dotados de múltiplas threads (multithreaded)**. 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 (single-thread)**, ele só poderia atender a um cliente de cada vez. A quantidade de tempo que um cliente teria de esperar para que sua requisição fosse atendida poderia ser enorme.

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. A criação de processos é demorada e exige

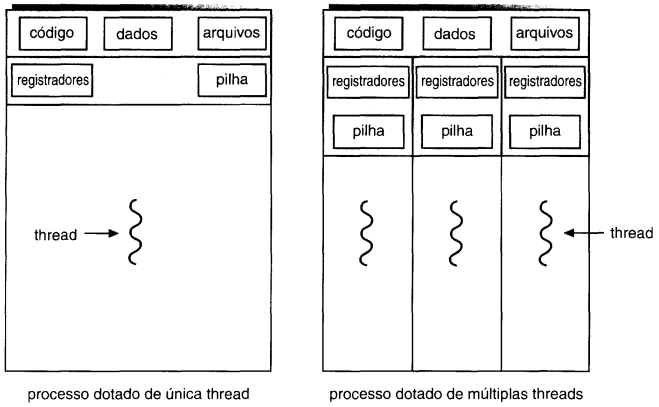



FIGURA 5.1 *Processos com thread única e múltiplas threads.*

muitos recursos, como mostrado no capítulo anterior. 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 sirva ao mesmo propósito. Essa técnica dotaria um processo servidor Web de múltiplas threads. O servidor criaria uma thread separada, que escutaria as requisições do cliente; quando uma requisição fosse feita, em vez de criar outro processo, ele criaria outra thread para atender à requisicão.

As threads também desempenham um papel vital nos sistemas com remote procedure calls (RPC). Lembre-se, do Capítulo 4, de que as RPCs permitem a comunicação entre processos, oferecendo 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 atenda a várias requisições simultâneas. Os sistemas de RMI 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.

### 5.1.2 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 ainda poderia permitir a interação do usuário em uma thread enquanto uma imagem é carregada em outra thread.
2. **Compartilhamento de recursos:** como padrão, as threads compartilham memória e os recursos do processo ao qual pertencem. O benefício do compartilhamento de código 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. **Utilização de arquiteturas multiprocessadas:** 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 uma CPU, não importa quantas estejam à disposição. O uso de múltiplas threads em uma máquina de múltiplas CPUs aumenta a concorrência.

### 5.1.3 Threads de usuário e de kernel

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 gerenciados sem o suporte do kernel, enquanto as threads de kernel são admitidas e gerenciadas diretamente pelo sistema operacional. A maioria dos sistemas operacionais contemporâneos – incluindo Windows XP, Solaris e Tru64 UNIX (originalmente, Digital UNIX) – admite threads de kernel. Na seção 5.2, vamos explicar o relacionamento entre as threads de usuário e de kernel com mais detalhes.

### 5.1.4 Bibliotecas de threads

Uma **biblioteca de threads** oferece 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 oferecer 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 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) Java e (3) Win32. Uma implementação do padrão POSIX pode ser do primeiro ou do segundo tipo. A biblioteca de threads Win23 é uma biblioteca no nível do kernel. A API Java para threads pode ser implementada por Pthreads ou Win32, ou possivelmente por outra biblioteca. Abordamos Pthreads e Java, mais adiante neste capítulo, nas Seções 5.4 e 5.7, respectivamente, e exploramos Win32 na Seção 5.5, que aborda o Windows. Além disso, examinaremos o suporte para thread no sistema operacional Linux, na Seção 5.6, embora o Linux não se refira a eles como *threads*.

## 5.2 Modelos de múltiplas threads (multithreading)

Na Seção 5.1.3, distinguimos entre threads nos níveis de usuário e kernel. Por fim, é preciso que haja um relacionamento entre esses dois tipos de estruturas. Nesta seção, examinamos três formas comuns de estabelecer esse relacionamento.

### 5.2.1 Modelo muitos-para-um

O modelo muitos-para-um (Figura 5.2) associa muitas threads no nível do usuário a uma thread de kernel. O gerenciamento de threads é feito pela biblioteca de threads no espaço do usuário, de modo que é eficiente; mas o processo inteiro será bloqueado se uma thread fizer uma chamada de sistema bloque-

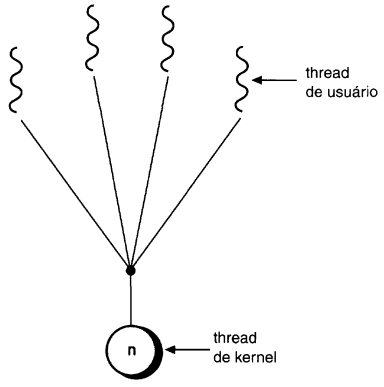



FIGURA 5.2 *Modelo muitos-para-um.*

ante. 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 de threads disponível para o Solaris – utiliza esse modelo, assim como **GNU Portable Threads.**

### 5.2.2 Modelo um-para-um

O modelo um-para-um (Figura 5.3) 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 – incluindo Windows 95/98/NT/2000/XP – implementam o modelo um-para-um.

### 5.2.3 Modelo muitos-para-muitos

O modelo muitos-para-muitos (Figura 5.4) 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 a determinada aplicação ou a 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 um thread realiza uma chamada de sistema bloqueante, o kernel pode escalonar outra thread

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 5.5), é aceita por sistemas operacionais como o IRIX, o HP-UX e 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.

## 5.3 Aspectos do uso de threads

Nesta seção, vamos discutir alguns dos aspectos a serem considerados com programas dotados de múltiplas threads.

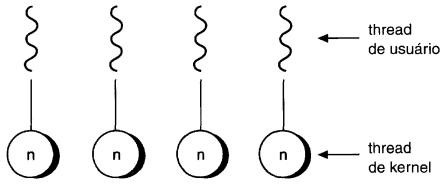



FIGURA 5.3 *Modelo um-para-um.*

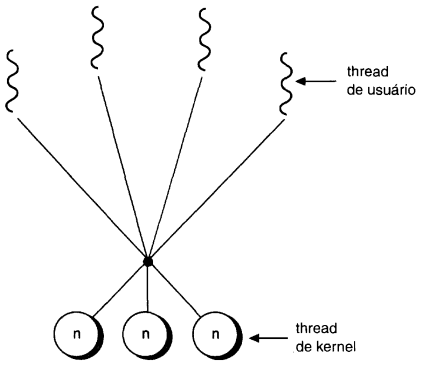



FIGURA 5.4 *Modelo muitos-para-muitos.*

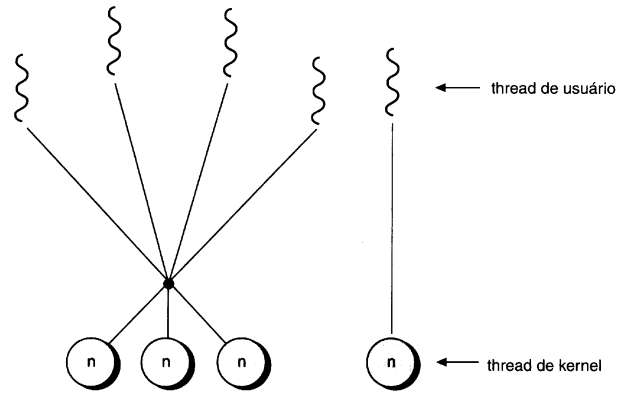



FIGURA 5.5 *Modelo de dois níveis.*

### 5.3.1 As chamadas de sistema fork( ) e exec( )

No Capítulo 4, descrevemos como a chamada de sistema `fork( )` é utilizada para criar um processo separado, duplicado. Em um programa dotado de múltiplas threads, a semântica das chamadas de sistema `fork( )` e `exec( )` muda.

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( )` atua da mesma maneira descrita no Capítulo 4. Ou seja, se um thread invocar a chamada de sistema `exec( )`, o programa especificado no parâmetro de `exec( )` substituirá o processo inteiro – incluindo todas as threads e LWPs.

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 é 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.

### 5.3.2 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, permitindo 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.

Ao contrário, 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 o thread verifique se deve ser cancelada em um ponto em que possa ser cancelada com segurança. Pthreads refere-se a esses pontos como **pontos de cancelamento (cancellation points)**.

### 5.3.3 Tratamento de sinais

Um **sinal** é usado nos sistemas UNIX para notificar a um processo a ocorrência de um determinado 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.

Um exemplo de um sinal síncrono inclui 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 padrão** executado pelo kernel ao tratar desse sinal. Essa ação padrão pode ser modificada por uma função **tratadora de sinal definida pelo usuário**. Nesse caso, a função definida pelo usuário é chamada para tratar do sinal, no lugar da ação padrão. 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 ao qual o sinal se aplica.
2. Entregar o sinal a cada thread no processo.
3. Entregar o sinal a certas 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.

Embora o Windows não ofereça suporte explícito para sinais, eles podem ser simulados por meio de **chamadas de procedimento assíncronas (APCs – Asynchronous Procedure Calls)**. A facilidade 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 assíncro-

no no UNIX. Contudo, enquanto o UNIX precisa disputar com a forma de lidar com sinais em um ambiente multithreads, a facilidade de APC é mais direta, pois uma APC é entregue a uma thread em particular, em vez de um processo.

### 5.3.4 Bancos de threads

Na Seção 5.1, mencionamos o uso de multithreading em um servidor Web. Nessa situação, sempre que o servidor recebe uma requisição, ele cria uma thread separado para atender à 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 à 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 concorrente no sistema. As threads ilimitadas poderiam esgotar os recursos do sistema, como tempo de CPU ou memória. Uma solução para esse problema é usar **bancos de threads**.

A idéia 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 ficam 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 pools de threads oferecem estes benefícios:

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 oferecem o benefício adicional de ter um banco menor – consumindo, assim, menos memória – quando a carga no sistema for baixa.

### 5.3.5 Dados específicos da thread

As threads pertencentes a um processo compartilham os dados do processo. Na realidade, esse compartilhamento de dados oferece um dos benefícios da programação multithreads. Contudo, em algumas circunstâncias, cada thread poderia precisar de sua própria cópia de certos 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 – oferece alguma forma de suporte para dados específicos da thread. Java também oferece suporte, e exploraremos isso na Seção 5.7.5.

### 5.3.6 Ativações do escalonador (Scheduler Activations)

Uma questão final a ser considerada com programas multithreads refere-se à comunicação entre o kernel e a biblioteca de threads, que pode ser exigida pelos modelos muitos-para-muitos e de dois níveis, discutidos na Seção 5.2.3. 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 LWP, é mostrada na Figura 5.6. Para a biblioteca de 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 um 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.

Uma aplicação pode exigir qualquer quantidade de LWPs para ser executada de modo eficiente. Considere uma aplicação ligada à CPU executando em um sistema monoprocessado. Nesse cenário, somente uma thread pode estar executando ao mesmo tempo, de modo que um LWP é suficiente. No entanto, uma aplicação com uso intenso de E/S pode exigir várias 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 de threads de usuário e o kernel é conhecido como **ativação do escalonador (schedule actiation)**. Isso funciona da seguinte forma: o kernel oferece a uma aplicação um conjunto de processadores virtuais (LWPs), e a aplicação pode escalonar threads de usuário para um processador virtual disponível. Além do mais, o kernel precisa informar sobre certos eventos a uma aplicação. Esse procedimento é conhecido como **upcall**. Upcalls são tratados pela biblioteca de 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 escalona 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 de 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 escalona uma thread elegível parra executar em um processador virtual disponível.

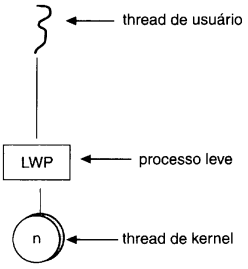



FIGURA 5.6 *Processo leve (LWP).*

## 5.4 Pthreads

**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, Tru64 UNIX e Mac OS X. Também existem implementações *shareware* em domínio público para os diversos sistemas operacionais Windows.

Nesta seção, apresentamos parte da API Pthreads como um exemplo de uma biblioteca de threads no nível do usuário. Vamos nos referir a ela como uma biblioteca no nível do usuário porque não existe um relacionamento distinto entre uma thread criada usando a API Pthreads e quaisquer threads de kernel associadas. O programa em C mostrado na Figura

5.7 demonstra a API Pthreads básica para a construção de um programa multithreads. Se você estiver interessado em mais detalhes sobre a programação com a API Pthreads, consulte as Notas Bibliográficas.

O programa mostrado na Figura 5.7 cria uma thread separada para o somatório de um inteiro não negativo. Em um programa Pthreads, threads separadas iniciam a execução em uma função especificada. Na Figura 5.7, 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.

Agora, oferecemos uma visão mais detalhada desse programa. Todos os programas Pthreads precisam incluir o arquivo de cabeçalho `pthread.h`. A instrução `pthread_t tid` declara o identificador para a

```
#include <pthread.h>
#include <stdio.h>

int sum; /* esses dados são compartilhados pelo(s) fluxo(s) */
void *runner(void *param); /* a thread */

int main(int argc, char *argv[ ])
{
  pthread_t tid; /* o identificador do fluxo */
  pthread_attr_t attr; /* conjunto de atributos para o fluxo */

  if (argc != 2) {
    fprintf(stderr,"uso: a.out <valor inteiro>\n");
    exit( );
  }
  if (atoi(argv[1]) < 0) {
    fprintf(stderr,"%d precisa ser>= 0\n",atoi(argv[1]));
    exit( );
  }

  /* apanha os atributos padrão */
  pthread_attr_init(&attr);
  /* cria o thread */
  pthread_create(&tid,&attr,runner,argv[1]);
  /* agora espera que a thread termine */
  pthread_join(tid,NULL);
  printf("soma = %d\n",sum);
}

/* O fluxo começará a controlar nesta função */
void *runner(void *param)
{
  int i, upper = atoi(param);
  sum = 0;

  if (upper > 0) {
      for (i = 1; i <= upper; i++)
        sum += i;
  }

  pthread_exit(0);
}
```

FIGURA 5.7 *Programa em C multithreads usando a API Pthreads.*

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. (No Capítulo 6, 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]`.

Neste ponto, o programa possui duas threads: a thread inicial em `main( )` e a thread realizando o somatório na função `runner( )`. Após a criação da segunda thread, a thread `main( )` esperará até que a thread `runner( )` termine, chamando a função `pthread_join( )`. A thread `runner( )` terminará quando chamar a função `pthread_exit( )`. Quando a thread `runner( )` tiver retornado, a thread `main( )` informará o valor da soma dos dados compartilhados.

## 5.5 Threads no Windows XP

O Windows XP implementa a API Win32. A API Win32 é a principal API para a família de sistemas operacionais da Microsoft (Windows 95/98/NT, Windows 2000 e Windows 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. Cada processo pode conter uma ou mais threads. O Windows XP utiliza o mapeamento um-para-um, descrito na Seção 5.2.2, no qual cada thread no nível do usuário é associada a uma thread de kernel. Entretanto, o Windows XP também oferece suporte para uma biblioteca **fiber**, que oferece a funcionalidade do modelo muitos-para-muitos (Seção 5.2.3). Cada 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. De modo semelhante, cada thread possui uma pilha do kernel usada quando a thread está executando no modo do kernel.
- Uma área de armazenamento privada, usada por diversas bibliotecas em tempo de execução e bibliotecas de vínculo dinâmico (DLLs – Dynamic Link Libraries).

O conjunto de registradores, pilhas e área de armazenamento privado são conhecidos como o 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 do 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 do usuário. Entre outros campos, o TEB contém uma pilha do modo do usuário e um array para dados específicos da thread (que o Windows XP chama de **armazenamento local a thread**).

## 5.6 Threads no Linux

O Linux oferece uma chamada de sistema `fork( )` com a funcionalidade tradicional da duplicação de um processo. O Linux também oferece a chamada de sis-

tema clone( ), semelhante à criação de uma thread. clone( ) comporta-se de modo muito parecido com fork( ), exceto que, em vez de criar uma cópia do processo que chama, ele cria um processo separado, que compartilha o espaço de endereços do processo que chama. Esse compartilhamento do espaço de endereços do processo que chama permite que uma tarefa clonada se comporte como uma thread separada.

O compartilhamento do espaço de endereços é permitido devido ao modo como um processo é representado no kernel do Linux. Existe uma estrutura de dados exclusiva do kernel para cada processo no sistema. Todavia, a estrutura dos dados, em vez de armazenar os dados para o processo, 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, um novo processo é criado junto com uma *cópia* de todas as estruturas de dados associadas do processo pai. Um novo processo também é criado quando é feita a chamada de sistema clone( ). Entretanto, em vez de copiar todas as estruturas de dados, o novo processo *aponta* para as estruturas de dados do processo pai, permitindo assim que o processo filho compartilhe a memória e outros recursos do processo do pai. Um conjunto de sinalizadores (flags) é passado como um parâmetro para a chamada de sistema clone( ). Esse conjunto de sinalizadores é usado para indicar quanto do processo pai deve ser compartilhado com o filho. Se nenhum dos sinalizadores estiver marcado, não haverá compartilhamento; e clone( ) atua da mesma forma que fork( ). Se todos os sinalizadores estiverem marcados, o processo filho compartilha tudo com o pai. Outras combinações de sinalizadores permitem diversos níveis de compartilhamento entre esses dois extremos. O kernel do Linux também cria vários threads de kernel, designadas para tarefas específicas, como o gerenciamento de memória.

É interessante que o Linux não distingue entre processos e threads. Na verdade, o Linux utiliza o termo *tarefa* – em vez de *processo* ou *thread* – quando se refere a um fluxo de controle dentro de um programa. Contudo, várias implementações Pthreads estão disponíveis para o Linux; consulte a Bibliografia para obter os detalhes.

## 5.7 Threads em Java

Como já visto, o suporte para threads pode ser fornecido no nível do usuário com uma biblioteca do tipo Pthreads. Além do mais, a maioria dos sistemas operacionais também oferece suporte para threads no nível do kernel. Java é uma dentre um pequeno número de linguagens que oferece suporte no nível de linguagem para a criação e o gerenciamento de threads. Todavia, como as threads são gerenciados pela máquina virtual Java (JVM), e não por uma biblioteca no nível do usuário ou do kernel, é difícil classificar as threads Java como sendo no nível do usuário ou do kernel. Nesta seção, apresentamos as threads Java como uma alternativa para os modelos estritos no nível do usuário ou do kernel. Também discutimos como uma thread Java pode ser associada a thread de kernel subjacente.

Todos os programas Java incluem pelo menos uma única thread de controle. Até mesmo um programa Java simples, consistindo apenas em um método main( ), é executado como uma única thread na JVM. Além disso, Java oferece comandos que permitem que o desenvolvedor crie e manipule threads de controle adicionais dentro do programa.

### 5.7.1 Criação da thread

Um modo de criar uma thread explicitamente é criar uma nova classe derivada da classe Thread e redefinir o método run( ) da classe Thread. Essa técnica é mostrada na Figura 5.8.

Um objeto dessa classe derivada será executado como uma thread de controle separada na JVM. No entanto, a criação de um objeto derivado da classe Thread não cria a nova thread; em vez disso, é o método start( ) que cria essa nova thread. A chamada do método start( ) para o novo objeto (1) aloca memória e inicializa uma nova thread na JVM e (2) chama o método run( ), tornando a thread elegível para ser executada pela JVM. (Nota: nunca chame o método run( ) diretamente. Chame o método start( ), e ele chamará o método run( ) em seu favor.)

Quando esse programa é executado, duas threads são criadas pela JVM. A primeira é a thread associada à aplicação – a thread que inicia a execução no método main( ). A segunda thread é a thread run-

```
class Worker1 extends Thread
{
   public void run( ) {
      System.out.println("Eu sou uma thread trabalhadora ");
   }
}

public class First
{
   public static void main(String args[ ]) {
      Thread runner = new Worker1( );

      runner.start( );

      System.out.println("Eu sou a thread principal");
   }
}
```

FIGURA 5.8 *Criação da thread estendendo a classe* `Thread`.

ner, criada com o método `start( )`. A thread runner inicia a execução em seu método `run( )`.

Outra opção para criar uma thread separada 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( )`. (A classe `Thread`, além de definir métodos estáticos e de instância, também implementa a interface `Runnable`. Isso explica por que uma classe derivada da thread precisa definir um método `run( )`.)

A implementação da interface `Runnable` é semelhante à extensão da classe `Thread`. A única mudança é que "`extends Thread`" é substituído por "`implements Runnable`":

```
class Worker2 implements Runnable
{
    public void run( ) {
      System.out.println("Eu sou uma thread
trabalhadora.");
    }
}
```

Contudo, a criação de uma nova thread a partir de uma classe que implementa `Runnable` é ligeiramente diferente da criação de uma thread a partir de uma classe que estende `Thread`. Como a nova classe não estende `Thread`, ela não tem acesso aos métodos estáticos ou de instância – como o método `start( )` – da classe `Thread`. Contudo, um objeto da classe `Thread` ainda é necessário, pois é o método `start( )` que cria uma nova thread de controle. A Figura 5.9 mostra como as threads podem ser criadas usando a interface `Runnable`.

Na classe `Second`, um novo objeto `Thread` é criado, recebendo um objeto `Runnable` em seu construtor. Quando a thread é criada com o método `start( )`, a nova thread inicia a execução no método `run( )` do objeto `Runnable`.

Por que Java admite duas técnicas para a criação de threads? Qual técnica é mais apropriada para uso em que situações?

```
public class Second
{
   public static void main(String args[ ]) {
      Thread thrd = new Thread(new Worker2( ));

      thrd.start( );

      System.out.println("Eu sou a thread principal");
   }
}
```

FIGURA 5.9 *Criação de thread implementando a interface* `Runnable`.

A primeira pergunta é fácil de responder. Como Java não admite herança múltipla, se uma classe já for derivada de outra classe, ela também não poderá estender a classe `Thread`. Um bom exemplo é que um applet já estende a classe `Applet`. Para usar multithreads em um applet, você estende a classe `Applet` e implementa a interface `Runnable`:

```
public class ThreadedApplet extends Applet
implements Runnable
{
    . . .
}
```

A resposta para a segunda pergunta é menos óbvia. Os puristas orientados a objeto poderiam dizer que, a menos que você esteja aperfeiçoando a classe `Thread`, a classe não deverá ser estendida. (Esse ponto é discutível, pois muitos dos métodos na classe `Thread` são definidos como `final`.) Não tentaremos determinar a resposta correta neste debate. Para este texto, adotamos a prática de implementar a interface `Runnable`, pois essa técnica parece ser a mais utilizada atualmente.

### 5.7.2 Estados da thread

Uma thread em Java pode estar em um destes quatro estados:

1. *Novo:* uma thread está nesse estado quando um objeto para a thread é criado (ou seja, a instrução `new`).
2. *Executável:* 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, esse fluxo passa do estado novo para o estado executável. Uma thread no estado executável é elegível para ser executada pela JVM. Observe que Java não distingue entre uma thread elegível para execução e uma thread sendo executada. Uma thread sendo executada ainda está no estado executável.
3. *Bloqueado:* uma thread torna-se bloqueada se realizar uma instrução de bloqueio – por exemplo, realizando E/S – ou se invocar certos métodos de Thread da Java, como `sleep( )`.
4. *Morto:* uma thread passa para o estado morto quando o método `run( )` termina.

Não é possível determinar o estado exato de uma thread, embora o método `isAlive( )` retorne um valor `boolean` que um programa pode usar para determinar se uma thread está ou não no estado morto. A Figura 5.10 ilustra os diferentes estados da thread e rotula diversas transições possíveis.

### 5.7.3 Unindo threads

A thread criada no programa da Figura 5.9 é executada independente da thread que o cria (a thread associada ao método `main( )`). Em situações em que a thread que cria deseja esperar por quaisquer threads que tenha criado, Java oferece o método `join( )`. Chamando `join( )`, a thread que cria é capaz de esperar até o método `run( )` da thread associada terminar antes de continuar sua operação. O método `join( )` é útil em situações em que a thread que cria só pode continuar depois de uma thread trabalhadora ter sido concluída. Por exemplo, suponha que a thread *A* crie a thread *B* e a thread *B* realize cálculos que a thread *A* exige. Nesse cenário, a thread *A* se unirá a thread *B*.

Ilustramos o método `join( )` modificando o código da Figura 5.9 da seguinte maneira:

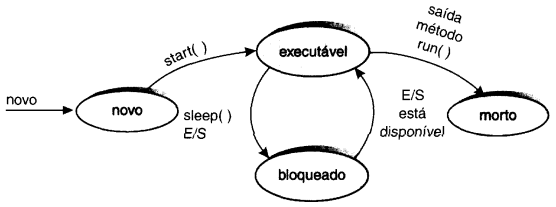



FIGURA 5.10 *Estados da thread em Java.*

```
Thread thrd = new Thread(new Worker2( ));
thrd.start( );

try {
    thrd.join( );
} catch (InterruptedException ie) { }
```

Observe que `join( )` pode lançar uma InterruptedException, que ignoraremos por enquanto. Vamos discutir sobre o tratamento dessa exceção no Capítulo 7.

### 5.7.4 Cancelamento de thread

Na Seção 5.3.2, discutimos questões referentes ao cancelamento de thread. 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. O Capítulo 8 explica por que `stop( )` foi desaprovado.

É possível cancelar uma thread usando o cancelamento adiado, conforme descrevemos na Seção 5.3.2. Lembre-se de que 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 5.11 ilustra como o cancelamento adiado funciona quando o método `isInterrupted( )` é utilizado.

Uma instância de um `InterruptibleThread` pode ser interrompida usando o seguinte código:

```
Thread thrd = new Thread(new
InterruptibleThread( ));
thrd.start( );
. . .
thrd.interrupt( );
```

Neste 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 cha-

```
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 estou interrompida!");
            break;
         }
      }
      // limpa e termina
   }
}
```

FIGURA 5.11 *Cancelamento adiado usando o método isInterrupted( ).*

mar 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 freqüência esse status de interrupção. Tradicionalmente, Java não desperta uma thread bloqueada em uma operação de E/S usando o pacote java.io. Qualquer thread bloqueada realizando E/S nesse pacote não 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.

### 5.7.5 Dados específicos da thread

Na Seção 5.3.5, descrevemos os dados específicos da thread, permitindo que uma thread tenha sua própria cópia privada dos dados. À primeira vista, pode parecer que Java não precisa de dados específicos da thread, e esse ponto de vista está parcialmente correto. Tudo o que é preciso para que cada thread tenha seus próprios dados privados é criar threads subclassificando a classe Thread e declarar dados de instância em sua classe. Essa técnica funciona bem quando 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 –, então uma técnica alternativa é necessária.

A API Java oferece 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 5.12, 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 exce-

```
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 5.12 *Usando a classe ThreadLocal.*

ção a errorCode usando o método set( ) da classe ThreadLocal. Agora considere um cenário em que duas threads – digamos, thread 1 e thread 2 – chamam 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 5.13 ilustra como uma thread pode perguntar sobre o valor de errorCode( ) depois de chamar transaction( ).

### 5.7.6 A JVM e o sistema operacional do host

A JVM normalmente é implementada em cima de um sistema operacional host. Essa configuração permite que a JVM esconda os detalhes da implementação do sistema operacional subjacente e ofereça 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 2000 utiliza o modelo um-para-um; portanto, cada thread Java para uma JVM rodando nesses sistemas 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 implementou a JVM usando o modelo muitos-para-um (chamado **green threads**). 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.

```
class Worker implements Runnable
{
   private static Service provider;

   public void run( ) {
      provider.transaction( );
      System.out.println(provider.getErrorCode( ));
   }
}
```

FIGURA 5.13 *Perguntando o valor dos dados de ThreadLocal.*

### 5.7.7 Exemplo de solução com multithreads

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 5.14 primeiro cria uma caixa de correio para a colocação de mensagens em buffers, usando a classe MessageQueue desenvolvida no Capítulo 4. Depois, ela cria threads produtor e consumidor separados (Figuras 5.15 e 5.16, 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 não é de bloqueio, o consumidor precisa verificar se a mensagem apanhada é nula.

## 5.8 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 a capacidade de tirar proveito das arquiteturas multiprocessadas.

As threads no nível do usuário são threads visíveis ao programador e desconhecidas do kernel. Uma biblioteca de threads no espaço do usuário gerencia as threads no nível do usuário. 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. 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.

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 trata-

```
public class Factory
{
   public Factory( )
   {
      // Primeiro cria o buffer de mensagens
      Channel mailBox = new MessageQueue( );

      // Cria as threads produtor e consumidor e passa
      // a cada thread uma referência ao objeto mailBox
      Thread producerThread = new Thread(
        new Producer(mailBox));
      Thread consumerThread = new Thread(
        new Consumer(mailBox));

      ;; Inicia as threads.
      producerThread.start( );
      consumerThread.start( );
   }

   public static void main(String args[ ]) {
      Factory server = new Factory( );
   }
}
```

FIGURA 5.14 *A classe Factory.*

```
import java.util.Date;

class Producer implements Runnable
{
   private Channel mbox;

   public Producer(Channel mbox) {
      this.mbox = mbox;
   }

   public void run( ) {
      Date message;

      while (true) {
         // dorme um pouco
         SleepUtilities.nap( );

         // produz item e o inclui no buffer
         message = new Date( );

         System.out.println("Produtor produziu " + message);
         mbox.send(message);
      }
   }
}
```

FIGURA 5.15 *Thread Producer.*

```
import java.util.Date;

class Consumer implements Runnable
{
   private Channel mbox;

   public Consumer(Channel mbox) {
      this.mbox = mbox;
   }

   public void run( ) {
      Date message;

      while (true) {
         // dorme um pouco
         SleepUtilities.nap( );

         // consome um item do buffer
         message = (Date)mbox.receive( );

         if (message != null)
            System.out.println("Consumidor consumiu " + message);
      }
   }
}
```

FIGURA 5.16 *Thread Consumer.*

mento de sinais e os dados específicos da thread. A maior parte dos sistemas operacionais modernos oferece suporte do kernel para as threads; entre eles estão Windows NT e Windows XP, Solaris e Linux. A API Pthreads oferece um conjunto de funções para criar e gerenciar threads no nível do usuário. Java oferece uma API semelhante para dar suporte às threads. Contudo, como as threads Java são controladas pela JVM, e não pela biblioteca de threads no nível do usuário ou kernel, elas não estão sob a categoria de threads no nível do usuário ou do kernel.

## Exercícios

**5.1** Forneça dois exemplos de programação em que o uso de multithreads oferece melhor desempenho do que uma solução com uma única thread.

**5.2** Forneça dois exemplos de programação em que o uso de multithreads *não* oferece melhor desempenho do que uma solução com uma única thread.

**5.3** Cite duas diferenças entre as threads no nível do usuário e threads no nível do kernel. Sob que circunstâncias um tipo é melhor que o outro?

**5.4** Descreva as ações tomadas por um kernel para a troca de contexto entre as threads no nível do kernel.

**5.5** Descreva as ações tomadas por uma biblioteca de threads para a troca de contexto entre threads no nível do usuário.

**5.6** Que recursos são usados quando uma thread é criada? Como eles diferem daqueles usados quando um processo é criado?

**5.7** Suponha que um sistema operacional mapeie threads no nível do usuário no kernel usando o modelo muitos-para-muitos, no qual o mapeamento é feito por LWPs. Além disso, o sistema permite que os desenvolvedores criem threads de tempo real. É necessário associar uma thread de tempo real a um LWP? Explique.

**5.8** Um programa Pthreads que realiza a função de somatório foi mostrado na Seção 5.4. Reescreva esse programa em Java.

**5.9** Escreva um programa em Java ou Pthreads multithreads, que gere a seqüência de Fibonacci. Esse programa deverá atuar da seguinte maneira: o usuário executará o programa e informará na linha de comandos a quantidade de números de Fibonacci que o programa terá de gerar. O programa, então, criará uma thread separada, que gerará os números da seqüência de Fibonacci.

**5.10** Escreva um programa em Java ou Pthreads multithreads, 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.

**5.11** Escreva um programa em Java ou Pthreads multithreads, que realize a multiplicação de matriz. Especificamente, use 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* ($C_{i,j}$) é a soma dos produtos dos elementos para a linha *i* na matriz *A* e coluna *j* na matriz *B*. Ou seja,

$$C_{i,j} = \sum_{n=1}^{K} A_{i,n} \times B_{n,j}$$

Por exemplo, se *A* fosse uma matriz 3 por 2 e *B* fosse uma matriz 2 por 3, o elemento C3,1 seria a soma de $A_{3,1} \times B_{1,1}$ e $A_{3,2} \times B_{2,1}$. Calcule cada elemento $C_{i,j}$ em uma thread separada. Isso envolverá a criação de $M \times N$ threads.

**5.12** Modifique o servidor de data baseado em socket, do Capítulo 4, de modo que o servidor atenda à requisição de cada cliente em uma thread separada.

## Notas bibliográficas

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 de um 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. Zabatta e Young [1998] compara as threads do Windows NT e Solaris em um multiprocessador simétrico. Pinilla e Gill [2003] comparam o desempenho da thread Java nos sistemas Linux, Windows e Solaris.

Vahalia [1996] aborda o uso de threads em diversas versões do UNIX. Mauro e McDougall [2001] descrevem os desenvolvimentos recentes no uso de threads pelo kernel do Solaris. Solomon e Russinovich [2000] discutem o uso de threads no Windows 2000. Bovet e Cesati [2002] explicam como o Linux trata da questão das threads.

As informações sobre a programação Pthreads são dadas em Lewis e Berg [1998] e Butenhof [1997]. Oaks e Wong [1999], Lewis e Berg [2000], e Holub [2000] discutem o uso de multithreads em Java. Beveridge e Wiener [1997] discutem o uso de multithreads com sistemas Win32.

CAPÍTULO 6

# 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 5, introduzimos as threads ao modelo de processo. Nos sistemas operacionais que admitem, 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 idéias específicas da thread.

## 6.1 Conceitos básicos

Em um sistema monoprocessado, 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 idéia 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.

### 6.1.1 Ciclo de Burst CPU- E/S

O sucesso do escalonamento da CPU depende da seguinte 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 6.1).

As durações dos burst de CPU foram medidas extensivamente. Embora variem bastante de um pro-

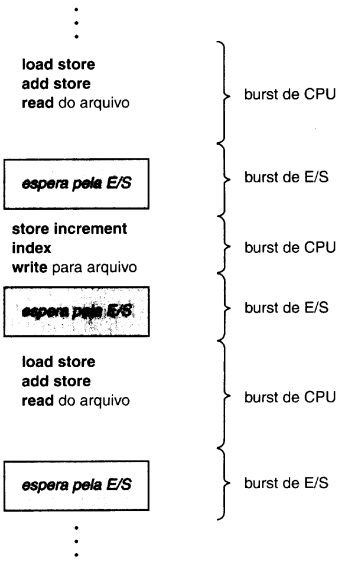



FIGURA 6.1 *Alternando a seqüência de bursts CPU-E/S.*

cesso para outro e de um computador para outro, costumam ter uma curva de freqüência semelhante à que aparece na Figura 6.2. 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 I/O-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.

### 6.1.2 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.

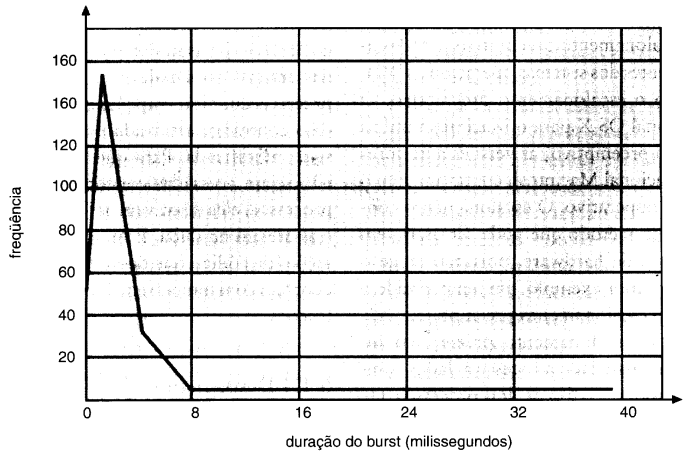



FIGURA 6.2 *Histograma dos tempos de burst da CPU.*

### 6.1.3 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 de espera 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 subseqüentes dos sistemas operacionais Windows têm usado o escalonamento preemptivo. O sistema operacional OS X para o Macintosh 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 7.

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 se apropria 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 'Escalonamento em múltiplos processadores" e "Escalonamento em tempo real".

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 freqüência e contêm poucas instruções.

### 6.1.4 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 o modo do 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)**.

## 6.2 Critérios de escalonamento

Diferentes algoritmos de escalonamento de CPU possuem diferentes propriedades e podem 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 uma 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 (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*. 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 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 é limitado pela velocidade do dispositivo de saída.

É importante maximizar a utilização e a throughput da CPU, reduzindo o turnaround, 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, ao invés 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 seqüência de centenas de bursts de CPU e bursts de

E/S. No entanto, para simplificar, consideramos 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 "Avaliação de algoritmo", mais adiante neste capítulo.

## 6.3 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.

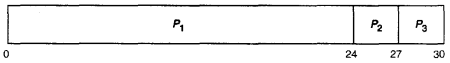
### 6.3.1 Escalonamento First Come First Server – FCFS

Certamente o mais simples dos algoritmos de escalonamento de CPU é o algoritmo de escalonamento **First Come First Serve – FCFS (primeiro a chegar, primeiro a ser atendido)**. 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 PCB 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.

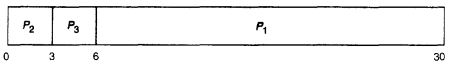
Entretanto, o tempo de espera médio sob a política FCFS normalmente é muito longo. Considere o seguinte conjunto de processos que chegam no instante 0, com a quantidade de tempo de burst de CPU indicada em milissegundos:

| 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, obtemos o resultado mostrado no seguinte **diagrama de Gantt**:




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 Gantt:




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 IO-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 I/O-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.

O algoritmo de escalonamento FCFS é não preemptivo. Quando a CPU é alocada a um processo,

ele 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 mantivesse a CPU por um período estendido.

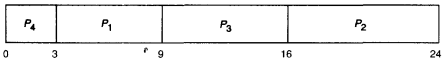
### 6.3.2 Escalonamento Shortest Job First – SJF

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 a maioria das pessoas e livros-texto refere-se a esse tipo de escalonamento como SJF.

Como um 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. Se estivéssemos usando o esquema de escalonamento FCFS, então o tempo de espera médio seria 10,25 milissegundos.

O algoritmo de escalonamento SJF provavelmente é o *ideal*, pois oferece 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. Conseqüentemente, 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 freqüência no escalonamento de longo prazo.

Embora o algoritmo SJF seja ideal, ele não pode ser implementado no nível do escalonamento de CPU 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. Assim, 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. Seja $t_n$ o tempo do enésimo burst de CPU, e seja $\tau_{n+1}$ nosso valor previsto para o próximo burst de CPU. Então, para $\alpha$, $0 \leq \alpha \leq 1$, defina

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\tau_n$$

Essa fórmula define uma **média exponencial**. O valor de $\tau_n$ contém nossa informação mais recente; $\tau_n$ armazena o histórico passado. O parâmetro $\alpha$ 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 $\tau_0$ inicial pode ser definido como uma constante ou como uma média geral do sistema. A Figura 6.3 mostra uma média exponencial com $\alpha = 1/2$ e $\tau_0 = 10$.

Para entender o comportamento da média exponencial, podemos expandir a fórmula para $\tau_{n+1}$ substituindo por $\tau_n$, para encontrar

$$\tau_{n+1} = \alpha t_n + (1 - \alpha)\alpha\tau_{n-1} + \ldots +$$
$$(1 - \alpha)^j \alpha\tau_{n-j} + \ldots + (1 - \alpha)^{n+1}\tau_0.$$

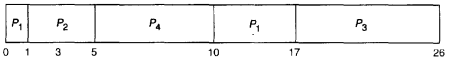
Como tanto $\alpha$ quanto $(1 - \alpha)$ são menores ou iguais a 1, cada termo sucessivo possui peso menor do que seu predecessor.

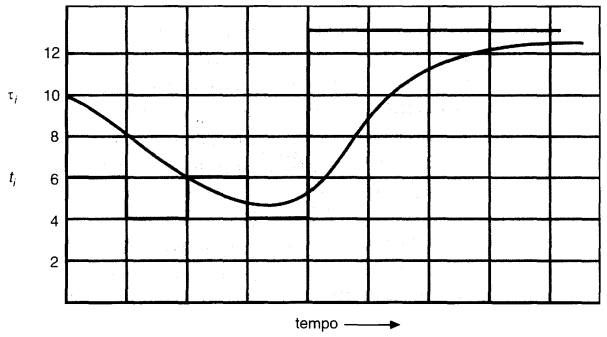
O algoritmo SJF pode ser preemptivo ou não preemptivo. A opção surge quando um novo processo chega na fila de prontos durante a execução de um processo anterior. O próximo burst de CPU do novo processo que chega pode ser menor do que aquilo que resta do processo atualmente em execução. Um algoritmo SJF 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 um 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:




| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| burst de CPU($t_i$) | 6 | 4 | 6 | 4 | 13 | 13 | 13 | ... |
| "chute"($\tau_i$) 10 | 8 | 6 | 6 | 5 | 9 | 11 | 12 | ... |

FIGURA 6.3 *Previsão de tempo do próximo burst de CPU.*

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.

### 6.3.3 Escalonamento por prioridade

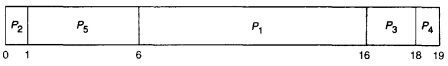
O algoritmo SJF é um caso especial do **algoritmo de escalonamento por prioridade**. Uma prioridade é associada a cada processo, e a CPU é alocada ao processo com a maior prioridade. Processos com a mesma prioridade são escalonados na ordem FCFS. Um algoritmo SJF é um algoritmo por prioridade no qual a prioridade ($p$) é o inverso do próximo burst de CPU (previsto). 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 4095. Contudo, não existe um acordo geral com relação a 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 um exemplo, considere o seguinte conjunto de processos, considerados como tendo chegado no momento 0, na ordem $P_1, P_2, \ldots, P_5$, 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 diagramâ de Gantt:




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 simplesmente 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, um constante de processos de prioridade mais alta pode impedir que um processo de baixa prioridade sequer tenha atenção da CPU. Em geral, poderá acontecer uma destas duas coisas: ou o processo por fim será executado (às 2 horas da manhã de domingo, quando o sistema não estiver sobrecarregado) ou o sistema do computador 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), poderíamos aumentar em 1 a prioridade do processo aguardando a cada 15 minutos. Por fim, até mesmo um processo com uma 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.

### 6.3.4 Escalonamento Round-Robin

O **algoritmo de escalonamento Round-Robin – RR (revezamento)** foi criado para sistemas de tempo compartilhado. Ele é semelhante ao escalonamento FCFS, mas a preempção é acrescentada para alternar 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. 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.

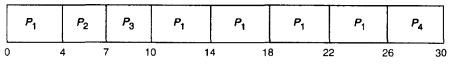
Para implementar o escalonamento RR, mantemos a fila de prontos como uma fila de processos FIFO. Novos processos são acrescentados ao final da fila de prontos. O escalonador de CPU apanha o primeiro processo da fila de prontos, define um temporizador para interromper após 1 quantum de tempo e despacha o processo.

Em seguida, acontecerá uma destas duas coisas: 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 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 é *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 recebe $1/n$ 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. Se o quantum de tempo for extremamente pequeno (digamos, 1 milissegundo), a técnica RR será chamada **processor sharing (compartilhamento de processador)** e (teoricamente) cria 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 realidade, 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. Vamos considerar 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 6.4).

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 também depende do tamanho do quantum de tempo. Como podemos ver na Figura 6.5, o turnaround médio de um conjunto de processos não necessariamente melhora com o aumento de tamanho do quantum de tempo. Em geral, o turnaround 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 médio é 29. Contudo, se o quantum de tempo for 10, o turnaround médio cai para 20. Se o tempo de troca de contexto for incluído, o turnaround médio aumenta para um quantum de tempo menor, pois mais trocas de contexto são necessárias.

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, o escalonamento RR se degenera para a política FCFS. Uma regra prática é que 80% dos bursts de CPU devem ser menores do que o quantum de tempo.

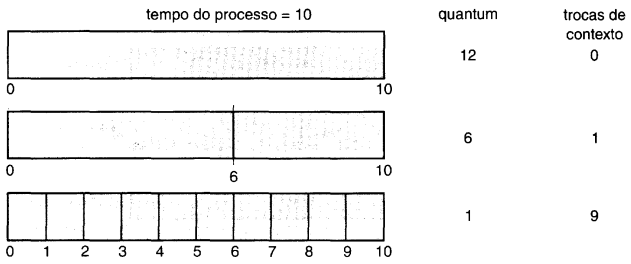
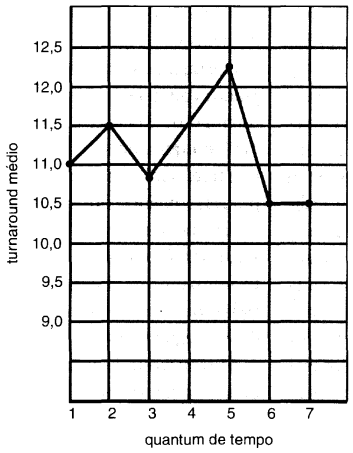



FIGURA 6.4 *O modo como um quantum de tempo menor aumenta as trocas de contexto.*




| processo | tempo |
|---|---|
| $P_1$ | 6 |
| $P_2$ | 3 |
| $P_3$ | 1 |
| $P_4$ | 7 |

FIGURA 6.5 *O modo como o turnaround varia com o quantum de tempo.*

### 6.3.5 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 (fila multinível)** divide a fila de prontos (ready queue) em várias filas separadas (Figura 6.6). Os processos são atribuídos 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

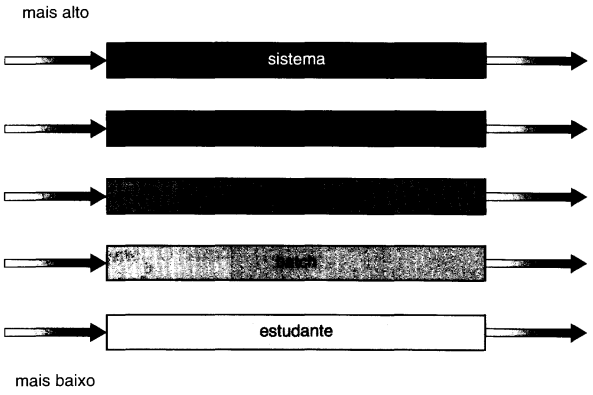



FIGURA 6.6 *Escalonamento de multilevel queue.*

processos de primeiro e segundo plano. A fila de primeiro plano poderia ser escalonada por um algoritmo RR, enquanto a fila de segundo plano seria escalonada por um algoritmo FCFS.

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 um 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 uma certa parte do tempo de CPU, que pode então ser escalonado entre os diversos processos em sua fila. Por exemplo, no exemplo da fila de segundo-primeiro plano, 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.

### 6.3.6 Escalonamento Multilevel Feedback-Queue

Em geral, em um algoritmo de escalonamento de fila multinível, os processos recebem uma fila permanentemente na entrada do sistema. Os processos não se movem entre as filas. Por exemplo, se houver filas separadas para processos de primeiro e segundo plano, 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 mova entre as filas. A idéia é separar os processos com diferentes 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 6.7). 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.

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, então será substituído e colocado na fila 2. Os processos na fila 2 são executados com base na política FCFS, mas são executados somente quando as filas 0 e 1 estiverem vazias.

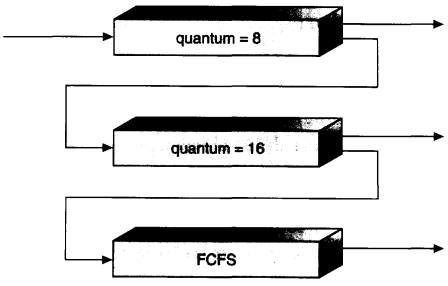



FIGURA 6.7 *Multilevel feedback-queues.*

Esse algoritmo de escalonamento oferece 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:

- O número de filas
- O algoritmo de escalonamento para cada fila
- O método usado para determinar quando atualizar um processo para uma fila de maior prioridade
- O método usado para determinar quando rebaixar um processo para uma fila de menor prioridade
- O 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.

## 6.4 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 problema do escalonamento será relativamente mais complexo. Muitas possibilidades foram experimentadas; e, como vimos com o escalonamento de CPU com único processador, não existe uma solução melhor. Nesta seção, vamos discutir os 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.

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.

Se vários processadores idênticos estiverem disponíveis, então poderá ocorrer o **compartilhamento de carga**. Poderíamos oferecer uma fila separada para cada processador. Nesse caso, no entanto, um processador poderia estar ocioso, com uma fila vazia, enquanto outro processador estivesse extremamente ocupado. Para evitar essa situação, podemos usar uma única fila de prontos comum. Todos os processos entram em uma fila e são escalonados para qualquer processador disponível.

Nesse esquema, uma dentre duas técnicas de escalonamento pode ser usada. Uma técnica tem todas as decisões de escalonamento, processamento de E/S e outras atividades do sistema tratadas por um único processador – 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, aliviando a necessidade de compartilhamento de dados.

Uma segunda técnica utiliza o **multiprocessamento simétrico (SMP)**, no qual cada processador é auto-escalonado. Cada processador examina a fila de prontos comum e seleciona um processo para executar. Como veremos no Capítulo 7, se tivermos vários processadores tentando acessar e atualizar uma estrutura de dados comum, cada processador 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 NT, windows 2000, Windows XP, Solaris, Linux e Mac OS X.

## 6.5 Escalonamento em tempo real

No Capítulo 1, discutimos a importância cada vez maior dos sistemas operacionais de tempo real.

Aqui, descrevemos a facilidade de escalonamento necessária para dar suporte à computação de tempo real dentro de um sistema computadorizado de uso geral.

A computação em tempo real pode ser de dois tipos. Os sistemas de **tempo real rígido (hard real time)** são exigidos para completar uma tarefa crítica dentro de uma quantidade de tempo garantida. Em geral, um processo é submetido junto com uma declaração da quantidade de tempo dentro da qual ele precisa completar ou realizar a E/S. O escalonador ou admite o processo, garantindo que o processo termine a tempo, ou rejeita a requisição por ser impossível. Tal garantia, feita sob a **reserva de recurso**, exige que o escalonador saiba exatamente quanto tempo é necessário para realizar cada tipo de função do sistema operacional; portanto, cada operação precisa ter garantia de levar uma quantidade de tempo máxima. Essa garantia é impossível em um sistema com armazenamento secundário ou de memória virtual, como mostraremos nos Capítulos de 9 a 14, pois esses subsistemas causam variação inevitável e imprevisível na quantidade de tempo necessária para executar determinado processo. Portanto, os sistemas de tempo real rígido são compostos de software de uso especial em execução no hardware dedicado ao seu processo crítico, e não possuem toda a funcionalidade dos computadores e sistemas operacionais modernos.

A computação em **tempo real flexível (soft real time)** é menos restritiva. Ela exige que os processos críticos tenham prioridade em relação aos menos favorecidos. Embora a inclusão da funcionalidade de tempo real flexível para um sistema de tempo compartilhado possa causar uma alocação injusta de recursos e resultar em atrasos maiores para alguns processos, ou até mesmo starvation, pelo menos isso pode ser alcançado. O resultado é um sistema de uso geral que também pode admitir multimídia, gráficos interativos de alta velocidade e uma série de tarefas que não funcionarão de forma aceitável em um ambiente que não aceita a computação de tempo real flexível.

A implementação da funcionalidade de tempo real requer um projeto cuidadoso do escalonador e aspectos relacionados do sistema operacional. Primeiro, o sistema precisa ter escalonamento por prioridade, e os processos de tempo real precisam ter a prioridade mais alta. A prioridade dos processos de tempo real não pode diminuir com o tempo, embora a prioridade dos processos que não são de tempo real possa diminuir. Em segundo lugar, a latência de despacho precisa ser pequena. Quanto menor a latência, mais rapidamente um processo de tempo real poderá iniciar sua execução quando estiver executável.

É relativamente simples garantir a primeira propriedade. Por exemplo, podemos impedir que os processos de tempo real envelheçam, garantindo assim que a prioridade dos diversos processos não mudará. Entretanto, garantir a segunda propriedade é muito mais complicado. O problema é que muitos sistemas operacionais são forçados a esperar o término de uma chamada de sistema ou até ocorrer um bloco de E/S antes de poder realizar uma troca de contexto. A latência de despacho desses sistemas pode ser longa, pois algumas chamadas de sistema são complexas e alguns dispositivos de E/S são lentos.

Para reduzir a latência de despacho, precisamos permitir que as chamadas de sistema aceitem preempção. Esse objetivo pode ser alcançado de diversas maneiras. Uma é inserir **pontos de preempção** nas chamadas de sistema de longa duração. Um ponto de preempção verifica se um processo de alta prioridade precisa ser executado. Se precisar, ocorre uma troca de contexto. Depois, quando o processo de alta prioridade terminar, o processo interrompido continua com a chamada de sistema. Os pontos de preempção podem ser colocados apenas em locais *seguros* no kernel – apenas onde as estruturas de dados do kernel não estão sendo modificadas. Como não é prático acrescentar mais do que alguns pontos de preempção em um kernel, a latência de despacho pode ser grande até mesmo quando são utilizados pontos de preempção.

Outro método para lidar com a preempção é tornar o kernel inteiro passível de preempção. Para que a operação correta seja garantida, todas as estruturas de dados do kernel precisam ser protegidas com o uso de vários mecanismos de sincronismo, discutidos no Capítulo 7. Com esse método, o kernel sempre poderá sofrer preempção, já que quaisquer dados do mesmo sendo atualizados estão protegidos contra modificação pelo processo de alta prioridade. Esse método é utilizado no Solaris.

O que acontece se um processo precisar ler ou modificar dados do kernel que estão sendo acessados por um processo de menor prioridade – ou por uma cadeia de processos de menor prioridade? O processo de maior prioridade terá de esperar até terminar um processo de menor prioridade. Esse problema, conhecido como **inversão de prioridade**, pode ser solucionado com o **protocolo de herança de prioridade**, em que todos os processos que estão acessando recursos necessários por um processo de maior prioridade herdam a prioridade maior, até terminar de usar os recursos em questão. Quando terminarem, sua prioridade retorna ao valor original.

Na Figura 6.8, mostramos a composição da latência de despacho. A **fase de conflito** da latência de despacho possui dois componentes:

1. Preempção de qualquer processo executando no kernel
2. Liberação, pelos processos de baixa prioridade, dos recursos necessários pelo processo de alta prioridade

Como um exemplo, no Solaris, a latência de despacho com a preempção desativada está acima de 100 milissegundos. Com a preempção ativada, ela é reduzida para menos de um milissegundo.

## 6.6 Escalonamento de thread

No Capítulo 5, 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 de 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, oferecendo exemplos específicos de escalonamento para Pthreads.

### 6.6.1 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 5.2.1) e muitos-para-muitos (Seção 5.2.3), 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 (PCS – Process Contention Scope)**, pois a disputa

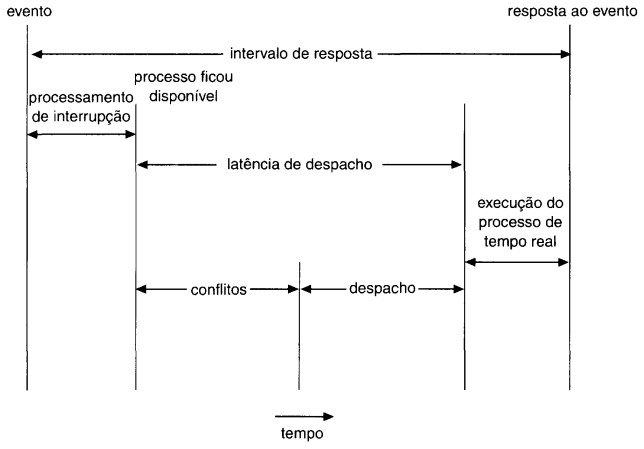



FIGURA 6.8 *Latência de despacho.*

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 (SCS – System Contention Scope)**. A disputa pela CPU com escalonamento SCS ocorre entre todas as threads no sistema. Os sistemas usando o modelo um-para-um (como Windows XP, Solaris 9 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 6.3.4) entre as threads de mesma prioridade.

### 6.6.2 Escalonamento Pthread

Na Seção 5.4, mostramos um exemplo de um programa POSIX Pthread. O padrão POSIX também oferece extensões para a computação de tempo real – POSIX.1b. Nesta seção, abordamos parte da API POSIX Pthread relacionada ao escalonamento de thread. Pthreads define duas classes de escalonamento para threads de tempo real:

- `SCHED_FIFO`
- `SCHED_RR`

`SCHED_FIFO` escalona threads de acordo com uma política FCFS usando uma fila FIFO, conforme esboçada na Seção 6.3.1. Contudo, não existe fatia de tempo entre threads de mesma prioridade. Portanto, a thread de tempo real de maior prioridade na frente da fila FIFO será entregue à CPU até terminar ou ser bloqueada. `SCHED_RR` é semelhante a `SCHED_FIFO`, exceto que oferece fatia de tempo entre as threads de mesma prioridade. Tanto `SCHED_FIFO` quanto `SCHED_RR` são designados como threads de tempo real. Pthreads oferece uma classe de escalonamento adicional – `SCHED_OTHER`, mas sua implementação não é definida e é específica do sistema; ela pode se comportar de forma diferente em diferentes sistemas.

Além disso, Pthreads permite duas políticas diferentes de escalonamento de thread:

- `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 (Seção 5.2.3), a política `PTHREAD_SCOPE_PROCESS` escalona as threads no nível do usuário para os LWPs disponíveis. A quantidade de LWPs é mantida pela biblioteca de threads, talvez usando ativações do escalonador (Seção 5.3.6). 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 (Seção 5.2.2).

Na Figura 6.9, ilustramos um programa Pthread que cria cinco threads separadas usando o algoritmo de escalonamento `SCHED_OTHER` e definindo a política de escalonamento como `PTHREAD_SCOPE_SYSTEM`. O algoritmo `SCHED_OTHER` é definido pela configuração dos atributos para o `attr` da thread, com a função `Pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM)`. A política de escalonamento é definida por meio da chamada de função `pthread_attr_setschedpolicy(&attr, SCHED_OTHER)`.

A API Pthread também permite que o programador altere a prioridade de uma thread.

## 6.7 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 do escalonamento das threads de kernel.

### 6.7.1 Exemplo: escalonamento no Solaris

O Solaris utiliza o escalonamento de thread com base em prioridade. Ele definiu quatro classes de escalonamento, que são, em ordem de prioridade:

```
#include <pthread.h>
#include <stdio.h>
#define NUM_THREADS 5

int main(int argc, char *argv[ ])
{
   int i;
   pthread_t tid[NUM_THREADS];
   pthread_attr_t attr;

   /* apanha os atributos padrão */
   pthread_attr_init(&attr);

   /* define o algoritmo de escalonamento como PROCESS ou SYSTEM */
   pthread_attr_setscope(&attr, PTHREAD_SCOPE_SYSTEM);

   /* define a política de escalonamento – FIFO, RT ou OTHER */
   pthread_attr_setschedpolicy(&attr, SCHED_OTHER);

   /* 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)
{
   printf("Eu sou uma thread\n");
   pthread_exit(0);
}
```

FIGURA 6.9 *API de escalonamento Pthread.*

1. Tempo real
2. Sistema
3. Tempo compartilhado
4. Interativo

Dentro de cada classe existem diferentes prioridades e diferentes algoritmos de escalonamento. O escalonamento no Solaris é ilustrado na Figura 6.10.

A classe de escalonamento padrão para um processo é tempo compartilhado. A política de escalonamento para 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 oferece um bom tempo de resposta para os processos interativos e uma boa 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 uma prioridade maior, para aumentar o desempenho.

A Figura 6.11 ilustra a tabela de despacho para o escalonamento de threads interativas e de tempo compartilhado. Essas duas classes de escalonamento

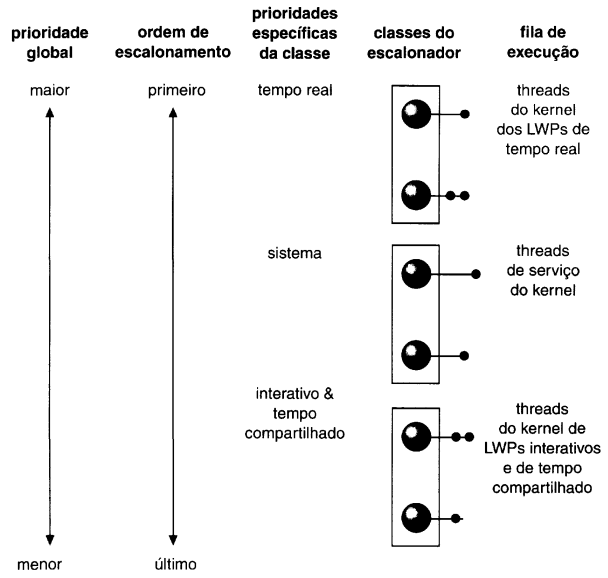



FIGURA 6.10 *Escalabilidade no Solaris.*

| 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 6.11 *Tabela de despacho do Solaris para threads interativas e de tempo compartilhado.*

incluem 60 níveis de prioridade, mas, para resumir, apresentaremos apenas algumas das prioridades, para ilustrar o escalonamento de thread do Solaris. A tabela de despacho mostrada na Figura 6.11 contém os seguintes campos:

- **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, oferecendo suporte para a política de escalonamento de fornecer um bom tempo de resposta para processos interativos.

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 fair share 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**).

O Solaris utiliza a classe do sistema para executar processos 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).

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.

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 round-robin. O Solaris tradicionalmente tem usado o modelo muitos-para-muitos (5.2.3), mas o Solaris 9 passou para o modelo um-para-um (5.2.2).

### 6.7.2 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. Entretanto, o Windows XP não é um sistema operacional de tempo real rígido, pois não garante que uma thread de tempo real começará a ser executada dentro de qualquer limite de tempo em particular.

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. 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 gerenciamento 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**.

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. Essas 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.

Dentro de cada uma das classes de prioridade existe 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 6.12. 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 é 10.

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, a menos que o pai

| | 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 6.12 *Prioridades no Windows XP.*

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 é reduzida. Contudo, a prioridade nunca é reduzida para menos do que a prioridade básica. A redução da prioridade da thread 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 pelo qual a thread estava esperando; por exemplo, uma thread que estava esperando uma E/S do teclado receberia um aumento grande, enquanto uma thread esperando por uma operação de disco receberia um aumento moderado. Essa estratégia costuma oferecer 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 E/S 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 oferecer um desempenho especialmente bom para esse processo. 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.

### 6.7.3 Exemplo: escalonamento no Linux

O Linux oferece dois algoritmos de escalonamento de processo separados. Um é um algoritmo de tempo compartilhado para o escalonamento preemptivo justo entre vários processos; o outro foi criado para tarefas de tempo real, nas quais as prioridades absolutas são mais importantes do que a imparcialidade. Na Seção 6.5, descrevemos uma situação em que os sistemas de tempo real precisam permitir que o kernel seja interrompido para manter baixa a latência de despacho. O Linux só permite que os processos executando no modo do usuário sejam interrompidos. Um processo não pode ser interrompido enquanto estiver executando no modo do kernel, mesmo que um processo de tempo real, com uma prioridade mais alta, esteja disponível para execução.

Parte da identidade de cada processo é uma classe de escalonamento, que define qual desses algoritmos será aplicado ao processo. As classes de escalonamento usadas pelo Linux são definidas nas extensões do padrão POSIX para cálculo de tempo real, que foram abordadas na Seção 6.6.2.

A primeira classe de escalonamento é para os processos de tempo compartilhado. Para os processos convencionais de tempo compartilhado, o Linux usa um algoritmo de prioridade Credit Based (**baseado em crédito**). Cada processo possui um certo número de créditos de escalonamento; quando uma nova tarefa precisa ser escolhida para execução, o processo com mais créditos é selecionado. Toda vez que ocorre uma interrupção com temporizador, o processo em execução perde um crédito; quando seu número de créditos atinge zero, ele é suspenso, e outro processo é escolhido.

Se um processo executável tiver créditos, então o Linux realizará uma operação de novo crédito, acrescentando créditos a *cada* processo no sistema (e não somente aos executáveis), de acordo com a seguinte regra:

$$\text{créditos} = \frac{\text{créditos}}{2} + \text{prioridade}$$

Esse algoritmo costuma misturar dois fatores: o histórico do processo e sua prioridade. Metade dos créditos que um processo ainda mantém será retida após o algoritmo ter sido aplicado, retendo algum histórico do comportamento recente do processo.

Os processos que estão executando o tempo todo costumam esgotar seus créditos rapidamente, mas os processos que gastam muito do seu tempo suspensos podem acumular créditos durante várias operações de novo crédito e, como conseqüência, acabam com uma contagem de crédito maior após essa operação. Esse sistema de crédito dá automaticamente uma prioridade alta aos processos interativos e I/O-bound, para os quais um tempo de resposta rápido é importante.

O uso de uma prioridade de processo no cálculo de novos créditos permite ajustar a prioridade de um processo. As tarefas batch no segundo plano podem receber uma prioridade baixa; elas receberão automaticamente menos créditos do que as tarefas dos usuários e, portanto, receberão uma porcentagem menor do tempo de CPU do que tarefas semelhantes, com prioridades maiores. O Linux utiliza esse sistema de prioridade para implementar o mecanismo de prioridade de processo *nice* padrão do UNIX. O escalonamento de tempo real do Linux é ainda mais simples. O Linux implementa as classes de escalonamento de tempo real exigidas pelo POSIX.1b: First Come First Served (FCFS) e Round-Robin (RR) (Seções 6.3.1 e 6.3.4). Nos dois casos, cada processo tem uma prioridade além de sua classe de escalonamento. No entanto, no escalonamento de tempo compartilhado, os processos de diferentes prioridades ainda podem competir um com o outro até certo ponto; no escalonamento de tempo real, o escalonador sempre executa o processo com a prioridade mais alta. Entre os processos com a mesma prioridade, ele executa o processo que esteve esperando por mais tempo. A única diferença entre escalonamento FCFS e RR é que os processos FCFS continuam a ser executados até saírem ou serem bloqueados, enquanto um processo por revezamento será interrompido depois de um tempo e será movido para o final da fila de escalonamento, de modo que os processos por revezamento com mesma prioridade compartilharão o tempo automaticamente entre si mesmos.

Observe que o escalonamento de tempo real do Linux é de tempo real flexível – ao invés de rígido. O escalonador oferece garantias estritas sobre as prioridades relativas dos processos de tempo real, mas o kernel não oferece quaisquer garantias sobre a rapidez com que um processo de tempo real será escalonado depois de esse processo se tornar executável. Lembre-se de que o código do kernel do Linux nunca pode ser interrompido pelo código no modo do usuário. Se uma interrupção chegar e acordar um processo de tempo real enquanto o kernel já estiver executando uma chamada de sistema em favor de outro processo, o processo de tempo real terá de esperar até terminar ou bloquear a chamada de sistema em execução.

## 6.8 Escalonamento de threads 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 executarão em preferência as 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. 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 (6.3.4) ou não – isso 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 sistema 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( );
   }
}
```

### 6.8.1 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 |
|---|---|
| Thread.MIN_PRIORITY | A prioridade mínima da thread. |
| Thread.MAX_PRIORITY | A prioridade máxima da thread. |
| Thread.NORM_PRIORITY | 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. Quando uma thread é criada, ela recebe a mesma prioridade da thread que a criou. A menos que seja especificada de outra forma, a prioridade padrão para todas as threads é `NORM_PRIORITY`. 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 6.13) aumenta a prioridade da thread em 1 a mais do que a prioridade padrão antes de realizar o restante do método `run( )`.

Como a JVM normalmente é implementada em cima de um sistema operacional host, a prioridade de uma thread Java está relacionada à prioridade da thread do kernel ao qual é mapeada. Em sistemas que admitem níveis de prioridade relativamente baixos, é possível que diferentes prioridades de thread em Java possam ser mapeadas para a mesma prioridade da thread do kernel. Por exemplo, `Thread.NORM_PRIORITY + 1` e `Thread.NORM_PRIORITY + 2` são mapeadas para a mesma prioridade do kernel no Windows NT. Nesse sistema, alterar a prioridade das threads Java pode não ter efeito sobre o modo como tais threads são escalonadas.

## 6.9 Avaliação de algoritmo

Como selecionamos um algoritmo de escalonamento de CPU para determinado sistema? Como vimos na Seção 6.3, 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 6.2, os critérios normalmente são definidos em termos de utilização de CPU, tempo de resposta ou throughput. Para selecionar um algorit-

```
public class HighThread implements Runnable
{
   public void run( ) {
      Thread.currentThread( ).setPriority(Thread.NORM_PRIORITY + 1);
      // restante do método run( )
        . . .
   }
}
```

FIGURA 6.13 *Definindo uma prioridade por meio de `setPriority( )`.*

mo, 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 a throughput de modo que o turnaround seja (na média) linearmente proporcional ao tempo de execução total

Quando os critérios de seleção tiverem sido definidos, desejamos avaliar os diversos algoritmos em consideração. Descrevemos os diversos métodos de avaliação que podemos usar nas Seções de 6.9.1 a 6.9.4.

### 6.9.1 Modelagem determinística

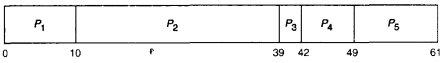
Uma classe importante dos métodos de avaliação é a **avaliação analítica**. A avaliação analítica usa o algoritmo indicado, e a carga de trabalho do sistema para produzir uma fórmula ou número que avalia o desempenho do algoritmo para essa carga de trabalho.

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 indicando em milissegundos:

| Processo | Tempo de burst |
|---|---|
| $P_1$ | 10 |
| $P_2$ | 29 |
| $P_3$ | 3 |
| $P_4$ | 7 |
| $P_5$ | 12 |

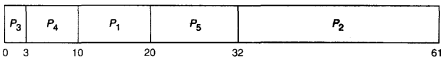
Considere os algoritmos de escalonamento FCFS, SJF e RR (quantum = 10 milissegundos) para esse conjunto de processos. Qual algoritmo daria o menor tempo de espera médio?

Para o algoritmo FCFS, executaríamos os processos como



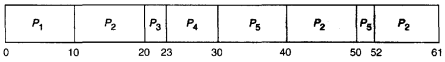

O tempo de espera é de 0 milissegundo para o processo $P_1$, 10 milissegundos para o processo $P_2$, 39 milissegundos para o processo $P_3$, 42 milissegundos para o processo $P_4$ e 49 milissegundos para o processo $P_5$. Assim, o tempo de espera médio é (0 + 10 + 39 + 42 + 49)/5 = 28 milissegundos.

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*, a política SJF resulta em menos de metade do tempo de espera médio obtido com o escalonamento FCFS; o algoritmo RR nos dá um valor intermediário.

A modelagem determinística é simples e rápida. Ela nos dá números exatos, permitindo que os algoritmos sejam comparados. 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 estar executando o mesmo programa várias vezes e podemos medir os 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 podem, então, ser 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.

### 6.9.2 Modelos de enfileiramento

Os processos executados em muitos sistemas variam de um dia para outro, de modo que não existe um conjunto estático de processos (ou tempos) para uso na modelagem determinística. Entretanto, o que pode ser determinada é a distribuição dos bursts de CPU e E/S. Essas distribuições podem ser medidas e depois aproximadas ou simplesmente 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 um exemplo, seja $n$ o tamanho médio da fila (excluindo o processo sendo atendido), seja $W$ o tempo médio de espera na fila, e seja $\lambda$ 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 de 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.

### 6.9.3 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 de computador. 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 distribui-

ções podem ser definidas matemática (uniforme, exponencial, Poisson) ou empiricamente. Se a distribuição tiver de ser definida empiricamente, as medições do sistema real sob 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 freqüência 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 seqüência de eventos reais (Figura 6.14). Depois, usamos essa seqüência para controlar a simulação. As fitas de rastreamento oferecem uma excelente maneira de comparar dois algoritmos com o mesmo conjunto de entradas reais. Esse método pode produzir resultados precisos para suas entradas.

As simulações podem ser caras, exigindo horas de tempo do computador. Uma simulação mais detalhada pode oferecer resultados mais precisos, mas também exige mais tempo do computador. Além disso, as fitas de rastreamento podem exigir uma grande quantidade de espaço para armazenamento. Finalmente, o projeto, a codificação e a depuração do simulador podem ser uma tarefa importante.

### 6.9.4 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 às 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 simplesmente 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 com qualquer avaliação algorítmica 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.

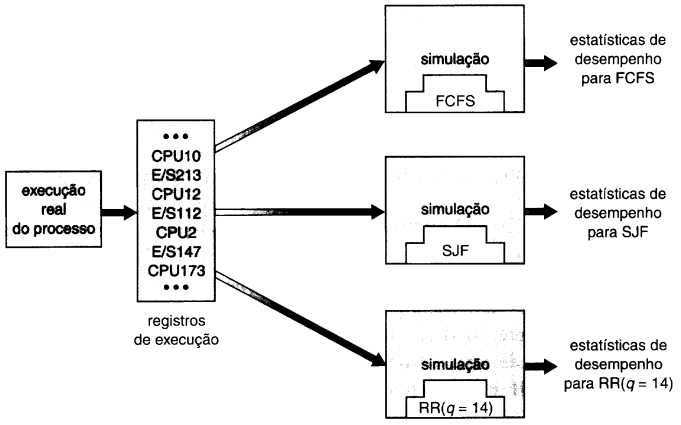



FIGURA 6.14 *Avaliação dos escalonadores de CPU pela simulação.*

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. Essa política resultou em uma situação em que 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 podem ser alterados pelos gerentes do sistema ou pelos usuários, para serem ajustados a uma aplicação específica ou conjunto de aplicações. Por exemplo, uma estação de trabalho que realiza aplicações gráficas de última geração 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. Outra técnica é usar APIs, como `yield( )` e `setPriority( )`, permitindo assim que as aplicações atuem de uma forma mais previsível. A desvantagem dessa técnica é que o ajuste do desempenho de um sistema ou aplicação não resulta em melhor desempenho em situações mais genéricas.

## 6.10 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 com que processos curtos esperem 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 simplesmente 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 se 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 a diferentes algoritmos serem utilizados para diversas classes de processos. O mais comum é 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 feedbak-queue permite aos processos mudarem de uma fila para outra.

Muitos sistemas de computador contemporâneos admitem vários processadores; cada processador realiza seu escalonamento de forma independente. Em geral, existe uma fila de processos (ou threads), todos disponíveis para execução. Cada processador toma uma decisão de escalonamento e seleciona a partir dessa fila.

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, onde 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 favo-

recem os processos interativos em detrimento dos processos batch e CPU-bound.

A JVM utiliza um algoritmo de escalonamento de thread baseada em prioridade, que favorece threads com maior prioridade. A especificação não indica se a JVM deve repartir o tempo das threads; isso fica a cargo da implementação específica da JVM.

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.

## Exercícios

**6.1** Um algoritmo de escalonamento de CPU determina a ordem para a execução de seus processos escalonados. Dados $n$ processos a serem escalonados em um processador sem preempção, quantos escalonamentos possíveis existem? Dê uma fórmula em termos de $n$.

**6.2** Defina a diferença entre escalonamento preemptivo e não preemptivo.

**6.3** Considere o seguinte conjunto de processos, com o tamanho do tempo de burst de CPU dado em milissegundos:

| Processo | Tempo de burst | Prioridade |
|---|---|---|
| $P_1$ | 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.

**a.** Desenhe quatro gráficos de Gantt que ilustrem a execução desses processos usando FCFS, SJF, uma prioridade não preemptiva (um número de prioridade menor significa uma prioridade mais alta) e o escalonamento RR (quantum = 1).

**b.** Qual é o turnaround de cada processo para cada um dos algoritmos de escalonamento no item *a*?

**c.** Qual é o tempo de espera de cada processo para cada um dos algoritmos de escalonamento no item *a*?

**d.** Qual dos escalonamentos na parte *a* resulta no menor tempo de espera médio (em relação a todos os processos)?

**6.4** Suponha que os seguintes processos cheguem para execução nos momentos indicados. Cada processo será executado pela quantidade de tempo indicada. Ao responder as perguntas, use o escalonamento não preemptivo, e baseie todas as decisões nas informações que você tem no momento em que a decisão deve ser tomada.

| 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 turnaround médio para esses processos com o algoritmo de escalonamento FCFS?

**b.** Qual é o turnaround médio pra esses processos com o algoritmo de escalonamento SJF?

**c.** O algoritmo SJF deveria melhorar o desempenho, mas observe que escolhemos executar o processo $P_1$ no momento 0 porque não sabíamos que dois outros processos mais curtos chegariam em breve. Calcule qual será o turnaround médio se a CPU ficar ociosa para a primeira 1 unidade e depois o escalonamento SJF é utilizado. Lembre-se de que os processos $P_1$ e $P_2$ estão aguardando durante esse tempo ocioso, de modo que seu tempo de espera poderá aumentar. Esse algoritmo poderia ser conhecido como *escalonamento por conhecimento do futuro*.

**6.5** Considere uma variante do algoritmo de escalonamento RR, no qual as entradas na fila de prontos são ponteiros para os PCBs.

**a.** Qual seria o efeito de colocar dois ponteiros para o mesmo processo na fila de prontos?

**b.** Quais seriam duas vantagens importantes e duas desvantagens desse esquema?

**c.** Como você modificaria o algoritmo RR básico para conseguir o mesmo efeito sem os ponteiros duplicados?

**6.6** Que vantagem existe em ter diferentes tamanhos de quantum de tempo em diferentes níveis de um sistema de enfileiramento multinível?

**6.7** Considere o seguinte 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 $\alpha$; quando está executando, sua prioridade muda a uma taxa $\beta$. Todos os processos recebem uma prioridade 0 quando entram na fila de prontos. Os parâmetros $\alpha$ e $\beta$ podem ser definidos para dar muitos algoritmos de escalonamento diferentes.

**a.** Qual é o algoritmo que resulta de $\beta > \alpha > 0$?

**b.** Qual é o algoritmo que resulta de $\alpha < \beta < 0$?

**6.8** Muitos algoritmos de escalonamento de CPU são parametrizados. Por exemplo, o algoritmo RR requer um parâmetro para indicar a fatia de tempo. As multilevel feedback queues exigem parâmetros para definir o número de filas, os algoritmos de escalonamento para cada fila, os critérios usados para mover processos entre filas, e assim por diante.

Esses algoritmos, na realidade são 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) permanece entre os seguintes pares de conjuntos de algoritmos?

**a.** Prioridade e SJF

**b.** Multilevel feedback queues e FCFS

**c.** Prioridade e FCFS

**d.** RR e SJF

**6.9** Suponha que um algoritmo de escalonamento (no nível de escalonamento de CPU de curto prazo) favorece os processos que têm usado o menor tempo do processador no passado recente. Por que esse algoritmo favorece programas I/O-bound e não causa starvation permanente nos programas CPU-bound?

**6.10** Explique as diferenças no grau ao qual os algoritmos de escalonamento a seguir são discriminados em favor de processos curtos:

**a.** FCFS

**b.** RR

**c.** Multilevel feedbak queues

**6.11** Explique a distinção entre o escalonamento PCX e SCS.

**6.12** Considere que um sistema operacional associe threads no nível do usuário ao kernel usando o modelo muitos-para-muitos, onde o mapeamento é feito usando LWPs. Além disso, o sistema permite que os desenvolvedores de programas criem threads de tempo real. É necessário vincular uma thread de tempo real a um LWP?

**6.13** Usando o algoritmo de escalonamento do Windows XP, qual é a prioridade numérica de uma thread para os cenários a seguir?

**a.** Uma thread na REALTIME_PRIORITY_CLASS com uma prioridade relativa HIGHEST.

**b.** Uma thread na NORMAL_PRIORITY_CLASS com uma prioridade relativa NORMAL.

**c.** Uma thread na HIGH_PRIORITY_CLASS com uma prioridade relativa ABOVE_NORMAL.

**6.14** Considere o algoritmo de escalonamento no sistema operacional Solaris para as threads de tempo compartilhado:

**a.** Qual é o quantum de tempo (em milissegundos) para uma thread com prioridade 10? E com prioridade 55?

**b.** 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?

**c.** 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?

## 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 6.7 foi sugerido por Kleinrock [1975].

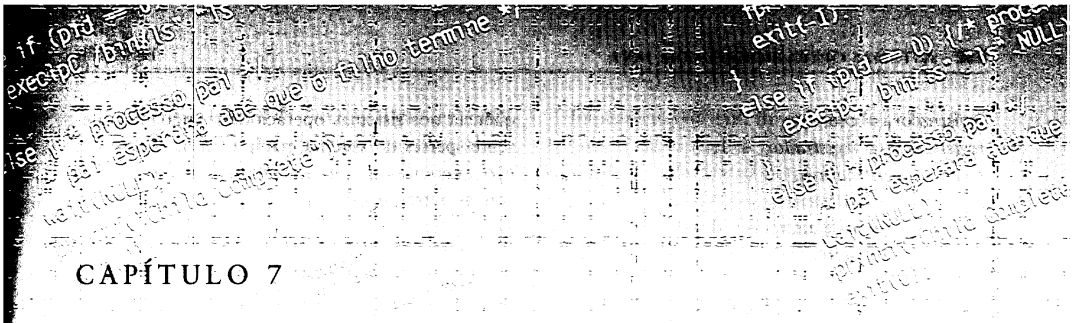
Anderson e outros [1989], e Lewis e Berg [1998] abordaram o escalonamento de thread. As discussões com relação a escalonamento com multiprocessadores foram apresentadas por Tucker e Gupta [1989], Zahorjan e McCann [1990], Feitelson e Rudolph [1990], e Leutenegger e Vernon [1990].

As discussões sobre escalonamento em sistemas de tempo real foram oferecidas por Abbot [1984], Jensen e outros [1985], Hong e outros [1989], e Khanna e outros [1992]. Um artigo especial sobre sistemas operacionais de tempo real foi editado por Zhao [1989].

Os escalonadores justos foram abordados por Henry [1984], Woodside [1986], e Kay e Lauder [1988].

A discussão sobre políticas de escalonamento usada no sistema operacional UNIX V foi apresentada por Bach [1987]; a discussão para o UNIX BSD 4.4 foi apresentada por McKusick e outros [1996]; e para o sistema operacional Mach, por Black [1990]. Bovet e Cesati [2002] abordam o escalonamento no Linux. O escalonamento no Solaris foi descrito por Mauro e McDougall [2001]. Solomon [1998], e Solomon e Russinovich [2000] discutiram o escalonamento no Windows NT e Windows 2000, respectivamente. Butenhof [1997], e Lewis e Berg [1998] aborda o escalonamento em sistemas Pthreads.

O escalonamento de thread em Java, de acordo com a especificação da JVM, é descrito por Lindholm e Yellin [1999]. Alguns livros de uso geral relacionados ao escalonamento de thread em Java são Holub [2000], Lewis e Berg [2000], e Oaks e Wong [1999].



CAPÍTULO 7

# Sincronismo de processos

Um processo cooperativo, conforme discutido no Capítulo 4, é aquele que pode afetar ou ser afetado por outros processos que estão sendo executados 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 5. 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 ou threads cooperativas, que compartilham um espaço de endereços lógico, permitindo a manutenção da coerência dos dados.

## 7.1 Segundo plano

No Capítulo 4, desenvolvemos um modelo de um sistema consistindo em processos ou threads seqüenciais cooperativas; todas executando de forma assíncrona e possivelmente compartilhando dados. Ilustramos esse modelo com um problema de produtor-consumidor, que representa os sistemas operacionais. O fragmento de código para a thread produtor é o seguinte:

```
while (count == BUFFER_SIZE)
   ; // não faz nada

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

O código para o consumidor é

```
while (count == 0)
   ; // não faz nada

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

Embora as rotinas do produtor e do consumidor estejam corretas separadamente, elas não funcionam de forma correta quando executadas ao mesmo tempo. O motivo é que as threads compartilham a variável `count`, que serve como contador para o número de itens no buffer. 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` poderia ser 4, 5 ou 6! O único resultado correto para `count` é 5, que só é gerado se o produtor e o consumidor forem executados em seqüência.

Podemos mostrar como o valor resultante 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

$registrador_1$ = count;
$registrador_1$ = $registrador_1$ + 1;
count = $registrador_1$;

onde $registrador_1$ é um registrador da CPU local. Da mesma forma, a instrução --count é implementada da seguinte maneira:

$registrador_2$ = count;
$registrador_2$ = $registrador_2$ − 1;
count = $registrador_2$;

onde novamente $registrador_2$ é um registrador da CPU local. Lembre-se de que, embora $registrador_1$ e $registrador_2$ possam ser o mesmo registrador físico (um acumulador, digamos), o seu conteúdo será salvo e restaurado pelo tratador de interrupção (Seção 2.1). Portanto, cada thread vê apenas seus próprios valores de registrador, e não os de outras threads.

A execução concorrente das instruções ++count e --count é equivalente a uma execução seqüencial onde 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

| | | | |
|---|---|---|---|
| $S_0$: *produtor* | executa | $registrador_1$ = count | {$registrador_1$ = 5} |
| $S_1$: *produtor* | executa | $registrador_1$ = $registrador_1$ + 1 | {$registrador_1$ = 6} |
| $S_2$: *consumidor* | executa | $registrador_2$ = count | {$registrador_2$ = 5} |
| $S_3$: *consumidor* | executa | $registrador_2$ = $registrador_2$ − 1 | {$registrador_2$ = 4} |
| $S_4$: *produtor* | executa | count = $registrador_1$ | {count = 6} |
| $S_5$: *consumidor* | executa | count = $registrador_2$ | {count = 4} |

observe que chegamos ao estado incorreto "*count* = 4", registrando que existem quatro buffers cheios, quando, na verdade, existem cinco buffers cheios. Se invertêssemos a ordem das instruções em $S_4$ e $S_5$, chegaríamos ao estado incorreto "*count* = 6".

Chegaríamos a esse estado incorreto porque permitimos que as duas threads manipulassem a variável count de forma concorrente. Essa situação, em que várias threads acessam e manipulam os mesmos dados concorrentemente e em que o resultado da execução depende da ordem específica em que ocorre o acesso é chamada de **condição de corrida (race condition)**. Essas situações ocorrem com freqüência nos sistemas operacionais, enquanto diferentes partes do sistema manipulam os recursos, e não queremos que as mudanças interfiram umas com as outras. A interferência inesperada pode causar adulteração de dados e falhas no sistema. Para nos protegermos contra uma condição de corrida, precisamos garantir que somente uma thread de cada vez poderá estar manipulando a variável count. Para isso, é preciso haver alguma forma de sincronismo das threads. Uma grande parte deste capítulo trata do sincronismo e da coordenação das threads.

## 7.2 O problema da seção crítica

Como um primeiro método de controle de acesso a um recurso compartilhado, declaramos uma seção do código como *crítica*; depois, regulamos o acesso a essa seção. Considere um sistema consistindo em $n$ threads $\{T_0, T_1, ..., T_{n-1}\}$. Cada thread possui um segmento de código, chamado **seção crítica (critical section)**, em que a thread pode alterar variáveis comuns, atualizando uma tabela, gravando um arquivo e assim por diante. O recurso importante do sistema é que, quando uma thread está executando em sua seção crítica, nenhuma outra thread pode ter permissão para executar em sua seção crítica. Assim, a execução de seções críticas pelas threads é *mutuamente exclusiva* no tempo. O desafio da seção crítica é criar um protocolo que as threads possam utilizar para a cooperação.

Uma solução para o problema da seção crítica precisa satisfazer os três requisitos a seguir:

1. *Exclusão mútua:* Se a thread $T_i$ está executando em sua seção crítica, então nenhuma outra poderá executar em suas seções críticas.
2. *Progresso:* Se nenhuma thread estiver executando em sua seção crítica e algumas threads quiserem entrar em suas seções críticas, então somente as threads que não estão executando em suas seções não críticas poderão participar na 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 outras threads têm permissão para entrar em suas seções críticas após uma thread ter feito uma requisição para entrar em sua se-

ção crítica e antes de essa requisição ser atendida. Esse limite impede *starvation* de qualquer thread isolada.

Consideramos que cada thread esteja sendo executada em uma velocidade diferente de zero. No entanto, não podemos fazer qualquer suposição a respeito da velocidade *relativa* das $n$ threads. Na Seção 7.3, examinaremos o problema da seção crítica e desenvolveremos uma solução que satisfaz esses três requisitos. As soluções não contam com quaisquer suposições a respeito das instruções de hardware ou do número de processadores que o hardware admite. Contudo, consideramos que as instruções básicas em linguagem de máquina (as instruções primitivas, como `load`, `store` e `test`) são executadas atomicamente. Ou seja, se duas dessas instruções forem executadas simultaneamente, o resultado será equivalente à sua execução seqüencial em alguma ordem desconhecida. Assim, se um `load` e um `store` são executados simultaneamente, o `load` receberá o valor antigo ou o valor novo, mas não alguma combinação dos dois.

## 7.3 Soluções com duas tarefas

Nesta seção, vamos considerar três implementações Java diferentes para coordenar as ações de duas threads diferentes. As threads são numeradas com $T_0$ e $T_1$. Por conveniência, ao representar $T_i$, usamos $T_j$ para indicar a outra thread, ou seja, $j = 1 - i$. Antes de examinar os diferentes algoritmos, apresentamos os arquivos de classe Java necessários.

Os três algoritmos implementarão a interface MutualExclusion mostrada na Figura 7.1, com cada algoritmo implementando os métodos `enteringCriticalSection( )` e `leavingCriticalSection( )`.

Implementamos cada thread usando a classe `Worker` mostrada na Figura 7.2. Antes de chamar sua seção crítica, cada thread chamará o método `enteringCriticalSection( )`, passando seu identificador de thread (que será 0 ou 1). Uma thread não retornará de `enteringCriticalSection( )` até ser capaz de entrar em sua seção crítica. Ao terminar sua seção crítica, uma thread chamará o método `leavingCriticalSection( )`.

As chamadas aos métodos estáticos `criticalSection( )` e `nonCriticalSection( )` representam onde cada thread realiza suas seções críticas e não críticas. Esses métodos fazem parte da classe `MutualExclusionUtilities` e simulam seções críticas e não críticas, dormindo por um período de tempo aleatório. (A classe `MutualExclusionUtilities` está disponível on-line.)

Usamos a classe `AlgorithmFactory` (Figura 7.3) para criar duas threads e testar cada algoritmo.

### 7.3.1 Algoritmo 1

Nossa primeira técnica é permitir que as threads compartilhem uma variável inteira comum, `turn`, inicializada com 0 ou 1. Se `turn == i`, então a thread $T_i$ tem permissão para executar sua seção crítica. Uma solução Java completa aparece na Figura 7.4.

Essa solução garante que somente uma thread de cada vez poderá estar em sua seção crítica. Entretanto, ela não satisfaz o requisito de progresso, pois exige alternância estrita das threads na execução de suas seções críticas. Por exemplo, se `turn == 0` e a thread $T_1$ estiver pronta para entrar em sua seção crítica, então $T_1$ não poderá fazer isso, embora $T_0$ possa estar em sua seção não crítica.

O algoritmo 1 utiliza o método `yield( )`, introduzido na Seção 6.8. A chamada do método `yield( )`

```
public interface MutualExclusion
{
   public static final int TURN_0 = 0;
   public static final int TURN_1 = 1;

   public abstract void enteringCriticalSection(int turn);
   public abstract void leavingCriticalSection(int turn);
}
```

FIGURA 7.1 *Interface MutualExclusion.*

```
public class Worker implements Runnable
{
   private String name;
   private int id;
   private MutualExclusion mutex;

   public Worker(String name, int id, MutualExclusion mutex) {
      this.name = name;
      this.id = id;
      this.mutex = mutex;
   }

   public void run( ) {
      while (true) {
         mutex.enteringCriticalSection(id);
         MutualExclusionUtilities.criticalSection(name);
         mutex.leavingCriticalSection(id);
         MutualExclusionUtilities.nonCriticalSection(name);
      }
   }
}
```

FIGURA 7.2 *A thread Worker.*

```
public class AlgorithmFactory
{
   public static void main(String args[ ]) {
      MutualExclusion alg = new Algorithm_1( );

      Thread first = new Thread(
         new Worker("Trabalhador 0", 0, alg));
      Thread second = new Thread(
         new Worker("Trabalhador 1", 1, alg));

      first.start( );
      second.start( );
   }
}
```

FIGURA 7.3 *A classe AlgorithmFactory.*

```
public class Algorithm_1 implements MutualExclusion
{
  private volatile int turn;

  public Algorithm_1( ) {
     turn = TURN_0;
  }

  public void enteringCriticalSection(int t) {
     while (turn != t)
        Thread.yield( );
  }

  public void leavingCriticalSection(int t) {
     turn = 1 - t;
  }
}
```

FIGURA 7.4 *Algoritmo 1.*

mantém a thread no estado Runnable, mas também permite que a JVM selecione outra thread Runnable para executar.

Essa solução também introduz uma nova palavra-chave em Java: `volatile`. A Especificação da Linguagem Java permite que um compilador realize certas otimizações, como o caching do valor de uma variável em um registrador da máquina, em vez de atualizar esse valor continuamente a partir da memória principal. Essas otimizações ocorrem quando o compilador reconhece que o valor da variável permanecerá inalterado, como na instrução

```
while (turn != t)
   Thread.yield( );
```

Contudo, se outra thread puder mudar o valor de `turn` – como acontece no algoritmo 1 –, então é desejável que o valor de `turn` seja atualizado da memória principal durante cada iteração. A declaração de uma variável como `volatile` impede que o compilador faça tais otimizações.

### 7.3.2 Algoritmo 2

O problema com o algoritmo 1 é que ele não retém informações suficientes sobre o estado de cada thread; ele se lembra apenas de qual thread tem permissão para entrar em sua seção crítica. Para remediar esse problema, podemos substituir a variável `turn` pelo seguinte:

```
boolean flag0;
boolean flag1;
```

Cada elemento é inicializado como `false`. Se um elemento for `true`, esse valor indica que a thread associada está pronta para entrar em sua seção crítica. A solução Java completa aparece na Figura 7.5. (Observe que um array de dois elementos do tipo `boolean` seria mais apropriado do que duas variáveis `boolean` separadas; porém, os dados precisam ser declarados como `volatile`, e a palavra-chave `volatile` não se estende para arrays.)

Nesse algoritmo, a thread $T_0$ primeiro define `flag0` como `true`, sinalizando que está pronta para entrar em sua seção crítica. Depois, $T_0$ verifica se a thread $T_1$ não está pronta também para entrar em sua seção crítica. Se $T_1$ estivesse pronta, então $T_0$ esperaria até $T_1$ indicar que não precisa mais estar na seção crítica (ou seja, até que `flag1` fosse `false`). Nesse ponto, $T_0$ entraria em sua seção crítica. Ao sair da seção crítica, $T_0$ definiria `flag0` como `false`, permitindo que outra thread (se houver uma aguardando) entre em sua seção crítica.

```
public class Algorithm_2 implements MutualExclusion
{
   private volatile boolean flag0;
   private volatile boolean flag1;

   public Algorithm_2( ) {
      flag0 = false;
      flag1 = false;
   }

   public void enteringCriticalSection(int t) {
      if (t == 0) {
         flag0 = true;
         while(flag1 == true)
            Thread.yield( );
      }
      else {
         flag1 = true;
         while (flag0 == true)
            Thread.yield( );
      }
   }

   public void leavingCriticalSection(int t) {
      if (t == 0)
         flag0 = false;
      else
         flag1 = false;
   }
}
```

FIGURA 7.5 *Algoritmo 2.*

Nessa solução, o requisito de exclusão mútua é satisfeito. Infelizmente, o requisito de progresso ainda não é atendido. Para ilustrar esse problema, considere a seguinte seqüência de execução: suponha que a thread $T_0$ defina `flag0` como `true`, indicando que deseja entrar em sua seção crítica. Antes de poder começar a executar o loop `while`, ocorre uma troca de contexto, e a thread $T_1$ define `flag1` como `true`. Cada thread entrará em um loop sem fim em sua instrução `while`, pois o valor do flag para a outra thread é `true`.

### 7.3.3 Algoritmo 3

Combinando as principais idéias do algoritmo 1 e do algoritmo 2, obtemos uma solução correta para o problema da seção crítica – que atende a todos os três requisitos. As threads compartilham três variáveis:

```
boolean flag0;
boolean flag1;
int turn;
```

Inicialmente, `flag0` e `flag1` são definidas como `false`, e o valor de `turn` não importa (ele é 0 ou 1). A Figura 7.6 apresenta a solução Java completa.

Para entrar em sua seção crítica, a thread $T_0$ primeiro define `flag0` como `true` e depois afirma que é a vez do outra thread entrar, se for apropriado (`turn == other`). Se as duas threads tentarem entrar ao mesmo tempo, `turn` será definido como 0 e 1 aproximadamente ao mesmo tempo. Apenas uma dessas atribuições persistirá; a outra ocorrerá, mas será imediatamente modificada. O valor final de `turn` decide qual das duas threads terá permissão para entrar em sua seção crítica primeiro.

```
public class Algorithm_3 implements MutualExclusion
{
   private volatile int turn;
   private volatile boolean flag0;
   private volatile boolean flag1;

   public Algorithm_3( ) {
      flag0 = false;
      flag1 = false;
      turn = TURN_0;
   }

   public void enteringCriticalSection(int t) {
      int other = 1 -- t;
      turn = other;

      if (t == 0) {
         flag0 = true;
         while ( (flag1 == true) && (turn == other) )
            Thread.yield( );
      }
      else {
         flag1 = true;
         while ( (flag0 == true) && (turn == other) )
            Thread.yield( );
      }
   }

   public void leavingCriticalSection(int t) {
      if(t == 0)
         flag0 = false;
      else
         flag1 = false;
   }
}
```

FIGURA 7.6 *Algoritmo 3.*

## 7.4 Hardware de sincronismo

Como acontece com outros aspectos do software, os recursos do hardware podem tornar a tarefa de programação mais fácil e melhorar a eficiência do sistema. Nesta seção, apresentamos instruções de hardware simples, disponíveis em muitos sistemas, e mostramos como podem ser usadas efetivamente na solução do problema de seção crítica.

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 seqüência 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.

Infelizmente, essa solução não é viável em um ambiente multiprocessado. Desabilitar interrupções em multiprocessadores pode ser algo demorado, pois os comandos para habilitar e desabilitar precisam ser passados a todos os processadores. Essa troca de mensagem atrasa a entrada em cada seção crítica e aumenta o custo da saída de cada seção crítica. Como resultado, a eficiência do sistema diminui bastante. Os sistemas que empregam esse método de sincronismo perdem a escalabilidade à medida que as CPUs são acrescentadas, pois o custo de comunicação aumenta com a quantidade de CPUs.

Todas as máquinas modernas, portanto, oferecem 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 so-

lucionar o problema da seção crítica de maneira relativamente simples. Em vez de discutirmos uma instrução específica para uma máquina específica, usaremos Java para separar os conceitos principais por trás desses tipos de instruções. A classe HardwareData, mostrada na Figura 7.7, ilustrará as instruções.

O método getAndSet( ) implementando a instrução *Get-and-Set* aparece na Figura 7.7. 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 seqüência, em alguma ordem qualquer.

Se a máquina admitir a instrução *Get-and-Set*, então 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 7.8 ilustra a estrutura da thread $T_i$.

```
public class HardwareData
{
   private boolean data;

   public HardwareData(boolean data) {
      this.data = data;
   }

   public boolean get( ) {
      return data;
   }

   public void set(boolean data) {
      this.data = data;
      }
   }

   public boolean getAndSet(boolean data) {
      boolean oldValue = this.get( );
      this.set(data);

      return oldValue;
   }

   public void swap(HardwareData other) {
      boolean temp = this.get( );

      this.set(other.get( ));
      other.set(temp);
   }
}
```

FIGURA 7.7 *Estrutura de dados para soluções de hardware.*

```
// lock é compartilhado por todas as threads
HardwareData lock = new HardwareData(false);

while (true) {
   while (lock.getAndSet(true))
      Thread.yield( );

   criticalSection( );
   lock.set(false);
   nonCriticalSection( );
}
```

FIGURA 7.8 *Thread usando o lock Get-and-Set.*

A instrução *Swap*, definida no método swap( ) na Figura 7.7, 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 da thread $T_i$ aparece na Figura 7.9.

```
// 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);

   criticalSection( );
   lock.set(false);
   nonCriticalSection( );
}
```

FIGURA 7.9 *Thread usando a instrução Swap.*

## 7.5 Semáforos

As soluções baseadas em hardware para o problema de seção crítica, apresentadas na Seção 7.4, são complicadas para programadores de aplicação. Para contornar essa dificuldade, podemos usar uma ferramenta de sincronismo chamada **semáforo**. Um semáforo *S* é 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"). As definições de `acquire( )` e `release( )` são as seguintes:

```
acquire(S) {
   while S <=0
      ; // nenhuma operação
   S--;
}

release(S) {
   S++;
}
```

As modificações feitas no valor inteiro do semáforo nas operações `acquire( )` e `release( )` precisam ser executadas de modo atômico. Ou seja, quando uma thread modificar o valor do semáforo, nenhuma outra thread poderá modificar esse mesmo valor de semáforo concorrentemente. Além disso, no caso de `acquire(S)`, o teste do valor inteiro de `S` (`S <= 0`) e sua possível modificação (`S--`) também devem ser executados sem interrupção. Na Seção 7.5.2, veremos como essas operações podem ser implementadas; primeiro, vejamos como os semáforos podem ser usados.

### 7.5.1 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**, pois são bloqueios (locks) que oferecem exclusão mútua (*mutual ex*clusion).

A estratégia geral para o uso de um semáforo binário para controlar o acesso a uma seção crítica é a seguinte (supondo que o semáforo seja inicializado em 1):

```
Semaphore S;

acquire(S);
criticalSection( );
release(S);
```

Assim, podemos usar o semáforo para controlar o acesso à seção crítica para um processo ou thread. Uma solução generalizada para várias threads aparece no programa Java da Figura 7.10. 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;
   private String name;

   public Worker(Semaphore sem, String name) {
      this.sem = sem;
      this.name = name;
   }

   public void run( ) {
      while (true) {
         sem.acquire( );

MutualExclusionUtilities.criticalSection(name);
         sem.release( );

MutualExclusionUtilities.nonCriticalSection(name);
      }
   }
}

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, "Trabalhador " + (new
Integer(i)).toString( ) ));

      for (int i = 0; i < 5; i++)
         bees[i].start( );
   }
}
```

FIGURA 7.10 *Sincronismo usando semáforos.*

Semáforos contadores podem ser usados para 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 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.

### 7.5.2 Implementação

A principal desvantagem das soluções de exclusão mútua da Seção 7.3, e do tipo de semáforo que acabamos de descrever, é que todas elas exigem 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 mutiprocessados, 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 bloqueio 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 um valor inteiro e 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(S){
   value--;
   if (value < 0) {
      acrescenta esse processo à lista
      block;
   }
}

release(S){
   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 essa implementação pode ter valores de semáforo negativos, embora, sob a definição clássica de semáforos com espera ocupada, o valor do semáforo nunca seja negativo. Se for negativo, sua magnitude é a quantidade de processos esperando por esse semáforo. Esse fato é o resultado da tro-

ca 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, seria 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 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 monoprocessado, 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 mutiprocessado, desabilitar interrupções não funciona. As instruções de diferentes processos (executando em diferentes processadores) podem ser intercaladas de alguma maneira arbitrária. Se o hardware não oferecer quaisquer instruções especiais, podemos empregar qualquer uma das soluções de software corretas para o problema de seção crítica (Seção 7.2), em que as seções críticas consistem nas operações `acquire( )` e `release( )`.

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 de maneira, não deverão ter mais do que 10 instruções). Assim, a seção crítica quase nunca é ocupada; a espera ocupada raramente ocorre, e somente por um curto período. Existe uma situação 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 7.8.5, veremos como os semáforos podem ser implementados em Java.

### 7.5.3 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**.

Como uma ilustração, consideramos um sistema consistindo em dois processos, $P_0$ e $P_1$, cada um acessando dois semáforos, $S$ e $Q$, definidos como valor 1:

| $P_0$ | $P_1$ |
|---|---|
| `acquire(S);` | `acquire(Q);` |
| `acquire(Q);` | `acquire(S);` |
| . | . |
| . | . |
| . | . |
| `release(S);` | `release(Q);` |
| `release(Q);` | `release(S);` |

Suponha que $P_0$ execute `acquire(S)` e depois $P_1$ execute `acquire(Q)`. Quando $P_0$ executar `acquire(Q)`, ele terá de esperar até $P_1$ executar `release(Q)`. De modo semelhante, quando $P_1$ executar acquire(S), ele terá de esperar até $P_0$ executar `release(S)`. 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

```
public class BoundedBuffer implements Buffer
{
   private static final int BUFFER_SIZE = 5;
   private Object[ ] buffer;
   private int in, out;
   private Semaphore mutex;
   private Semaphore empty;
   private Semaphore full;

   public BoundedBuffer( ) {
      // buffer está inicialmente vazio
      in = 0;
      out = 0;
      buffer = new Object[BUFFER_SIZE];

      mutex = new Semaphore(1);
      empty = new Semaphore(BUFFER_SIZE);
      full = new Semaphore(0);
   }

   public void insert(Object item) {
   // Figura 7.12
   }

   public Object remove( ) {
   // Figura 7.13
   }
}
```

FIGURA 7.11 *Solução para o problema de bounded buffer (produtor-consumidor) utilizando semáforos.*

```
public void insert(Object item) {
   empty.acquire( );
   mutex.acquire( );

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

   mutex.release( );
   full.release( );
}
```

FIGURA 7.12 *O método insert( ).*

aquisição e liberação de recursos; porém, outros tipos de eventos podem resultar em deadlocks, como mostraremos no Capítulo 8. Nesse capítulo, descrevemos diversos mecanismos para lidar com o problema do deadlock.

Outro problema relacionado a deadlocks é o **bloqueio indefinido** ou **starvation** – uma situação em

```
public Object remove( ) {
   full.acquire( );
   mutex.acquire( );

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

   mutex.release( );
   empty.release( );

   return item;
}
```

FIGURA 7.13 *O método remove( ).*

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).

## 7.6 Problemas clássicos de sincronismo

Nesta seção, apresentamos uma série de problemas de sincronismo diferentes, importantes principalmente porque são exemplos para uma grande classe de problemas de controle de concorrência. Esses problemas são usados para testar quase todo esquema de sincronismo recém-proposto. Os semáforos são usados para o sincronismo em nossas soluções.

### 7.6.1 O problema do bounded buffer

O problema do bounded buffer foi introduzido na Seção 7.1; ele normalmente é usado para ilustrar o poder das primitivas de sincronismo. Uma solução foi mostrada na Figura 7.11. Um produtor coloca um item no buffer chamando o método `insert( )`; os consumidores removem itens invocando `remove( )`.

O semáforo `mutex` oferece 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 7.14. O produtor alterna entre dormir por um tempo (a classe `SleepUtilities` está disponível on-line) produzindo uma mensagem, e tentar colocar essa mensagem no buffer por meio do método `insert( )`.

A thread consumidor aparece na Figura 7.15. O consumidor alterna entre dormir e consumir um item usando o método `remove( )`.

A classe `Factory` (Figura 7.16) cria as threads produtor e o consumidor, passando a cada uma delas uma referência ao objeto `BoundedBuffer`.

```
import java.util.Date;

public class Producer implements Runnable
{
   private  Buffer buffer;

   public Producer(Buffer 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 7.14 *Thread produtor.*

```
import java.util.Date;

public class Consumer implements Runnable
{
   private  Buffer buffer;

   public Consumer(Buffer 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 7.15 *Thread consumidor.*

### 7.6.2 O problema dos leitores-escritores

Um banco de dados deve 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 algu-

```
public class Factory
{
   public static void main(String args[ ]) {
      Buffer buffer = new BoundedBuffer( );

      // agora 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 7.16 *A classe Factory.*

ma 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 de 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 simplesmente 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.

Observamos que uma solução para qualquer um desses problemas pode resultar em starvation. No primeiro caso, os escritores podem se estagnar; no segundo caso, os leitores podem se 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 7.17. 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 7.18) funciona de modo semelhante.

Os métodos chamados em cada thread leitor e escritor são definidos na interface `RWLock` da Figura 7.19. A classe `Database` da Figura 7.20 implementa essa interface. O `readerCount` registra 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

```
public class Reader implements Runnable
{
   private RWLock db;

   public Reader(RWLock 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 7.17 *Um leitor.*

```
public class Writer implements Runnable
{
   private RWLock db;

   public Writer(RWLock 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 7.18 *Um escritor.*

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.

Os locks de leitura-escrita são mais úteis nas seguintes situações:

- Em aplicações fáceis de identificar quais threads só lêem 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

```
public interface RWLock
{
   public abstract void  acquireReadLock( );
   public abstract void  acquireWriteLock( );
   public abstract void  releaseReadLock( );
   public abstract void  releaseWriteLock( );
}
```

FIGURA 7.19 *A interface para o problema dos leitores-escritores.*

```
public class Database implements RWLock
{
   private int readerCount;
   private Semaphore mutex;
   private Semaphore db;

   public Database( ) {
      readerCount = 0;
      mutex = new Semaphore(1);
      db = new Semaphore(1);
   }

   public int acquireReadLock( ) {
      // Figura 7.21
   }

   public int releaseReadLock( ) {
      // Figura 7.21
   }

   public void acquireWriteLock( ) {
      // Figura 7.22
   }

   public void releaseWriteLock( ) {
      // Figura 7.22
   }
}
```

FIGURA 7.20 *O banco de dados para o problema dos leitores-escritores.*

```
public void acquireReadLock( ) {
   mutex.acquire( );
   ++readerCount;

   // se sou o primeiro leitor, diz a todos os outros
   // que o banco de dados está sendo lido
   if (readerCount == 1)
      db.acquire( );

   mutex.release( );
}

public void releaseReadLock( ) {
   mutex.acquire( );
   --readerCount;

   // se eu sou o último leitor, diz a todos os outros
   // que o banco de dados não está mais sendo lido
   if (readerCount == 0)
      db.release( );

   mutex.release( );
}
```

FIGURA 7.21 *Métodos chamados pelos leitores.*

```
public void acquireWriteLock( ) {
   db.acquire( );
}

public void releaseWriteLock( ) {
   db.release( );
}
```

FIGURA 7.22 *Métodos chamados pelos escritores.*

estabelecidos do que os semáforos ou locks de exclusão mútua, e o custo adicional para configurar um lock de leitura-escrita é compensado pela maior concorrência da permissão de leitores múltiplos.

### 7.6.3 O problema dos filósofos na mesa de jantar

Imagine cinco filósofos que gastam suas vidas pensando e comendo. Os filósofos compartilham uma mesa redonda comum, 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 cinco garfos (Figura 7.23). Quando um filósofo pensa, ele não interage com os colegas. De vez em quando, um filósofo tem fome e tenta pegar os dois garfos próximos a ele (os garfos que estão entre ele e seus vizinhos da esquerda e da direita). Um filósofo só pode pegar um garfo de cada vez. Obviamente, ele não pode pegar um garfo que já esteja na mão de um vizinho. Quando um filósofo com fome tem dois garfos ao mesmo tempo, ele come sem largar os grafos. Quando termina de comer, ele coloca os dois garfos na mesa e recomeça a pensar.

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 starvation.

Uma solução simples é representar cada garfo por um semáforo. Um filósofo tenta apanhar o garfo executando uma operação `acquire( )` sobre esse semáforo; ele solta um garfo executando a operação

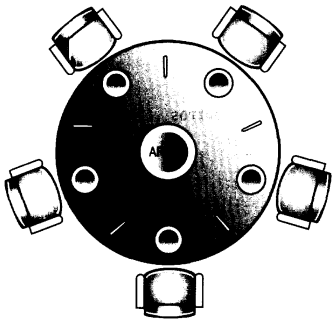


FIGURA 7.23 *A situação dos filósofos na mesa de jantar.*

`release( )` sobre os semáforos apropriados. Assim, os dados compartilhados são

```
Semaphore chopStick[ ] = new Semaphore [5];
```

onde todos os elementos de chopstick são inicializados com 1. A estrutura do filósofo *i* aparece na Figura 7.24.

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 garfo à sua esquerda. Todos os elementos de `chopstick` agora serão iguais a 0. Quando cada filósofo tentar pegar o garfo da direita, ele esperará indefinidamente.

```
while (true) {
   // apanha garfo da esquerda
   chopStick[i].acquire( );
   // apanha garfo da direita
   chopStick[(i + 1) % 5].acquire( );

   eating( );

   // retorna pauzinho da esquerda
   chopStick[i].release( );
   // retorna garfo da esquerda
   chopStick[(i + 1) % 5].release( );

   thinking( );
}
```

FIGURA 7.24 *A estrutura do filósofo i.*

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 a mesa.
- Só permitir que um filósofo apanhe os garfos se os dois garfos estiverem disponíveis (observe que ele precisa apanhá-los em uma seção crítica).
- Usar uma solução assimétrica; por exemplo, um filósofo ímpar apanha primeiro o garfo da esquerda, e depois o da direita, enquanto um filósofo par apanha o garfo da direita e depois o da esquerda.

Na Seção 7.7, 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 na mesa de jantar precisa de proteção contra a possibilidade de um dos filósofos ainda morrer de fome. Uma solução sem deadlock não necessariamente elimina a possibilidade de starvation.

## 7.7 Monitores

Embora os semáforos forneçam um mecanismo conveniente e eficaz para o sincronismo de processos, seu uso incorreto pode resultar em erros de temporização difíceis de detectar, pois esses erros só acontecem se ocorrerem determinadas seqüências de execução, e essas seqüências nem sempre ocorrem.

Vimos um exemplo desses erros com o uso de contadores em nossa solução para o problema de produtor-consumidor (Seção 7.1). 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 com o uso de 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 seqüência 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( );
criticalSection( );
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( );
criticalSection( );
mutex.acquire( );
```

  Nesse caso, ocorrerá um deadlock.

- Suponha agora que um processo omita 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 7.6.

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**.

Lembre-se de que 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. O pseudocódigo tipo Java, descrevendo a sintaxe de um monitor, é:

```
monitor nome-monitor
{
   // declarações de variáveis

   entrada pública p1(...){
   ...
   }
   entrada pública p2(...){
   }
}
```

A implementação interna de um tipo monitor não pode ser acessada pelas diversas threads. Um procedimento definido dentro de um monitor só pode acessar as variáveis declaradas localmente dentro do monitor, junto com quaisquer parâmetros formais passados ao procedimento. De modo semelhante, as variáveis locais só podem ser acessadas pelos procedimentos locais.

A construção do monitor proíbe o acesso simultâneo a procedimentos definidos dentro do monitor. Portanto, somente uma thread (ou processo) pode estar ativa dentro do monitor em determinado momento. Como conseqüência, o programador não precisa codificar esse sincronismo explicitamente; ele está embutido no tipo de monitor.

As variáveis de condição de tipo desempenham uma função especial em monitores em virtude das operações especiais `wait` e `signal`. Um programador que precisa escrever seu próprio esquema de sincronismo personalizado pode definir uma ou mais variáveis do tipo `condition`

```
condition x,y;
```

A operação

```
x.wait;
```

significa que a thread que chama essa operação é suspensa até outra thread chamar

```
x.signal;
```

A operação `signal` reassume exatamente uma thread. Se nenhuma thread estiver suspensa, então a operação `signal` não terá efeito; ou seja, o estado de x é como se a operação nunca tivesse sido executada (Figura 7.25). Compare esse esquema com a operação `release( )` com semáforos, que sempre afeta o estado do semáforo.

Agora suponha que, quando a operação `x.signal` é chamada por uma thread *P*, existe uma thread suspensa *Q* associada à condição *x*. Logicamente, se a thread suspensa *Q* tiver permissão para reassumir sua execução, a thread sinalizador *P* terá de esperar. Caso contrário, *P* e *Q* estariam ativas simultaneamente dentro do monitor. No entanto, observe que as duas threads, 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 e wait* 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.

Ilustramos esses conceitos apresentando uma solução sem deadlock para o problema dos filósofos na mesa de jantar. Essa solução impõe uma restrição de que um filósofo só pode apanhar os garfos 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:

```
int[ ] state = new int[5];
static final int THINKING = 0;
static final int HUNGRY = 1;
static final int EATING = 2;
```

O filósofo *i* só pode definir a variável `state[i] = EATING` se seus dois vizinhos não estiverem comendo; ou seja, as condições `(state [(i + 4) % 5] != EATING)` e `(state[(i + 1) % 5] != EATING)` são verdadeiras.

Também precisamos declarar

```
condition[ ] self = new condition[5];
```

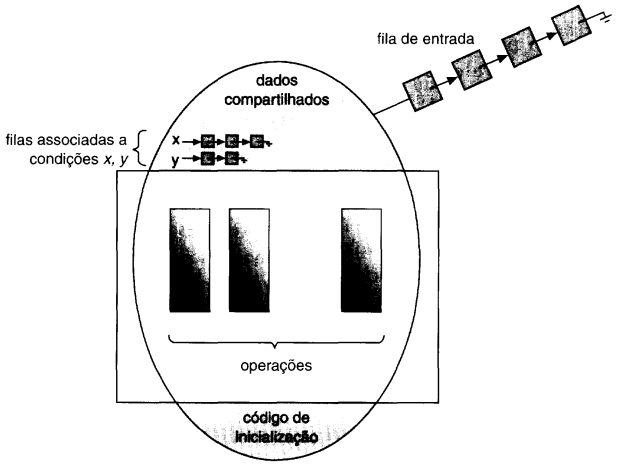



FIGURA 7.25 *Monitor com variáveis de condição.*

onde o filósofo *i* pode se atrasar quando estiver com fome, mas não consegue obter os garfos de que precisa.

Agora, estamos em posição de descrever nossa solução para o problema dos filósofos na mesa de jantar. A distribuição dos garfos é controlada pelo monitor dp, que é uma instância do tipo de monitor DiningPhilosophers, cuja definição usando um pseudocódigo tipo Java aparece na Figura 7.26. Cada filósofo, antes de começar a comer, precisa chamar a operação pickUp( ). 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 putDown( ) e começa a pensar. Assim, o filósofo *i* precisa chamar as operações pickUp( ) e putDown( ) na seguinte seqüência:

```
dp.pickUp(i);
eat( );
dp.putDown(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.

```
monitor DiningPhilosophers {
   int[ ] state = new int[5];
   static final int THINKING = 0;
   static final int HUNGRY = 1;
   static final int EATING = 2;
   condition[ ] self = new condition[5];

   public diningPhilosophers {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
   }

   public entry pickUp(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait;
   }

   public entry putDown(int i) {
      state[i] = THINKING;
      // testa vizinhos da esquerda e da direita
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   private test(int i) {
      if ( (state[(i + 4) % 5] != EATING) &&
         (state[i] == HUNGRY) &&
         (state[(i + 1) % 5] != EATING) ) {
            state[i] = EATING;
            self[i].signal;
      }
   }
}
```

FIGURA 7.26 *Uma solução de monitor para o problema dos filósofos na mesa de jantar.*

## 7.8 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 impondo 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)**.

### 7.8.1 Bounded buffer

A solução de memória compartilhada para o problema de bounded buffer, descrita no Capítulo 4, 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 7.1, a condição corrida sobre a variável `count` é compartilhada pelo produtor e consumidor. Esta seção resolve esses e outros problemas enquanto desenvolve uma solução por meio dos mecanismos de sincronismo Java.

#### 7.8.1.1 Espera ocupada

A espera ocupada foi introduzida na Seção 7.5.2, 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 com que uma thread chame o método `Thread.yield( )`. Lembre-se de que, como vimos na Seção 7.3.1, 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 poderia 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 as 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 uma condição desejada.

#### 7.8.1.2 Condição de corrida

Na Seção 7.1, vimos um exemplo das conseqüências de uma condição de corrida sobre a variável compartilhada `count`. A Figura 7.27 ilustra como o tratamento do acesso concorrente em Java aos dados compartilhados impede as condições de corrida.

Essa situação introduz uma nova palavra-chave: synchronized. Cada objeto em Java possui, associado a ele, um único lock. (Essa proteção é muito parecida com aquela oferecida por um monitor. As diferenças são descritas na Seção 7.8.7.) 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 estando sincronizado, para chamar o método é preciso obter o lock do objeto. Se o lock já estiver em posse de outra thread, a thread que chama o método sincronizado 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 sincronizado for chamado, a thread que chama se torna a proprietária do lock do objeto e pode entrar no método. O lock é li-

```
public synchronized void insert(Object item) {
   while (count == BUFFER_SIZE) {
      Thread.yield( );

   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;
}

public synchronized Object remove( ) {
   Object item;

   while (count == 0) {
      Thread.yield( );

   --count;
   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;

   return item;
}
```

FIGURA 7.27 *Métodos sincronizados insert( ) e remove( ).*

berado 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 a partir 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 JVMs ordena as threads no conjunto de espera de acordo com uma política FIFO.) A Figura 7.28 ilustra como o conjunto de entrada opera.

Se o produtor chamar o método `insert( )`, como mostra a Figura 7.27, 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( )`.

### 7.8.1.3 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.

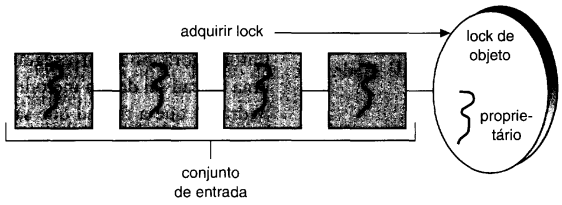



FIGURA 7.28 *Conjunto de entrada.*

Suponha que o buffer esteja cheio e o consumidor esteja dormindo. Se o produtor chamar o método `insert( )`, ele poderá continuar, pois o lock está 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 será 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.

### 7.8.1.4 Wait e notify

A Figura 7.29 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 (set wait)**, consistindo em um conjunto de threads. Esse conjunto de espera está inicialmente vazio. Quando uma thread entra em um método `synchronized`, ele tem a posse do lock do objeto. Todavia, essa thread pode determinar que ela é incapaz de continuar, porque uma certa 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. 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 7.29. 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

```
public synchronized void insert(Object item) {
   while (count == BUFFER_SIZE) {
      try {
         wait( );
      }
      catch (InterruptedException e) { }
   }

   ++count;
   buffer[in] = item;
   in = (in + 1) % BUFFER_SIZE;

   notify( );
}

public synchronized Object remove( ) {
   Object item;

   while (count == 0) {
      try {
         wait( );
      }
      catch (InterruptedException e) { }
   }

   --count;
   item = buffer[out];
   out = (out + 1) % BUFFER_SIZE;

   notify( );

   return item;
}
```

FIGURA 7.29 *Métodos `insert( )` e `remove( )` usando `wait( )` e `notify( )`.*

coloca o produto 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 7.30 ilustra os conjuntos de entrada e espera por um lock. (Observe que `wait( )` pode resultar em uma `InterruptedException`. Veremos isso na Seção 7.8.8.)

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 `synchronized insert( )` e `remove( )`, temos uma chamada

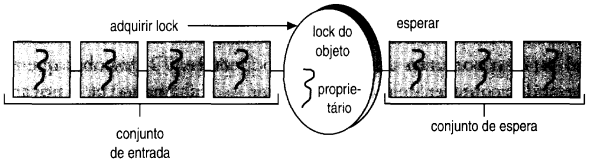



FIGURA 7.30 *Conjuntos de entrada e espera.*

para o método `notify( )`. A chamada para `notify( )`:

1. Apanha uma thread `T` qualquer na lista de threads no conjunto de espera
2. Move `T` do conjunto de espera para o conjunto de entrada
3. Define o estado de `T` de bloqueado para executável

`T` agora pode competir pelo lock com as outras threads. Quando `T` 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 7.29. Consideramos 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 o 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 a partir 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 7.31 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 7.6.1.

### 7.8.2 Notificações múltiplas

Conforme descrevemos na Seção 7.8.1.4, a chamada a `notify( )` seleciona arbitrariamente uma thre-

```
public class BoundedBuffer implements Buffer
{
   private static final int BUFFER_SIZE = 5;

   private int count, in, out;
   private Object[ ] buffer;

   public BoundedBuffer( ) {
      // buffer inicialmente está vazio
      count = 0;
      in = 0;
      out = 0;
      buffer = new Object[BUFFER_SIZE];
   }

   public synchronized void insert(Object item) {
      // Figura 7.29
   }

   public synchronized Object remove( ) {
      // Figura 7.29
   }
}
```

FIGURA 7.31 *Bounded buffer.*

ad 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 {*T1*, *T2*, *T3*, *T4*, *T5*} 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 7.32. Somente uma thread cujo número combina com o valor de `turn` pode prosseguir; todas as outras threads precisam esperar sua vez.

Considere o seguinte:

- `turn = 3`.
- *T1*, *T2* e *T4* estão no conjunto de espera do objeto.
- *T3* está atualmente no método `doWork( )`.

Quando a thread *T3* termina, ela define `turn` como 4 (indicando que é a vez de *T4*) e chama `notify( )`. A chamada a `notify( )` seleciona arbitrariamente uma thread no conjunto de espera. Se *T2* receber a notificação, ela retomará a execução a partir da chamada a `wait( )` e testará a condição no loop `while`. *T2* descobre que essa não é sua vez e, por isso, chama `wait( )` novamente. Por fim, *T3* e *T5* chamarão `doWork( )` e também chamarão o método `wait( )`, pois não é a vez nem de *T3* nem de *T5*. Agora, todas as cinco threads estão bloqueadas no conjunto de espera do objeto. Assim, temos outro deadlock para tratar.

```
// 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 ao próxima
   // thread esperando que é sua vez
   // de realizar algum trabalho.

   if (turn < 5)
      ++turn;
   else
      turn = 1;

   notify( );
}
```

FIGURA 7.32 *Método doWork( ).*

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, Java oferece 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 *T3* terminar e definir `turn` para 4, ela chamará `notifyAll( )`. Essa chamada tem o efeito de remover *T1*, *T2* e *T4* do conjunto de espera. As três threads, então, competem pelo lock do objeto mais uma vez. Por fim, *T1* e *T2* chamam `wait( )` e somente *T4* 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 onde 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( )`.

### 7.8.3 O problema de leitores-escritores

Agora, podemos oferecer 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 7.33, que implementa a interface `RWLock` mostrada na Figura 7.19. O `readerCount` registra o número de leitores; um valor > 0 indica que

o banco de dados está sendo lido. `dbWriting` é uma variável `boolean` 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.

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.

```
public class Database implements RWLock {
   private int readerCount;
   private boolean dbWriting;

   public Database( ) {
      readerCount = 0;
      dbWriting = false;
   }

   public synchronized void acquireReadLock( ) {
      // Figura 7.34
   }

   public synchronized void releaseReadLock( ) {
      // Figura 7.34
   }

   public synchronized void acquireWriteLock( ) {
      // Figura 7.35
   }

   public synchronized void releaseWriteLock( ) {
      // Figura 7.35
   }
}
```

FIGURA 7.33 *O banco de dados.*

```
public synchronized void acquireReadLock( ) {
   while (dbWriting == true) {
      try {
         wait( );
      }
      catch(InterruptedException e) { }
   }

   ++readerCount;
}

public synchronized void releaseReadLock( ) {
   --readerCount;

   // se sou o último leitor, diz aos escritores
   // que o banco de dados não está mais sendo lido
   if (readerCount == 0)
      notify( );
}
```

FIGURA 7.34 *Métodos chamados pelos leitores.*

```
public synchronized void acquireWriteLock( ) {
   while (readerCount > 0 || dbWriting == true) {
      try {
         wait( );
      }
      catch(InterruptedException e) { }
   }

   // quando não houver mais leitores ou escritores
   // indica que o banco de dados está sendo escrito
   dbWriting = true;
}

public synchronized void releaseWriteLock( ) {
   dbWriting = false;

   notifyAll( );
}
```

FIGURA 7.35 *Métodos chamados pelos escritores.*

### 7.8.4 Sincronismo em bloco

A quantidade de tempo entre o momento em que um lock é adquirido e quando ele é liberado é definida como o **escopo** do lock. 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`, Java também permite o sincronismo em bloco, conforme ilustrado na Figura 7.36. O acesso ao método `criticalSection( )` na Figura 7.36 exige a propriedade do lock do objeto `mutexLock`.

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 7.37.

### 7.8.5 Semáforos em Java

Java não oferece um semáforo, mas podemos construir um usando mecanismos de sincronismo padrão. A declaração dos métodos `acquire( )` e `release( )` como `synchronized` garante que cada operação será realizada atomicamente. A classe Semaphore mostrada na Figura 7.38 implementa um semáforo de contagem básico. Deixamos como exercício a modificação da classe `Semaphore` de modo que ela atue como um semáforo binário.

```
Object mutexLock = new Object( );
. . .
public void someMethod( ) {
   nonCriticalSection( );

   synchronized(mutexLock) {
      criticalSection( );
   }

   nonCriticalSection( );
}
```

FIGURA 7.36 *Sincronismo em bloco.*

### 7.8.6 Regras de sincronismo

A palavra-chave `synchronized` é uma construção simples, mas é importante conhecer algumas regras sobre seu comportamento.

```
Object mutexLock = new Object( );
. . .
synchronized(mutexLock) {
   try {
      mutexLock.wait( );
   }
   catch (InterruptedException ie) { }
}

synchronized(mutexLock) {
   mutexLock.notify( );
}
```

FIGURA 7.37 *Sincronismo em bloco usando `wait( )` e `notify( )`.*

```
public class Semaphore
{
   private int value;

   public Semaphore( ) {
      value = 0;
   }

   public Semaphore(int value) {
      this.value = value;
   }

   public synchronized void acquire( ) {
      while (value == 0) {
         try {
            wait( );
         }
         catch (InterruptedException e) { }
      }

      value --;
   }

   public synchronized void release( ) {
      ++value;

      notify( );
   }
}
```

FIGURA 7.38 *Implementação de semáforo em Java.*

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**.
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`, então ele pode ser invocado independente da propriedade do lock, mesmo enquanto outro método `synchronized` do mesmo objeto esteja sendo executado.
4. Se o conjunto de espera de um objeto estiver vazio, então uma chamada a `notify( )` ou `notifyAll( )` não terá efeito.
5. `wait( )`, `notify( )` e `notifyAll( )` só podem ser invocados a partir de métodos ou blocos sincronizados; caso contrário, será gerada uma exceção do tipo `IllegalMonitorStateException`.

### 7.8.7 Monitores em Java

Muitas linguagens de programação têm incorporado a idéia do monitor (discutido na Seção 7.7), incluindo Concurrent Pascal, Mesa, NeWs e Java. Muitas outras linguagens de programação modernas têm oferecido algum tipo de suporte de concorrência usando um mecanismo semelhante aos monitores. Discutimos o relacionamento dos monitores no sentido estrito com os monitores Java.

Já descrevemos como o sincronismo Java utiliza o lock de um objeto. De muitas maneiras, esse lock atua como um monitor. Assim, cada objeto Java possui um monitor associado. Uma thread pode adquirir o monitor de um objeto entrando em um método ou bloco sincronizado.

Com os monitores, as operações `wait` e `signal` podem ser aplicadas a variáveis de condição nomeadas, permitindo que uma thread espere por uma condição específica ou seja notificada quando uma condição específica tiver sido atendida. Entretanto, Java não oferece suporte a variáveis de condição nomeadas. Além do mais, cada monitor Java está associado a no máximo uma variável de condição não nomeada. As operações `wait( )`, `notify( )` e `notifyAll( )` só podem ser aplicadas a essa única variável de condição. Quando uma thread Java é despertada por meio de `notify( )` ou `notifyAll( )`, ela não recebe informações explicando por que foi despertada. Fica a cargo da thread reativada verificar por si só se a condição a qual estava esperando foi atendida.

Os monitores Java utilizam a técnica de sinalizar e continuar: quando uma thread é sinalizada com o método `notify( )`, ela só pode obter a posse do lock do monitor quando a thread notificando sair do método ou bloco `synchronized`.

### 7.8.8 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 5, 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. Essa estratégia leva em conta aplicações mais robustas, com multithreads, pois oferece 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 `Interrup-`

tedException se propague. Ou seja, em métodos em que `wait( )` é chamado, primeiro removemos os locks `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.

Como um exemplo, o método `acquire( )` na classe `Semaphore` mostrada na Figura 7.38 chama `wait( )`. A chamada a `wait( )` é colocada em um bloco `try-catch`; mas, se uma `InterruptedException` for apanhada, nós a ignoramos. O tratamento de `InterruptedException` de acordo com a estratégia esboçada anteriormente resulta na classe `Semaphore` mostrada na Figura 7.39. Observe que, como `acquire( )` agora é declarado como lançando uma `InterruptedException`, essa técnica de tratar `InterruptedException` agora exige a colocação de `acquire( )` em um bloco `try-catch`. Contudo, isso permite interromper uma thread bloqueada no método `acquire( )` de um semáforo.

```
public class Semaphore
{
   private int value;

   public Semaphore( ) {
      value = 0;
   }

   public Semaphore(int value) {
      this.value = value;
   }

   public synchronized void acquire( )
         throws InterruptedException {
      while (value == 0)
         wait( );

      value--;
   }

   public synchronized void release( ) {
      ++value;

      notify( );
   }
}
```

FIGURA 7.39 *Tratando de InterruptedException com semáforos Java.*

## 7.9 Exemplos de sincronismo

Em seguida, descrevemos os mecanismos de sincronismo fornecidos pelos sistemas operacionais Solaris, Windows XP e Linux, assim como a API Pthreads. Como ficará claro nesta seção, existem variações sutis e significativas nos métodos de sincronismo disponíveis nos diferentes sistemas.

### 7.9.1 Sincronismo no Solaris

Para controlar o acesso às seções críticas, o Solaris oferece 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 7.5 e 7.7. 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 "gira" 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 quando o lock não for liberado por um período relativamente rápido. Um lock mantido por uma thread dormindo provavelmente estará nessa categoria. Em um sistema monoprocessado, a thread que mantém o lock nunca estará em execução se o lock estiver sendo testado por outra thread, pois somente uma thread pode estar sendo executada em determinado momento. Portanto, em um sistema monoprocessado, 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, a espera pelo giro será bastante

ineficaz. Para esses segmentos de código maiores, as variáveis de condição e os semáforos são utilizados. Se o lock desejado já estiver sendo mantido, a thread 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 freqüência, 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 entrarão no turnstile para esse bloqueio. Quando o lock é liberado, o kernel seleciona uma thread a partir 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 subseqüentes 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** (Seção 6.5). Isso significa que, 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 6.5; os mecanismos de lock da thread no nível do usuário não oferecem essa funcionalidade.

Para otimizar o desempenho no Solaris, os desenvolvedores refinaram e ajustaram os métodos de bloqueio. Como os bloqueios são usados com freqüência e são usados para funções cruciais do kernel, o ajuste de sua implementação e uso pode produzir ganhos tremendos no desempenho.

### 7.9.2 Sincronismo no Windows XP

O sistema operacional Windows XP é um kernel multithread, que também oferece suporte para aplicações de tempo real e mutiprocessadores. Quando o kernel do Windows XP acessa um recurso global em um sistema monoprocessado, ele mascara temporariamente as interrupções para todos os tratadores de interrupção que também podem acessar o recurso global. Em um sistema de mutiprocessadores, 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 preempetada enquanto mantém um spinlock. Para o sincronismo da thread fora do kernel, o Windows XP oferece **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 libere-o quando ter-

minar. Os **eventos** são semelhantes às variáveis de condição; ou seja, eles podem notificar uma thread aguardando quando ocorrer uma condição desejada.

Os objetos despachantes podem estar em um estado **sinalizado** ou **não-sinalizado**. Um 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. 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 poderão retomar a execução. A quantidade de threads que o kernel seleciona na fila de fluxos esperando depende do tipo de objeto despachante que estão esperando. O kernel selecionará apenas uma thread a partir 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 um mutex como um 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:

1. será movida do estado de espera para o estado pronto e
2. obterá o mutex.

### 7.9.3 Sincronismo no Linux

O kernel do Linux é não preemptivo enquanto executa em modo kernel. Isso significa que um processo executando nesse modo não pode ser preemptado por um processo com uma prioridade mais alta. Essa política simplifica o projeto do kernel; condições de corrida sobre as estruturas de dados do kernel são evitadas, pois um processo não pode ser preemptado enquanto está sendo executado no kernel. No entanto, é possível ocorrer uma interrupção enquanto um processo está executando em modo kernel. Portanto, para seções críticas consistindo em seções curtas de código, o Linux desativa interrupções. Isso só é eficaz para seções críticas curtas, pois não desejamos desativar interrupções por longos períodos. Para seções críticas maiores, o Linux utiliza semáforos para bloquear dados do kernel. Alguns exemplos de estruturas de dados do kernel que são bloqueadas por semáforos incluem estruturas para gerenciamento de memória e sistemas de arquivo. Em máquinas com mutiprocessadores, o Linux utiliza spinlocks e também semáforos para proteger estruturas de dados compartilhadas do kernel.

### 7.9.4 Sincronismo no Pthreads

A API Pthreads oferece mutex e variáveis de condição para sincronismo de thread. Essa API está disponível para programadores e não faz parte de qualquer kernel em particular. Os mutex 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 7.7. Muitos sistemas que implementam Pthreads também oferecem semáforos, embora não façam parte do padrão Pthreads e, em vez disso, pertençam ao padrão POSIX.1b.

Existem extensões à API Pthreads, embora não sejam consideradas portáveis. Essas extensões incluem locks de leitura-escrita e spinlocks.

## 7.10 Transações atômicas

A exclusão mútua de seções críticas garante que as seções críticas são executadas atomicamente. Ou seja, se duas seções críticas forem executadas ao mesmo tempo, o resultado será equivalente à sua execução seqüencial em alguma ordem desconhecida. Embora essa propriedade seja útil em muitos do-

mínios de aplicação, em outros 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.

O restante desta seção está relacionado à área dos sistemas de banco de dados. Os **bancos de dados** tratam do armazenamento e da recuperação de dados e da coerência dos 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 forem usados em seu lugar. Nas Seções 7.10.2 a 7.10.4, descrevemos quais são essas técnicas de banco de dados e como podem ser usadas pelos sistemas operacionais.

### 7.10.1 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. Nesta seção, descrevemos diversos mecanismos para garantir a atomicidade da transação. Primeiro, consideramos um ambiente em que apenas uma transação pode ser executada de uma só vez. Então, consideramos o caso em que várias transações estão ativas concorrentemente.

Uma transação é uma unidade de programa que acessa e atualiza diversos itens de dados que podem residir no disco dentro de alguns arquivos. Pelo nosso ponto de vista, uma transação é uma seqüência 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 precisou abandonar sua execução normal por causa de algum erro lógico ou por causa de uma falha no sistema. Uma transação que terminou e completou sua execução com sucesso está **confirmada**; caso contrário, ela é **cancelada**. O efeito de uma transação confirmada não pode ser desfeito pelo cancelamento da transação.

Como uma transação cancelada já pode ter modificado os diversos dados acessados, o estado desses dados pode não ser igual ao estado que haveria se a transação tivesse sido executada atomicamente. Para que a propriedade de atomicidade seja garantida, uma transação cancelada não poderá ter efeito sobre o estado dos dados modificados. 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 aos dados de acesso.

- **Armazenamento estável:** As informações residindo no armazenamento estável *nunca* são perdidas (*nunca* deve ser considerado com um 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 uma maneira controlada (Seção 14.7).

Aqui, estamos preocupados apenas em garantir a atomicidade de transações em um ambiente em que falhas resultam em perda de informações no armazenamento volátil.

### 7.10.2 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 <$T_i$ `starts`> 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 <$T_i$ `commits`> 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 das 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( )`, 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 seus respectivos valores antigos e novos podem ser encontrados no log.

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 <$T_i$ `starts`>, mas não tiver o registro <$T_i$ `commits`>.
- A transação $T_i$ precisa ser refeita se o log tiver os registros <$T_i$ `starts`> e <$T_i$ `commits`>.

### 7.10.3 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 seqüência 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> subseqüente.

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 redo($T_k$).
- Para todas as transações $T_k$ em $T$ que não possuem um registro <$T_k$ commits> no log, executar undo($T_k$).

### 7.10.4 Transações atômicas concorrentes

Como cada transação é atômica, a execução concorrente de transações precisa ser equivalente ao caso em que essas transações são executadas em série, em alguma ordem qualquer. Essa propriedade, chamada **serialização**, pode ser mantida 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.

#### 7.10.4.1 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 seqüência de execução, chamada **roteiro** (schedule), está representada na Figura 7.40. No roteiro 1 da Figura 7.40,

| $T_0$ | $T_1$ |
|---|---|
| read($A$) | |
| write($A$) | |
| read($B$) | |
| write($B$) | |
| | read($A$) |
| | write($A$) |
| | read($B$) |
| | write($B$) |

FIGURA 7.40 *Roteiro 1: Um roteiro serial em que $T_0$ é seguida por $T_1$.*

| $T_0$ | $T_1$ |
|---|---|
| read($A$) | |
| write($A$) | |
| | read($A$) |
| | write($A$) |
| read($B$) | |
| write($B$) | |
| | read($B$) |
| | write($B$) |

FIGURA 7.41 *Roteiro 2: Um roteiro serializável concorrente.*

a seqüência 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.

Um roteiro em que cada transação é executada atomicamente é denominado **roteiro serial**. Cada roteiro serial consiste em uma seqüência de instruções de várias transações, na qual as instruções pertencentes a uma única transação aparecem juntas. Assim, para um conjunto de $n$ transações, existem $n!$ roteiros seriais válidos. Cada roteiro serial é correto, pois é equivalente à execução atômica das diversas 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 necessariamente implica 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 existem 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, consideramos o roteiro não serial 2 da Figura 7.41. 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.

Sejam $O_i$ e $O_j$ operações consecutivas de um roteiro $R$. 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 $R'$. Esperamos que $R$ seja equivalente a $R'$, 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 idéia de troca considerando novamente o roteiro 2 da Figura 7.41. 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. Independente do estado inicial do sistema, os dois roteiros produzem o mesmo estado final do sistema. Continuando com esse procedimento de troca de operações não conflitantes, obtemos:

- 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 7.40, que é um roteiro serial. Assim, mostramos que o roteiro 2 é equivalente a um roteiro serial. Esse resultado implica que, independente do estado inicial do sistema, o roteiro 2 produzirá o mesmo estado final que algum roteiro serial.

Se o roteiro $R$ puder ser transformado em um roteiro serial $R1$ por uma série de trocas de operações não conflitantes, dizemos que um roteiro $R$ é **serializável por conflito**. Assim, o roteiro 2 é serializável por conflito, pois pode ser transformado no roteiro serial 1.

### 7.10.4.2 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 esse item, mas não pode escrever $Q$.
- **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 $Q$, então $T_i$ 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 7.6.2.

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 bloqueio em duas fases em geral garante a serialização de conflitos (Exercício 7.21). Contudo, ele não garante ausência de deadlock. 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.

### 7.10.4.3 Protocolos Timestamp-based

Nos protocolos de lock descritos anteriormente, a ordem de serialização de cada par de transações em conflito é determinada no momento da execução, pelo primeiro lock que ambos requisitam e que envolve modos incompatíveis. Outro método para determinar a ordem de serialização é selecionar uma ordenação entre transações 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 processos 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`($Q$) é 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 $TS(T_i) <$ 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 $TS(T_i) \geq$ estampa-W($Q$), então a operação `read` é executada, e estampa-R($Q$) é definida para o maior valor dentre estampa-R($Q$) e $TS(T_i)$.
- Suponha que a transação $T_i$ emita `write`($Q$):
  - Se $TS(T_i) <$ 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 $TS(T_i) <$ estampa-W($Q$), então $T_i$ está tentando escrever um valor absoluto 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 pelo esquema de controle de concorrência como resultado da emissão de uma operação `read` ou `write` recebe uma nova estampa de tempo e é reiniciada.

Para ilustrar o protocolo de ordenação por estampa de tempo, consideramos o roteiro 3 da Figura 7.42 com as transações $T_2$ e $T_3$. Consideramos

| $T_2$ | $T_3$ |
|---|---|
| read($B$) | |
| | read($B$) |
| | write($B$) |
| read($A$) | |
| | read($A$) |
| | write($A$) |

FIGURA 7.42 *Um roteiro possível sob o protocolo de estampa de tempo.*

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.

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 7.22).

O protocolo de ordenação 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.

## 7.11 Resumo

Dada uma coleção de processos ou threads seqüenciais cooperativos que compartilham dados, a exclusão mútua precisa ser fornecida para impedir o surgimento de uma condição de corrida sobre os dados compartilhados. Uma solução é garantir que uma seção crítica do código esteja em uso apenas por um processo ou thread de cada vez. Existem diversas soluções de software para solucionar o problema da seção crítica, supondo que somente o interbloqueio de armazenamento esteja disponível.

A principal desvantagem das soluções de software é que não são fáceis de generalizar para multithreads ou para problemas mais complexos do que o problema de seção crítica. Os semáforos contornam essa dificuldade. Podemos usar semáforos para solucionar diversos problemas de sincronismo e podemos implementá-los 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 na 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.

Java oferece um mecanismo para coordenar as atividades multithread quando acessam dados compartilhados por meio dos mecanismos `synchronized`, `wait( )`, `notify( )` e `notifyAll( )`. O sincronismo em Java é fornecido no nível da linguagem. O sincronismo em Java é um exemplo de um mecanismo de sincronismo de nível mais alto, chamado de monitor. Além da Java, muitas linguagens forneceram suporte para monitores, incluindo Concurrent Pascal e Mesa.

Os sistemas operacionais também oferecem suporte para sincronismo de thread. Por exemplo, Solaris, Windows XP e Linux oferecem mecanismos, como semáforos, mutexes, spinlocks e variáveis de condição, para controlar o acesso aos dados compartilhados. A API Pthreads oferece 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.

Quando várias transações sobrepõem sua execução, a execução resultante não pode mais ser equivalente a quando essas transações são executadas atomicamente. Para garantir a execução correta, temos de usar um esquema de controle de concorrência para garantir a serialização. 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

**7.1** A primeira solução de software correta conhecida para o problema de seção crítica para duas threads foi desenvolvida por Dekker; ela aparece na Figura 7.43. As duas threads, $T_0$ e $T_1$, coordenam o compartilhamento de atividades de um objeto da classe `Dekker`. Mostre que o

```
public class Dekker implements MutualExclusion
{
   private volatile int turn = TURN_0;
   private volatile boolean flag0 = false;
   private volatile boolean flag1 = false;

   public void enteringCriticalSection(int t) {
      int other = 1 -- t;
      if (t == 0) {
         flag0 = true;
      else
         flag1 = true;

      if (t == 0) {
      while (flag1 == true) {
         if (turn == other) {
            flag0 = false;
            while (turn == other)
               Thread.yield( );
            flag0 = true;
         }
      }
      }
      else {
      while (flag0 == true) {
         if (turn == other) {
            flag1 = false;
            while (turn == other)
               Thread.yield( );
            flag1 = true;
         }
      }
      }
   }

   public void leavingCriticalSection(int t) {
      turn = 1 -- t;
      if (t == 0)
         flag0 = false;
      else
         flag1 = false;
   }
}
```

FIGURA 7.43 *Algoritmo de Dekker para a exclusão mútua.*

algoritmo satisfaz a todos os três requisitos para o problema da seção crítica.

**7.2** No Capítulo 5, mostramos uma solução multithread para o problema de bounded buffer, que usava a troca de mensagens. A classe `MessageQueue` não é considerada segura para threads, significando que uma condição de corrida é possível quando várias threads tentam acessar a fila ao mesmo tempo. Modifique a classe MessageQueue usando o sincronismo Java para torná-la segura para threads.

**7.3** Implemente a interface `Channel` do Capítulo 4 de modo que os métodos `send( )` e `receive( )` sejam de bloqueantes. Isso exigirá armazenar as mensagens em um array de tamanho fixo. Garanta que sua implementação seja segura para threads (usando o sincronismo Java) e que as mensagens sejam armazenadas na ordem FIFO.

**7.4** A classe `HardwareData` da Figura 7.7 utiliza a idéia 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 realizado atomicamente. Reescreva a classe `HardwareData` usando o sincronismo Java, de modo que seja segura para threads.

**7.5** Os servidores podem ser projetados de modo a limitar o número de conexões abertas. Por exemplo, um servidor pode querer ter apenas *N* conexões de socket em determinado momento. Assim que *N* conexões forem feitas, o servidor não aceitará outra conexão que chegue até uma conexão existente ser liberada. Explique como os semáforos podem ser usados por um servidor para limitar o número de conexões simultâneas.

**7.6** Crie uma classe `BinarySemaphore` que implemente um semáforo binário.

**7.7** A instrução `wait( )` em todos os exemplos de programa Java fazia parte de um loop `while`. Explique por que você sempre usaria uma instrução `while` ao usar `wait( )` e por que você nunca a usaria em uma instrução `if`.

**7.8** A solução para o problema de leitores-escritores não impede o starvation dos escritores que estão esperando: se o banco de dados estiver sendo lido e houver um escritor esperando, leitores subseqüentes terão permissão para ler o banco de dados antes de o escritor poder escrever. Modifique a solução de modo que os escritores esperando não sofram starvation.

**7.9** Uma solução baseada em monitor para o problema dos filósofos na mesa de jantar, escrita em pseudocódigo tipo Java e usando variáveis de condição, foi dada na Seção 7.7. Desenvolva uma solução para o problema dos filósofos na mesa de jantar usando o sincronismo Java.

**7.10** A solução que demos para o problema dos filósofos na mesa de jantar não impede a starvation de um filósofo. Por exemplo, dois filósofos – digamos, $filósofo_1$ e $filósofo_3$ – poderiam alternar entre comer e pensar, de modo que o $filósofo_2$ nunca poderia comer. Usando o sincronismo Java, desenvolva uma solução para o problema dos filósofos na mesa de jantar que impeça o starvation de um filósofo.

**7.11** Na Seção 7.4, mencionamos que a desativação de interrupções com freqüência poderia afetar o relógio do sistema. Explique por que isso poderia acontecer e como esses efeitos poderiam ser minimizados.

**7.12** Dê os motivos pelos quais Solaris, Windows XP e Linux implementam mecanismos de lock múltiplos. 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.

**7.13** Conforme descrevemos na Seção 7.9.3, o Linux utiliza não preempção e a desativação de interrupção para proteger os dados do kernel em sistemas monoprocessados, garantindo assim que no máximo um processo de cada vez esteja ativo no kernel. Entretanto, em sistemas com mutiprocessadores simétricos (SMP), dois processos podem estar ativos no kernel ao mesmo tempo, enquanto são executados em processadores diferentes. A primeira versão do Linux que admitiu SMP (o kernel do Linux 2.0) permitia que vários processos fossem executados concorrentemente em diferentes processadores; porém, apenas um processo por vez poderia estar sendo executado no modo kernel. Comente a eficácia dessa estratégia de SMP.

**7.14** 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 certo ponto (a barreira), eles poderão continuar.

O segmento de código a seguir estabelece uma barreira e cria 10 threads `Worker` que serão sincronizadas de acordo com a barreira:

```
Barrier jersey = new Barrier(10);
for (int i = 0; i < 10; 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;
// 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. Implemente uma classe de barreira usando o sincronismo Java. Essa classe oferecerá um construtor e um método `waitForOthers( )`.

**7.15** Crie um banco de threads (ver Capítulo 5) 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 5.7.4). Naturalmente, isso exige que as tarefas Runnable executadas pelo banco de threads verifiquem seu status de interrupção.

**7.16** *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, então o cliente vai embora. Se o barbeiro estiver ocupado, mas houver cadeiras disponíveis, então 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.

**7.17** *O problema dos fumantes.* Considere um sistema com três processos *fumantes* e um processo *agente*. Cada fumante continuamente enrola um cigarro e depois o fuma. Mas, para enrolar e fumar um cigarro, o fumante precisa de três ingredientes: tabaco, papel e fósforos. Um dos processos fumantes tem papel, outro tem tabaco e o terceiro tem fósforos. O agente possui um estoque infinito de todos os três materiais. O agente coloca dois dos ingredientes na mesa. O fumante que possui o ingrediente restante, então, enrola e fuma um cigarro, sinalizando para o agente quando terminar. O agente, então, coloca outros dois dos três ingredientes, e o ciclo se repete. Escreva um programa para sincronizar o agente e os fumantes usando o sincronismo Java.

**7.18** Explique as diferenças, em termos de custo, entre os três tipos de armazenamento: volátil, não volátil e estável.

**7.19** Explique a finalidade do mecanismo de checkpoint. Com que freqüência os pontos de verificação devem ser realizados? Descreva como a freqüência dos pontos de verificação afeta:

- O desempenho do sistema quando não corre falha
- O tempo gasto para a recuperação de uma falha do sistema
- O tempo gasto para a recuperação de uma falha no disco

**7.20** Explique o conceito de atomicidade da transação.

**7.21** Mostre que o protocolo de lock em duas fases garante a serialização de conflitos.

**7.22** Mostre que alguns roteiros são possíveis sob o protocolo de lock em duas fases, mas não é possível sob o protocolo de estampa de tempo, e vice-versa.

## Notas bibliográficas

Os algoritmos de exclusão mútua 1 e 2 para duas tarefas foram discutidos inicialmente no trabalho clássico de Dijkstra [1965a]. O algoritmo de Dekker (Exercício 7.1) – 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] (algoritmo 3).

Dijkstra [1965b] apresentou a primeira solução para o problema de exclusão mútua para $n$ processos. Essa solução, porém, não possui um limite máximo sobre a quantidade de tempo que um processo precisa esperar antes que o processo tenha permissão para entrar na seção crítica.

Knuth [1966] apresentou o primeiro algoritmo com um limite; seu limite foi $2^n$ vezes. Uma melhoria do algoritmo de Knuth, por deBruijn [1967], reduziu o tempo de espera para $n^2$ vezes, e depois disso Eisenberg e McGuire [1972] tiverem sucesso ao reduzir o tempo para o limite inferior de $n - 1$ vezes. Lamport [1974] apresentou um esquema diferente para solucionar o problema de exclusão mútua – o algoritmo do padeiro. Ele também exige $n - 1$ vezes, mas é mais fácil de programar e entender. Burns [1978] desenvolveu o algoritmo de solução de 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].

Informações sobre soluções de hardware para o sincronismo de processos poderão ser encontradas em Patterson e Hennessy [1998].

O conceito de semáforo foi sugerido por Dijkstra [1965a]. Patil [1971] examinou a questão de 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 a limitação 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 na mesa de jantar e o problema do barbeiro dorminhoco (Exercício 7.16) foram sugeridos por Dijkstra [1965a] e Dijkstra [1971]. O problema dos fumantes (Exercício 7.17) 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 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. Um artigo descrevendo as classificações para monitores foi publicado por Buhr e outros [1995]. Discussões gerais referentes à programação concorrente foram oferecidas por Ben-Ari [1990], e Burns e Davies [1993].

Os detalhes de como a Java sincroniza as threads podem ser encontrados em Oaks e Wong [1999], Holub [2000], e Lewis e Berg [2000]. Lea [2000] apresenta muitos padrões de projeto para a programação concorrente em Java. O Java Report [1998] é dedicado a assuntos avançados de multithreads e sincronismo em Java.

As primitivas de sincronismo para Windows 2000 são discutidas por Solomon e Russinovich [2000]. Os detalhes dos mecanismos de sincronismo usados no Solaris e Linux são apresentados por Mauro e McDougall [2001], e Bovet e Cesati [2002], respectivamente. Butenhof [1997] discute as questões de sincronismo na API Pthreads.

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 bloqueio 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].

CAPÍTULO 8

# 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. 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 7, em conjunto com os semáforos.

Talvez a melhor ilustração de um deadlock possa ser tirada de uma lei que passou pela 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 impedir ou lidar com deadlocks. A maioria dos sistemas operacionais modernos não provê facilidades de prevenção de deadlock, mas esses recursos provavelmente serão acrescentados em breve. Os problemas de deadlock só podem se tornar mais comuns, dadas as tendências atuais, incluindo grandes quantidades de processos, programas multithreads, muito mais recursos dentro de um sistema e uma ênfase em servidores de arquivos e banco de dados de longa duração, ao invés de sistemas batch.

## 8.1 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 dos quais consistindo em alguma quantidade de instâncias idênticas. Espaço de memória, ciclos de CPU, arquivos e dispositivos de E/S (como impressoras e unidades de fita) 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* poderia 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 foram definidas corretamente. Por exemplo, um sistema pode ter duas impressoras. Essas duas impressoras podem ser definidas como estando na mesma classe de recurso; isso 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 sendo equivalentes, e classes de recurso separadas poderão ser definidas 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 quan-

tos 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 seqüência:

1. **Requisitar:** Se a requisição não puder ser concedida imediatamente (por exemplo, se o recurso estiver sendo usado por outro processo), então 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 3. Alguns exemplos são as chamadas de sistema `request` e `release device`, `open` e `close 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 em um objeto Java, por meio da palavra-chave `synchronized`. Para cada uso de um recurso por um processo ou thread, o sistema operacional verifica para garantir que o processo tenha requisitado e tenha recebido um recurso. Uma tabela do sistema registra se cada recurso está liberado ou alocado; para cada recurso alocado, ele também registra o processo ao qual é alocado. 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 principalmente aqui são 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 4).

Para ilustrar um estado de deadlock, considere um sistema com três unidades de fita. Suponha que cada um de três processos mantenha uma dessas unidades de fita. Se cada processo agora requisitar outra unidade de fita, os três processos estarão em um estado de deadlock. Cada um está esperando pelo evento "unidade de fita está liberada", que só pode ser causado por um dos três outros processos esperando. Esse exemplo ilustra um deadlock envolvendo o mesmo recurso.

Os deadlocks também podem envolver diferentes tipos de recursos. Por exemplo, considere um sistema com uma impressora e uma unidade de fita. Suponha que o processo $P_i$ esteja mantendo a unidade de fita e o processo $P_j$ esteja mantendo a impressora. Se $P_i$ requisitar a impressora e $P_j$ requisitar a unidade de fita, 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 (como locks de objeto).

## 8.2 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.

### 8.2.1 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 retém, depois de esse processo ter concluído sua tarefa.
4. **Espera circular:** Existe um conjunto $\{P_0, P_1, ..., P_n\}$ de processos esperando, de modo que $P_0$ aguarda um recurso retido por $P_1$, $P_1$ aguarda um recurso retido por $P_2$, ..., $P_{n-1}$ 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 8.4, que é útil considerar cada condição separadamente.

### 8.2.2 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 particionado em dois tipos de nós diferentes: $P = \{P_1, P_2, ..., P_n\}$, o conjunto consistindo em todos os processos ativos no sistema, e $R = \{R_1, R_2, ..., R_m\}$, 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 é *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 é excluída.

O grafo de alocação de recursos mostrado na Figura 8.1 representa a seguinte situação:

- 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:
  - Processo $P_1$ tem a posse de uma instância do recurso $R_2$ e aguardando uma instância do recurso $R_1$.
  - Processo $P_2$ tem a posse de uma instância de $R_1$ e $R_2$ e aguardando uma instância do recurso $R_3$.
  - Processo $P_3$ tem a posse de uma instância de $R_3$.

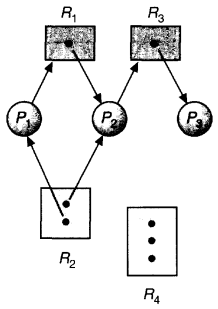



FIGURA 8.1 *Grafo de alocação de recursos.*

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, então 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 8.1. 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 8.2). Nesse ponto, existem dois ciclos mínimos no sistema:

$$P_1 \rightarrow R_1 \rightarrow P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$
$$P_2 \rightarrow R_3 \rightarrow P_3 \rightarrow R_2 \rightarrow P_2$$

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 8.3. Nesse exemplo, também temos um ciclo

$$P_1 \rightarrow R_1 \rightarrow P_3 \rightarrow R_2 \rightarrow P_1$$

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 um estado de deadlock. Se houver um ciclo, então o sistema pode ou não estar em um estado de deadlock. Essa observação é importante quando lidamos com o problema de deadlock.

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 8.4.

Nesse exemplo, `threadA` tenta obter a posse dos locks dos objetos na ordem de (1) `mutexX`, (2) `mutexY`; e `threadB` tenta usar a ordem de (1) `mutexY`, (2) `mutexX`. O deadlock é possível no seguinte cenário:

```
threadA → mutexY → threadB → mutexX → threadA
```

Observe que, embora o deadlock seja possível, ele não ocorrerá se `threadA` for capaz de obter e liberar os locks do `mutexX` e `mutexY` 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.

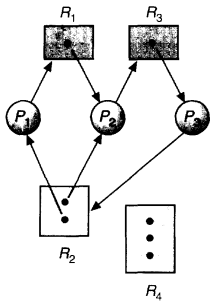



FIGURA 8.2 *Grafo de alocação de recursos com um deadlock.*

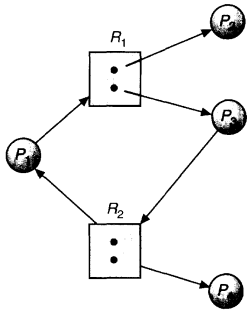



FIGURA 8.3 *Grafo de alocação de recursos com um ciclo, mas sem deadlock.*

```
class Mutex { }

class A implements Runnable
{
   private Mutex first, second;
   public A(Mutex first, Mutex second) {
      this.first = first;
      this.second = second;
   }

   public void run( ) {
      synchronized (first) {
         // faz alguma coisa
         synchronized (second) {
            // faz alguma coisa
         }
      }
   }
}

class B implements Runnable
{
   private Mutex first, second;
   public B(Mutex f, Mutex second) {
      this.first = first;
      this.second = second;
   }

   public void run( ) {
      synchronized (second) {
         // faz alguma coisa
         synchronized (first) {
            // faz alguma coisa
         }
      }
   }
}

public class DeadlockExample
{
   // Figura 8.5
}
```

FIGURA 8.4 *Exemplo de deadlock.*

## 8.3 Métodos para tratamento de deadlocks

Essencialmente, 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 um estado de deadlock.

```
public static void main(String arg[ ]) {
   Mutex mutexX = new Mutex( );
   Mutex mutexY = new Mutex( );

   Thread threadA = new Thread(new
A(mutexX,mutexY));
   Thread threadB = new Thread(new
B(mutexX,mutexY));

   threadA.start( );
   threadB.start( );
}
```

FIGURA 8.5 *Criando as threads.*

- Podemos permitir que o sistema entre em um 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.

A seguir, vamos fazer uma breve descrição de cada um desses três métodos para tratamento de deadlocks. Depois, nas Seções de 8.4 a 8.7, 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.

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 8.2.1) não poderá ser satisfeita. Esses métodos previnem deadlocks restringindo o modo como as requisições de recursos podem ser feitas. Discutiremos esses métodos na Seção 8.4.

**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 co-

nhecimento 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 8.5.

Se um sistema não emprega um algoritmo para prevenção de deadlock ou um para evitar deadlock, então uma situação de deadlock poderá ocorrer. Nesse ambiente, o sistema pode provê 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 8.6 e 8.7.

Se um sistema não garantir que um deadlock nunca ocorrerá nem provê um mecanismo para detecção e recuperação de deadlock, então podemos chegar a uma situação em que o sistema está em um 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 um 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 freqüência (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 um estado "congelado", mas não em um 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.

Como já observamos, a JVM não faz nada para controlar os deadlocks; fica a critério do desenvolvedor da aplicação a escrita de programas livres de 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 5, apresentamos as threads Java e parte da API que permite que os usuários criem e manipulem threads. Dois métodos adicionais da classe `Thread` são os métodos `suspend( )` e `resume( )`, que foram desaprovados na Java 2 porque poderiam levar ao deadlock. 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.

`stop( )` também foi desaprovado, mas não porque pode levar ao deadlock. Ao contrário da situação em que uma thread foi suspensa, quando uma thread tiver terminada, ela libera todos os locks que possui. Contudo, os locks em geral são usados na seguinte seqüência: (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 5.7.4, explicamos como terminar uma thread usando uma técnica diferente do 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 8.6 é um applet multithreads, que exibe a hora do dia. Quando esse applet é iniciado, ele cria uma segunda thread (que chamaremos de *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;
   provate 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 8.7
   }

   public void stop( ) {
      // Figura 8.7
   }

   public void paint(Graphics g) {
      g.drawString( new java.util.Date( ).toString( ), 10, 30);
   }
}
```

FIGURA 8.6 *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, então 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( )`

```
// 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
// da página onde o applet se encontra
public void stop( ) {
   synchronized(mutex) {
      ok = false;
   }
}
```

FIGURA 8.7 *Métodos start( ) e stop( ) para o applet.*

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 estiverem compartilhando essa variável, o acesso a ela será controlado por um bloco `synchronized`. Esse programa aparece na Figura 8.6.

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.

## 8.4 Prevenção de deadlock

Conforme observamos na Seção 8.2.1, 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.

### 8.4.1 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.

### 8.4.2 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 fita 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 fita, o arquivo de disco e a impressora. Ele manterá a impres-

sora por toda a sua execução, embora só precise dela no final.

O segundo método permite ao processo requisitar apenas a unidade de fita e o arquivo de disco. Ele copia da unidade de fita para o disco e depois libera a unidade de fita 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 fita 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.

Em segundo lugar, 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.

Essa solução também não é prática em Java, pois um processo requisita recursos (locks) entrando com métodos ou `synchronized` ou blocos. Como os recursos de locks são requisitados dessa maneira, é difícil escrever uma aplicação que siga qualquer um dos protocolos indicados.

### 8.4.3 Não preempção

A terceira condição necessária 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 são preemptados. Em outras palavras, esses recursos são 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 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, apropriamo-nos dos 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.

### 8.4.4 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 $R = \{R_1, R_2, ..., R_m\}$ 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 um-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 $F$ poderia ser definida da seguinte forma:

$F$(unidade de fita) = 1
$F$(unidade de disco) = 5
$F$(impressora) = 12

Agora, podemos considerar o seguinte protocolo para prevenir deadlocks: cada processo só pode requisitar recursos em ordem crescente de enumera-

ção. 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 $F(R_j) > F(R_i)$. Se várias instâncias do mesmo recurso forem necessárias, uma *única* requisição para todas elas deverá ser emitida. 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 $F(R_i) \geq F(R_j)$.

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 $\{P_0, P_1, ..., P_n\}$, 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 $R_n$ 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 $F(R_0) < F(R_1) < ... < F(R_n) < F(R_0)$. 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 no sistema. Todas as requisições de locks de objetos precisam ser feitas em ordem crescente. Por exemplo, se a ordenação de lock no programa Java mostrado na Figura 8.4 fosse

$F($`mutexX`$) = 1$
$F($`mutexY`$) = 5$

então a classe B 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 $F$(unidade de fita) < $F$(impressora).

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 a **witness**. Witness com locks de exclusão mútua que protegem seções críticas, conforme descrevemos no Capítulo 7; ele atua mantendo dinamicamente o relacionamento das ordens dos lock em um sistema. Vamos usar o programa mostrado na Figura 8.4 como um exemplo. Suponha que `threadA` seja o primeiro a obter os locks e faça isso na ordem (1) `mutexX`, (2) `mutexY`. A witness registra o relacionamento de que `mutexX` precisa ser adquirido antes de `mutexY`. Se `threadB` mais tarde obtiver os locks fora de ordem, a testemunha gerará uma mensagem de aviso no console do sistema.

## 8.5 Evitar deadlock

Os algoritmos de prevenção de deadlock, discutidos na Seção 8.4, previnem os deadlocks restringindo o modo como as requisições são feitas. As restrições 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 seqüência completa de requisições e liberações para cada processo, podemos decidir, para cada requisitação, 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 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 um estado de deadlock. Tal algoritmo define a técnica para **evitar 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, exploramos dois algoritmos para evitar deadlock.

### 8.5.1 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 um estado seguro somente se houver uma **seqüência segura**. Uma seqüência de processos $<P_1, P_2, ..., P_n>$ é uma seqüência 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 seqüência, então o estado do sistema é 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 8.8). Um estado inseguro *pode* ocasionar um deadlock. Desde que o estado seja seguro, ó sistema operacional poderá evitar

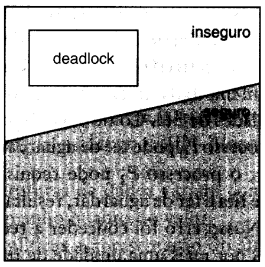



FIGURA 8.8 *Espaço dos estados seguro, inseguro e com deadlock.*

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.

Para ilustrar, consideramos um sistema com 12 unidades de fita magnética e 3 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 3 unidades de fita livres.)

| | Necessidades máximas | Necessidades atuais |
|---|---|---|
| $P_0$ | 10 | 5 |
| $P_1$ | 4 | 2 |
| $P_2$ | 9 | 2 |

No instante $t_0$, o sistema está em um estado seguro. A seqüência $<P_1, P_0, P_2>$ 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).

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 re-

ceber 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. Como elas não estão disponíveis, o processo $P_0$ pode ter de aguardar. De modo semelhante, o processo $P_2$ pode requisitar mais 6 unidades de fita e ter de aguardar, resultando em um deadlock. Nosso 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 um estado seguro, podemos definir os algoritmos para evitar deadlocks que garantem que o sistema nunca estará em condição de deadlock. A idéia é garantir que o sistema sempre permanecerá em um estado seguro. A princípio, o sistema está em um estado seguro. Sempre que um processo requisitar um recurso disponível, o sistema terá de decidir se o recurso pode ser alocado ou se o processo precisa esperar. A requisição só é concedida se a alocação sair do sistema em um estado seguro.

Nesse esquema, se um processo requisitar um recurso disponível, ele ainda pode ter de esperar. Assim, a utilização de recursos pode ser menor do que seria sem um algoritmo para evitar deadlock.

### 8.5.2 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 8.2.2, 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 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 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 um estado seguro. Se for encontrado um ciclo, então a alocação colocará o sistema em um 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 8.9. 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 8.10). Um ciclo indica que o sistema está em um estado inseguro. Se $P_1$ requisitar $R_2$, e $P_2$ requisitar $R_1$, então haverá um deadlock.

### 8.5.3 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

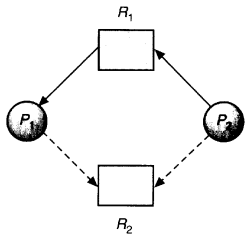



FIGURA 8.9 *Grafo de alocação de recursos para evitar deadlock.*

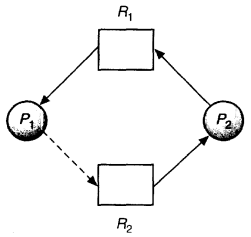



FIGURA 8.10 *Um estado inseguro em um grafo de alocação de recursos.*

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 recursos 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 um 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. Seja $n$ o número de processos no sistema e seja $m$ o número de tipos de recursos. Usamos as seguintes estruturas de dados:

- **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 $n$ x $m$ 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 $n$ x $m$ define o número de recursos de cada tipo alocados 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 $n$ x $m$ 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, vamos estabelecer 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 $i = 1, 2, ..., n$. Por exemplo, se $X = (1,7,3,2)$ e $Y = (0,3,2,1)$, então $Y \leq X$. $Y < X$ 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*, respectivamente. 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.

### 8.5.3.1 Algoritmo de segurança

O algoritmo para descobrir se um sistema está ou não em um estado seguro pode ser descrito da seguinte maneira:

1. Sejam *Trabalho* e *Fim* vetores de tamanho $m$ e $n$, respectivamente. Inicialize *Trabalho* = *Disponível* e *Fim*[$i$] = *false* para $i = 0, ..., n$-1.
2. Encontre um $i$ tal que
   a. *Fim*[$i$] == *false*
   b. $Necessário_i \leq Trabalho$
   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*[$i$] == *true* para todo $i$, então o sistema está em um estado seguro.

Esse algoritmo pode exigir uma ordem de $m \times n^2$ operações para decidir se um estado é seguro.

### 8.5.3.2 Algoritmo de requisição de recursos

Seja $Requisição_i$ o vetor de requisição para o processo $P_i$. Se $Requisição_i[j] == k$, então o processo $P_i$ de-

seja $k$ 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:

   $Disponível = Disponível - Requisição_i;$
   $Alocação_i = Alocação_i + Requisição_i;$
   $Necessário_i = Necessário_i - Requisição_i;$

   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, então $P_i$ terá de esperar por $Requisição_i$, e o antigo estado de alocação de recursos será restaurado.

### 8.5.3.3 Um exemplo ilustrativo

Considere um sistema com cinco processos, de $P_0$ a $P_4$, e três tipos de recursos, $A$, $B$, $C$. O recurso $A$ possui 10 instâncias, o recurso $B$ possui 5 instâncias, e o recurso $C$ possui 7 instâncias. Suponha que, no instante $T_0$, o seguinte instantâneo do sistema tenha sido tirado:

| | Alocação | Máximo | Disponível |
|---|---|---|---|
| | $A$ $B$ $C$ | $A$ $B$ $C$ | $A$ $B$ $C$ |
| $P_0$ | 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

| | Necessário |
|---|---|
| | $A$ $B$ $C$ |
| $P_0$ | 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 um estado seguro. Na realidade, a seqüência $<P_1, P_3, P_4, P_2, P_0>$ 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 $Requisiçãodo_1 \leq Disponíveis$ – 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:

| | Alocação | Necessário | Disponível |
|---|---|---|---|
| | $A$ $B$ $C$ | $A$ $B$ $C$ | $A$ $B$ $C$ |
| $P_0$ | 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 seqüência $<P_1, P_3, P_4, P_0, P_2>$ 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 como um exercício a implementação do algoritmo do banqueiro em Java.

## 8.6 Detecção de deadlock

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 aos 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.

### 8.6.1 Única instância de cada recurso

Se todos os recursos tiverem apenas uma única instância, então podemos definir um algoritmo de detecção de deadlock que usa uma variante do grafo de alocação de recursos, chamada grafo *wait-for*. Obtemos esse grafo a partir 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 8.11, apresentamos um grafo de alocação de recursos e o 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.

### 8.6.2 Várias instâncias de um 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 8.5.3):

- **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 $R_j$.

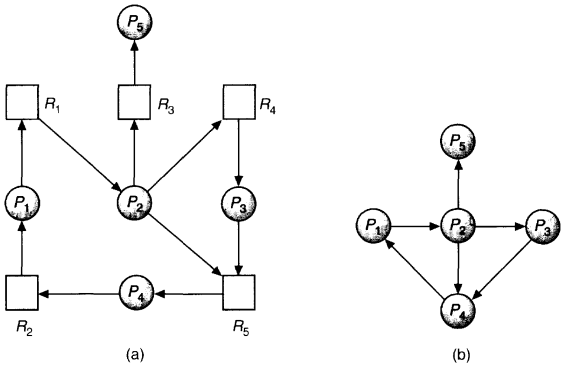



FIGURA 8.11 *(a) Grafo de alocação de recursos. (b) Grafo wait-for correspondente.*

A relação ≤ entre dois vetores é definida da mesma forma que na Seção 8.5.3. 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 seqüência de alocação possível em busca dos processos que ainda precisam ser completados. Compare esse algoritmo com o algoritmo do banqueiro, da Seção 8.5.3.

1. Sejam *Trabalho* e *Fim* vetores de tamanho *m* e *n*, respectivamente. Inicialize *Trabalho* = *Disponível*. Para $i = 0, ..., n-1$, 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 um estado de deadlock. Além do mais, se $Fim[i] == false$, então o processo $P_i$ está em um 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*, *C*. O recurso *A* possui 7 instâncias, o recurso *B* possui 2 instâncias e o recurso *C* possui 6 instâncias. Suponha que, no instante $T_0$, tenhamos o seguinte estado de alocação de recursos:

| | Alocação | Requisição | Disponível |
|---|---|---|---|
| | *A B C* | *A B C* | *A B C* |
| $P_0$ | 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 um estado de deadlock. Na realidade, se executarmos nosso algoritmo, veremos que a seqüência $<P_0, P_2, P_3, P_1, P_4>$ 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:

| | Requisição |
|---|---|
| | *A B C* |
| $P_0$ | 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 $P_4$.

### 8.6.3 Uso do algoritmo de detecção

Quando devemos invocar o algoritmo de detecção? A resposta depende de dois fatores:

1. Com que *freqüência* um deadlock provavelmente ocorrerá?
2. *Quantos* processos serão afetados pelo deadlock quando ele acontecer?

Se os deadlocks ocorrem com freqüência, então o algoritmo de detecção deverá ser invocado também com freqüência. 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, 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 pode gerar um custo adicional considerável em tempo de computação. Uma alternativa menos dispendiosa é envolver o algoritmo em intervalos menos freqüentes – 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 com que a utilização da CPU caia.) Se o algoritmo de detecção for chamado em momentos arbitrários, poderá haver muitos ciclos no grafo de recursos. Em geral, não poderíamos dizer quais dos muitos processos em deadlock "causaram" o deadlock.

## 8.7 Recuperação do deadlock

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. A 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.

### 8.7.1 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 estivesse no meio da atualização de um arquivo, seu cancelamento deixará esse arquivo em um estado incorreto. De modo semelhante, se o processo estivesse no meio da impressão de dados em uma impressora, o sistema teria 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

### 8.7.2 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 que o ciclo de deadlock ser desfeito.

Se a preempção tiver de lidar com deadlocks, então três questões precisam 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 apropriarmos 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 a partir desse 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.

## 8.8 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. Em especial, existem três métodos para lidar com os deadlocks:

- Usar algum protocolo para prevenir ou evitar deadlocks, garantindo que o sistema nunca entrará em um 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, bem como a JVM.

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, então um esquema de detecção e recuperação terá de 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 completa sua tarefa designada.

Finalmente, 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

**8.1** Liste três exemplos de deadlocks que não estão relacionados a um ambiente de sistema computadorizado.

**8.2** É possível ter um deadlock envolvendo apenas um processo com uma única thread? Explique sua resposta.

**8.3** Considere o deadlock de trânsito representado na Figura 8.12.

**a.** Mostre que as quatro condições necessárias para o deadlock realmente são atendidas neste exemplo.

**b.** Indique uma regra simples para evitar deadlocks nesse sistema.

**8.4** Suponha que um sistema esteja em um estado inseguro. Mostre que é possível que os processos completem sua execução sem entrar em um estado de deadlock.

**8.5** 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 locks para cinco objetos Java *A...E*, o deadlock é possível. Podemos prevenir o deadlock acrescentando um sexto objeto *F*. Sempre que uma thread quiser obter o lock para qualquer objeto *A...E*, ela primeiro terá de obter o lock do objeto *F*. Essa solução é conhecida como **contenção**: os locks dos objetos *A...E* estão contidos dentro do lock do objeto *F*. Compare esse esquema com o esquema de espera circular da Seção 8.4.4.

**8.6** 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; ele pode querer mais)

**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.

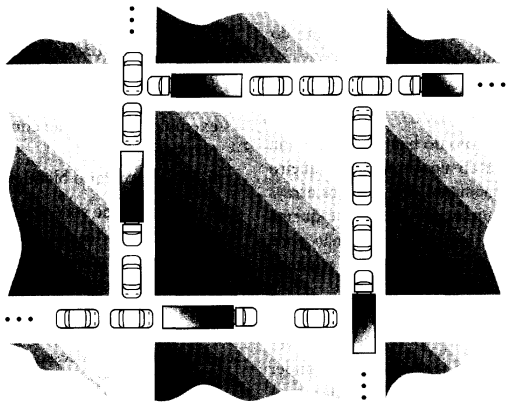


FIGURA 8.12 *Deadlock de trânsito para o Exercício 8.3.*

**8.7** Prove que o algoritmo de segurança apresentado na Seção 8.5.3 exige uma ordem de $m \times n^2$ operações.

**8.8** 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.

**8.9** 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 $m + n$.

**8.10** Considere um sistema computadorizado que execute 5.000 tarefas por mês sem qualquer esquema para prevenção ou para evitar deadlock. Os deadlocks ocorrem cerca de duas vezes por mês, e o operador precisa terminar e reexecutar cerca de 10 tarefas por deadlock. Cada tarefa custa cerca de $2 (em tempo de CPU), e as tarefas terminadas costumam estar prontas pela metade quando são canceladas.

Um programador de sistemas estimou que um algoritmo para evitar deadlock (como o algoritmo do banqueiro) poderia ser instalado no sistema com um aumento no tempo de execução médio por tarefa de cerca de 10%. Como a máquina atualmente possui um tempo ocioso de 30%, todas as 5.000 tarefas por mês ainda poderiam ser executadas, embora o turnaround aumentasse em cerca de 20% na média.

**a.** Quais são os argumentos a favor da instalação do algoritmo para evitar deadlock?

**b.** Quais são os argumentos contra a instalação do algoritmo para evitar deadlock?

**8.11** Podemos obter o algoritmo do banqueiro para um único tipo de recurso a partir do algoritmo do banqueiro geral reduzindo o dimensionamento dos diversos arrays por 1. Mostre, por meio de um exemplo, que o esquema do banqueiro para múltiplos tipos de recursos não pode ser implementado pela aplicação individual do esquema de único recurso a cada recurso.

**8.12** Um sistema pode detectar que algum de seus processos está sofrendo stariation? Se você responder "sim", explique como isso pode ser feito. Se responder "não", explique como o sistema pode lidar com o problema da starvation.

**8.13** Considere o seguinte instantâneo de um sistema:

| | Alocação | Máximo | Disponível |
|---|---|---|---|
| | *A B C D* | *A B C D* | *A B C D* |
| $P_0$ | 0 0 1 2 | 0 0 1 2 | 1 5 2 0 |
| $P_1$ | 1 0 0 0 | 1 7 5 0 | |
| $P_2$ | 1 3 5 4 | 2 3 5 6 | |
| $P_3$ | 0 6 3 2 | 0 6 5 2 | |
| $P_4$ | 0 0 1 4 | 0 6 5 6 | |

Responda as seguintes perguntas usando o algoritmo do banqueiro:

**a.** Qual é o conteúdo da matriz *Necessário*?

**b.** O sistema está em um estado seguro?

**c.** Se uma requisição do processo $P_1$ chegar para (0,4,2,0), a requisição poderá ser concedida imediatamente?

**8.14** Considere a seguinte política de alocação de recursos. As requisições e as 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, esperando por recursos. Se tiverem os recursos desejados, então esses recursos são retirados e dados ao processo requisitante. O vetor de recursos que um processo esperando precisa é aumentado para incluir os recursos que foram retirados.

Por exemplo, considere um sistema com três tipos de recurso e o vetor *Disponível* inicializado como (4,2,2). Se o processo $P_0$ pedir (2,2,1), ele os receberá. Se $P_1$ pedir (1,0,1), ele os receberá. Depois, se $P_0$ pedir (0,0,1), ele será bloqueado (recurso não disponível). Se $P_2$ agora pedir (2,0,0), ele receberá o único disponível (1,0,0) junto com um que estava alocado para $P_0$ (pois $P_0$ está bloqueado). O vetor *Alocação* de $P_0$ diminui para (1,2,1), e seu vetor *Necessário* cresce para (1,0,1).

**a.** Pode ocorrer um deadlock? Se puder, dê um exemplo. Se não, que condição necessária não pode ocorrer?

**b.** Pode ocorrer o bloqueio indefinido (starvation)?

**8.15** Suponha que você tenha codificado o algoritmo de segurança para evitar deadlock e agora tenha sido requisitado a implementar o algoritmo de detecção de deadlock. É possível fazer isso usando o código do algoritmo de segurança e redefinindo $Máximo_i = Esperando_i + Alocação_i$, onde $Esperando_i$ é um vetor especificando os recursos pelos quais o processo $i$ está esperando e $Alocação_i$ é conforme definido na Seção 8.5? Explique sua resposta.

**8.16** Escreva um programa em Java que ilustre o deadlock por meio de métodos **synchronized** chamando outros métodos **synchronized**.

**8.17** Escreva um programa em Java que ilustre o deadlock por meio de threads separadas tentando realizar operações sobre semáforos diferentes.

**8.18** Escreva um programa em Java multithreads que implemente o algoritmo do banqueiro, discutido na Seção 8.5.3. Crie *n* threads que requisitam e liberam recursos do banco. O banqueiro só concederá a requisição se deixar o sistema em um estado seguro. Garanta que o acesso aos dados compartilhados seja seguro para a thread, empregando o sincronismo de threads em Java, conforme discutimos na Seção 7.8.

**8.19** Um túnel de estrada de ferro com um único conjunto de trilhos conecta duas cidades. A estrada de ferro pode obter um deadlock se um trem indo para o norte e um trem indo para o sul entrarem no túnel ao mesmo tempo (os trens não podem recuar). Escreva um programa em Java que previna o deadlock, usando semáforos ou sincronismo Java. Inicialmente, não se preocupe com o starvation (a situação em que os trens indo para o norte impedem que os trens indo para o sul usem o túnel, ou vice-versa), e não se preocupe com os trens deixando de parar e colidindo um com o outro.

**8.20** Modifique sua solução para o Exercício 8.19 de modo que fique livre de starvation.

## 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 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 recurso por Dijkstra [1965a] e foi estendido para múltiplos recursos por Habermann [1969]. Os Exercícios 8.8 e 8.9 são de Holt [1971].

O algoritmo de detecção de deadlock para múltiplas instâncias de um recurso, descrito na Seção 8.6.2, foi apresentado por Coffman e outros [1971].

Bach [1987] descreve quantos dos algoritmos no kernel do UNIX tradicional tratam do deadlock.

O verificador de ordem de lock conhecido como witness é apresentado em Baldwin [2002].