

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

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

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

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

Ao contrário de Ada, Dave e Mike, que se concentraram cada um em um capítulo, Shivakant Mishra, da Universidade do Colorado, em Boulder, atuou como um sistema dis-

tribuído, lendo e comentando muitos capítulos e também fornecendo significativa quantidade de novos exercícios e problemas de programação do início ao fim do livro.

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

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

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

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

Andrew S. Tanenbaum

# Capítulo

# 1

## Introdução

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

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

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

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

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

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

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

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

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

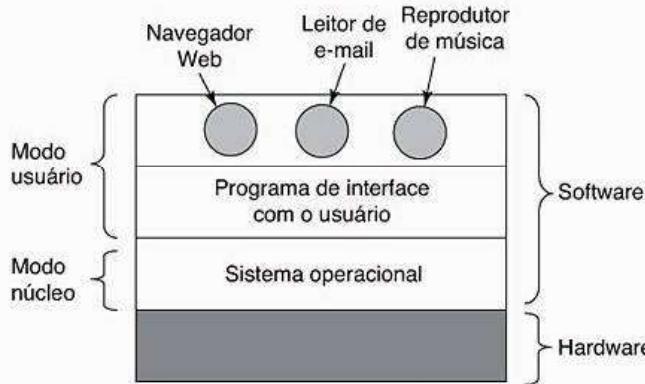


Figura 1.1 Onde o sistema operacional se encaixa.

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

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

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

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

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

## 1.1 O que é um sistema operacional?

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

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

### 1.1.1 O sistema operacional como uma máquina estendida

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

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

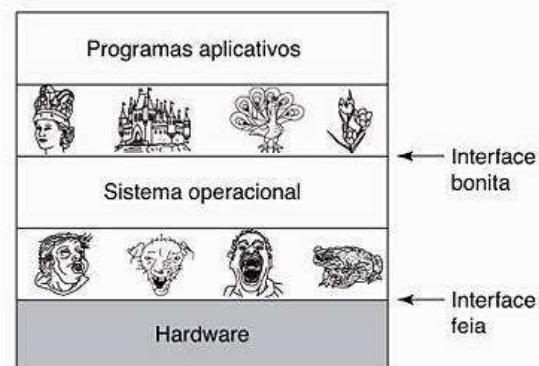
Sem entrar em detalhes *de fato*, é claro que um programador de nível médio provavelmente não se envolverá profundamente com os detalhes de programação das unidades de discos flexíveis (ou discos rígidos, que são piores). Em vez disso, o programador busca lidar com essas uni-

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

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

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

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



**Figura 1.2** Sistemas operacionais transformam hardware feio em abstrações bonitas.

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

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

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

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

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

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

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

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

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

## 1.2 História dos sistemas operacionais

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

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

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

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

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

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

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

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

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

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

Essas máquinas — então denominadas **computadores de grande porte** (*mainframes*) — ficavam isoladas em salas especiais com ar-condicionado, operadas por equipes profissionais. Somente grandes corporações, agências governamentais ou universidades podiam pagar vários milhões de dólares para tê-las. Para uma **tarefa** (isto é, um programa ou um conjunto de programas) ser executada, o

programador primeiro escrevia o programa no papel (em Fortran ou em linguagem assembly) e depois o perfurava em cartões. Ele então levava o maço de cartões para a sala de entradas, entregava-o a um dos operadores e ia tomar um café até que a saída impressa estivesse pronta.

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

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

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

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



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

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

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

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

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

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

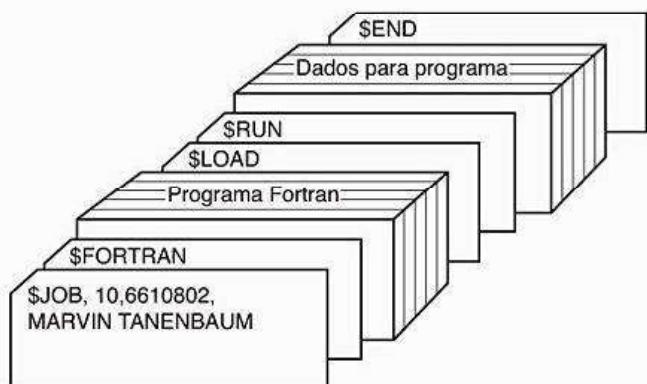


Figura 1.4 Estrutura de uma tarefa típica FMS.

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

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

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

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

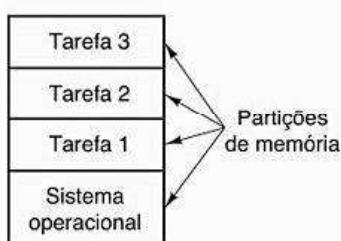
Não havia como a IBM (ou qualquer outro fabricante) elaborar um software que resolvesse todos esses requisitos conflitantes. O resultado foi um sistema operacional enorme e extraordinariamente complexo, provavelmente duas ou três vezes maior que o FMS. Eram milhões de linhas

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

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

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

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



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

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

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

O anseio por respostas mais rápidas abriu caminho para o tempo compartilhado ou **timesharing**, uma variante da multiprogramação na qual cada usuário se conectava por meio de um terminal on-line. Em um sistema de tempo compartilhado, se 20 usuários estivessem conectados e 17 deles estivessem pensando, falando ou tomando café, a CPU podia ser ciclicamente alocada a cada uma das três tarefas que estivessem requisitando a CPU. Como, ao depurar programas, emitem-se normalmente comandos curtos (por exemplo, compile uma rotina<sup>1</sup> de cinco páginas) em vez de comandos longos (por exemplo, ordene um arquivo de um milhão de registros), o computador era capaz de fornecer um serviço rápido e interativo a vários usuários e ainda processar grandes lotes de tarefas em background (segundo plano) nos instantes em que a CPU estivesse ociosa. O primeiro sistema importante de tempo compartilhado, o **CTSS** (*compatible time sharing system* — sistema compatível de tempo compartilhado), foi desenvolvido no MIT em um 7094 modificado (Corbató et al., 1962). Contudo, o tempo compartilhado só se popularizou durante a terceira geração, período em que a necessária proteção em hardware foi largamente empregada.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

#### 1.2.4] A quarta geração (1980–presente) – computadores pessoais

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

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

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

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

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

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

O CP/M, o MS-DOS e outros sistemas operacionais dos primeiros microcomputadores eram todos baseados na di-

gitação de comandos em um teclado, feita pelo usuário. Isso finalmente mudou graças a um trabalho de pesquisa de Doug Engelbart no Stanford Research Institute nos anos 1960. Engelbart inventou uma interface gráfica completa — voltada para o usuário — com janelas, ícones, menus e mouse, denominada **GUI** (*graphical user interface*). Essas ideias de Engelbart foram adotadas por pesquisadores do Xerox Parc e incorporadas às máquinas que eles projetaram.

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

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

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

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

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

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

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

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

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

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

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

Os verdadeiros sistemas operacionais distribuídos requerem muito mais do que apenas adicionar algum código a um sistema operacional monoprocessador, pois os sistemas distribuídos e centralizados são muito diferentes em pontos fundamentais. Por exemplo, é comum que sistemas distribuídos permitam que aplicações sejam executadas em

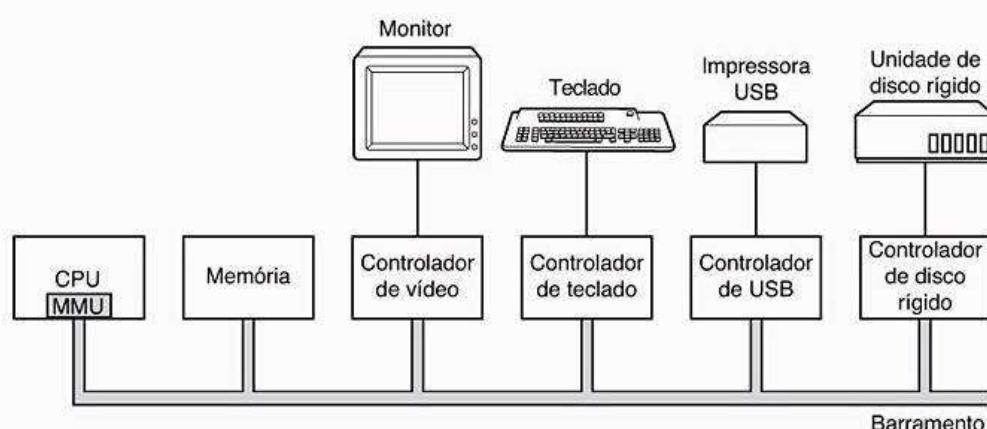
vários processadores ao mesmo tempo, o que exige algoritmos mais complexos de escalonamento de processadores para otimizar o paralelismo.

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

### 1.3 Revisão sobre hardware de computadores

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

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



**Figura 1.6** Alguns dos componentes de um computador pessoal simples.

### 1.3.1] Processadores

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

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

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

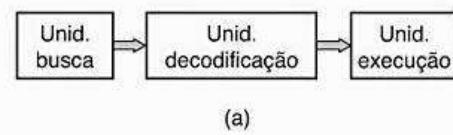
Outro registrador especial é o **ponteiro de pilha**, que aponta para o topo da pilha atual na memória. A pilha contém uma estrutura para cada rotina chamada, mas que ainda não encerrou. Uma estrutura de pilha da rotina contém os parâmetros de entrada, as variáveis locais e as variáveis temporárias que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

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

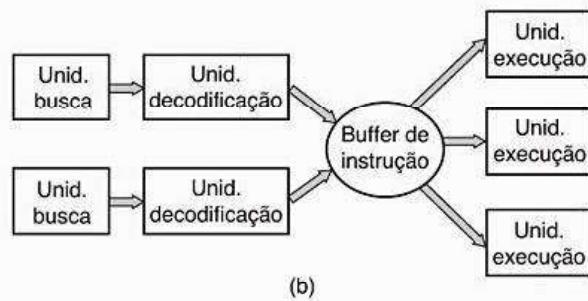
O sistema operacional deve estar ‘ciente’ de todos os registradores. Quando o sistema operacional compartilha o tempo da CPU, ele muitas vezes interrompe a execução de um programa e (re)inicia outro. Toda vez que ele faz isso, o sistema operacional precisa salvar todos os registradores para que eles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

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

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



(a)



(b)

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

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

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

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

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

### Chips multithread e multinúcleo

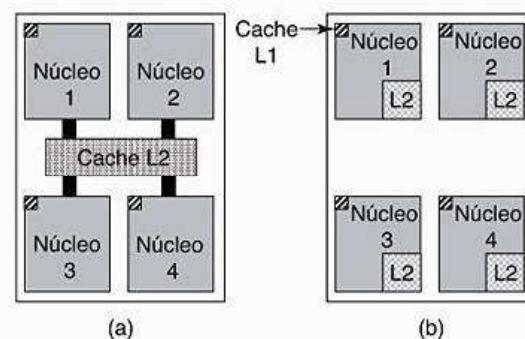
A lei de Moore afirma que o número de transistores de um chip dobra a cada 18 meses. Essa ‘lei’ não é um tipo de lei da física, como a conservação do momento, mas é uma observação do cofundador da Intel, Gordon Moore, sobre a rapidez com que os engenheiros de processo das companhias de semicondutores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

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

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

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

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



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

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

### 1.3.2 | Memória

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

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

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

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

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

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

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

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

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

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

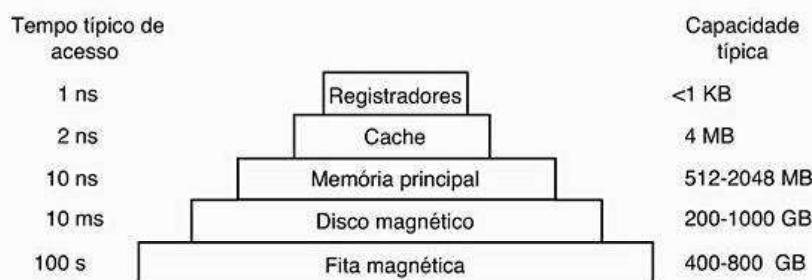


Figura 1.9 Hierarquia de memória típica. Os números são aproximações.

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

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

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

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

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

A **EEPROM** (*electrically erasable ROM* — ROM eletricamente apagável) e a **flash RAM** também são memórias de

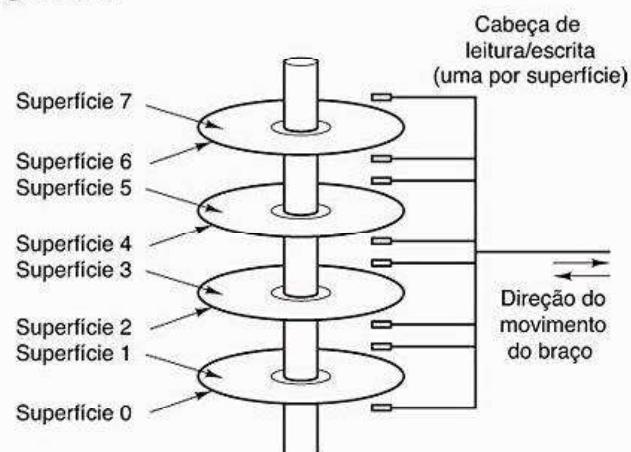
acesso aleatório não voláteis, mas diferentes da ROM, pois podem ser apagadas e reescritas. Contudo, escrever nelas leva várias vezes mais tempo que escrever em uma RAM. Desse modo, são usadas como as ROMs comuns, só que com a característica adicional de possibilitar correção de erros em programas por meio da regravação.

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

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

### 1.3.3 | Discos

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



**Figura 1.10** Estrutura de uma unidade de disco.

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

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

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

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

### 1.3.4 | Fitas

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

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

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

### 1.3.5 | Dispositivos de E/S

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

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

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

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

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

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

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

Em alguns computadores, os registradores dos dispositivos são mapeados no espaço de endereçamento do sistema operacional (os endereços que ele pode usar). Assim, eles podem ser lidos e escritos como se fossem palavras da memória principal. Para esses computadores, nenhuma instrução especial de E/S é necessária e os programas do usuário podem ser mantidos distantes do hardware dei-

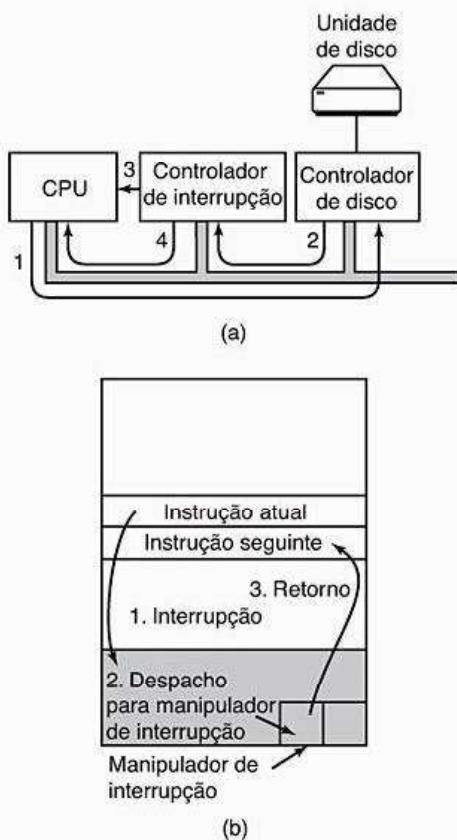
xando esses endereços de memória fora de seu alcance (por exemplo, usando-se registradores-base e limite). Em outros computadores, os registradores de dispositivo são colocados em um espaço especial de portas de E/S, e cada registrador tem seu endereço de porta. Nessas máquinas, instruções especiais IN e OUT são disponíveis em modo núcleo para que se permita que os drivers leiam e escrevam os registradores. O primeiro esquema elimina a necessidade de instruções especiais de E/S, mas consome parte do espaço de endereçamento. O último esquema não gasta o espaço de endereçamento, no entanto requer instruções especiais. Ambos são amplamente usados.

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

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

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

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



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

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

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

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

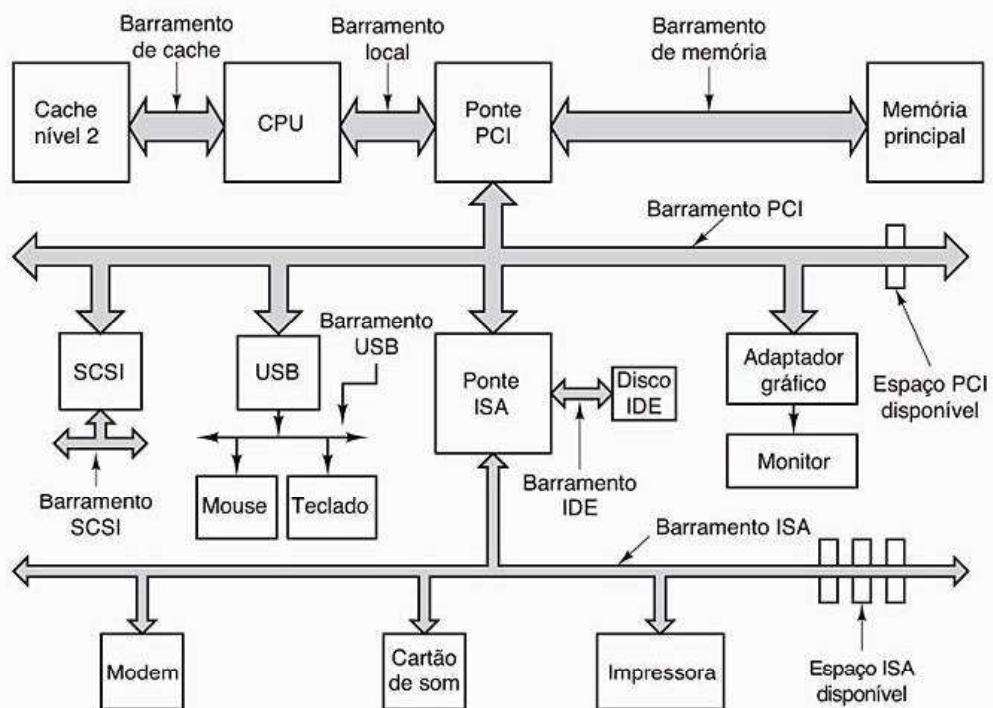
quanto outro manipulador de interrupção estiver em execução. Por isso, a CPU tem uma maneira de desabilitar as interrupções e, então, reabilitá-las depois. Enquanto as interrupções estiverem desabilitadas, todos os dispositivos que terminem suas atividades continuam a emitir sinais de interrupção, mas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

### 1.3.6 | Barramentos

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

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

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



**Figura 1.12** A estrutura de um sistema Pentium grande.

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

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

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

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

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

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

Até aqui, tudo bem. O problema começava quando o usuário trazia uma placa de som e uma placa de modem e ocorria de ambas usarem, digamos, a interrupção 4. Elas conflitariam e não funcionariam juntas. A solução era incluir chaves DIP ou jumpers em todas as placas de E/S e instruir o usuário a configurá-las selecionando um nível de interrupção e endereços de dispositivos de E/S que não conflitassem com quaisquer outros no sistema do usuário.

Adolescentes que devotavam suas vidas às complexidades do hardware do PC podiam fazê-lo, às vezes sem cometer erros. Infelizmente, nem todos tinham esse know-how, o que levava ao caos.

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

### 1.3.7 | Inicializando o computador

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

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

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

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

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

## 1.4 | O zoológico de sistemas operacionais

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

### 1.4.1 | Sistemas operacionais de computadores de grande porte

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

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

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

#### **1.4.2 | Sistemas operacionais de servidores**

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

#### **1.4.3 | Sistemas operacionais de multiprocessadores**

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

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

#### **1.4.4 | Sistemas operacionais de computadores pessoais**

A categoria seguinte é o sistema operacional de computadores pessoais. Os computadores modernos dão suporte a multiprogramação, muitas vezes com dezenas de programas iniciados. Seu trabalho é oferecer uma boa interface para um único usuário. São amplamente usados para processadores de texto, planilhas e acesso à Internet. Exemplos comuns são Linux, FreeBSD, Windows Vista e o

sistema operacional do Macintosh. Sistemas operacionais de computadores pessoais são tão amplamente conhecidos que é provável que precisem, aqui, de pouca introdução. Na verdade, muitas pessoas nem mesmo sabem da existência de outros tipos de sistemas operacionais.

#### **1.4.5 | Sistemas operacionais de computadores portáteis**

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

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

#### **1.4.6 | Sistemas operacionais embarcados**

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

#### **1.4.7 | Sistemas operacionais de nós sensores (*sensor node*)**

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

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

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

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

#### 1.4.8 | Sistemas operacionais de tempo real

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

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

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

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

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

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

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

### 1.5 Conceitos sobre sistemas operacionais

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

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

### 1.5.1 | Processos

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

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

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

Assim, um processo (suspenso) é constituído de seu espaço de endereçamento, normalmente chamado de **íma-**

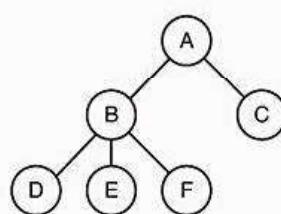
**gem do núcleo** (em homenagem às memórias de núcleo magnético usadas antigamente), e de sua entrada na tabela de processos, a qual armazena o conteúdo de seus registradores, entre outros itens necessários para reiniciar o processo mais tarde.

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

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

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

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



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

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

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

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

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

## 1.5.2 | Espaços de endereçamento

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

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

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

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

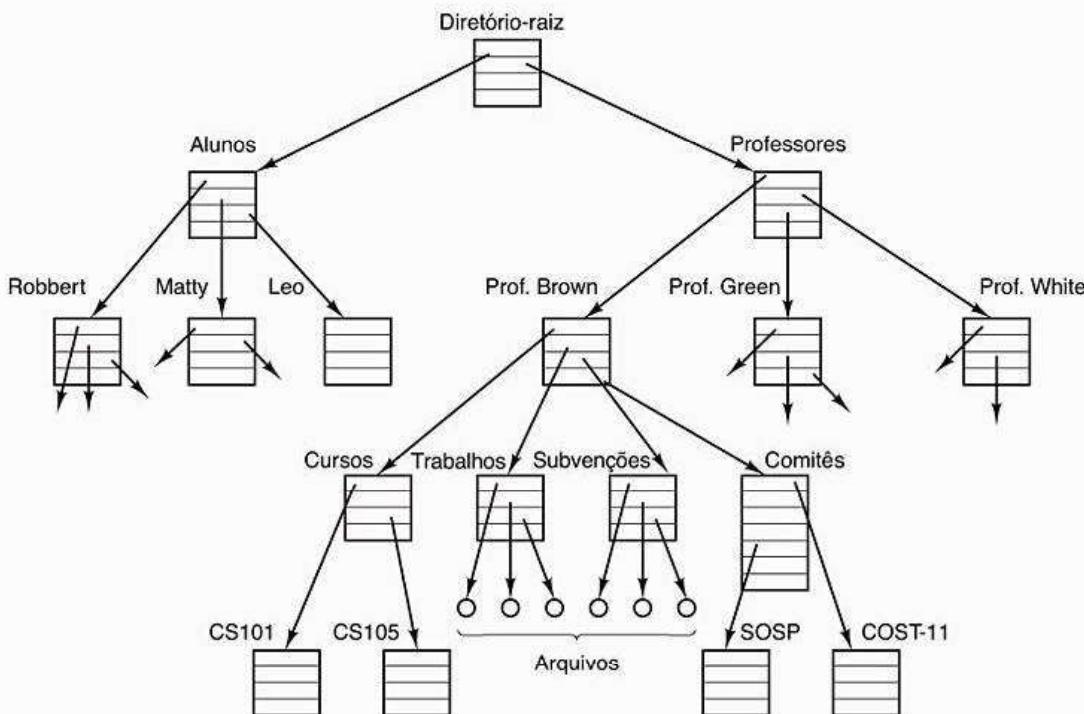
Contudo, em muitos computadores os endereços são de 32 ou 64 bits, dando um espaço de endereçamento de  $2^{32}$  ou  $2^{64}$  bytes, respectivamente. O que acontece se um processo tiver maior espaço de endereçamento que a memória principal do computador e o processo quiser utilizá-lo completamente? Nos primeiros computadores, esse processo não tinha sorte. Atualmente, existe uma técnica chamada memória virtual, como mencionado anteriormente, na qual o sistema operacional mantém parte do espaço de endereçamento na memória principal e parte no disco, trocando os pedaços entre eles conforme a necessidade. Em essência, o sistema operacional cria a abstração de um espaço de endereçamento como o conjunto de endereços ao qual um processo pode se referir. O processo de endereçamento é desacoplado da memória física da máquina e pode ser maior ou menor que a memória física. O gerenciamento de espaços de endereçamento e da memória física forma uma parte importante das atividades do sistema operacional; por isso, o Capítulo 3 é dedicado a esse tópico.

## 1.5.3 | Arquivos

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

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

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



**Figura 1.14** Sistema de arquivos para um departamento universitário.

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

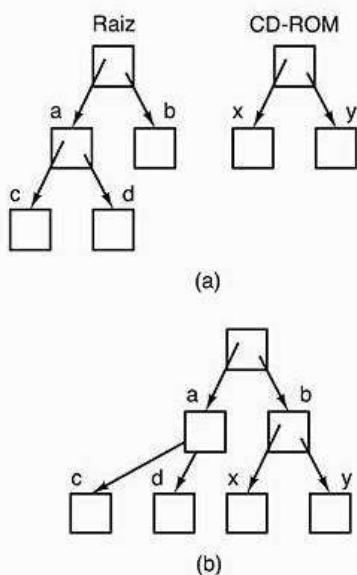
Cada arquivo dentro da hierarquia de diretórios pode ser especificado fornecendo-se o **caminho** (*path name*) a partir do topo da hierarquia de diretórios, o **diretório-raiz**. Esses caminhos absolutos formam uma lista de diretórios que deve ser percorrida a partir do diretório-raiz para chegar até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

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

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

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

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

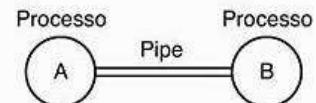


**Figura 1.15** (a) Antes da montagem, os arquivos no CD-ROM não estão acessíveis. (b) Após a montagem, tornam-se parte da hierarquia de arquivos.

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

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

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



**Figura 1.16** Dois processos conectados por um pipe.

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

#### 1.5.4 | Entrada e saída

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

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

#### 1.5.5 | Segurança

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

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

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

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

### **1.5.6 | O interpretador de comandos (shell)**

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

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

```
date
```

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

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

```
date >arq
```

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

```
sort <arq1>arq2
```

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

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

```
cat arq1 arq2 arq3 | sort >/dev/lp
```

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

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

```
cat arq1 arq2 arq3 | sort >/dev/lp &
```

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

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

### **1.5.7 | Ontogenia recapitula a filogenia**

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

Algo ligeiramente análogo tem acontecido na indústria da computação. Cada nova espécie (computador de grande porte, minicomputador, computador pessoal, computador embarcado, cartões inteligentes etc.) parece passar pelo mesmo desenvolvimento de seus ancestrais, tanto no que se refere ao hardware como ao software. Esquecemos-nos frequentemente de que muito do que acontece na indústria da computação e em muitos outros campos é orientado pela tecnologia. A razão pela qual os romanos não tinham

carros não é porque eles gostavam muito de caminhar. É porque eles não sabiam como construí-los. Computadores pessoais *não* existem porque milhões de pessoas têm um desejo contido por muitos séculos de ter um computador, mas porque agora é possível fabricá-los de modo mais barato. Muitas vezes nos esquecemos de como a tecnologia afeta nossa visão dos sistemas e de que convém refletir sobre o assunto de vez em quando.

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

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

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

### Memórias grandes

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

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

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

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

### Hardware de proteção

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

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

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

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

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

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

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

## Discos

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

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

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

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

## Memória virtual

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

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

## 1.6 Chamadas de sistema (system calls)

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

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

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

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

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

```
contador = read(arq, buffer, nbytes);
```

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

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

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

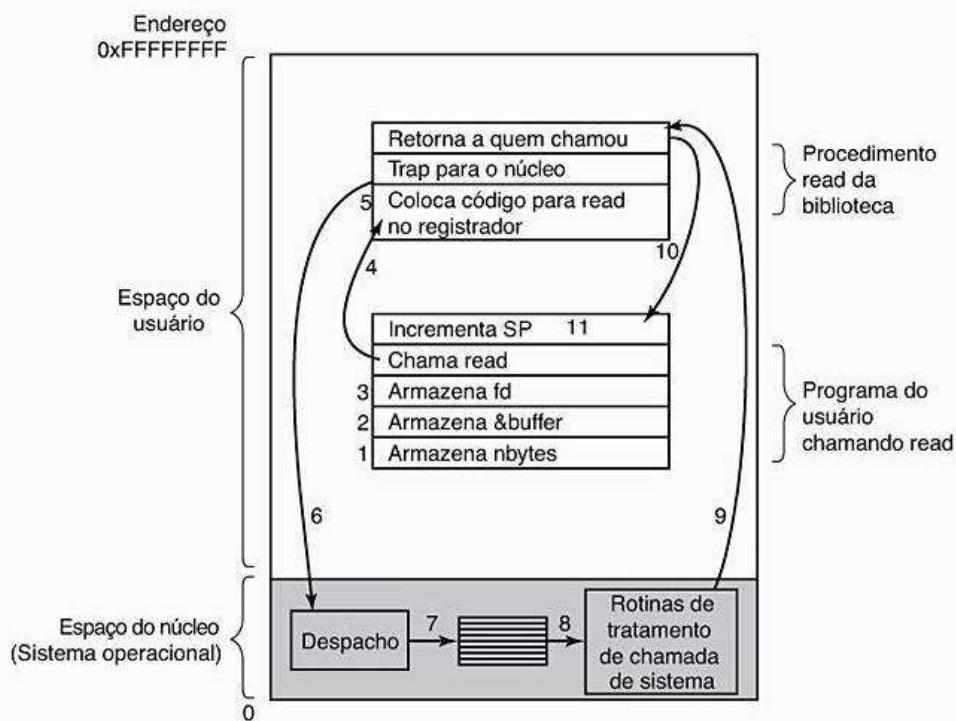
As chamadas de sistema são realizadas em uma série de passos. Para melhor esclarecer esse conceito, examinemos a chamada *read* discutida anteriormente. Antes da chamada da rotina *read* de biblioteca, que é na verdade quem faz a chamada de sistema *read*, o programa que chama *read*, antes de tudo,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

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

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

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

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



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

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

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

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

Os serviços oferecidos por essas chamadas determinam a maior parte do que o sistema operacional deve realizar, já que o gerenciamento de recursos em computadores pessoais é mínimo (pelo menos, se comparado a grandes máquinas com vários usuários). Os serviços englobam coisas

como criar e finalizar processos, criar, excluir, ler e escrever arquivos, gerenciar diretórios e realizar entrada e saída.

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

### 1.6.1] Chamadas de sistema para gerenciamento de processos

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

## Gerenciamento de processos

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

## Gerenciamento de arquivos

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

## Gerenciamento do sistema de diretório e arquivo

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

## Diversas

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

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

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

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

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

```
#define TRUE 1

while (TRUE) {
    type _prompt();
    read _command(command, parameters);

    if (fork() != 0) {
        /* Código do processo pai. */
        waitpid(-1, &status, 0);
    } else {
        /* Código do processo filho. */
        execve(command, parameters, 0);
    }
}

/*
 * repita para sempre
 * mostra prompt na tela
 * lê entrada do terminal
 */

/*
 * cria processo filho
 */

/*
 * aguarda o processo filho acabar
 */

/*
 * executa o comando
 */
```

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

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

Consideremos o caso de um comando como

`cp arq1 arq2`

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

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

`main(argc, argv, envp)`

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

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

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

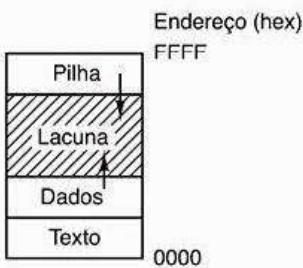
Se a chamada exec parecer-lhe complicada, não se desespere: ela é (semanticamente) a mais complexa de todas as chamadas de sistema em POSIX. Todas as outras são muito mais simples. Como um exemplo das simples, considere a chamada exit, que os processos devem usar para terminar sua execução. Ela possui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

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

### 1.6.2 | Chamadas de sistema para gerenciamento de arquivos

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

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



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

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

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

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

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

### 1.6.3 | Chamadas de sistema para gerenciamento de diretórios

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

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

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

The diagram illustrates two separate file systems at different times:

**(a)** Two boxes represent file systems. The left box is labeled `/usr/ast` and contains three entries: i-node 16 (correio), i-node 81 (jogos), and i-node 40 (teste). The right box is labeled `/usr/jim` and contains four entries: i-node 31 (bin), i-node 70 (memo), i-node 59 (f.c.), and i-node 38 (prog1).

**(b)** The same two boxes are shown again. In the `/usr/ast` box, the entry for i-node 40 (teste) has been removed. In the `/usr/jim` box, the entry for i-node 70 (nota) has been added.

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

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

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

Conforme mencionado anteriormente, a chamada de sistema `mount` permite que dois sistemas de arquivo sejam unificados. Uma situação comum é ter em disco rígido o

sistema de arquivos-raiz contendo as versões binárias (executáveis) dos comandos comuns e outros arquivos intensivamente usados. O usuário pode, então, inserir um disquete com arquivos a serem lidos na unidade de CD-ROM.

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

```
mount("/dev/fd0", "/mnt", 0);
```

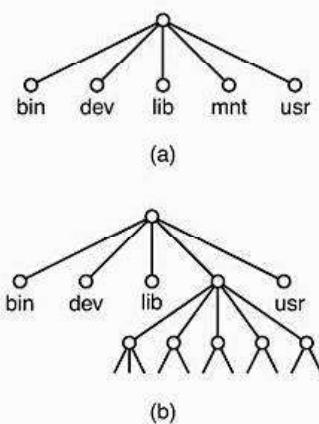
no qual o primeiro parâmetro é o nome de um arquivo especial de blocos para a unidade de disco 0, o segundo parâmetro é o local na árvore onde ele será montado e o terceiro parâmetro informa se o sistema de arquivos deve ser montado como leitura e escrita ou apenas para leitura.

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

#### 1.6.4 | Outras chamadas de sistema

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

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



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

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

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

```
chmod("arq", 0644);
```

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

O POSIX define várias rotinas para lidar com o tempo. Por exemplo, a chamada *time* retorna o tempo atual em segundos, com 0 correspondendo à meia-noite do dia 1º de janeiro de 1970 (como se nesse instante o dia estivesse começando, e não terminando). Em computadores com palavras de 32 bits, o valor máximo que a chamada *time* pode retornar é  $2^{32} - 1$  segundos (supondo que esteja usando inteiros sem sinal). Esse valor corresponde a pouco mais de 136 anos. Assim, no ano 2106, os sistemas UNIX de 32 bits irão entrar em pane, como o famoso bug do milênio em 2000, que foi evitado graças aos grandes esforços da indústria de TI para corrigir esse problema. Se você atualmente possui um sistema UNIX de 32 bits, aconselho-o a trocá-lo por um de 64 bits em algum momento antes de 2106.

### 1.6.5 | A API Win32 do Windows

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

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

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

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

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

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

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

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

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

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

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

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

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

## 1.7 Estrutura de sistemas operacionais

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

### 1.7.1 Sistemas monolíticos

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

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

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

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

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

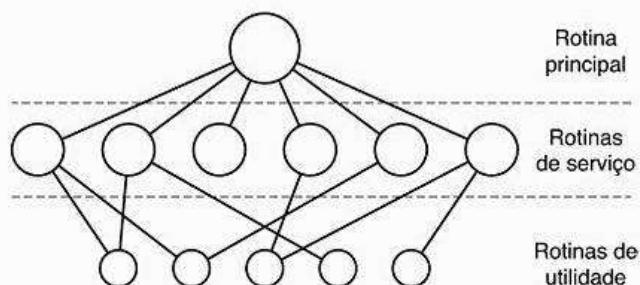
Contudo, mesmo em sistemas monolíticos, é possível ter um mínimo de estrutura. Os serviços (chamadas de sistema) providos pelo sistema operacional são requisitados colocando-se os parâmetros em um local bem definido (na pilha, por exemplo) e, então, executando uma instrução de desvio de controle (*trap*).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

Essa organização sugere uma estrutura básica para o sistema operacional:

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

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

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



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

### 1.7.2 | Sistemas de camadas

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

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

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

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

**Tabela 1.3** Estrutura do sistema operacional THE.

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

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

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

### 1.7.3 | Micronúcleo

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

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

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

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

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

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

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

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

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

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

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

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

#### 1.7.4 | O modelo cliente-servidor

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

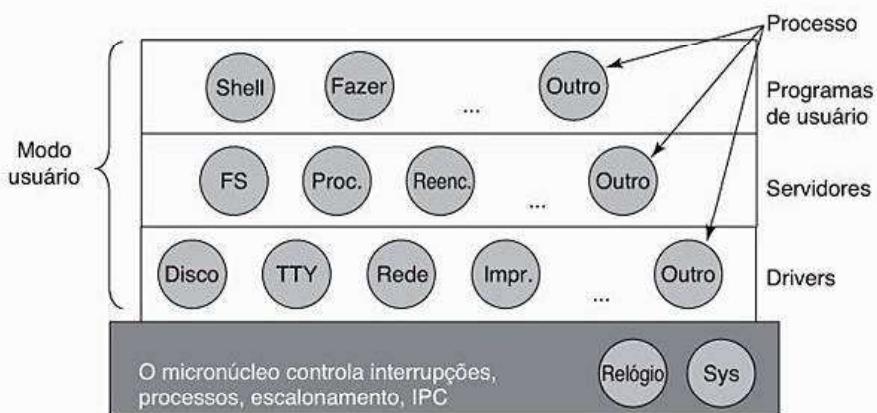


Figura 1.23 Estrutura do sistema MINIX 3.

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

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

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

### 1.7.5 | Máquinas virtuais

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

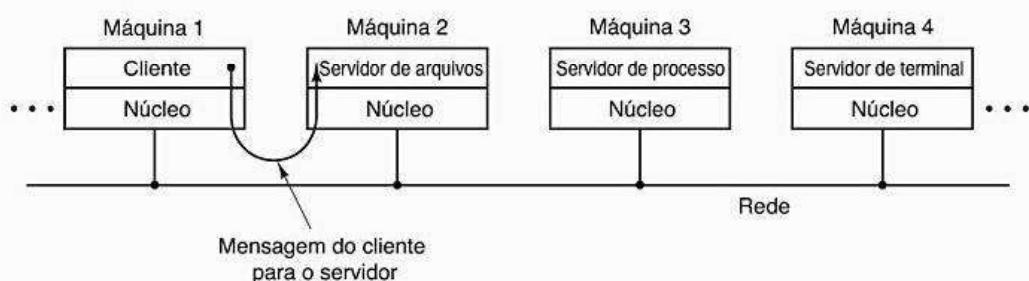
diferente, que a IBM finalmente adotou. Um descendente linear desse sistema, chamado **z/VM**, agora é amplamente usado nos computadores de grande porte atuais da IBM, a série **z**, que são muito utilizados em grandes centros de dados corporativos, por exemplo, como servidores de comércio eletrônico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

### VM/370

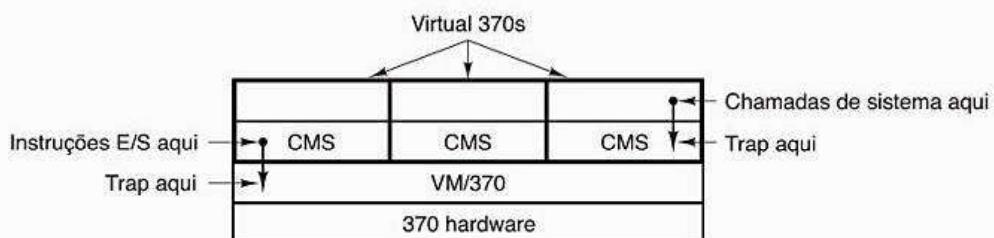
Esse sistema, originalmente denominado CP/CMS e depois renomeado VM/370 (Seawright e MacKinnon, 1979), foi baseado em uma observação perspicaz: um sistema de tempo compartilhado fornece (1) multiprogramação e (2) uma máquina estendida com uma interface mais conveniente do que a que o hardware oferece.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

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

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



**Figura 1.24** O modelo cliente-servidor em uma rede.



**Figura 1.25** Estrutura do VM/370 com CMS.

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

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

### Máquinas virtuais redescobertas

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

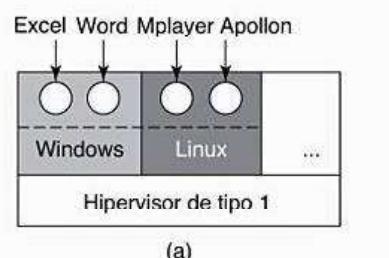
Primeiro as necessidades. Muitas empresas tradicionalmente executavam seus servidores de correio, servidores da Web, servidores FTP e outros em computadores separados, algumas vezes com sistemas operacionais diferentes. Elas perceberam a virtualização como um modo de executar todos eles na mesma máquina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

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

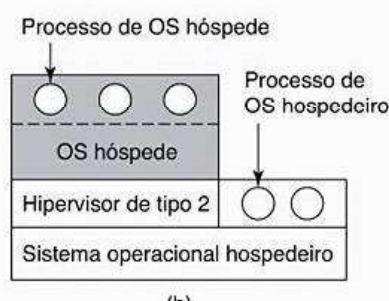
lhes permite controlar o software do servidor) e hospedagem dedicada (que lhes oferece uma máquina própria, que é muito flexível mas pouco econômica para sites da Web de pequeno a médio porte). Quando uma empresa de hospedagem na Web aluga máquinas virtuais, uma única máquina física pode executar muitas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

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

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



(a)



(b)

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

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

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

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

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

### A máquina virtual Java

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

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

### 1.7.6 | Exonúcleo

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

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

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

## 1.8 O mundo de acordo com a linguagem C

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

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

### 1.8.1] A linguagem C

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

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

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

que declaram que *c1* e *c2* são variáveis do tipo caractere e que *p* é uma variável que aponta para um caractere (isto é, contém o endereço dele). A primeira atribuição armazena o código ASCII para o caractere 'c' na variável *c1*. A segunda atribui o endereço de *c1* à variável ponteiro *p*. A terceira atribui os conteúdos da variável apontada por *p* à variável *c2*; desse modo, depois que esses comandos são executados, *c2* também contém o código ASCII para 'c'. Na teoria, os ponteiros são tipificados; assim, programadores não deveriam atribuir o endereço de um número em ponto flutuante a um ponteiro de caractere, mas, na prática, os compiladores aceitam essas atribuições, embora algumas vezes com uma advertência. Os ponteiros são uma construção muito eficaz, mas também uma grande fonte de erros quando usados sem cuidado.

C não tem, entre outras coisas, cadeia de caracteres incorporadas, threads, pacotes, classes, objetos, segurança de tipos (*type safety*) e coletor de lixo. O último é um defeito fatal para sistemas operacionais. Todo o armazenamento em C é estático ou explicitamente alocado e liberado pelo programador, normalmente com a função biblioteca *malloc* e *free*. É a última propriedade — controle total do programador sobre a memória — com ponteiros explícitos que torna a linguagem C atraente para a escrita de sistemas operacionais. Os sistemas operacionais são, até certo ponto, sistemas em tempo real, mesmo no caso de sistemas de propósito geral. Quando ocorre uma interrupção, o sistema operacional pode ter apenas alguns microssegundos para executar alguma ação ou perder informações críticas. A entrada em operação do coletor de lixo em um momento arbitrário é intolerável.

### 1.8.2] Arquivos de cabeçalho

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

```
#define BUFFER_SIZE 4096
```

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

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

que permitem que o programador escreva

```
i = max(j, k+1)
```

e obtenha

```
i = (j > k+1 ? j : k+1)
```

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

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

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

### 1.8.3] Grandes projetos de programação

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

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

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

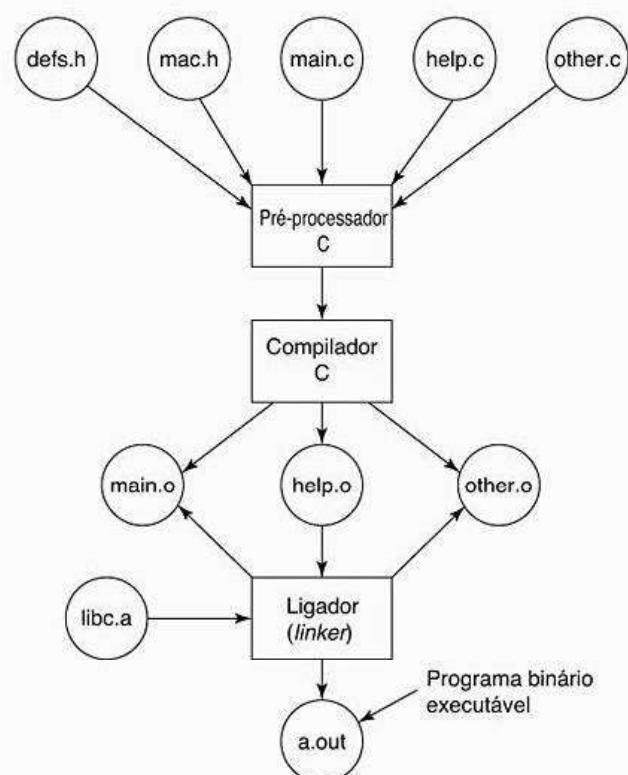
Uma vez que os sistemas operacionais são muito grandes (cinco milhões de linhas de código não são incomuns), compilar tudo novamente a cada vez que um arquivo fosse alterado seria inaceitável. Por outro lado, alterar um arquivo de cabeçalho importante que esteja incluído em milhares de outros arquivos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

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

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

#### 1.8.4] O modelo de execução

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



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

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

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

### 1.9 Pesquisas em sistemas operacionais

A ciência da computação avança muito rapidamente e é difícil dizer para onde se dirige. Pesquisadores em universidades e em laboratórios industriais estão sempre desenvolvendo novas ideias; algumas delas não levam a nada, porém outras tornam-se a base para futuros produtos

e causam altos impactos à indústria e aos consumidores. Fazer uma retrospectiva de como as coisas evoluíram é mais fácil do que predizer como evoluirão. Separar o joio do trigo é muito difícil, porque muitas vezes uma ideia leva de 20 a 30 anos para causar algum impacto.

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

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

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

ACM	<a href="http://www.acm.org">http://www.acm.org</a>
IEEE Computer Society	<a href="http://www.computer.org">http://www.computer.org</a>
USENIX	<a href="http://www.usenix.org">http://www.usenix.org</a>

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

## 1.10 Delineamento do restante deste livro

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

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

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

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

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

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

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

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

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

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

## 1.11 Unidades métricas

Para evitar qualquer confusão, é melhor dizer claramente que, neste livro, como na ciência da computação em geral, são usadas unidades métricas, e não as tradicionais unidades inglesas (sistema *furlong-stone-fortnight*). Os principais prefixos métricos são relacionados na Tabela 1.4. Normalmente esses prefixos são abreviados por suas primeiras letras, com as unidades maiores que 1 em letras maiúsculas. Assim, um banco de dados de 1 TB ocupa  $10^{12}$  bytes de memória e um tique de relógio de 100 pseg (ou 100 ps) ocorre a cada  $10^{-10}$  segundos. Como ambos os prefixos, mili e micro, começam com a letra 'm', foi necessário fazer uma escolha. Normalmente, 'm' é para mili e 'μ' (a letra grega *mu*) é para micro.

Convém também observar que, para medir tamanhos de memória, as unidades têm significados um pouco diferentes. O quilo corresponde a  $2^{10}$  (1.024), e não a  $10^3$  (1.000), pois as memórias são sempre expressas em potências de 2. Assim, uma memória de 1 KB contém 1.024 bytes, e não 1.000 bytes. De maneira similar, uma memória de 1 MB contém  $2^{20}$  (1.048.576) bytes, e uma memória de 1 GB contém  $2^{30}$  (1.073.741.824) bytes. Contudo, uma linha de comunicação de 1 Kbps transmite a 1.000 bits por segundo, e uma rede local (LAN) de 10 Mbps transmite a 10.000.000 bits por segundo, pois essas velocidades não são potências de 2. Infelizmente, muitas pessoas tendem a misturar esses dois sistemas, especialmente em tamanhos de disco. Para evitar ambiguidade, neste livro usaremos os símbolos KB, MB e GB para  $2^{10}$ ,  $2^{20}$  e  $2^{30}$  bytes, respectivamente, e os símbolos Kbps, Mbps e Gbps para  $10^3$ ,  $10^6$ ,  $10^9$  bits/segundo, respectivamente.

## 1.12 Resumo

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

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

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

<b>Exp.</b>	<b>Explicito</b>	<b>Prefixo</b>	<b>Exp.</b>	<b>Explicito</b>	<b>Prefixo</b>
$10^{-3}$	0,001	mili	$10^3$	1.000	quilo
$10^{-6}$	0,000001	micro	$10^6$	1.000.000	mega
$10^{-9}$	0,000000001	nano	$10^9$	1.000.000.000	giga
$10^{-12}$	0,000000000001	pico	$10^{12}$	1.000.000.000.000	tera
$10^{-15}$	0,000000000000001	femto	$10^{15}$	1.000.000.000.000.000	peta
$10^{-18}$	0,000000000000000001	atto	$10^{18}$	1.000.000.000.000.000.000	exa
$10^{-21}$	0,000000000000000000001	zepto	$10^{21}$	1.000.000.000.000.000.000.000	zetta
$10^{-24}$	0,000000000000000000000000001	yocto	$10^{24}$	1.000.000.000.000.000.000.000.000	yotta

**Tabela 1.4** Os principais prefixos métricos.

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

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

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

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

## Problemas

1. O que é multiprogramação?
2. O que é a técnica de spooling? Você acha que computadores pessoais avançados terão o spooling como uma característica-padrão no futuro?
3. Nos primeiros computadores, todo byte de dados lido ou escrito era tratado pela CPU (isto é, não havia DMA). Quais as implicações disso para a multiprogramação?
4. A ideia de família de computadores foi introduzida nos anos 1960 com os computadores de grande porte IBM System/360. Essa ideia está morta e sepultada ou ainda vive?
5. Uma razão para a demora da adoção das interfaces gráficas GUI era o custo do hardware necessário para dar suporte a elas. De quanta RAM de vídeo se precisa para dar suporte a uma tela de texto monocromática com 25 linhas × 80 colunas de caracteres? Quanto é necessário para dar suporte a um mapa de bits com  $1.024 \times 768$  pixels de 24 bits? Qual é o custo dessa RAM em preços de 1980 (5 dólares/KB)? Quanto custa agora?
6. Há várias metas de projeto na construção de um sistema operacional; por exemplo, utilização de recursos, oportunidade, robustez etc. Dê um exemplo de duas metas de projeto que possam ser contraditórias.
7. Das instruções a seguir, quais só podem ser executadas em modo núcleo?
  - (a) Desabilite todas as interrupções.
  - (b) Leia o horário do relógio.
  - (c) Altere o horário do relógio.
  - (d) Altere o mapa de memória.
8. Considere um sistema que tem duas CPUs e cada CPU tem dois threads (hyperthreading). Suponha que três programas,  $P_0$ ,  $P_1$  e  $P_2$ , sejam iniciados com tempos de execução de 5, 10 e 20 ms, respectivamente. Quanto tempo seria necessário para concluir a execução desses programas? Suponha que todos os três programas sejam 100% CPU bound (limitados pela CPU, ou seja, que não fazem E/S), não bloqueiem durante a execução e não mudem de CPUs uma vez realizada a atribuição.
9. Um computador tem um pipeline de quatro estágios. Cada estágio leva o mesmo tempo para fazer seu trabalho — digamos, 1 ns. Quantas instruções por segundo essa máquina pode executar?
10. Considere um sistema de computador que tem memória cache, memória principal (RAM) e disco. O sistema operacional usa memória virtual. São necessários 2 ns para acessar uma palavra a partir da cache, 10 ns para acessar uma palavra a partir da RAM e 10 ms para acessar uma palavra a partir do disco. Se a taxa de acerto da cache é de 95% e a da memória principal (após uma falta de cache) é de 99%, qual é o tempo médio de acesso a uma palavra?
11. Um revisor alerta sobre um erro de ortografia no original de um livro-texto sobre sistemas operacionais que está para ser impresso. O livro tem aproximadamente 700 páginas, cada uma com 50 linhas de 80 caracteres. Quanto tempo será preciso para percorrer eletronicamente o texto no caso de a cópia estar em cada um dos níveis de memória da Figura 1.9? Para métodos de armazenamento interno, considere que o tempo de acesso é dado por caractere; para discos, considere que o tempo é por bloco de 1.024 caracteres; e para fitas, que o tempo dado é a partir do início dos dados com acesso subsequente na mesma velocidade que o acesso a disco.
12. Quando um programa de usuário faz uma chamada de sistema para ler ou escrever um arquivo em disco, ele fornece uma indicação de qual arquivo ele quer, um ponteiro para o buffer de dados e um contador. O controle é, então, transferido ao sistema operacional, que chama o driver apropriado. Suponha que o driver inicie o disco, termine e só volte quando uma interrupção ocorrer. No caso da leitura do disco, obviamente quem chama deverá ser bloqueado (pois não há dados para ele). E no caso da escrita no disco? Quem chama precisa ser bloqueado aguardando o final da transferência do disco?
13. O que é uma instrução trap? Explique seu uso em sistemas operacionais.
14. Qual é a diferença fundamental entre um trap e uma interrupção?
15. Por que é necessária uma tabela de processos em sistemas de compartilhamento de tempo? Essa tabela também é essencial em sistemas de computador pessoal (PC), nos quais existe apenas um processo, que detém o comando de toda a máquina até que ele termine?
16. Há alguma razão para se querer montar um sistema de arquivos em um diretório não vazio? Se há, qual é?

17. Qual é a finalidade de uma chamada de sistema em um sistema operacional?
18. Para cada uma das seguintes chamadas de sistema, dê uma condição que faça com que elas falhem: fork, exec e unlink.
19. Considere count = write(fd, buffer, nbytes); essa chamada pode retornar algum valor em count que seja diferente de nbytes? Em caso afirmativo, por quê?
20. Um arquivo cujo descritor é fd contém a seguinte sequência de bytes: 3, 1, 4, 1, 5, 9, 2, 6, 5, 3, 5. São executadas as seguintes chamadas de sistema: lseek(fd, 3, SEEK\_SET); read(fd, &buffer, 4); onde a chamada lseek faz uma busca ao byte 3 do arquivo. O que o buffer contém ao final da leitura?
21. Imagine que um arquivo de 10 MB esteja armazenado em um disco na mesma trilha (trilha #:50) em setores consecutivos. O braço do disco está situado sobre a trilha número 100. Quanto tempo é necessário para recuperar esse arquivo a partir do disco? Suponha que a transferência do braço de um cilindro a outro leve cerca de 1 ms e cerca de 5 ms para que o setor onde o início do arquivo está armazenado faça a rotação sob a cabeça. Além disso, suponha que a leitura ocorra a uma taxa de 100 MB/s.
22. Qual é a diferença essencial entre um arquivo especial de blocos e um arquivo especial de caracteres?
23. No exemplo dado na Figura 1.17, a rotina de biblioteca é denominada *read* e a própria chamada de sistema é denominada *read*. É essencial que ambas tenham o mesmo nome? Em caso negativo, qual é a mais importante?
24. O modelo cliente-servidor é muito usado em sistemas distribuídos. Ele pode ser também utilizado em um sistema de um único computador?
25. Para um programador, uma chamada de sistema se parece com qualquer outra chamada a uma rotina de biblioteca. É importante que um programador saiba quais rotinas de biblioteca resultam em chamadas de sistema? Sob quais circunstâncias e por quê?
26. A Figura 1.23 mostra que várias chamadas de sistema em UNIX não têm equivalentes na API do Win32. Para cada chamada relacionada que não tenha equivalente no Win32, quais são as consequências para o programador de converter um programa UNIX para executar no Windows?
27. Um sistema operacional portátil é aquele que tem portabilidade de uma arquitetura de sistema a outra sem sofrer nenhuma modificação. Explique por que é inviável construir um sistema operacional que seja completamente portátil. Descreva duas camadas de alto nível obtidas ao projetar um sistema operacional que seja altamente portátil.
28. Explique como a separação entre política e mecanismo pode ajudar na construção de sistemas operacionais baseados em micronúcleos.
29. Eis algumas questões para praticar conversão de unidades:
  - (a) Quanto dura um microano em segundos?
  - (b) Micrômetros muitas vezes são chamados de mícrons. Qual o tamanho de um gigamícron?
  - (c) Quantos bytes há em 1 TB de memória?
  - (d) A massa da Terra é de seis mil yottagramas. Qual é esse peso em quilogramas?
30. Escreva um shell que seja similar ao da Figura 1.18, mas que contenha código suficiente e realmente funcione para que seja possível testá-lo. Você também pode adicionar alguns aspectos, como redirecionamento de entrada e saída, pipes e tarefas em background.
31. Se você tem disponível um sistema pessoal do tipo UNIX (Linux, MINIX, FreeBSD etc.), em que se possa provocar uma falha e reiniciar seguramente, então escreva um script do shell que tente criar um número ilimitado de processos filhos e observe o que acontece. Antes de executar o experimento, digite sync para que o shell descarregue os buffers do sistema de arquivos no disco para evitar danos ao sistema de arquivos. **Observação:** não tente fazer isso em sistemas compartilhados sem antes obter a permissão do administrador do sistema. As consequências serão instantaneamente óbvias, você será pego e poderão sobrevir punições.
32. Examine e tente interpretar o conteúdo de um diretório do tipo UNIX ou Windows com uma ferramenta como o programa *od* do UNIX ou o programa *Debug* do MS-DOS. *Dica:* o modo como você faz isso depende do que o SO permite. Um truque que pode funcionar é criar um diretório em um disco flexível com um sistema operacional e, então, ler os dados do disco usando um sistema operacional diferente que permita esse acesso.

# Capítulo 2

## Processos e threads

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

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

### 2.1 Processos

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

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

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

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

#### 2.1.1 O modelo de processo

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

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

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

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

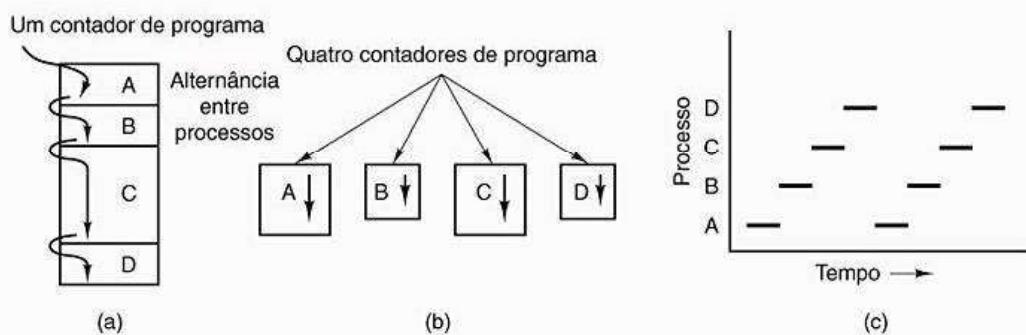
de tempo prefixado de milissegundos —, é preciso tomar medidas especiais para que esses eventos ocorram. Contudo, em geral a maioria dos processos não é afetada pelo aspecto inerente de multiprogramação da CPU ou pelas velocidades relativas dos diversos processos.

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

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

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

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



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

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

### **2.1.2 | Criação de processos**

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

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

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

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

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

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

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

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

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

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

Por outro lado, no Windows, uma única chamada de função do Win32, *CreateProcess*, trata tanto do pro-

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

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

### **2.1.3] Término de processos**

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

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

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

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

```
cc foo.c
```

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

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

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

### **2.1.4] Hierarquias de processos**

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

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

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

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

### 2.1.5 | Estados de processos

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

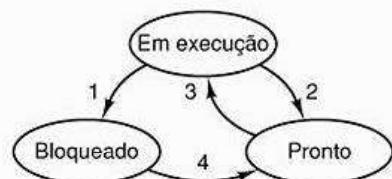
```
cat chapter1 chapter2 chapter3 | grep tree
```

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

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

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

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



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

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

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

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

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

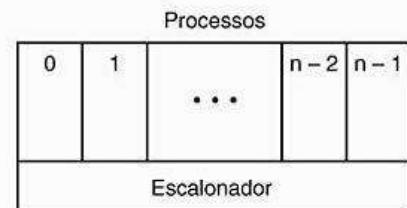
Com o modelo de processo, torna-se muito mais fácil saber o que está ocorrendo dentro do sistema. Alguns dos processos chamam programas que executam comandos digitados por um usuário. Outros processos são parte do sistema e manejam tarefas como fazer requisições por serviços de arquivos ou gerenciar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

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

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

### 2.1.6 | Implementação de processos

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



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

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

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

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

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

**Tabela 2.1** Alguns dos campos de um processo típico de entrada na tabela.

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

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

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

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

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

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

### 2.1.7 | Modelando a multiprogramação

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

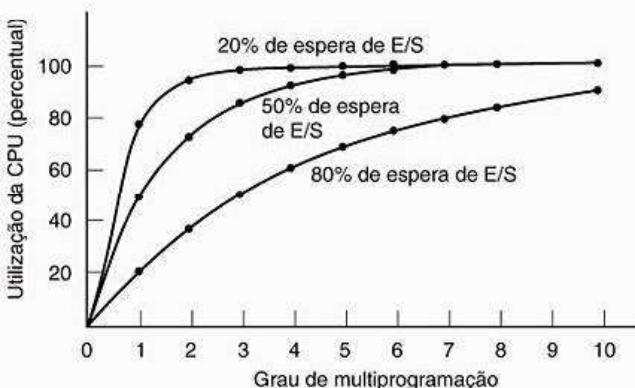
Um modelo melhor é examinar o emprego da CPU do ponto de vista probabilístico. Imagine que um processo passe uma fração  $p$  de seu tempo esperando que os dispositivos de E/S sejam concluídos. Com  $n$  processos na memória simultaneamente, a probabilidade de que todos os  $n$  processos estejam esperando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

$$\text{utilização da CPU} = 1 - p^n$$

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

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

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



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

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

Embora muito simples, o modelo da Figura 2.4 pode, mesmo assim, ser usado para previsões específicas, ainda que aproximadas, de desempenho da CPU. Suponha, por exemplo, que um computador tenha 512 MB de memória, com um sistema operacional que use 128 MB, e que cada programa de usuário também empregue 128 MB. Esses tamanhos possibilitam que três programas de usuário estejam simultaneamente na memória. Considerando-se que, em média, um processo passa 80 por cento de seu tempo em espera por E/S, tem-se uma utilização da CPU (ignorando o gasto extra — overhead — causado pelo sistema operacional) de  $1 - 0,8^3$ , ou cerca de 49 por cento. A adição de mais 512 MB de memória permite que o sistema aumente seu grau de multiprogramação de 3 para 7, elevando assim a utilização da CPU para 79 por cento. Em outras palavras, a adição de 512 MB aumentará a utilização da CPU em 30 por cento.

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

## 2.2 Threads

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

### 2.2.1 O uso de thread

Por que alguém desejaría ter um tipo de processo dentro de um processo? Consta-se que há várias razões para existirem esses miniprocessos, chamados **threads**. Exami-

nemos alguns deles agora. A principal razão para existirem threads é que em muitas aplicações ocorrem múltiplas atividades ao mesmo tempo. Algumas dessas atividades podem ser bloqueadas de tempos em tempos. O modelo de programação se torna mais simples se decomponemos uma aplicação em múltiplos threads sequenciais que executam em quase paralelo.

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

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

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

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

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

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

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

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

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

Enquanto estamos nesse exemplo, por que não adicionar um terceiro thread? Muitos processadores de texto são capacitados para salvar automaticamente todo o arquivo no disco a cada intervalo de tempo em minutos, a fim de

proteger o usuário contra a perda de um dia de trabalho, caso ocorra uma falha no programa ou no sistema ou mesmo uma queda de energia. O terceiro thread pode fazer os backups em disco sem interferir nos outros dois. A situação dos três threads está ilustrada na Figura 2.5.

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

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

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

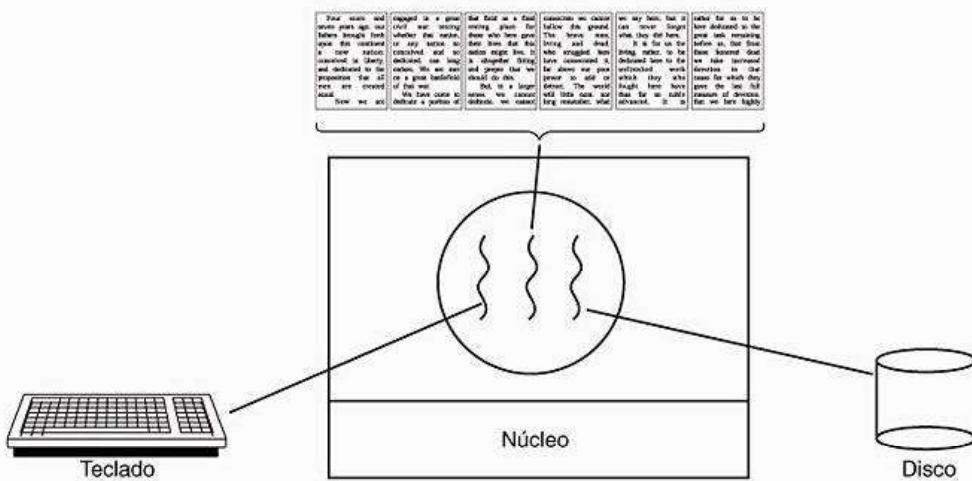


Figura 2.5 Um processador de textos com três threads.

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

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

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

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

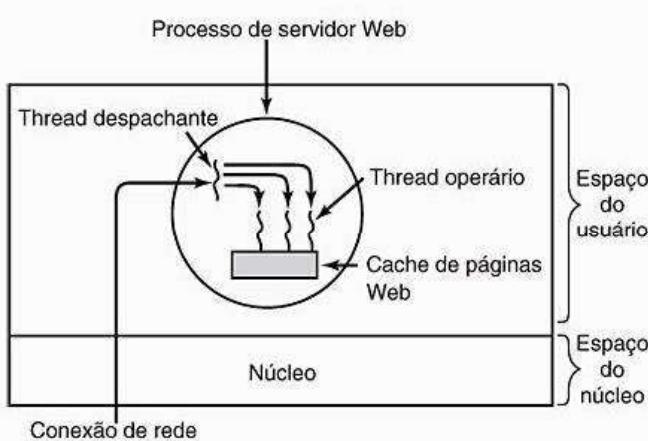
sente na cache de páginas da Web. Se estiver, entrega-a ao cliente e o operário bloqueia esperando uma nova requisição. Do contrário, ele busca a página no disco, entrega-a ao cliente e bloqueia esperando uma nova requisição.

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

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

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

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



**Figura 2.6** Um servidor Web multithread.

```

while (TRUE) {
    get_next_request(&buf);
    handoff_work(&buf);
}

(a)

while (TRUE) {
    wait_for_work(&buf);
    look_for_page_in_cache(&buf, &page);
    if (page_not_in_cache(&page))
        read_page_from_disk(&buf, &page);
    return_page(&page);
}

(b)

```

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

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

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

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

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

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

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

### 2.2.2 | O modelo de thread clássico

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

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

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

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

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

I Tabela 2.3 Três modos de construir um servidor.

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

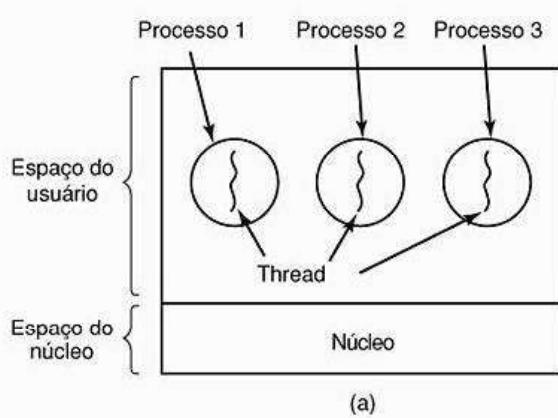
Quando um processo com múltiplos threads é executado em um sistema com uma única CPU, os threads esperam a vez para executar. Na Figura 2.1 vimos como a multiprogramação de processos funciona. Ao chavear entre vários processos, o sistema dá a ilusão de processos sequenciais distintos executando em paralelo.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

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

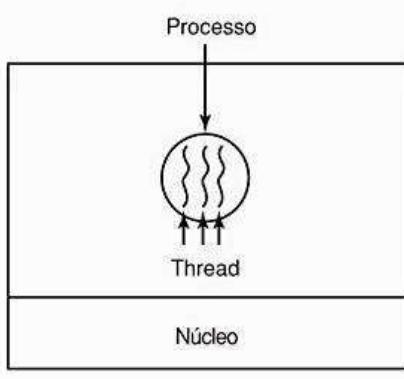
que eles também compartilham as mesmas variáveis globais. Como cada thread pode ter acesso a qualquer endereço de memória dentro do espaço de endereçamento do processo, um thread pode ler, escrever ou até mesmo apagar completamente a pilha de outro thread. Não há proteção entre threads porque (1) é impossível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

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

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



(a)



(b)

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

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

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

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

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

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

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

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

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

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

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

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

### 2.2.3 | Threads POSIX

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

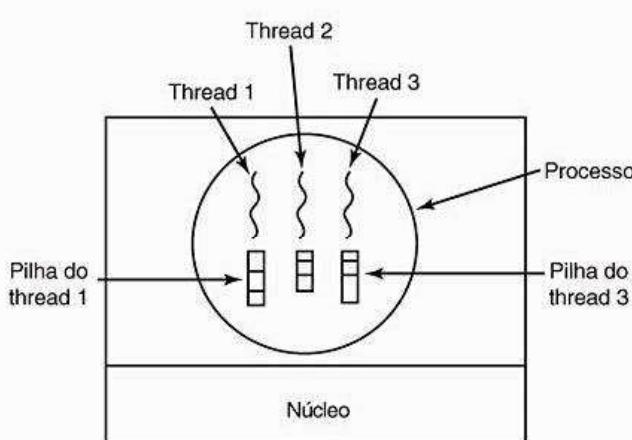


Figura 2.9 Cada thread tem sua própria pilha.

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

**Tabela 2.5** Algumas das chamadas de função de Pthreads.

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

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

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

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

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

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

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

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

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

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

## 2.2.4 | Implementação de threads no espaço do usuário

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

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

Todas essas implementações apresentam a mesma estrutura geral, ilustrada na Figura 2.11(a). Os threads executam no topo de um sistema denominado sistema de tempo de execução (*runtime*), que é uma coleção de rotinas que gerenciam threads. Já vimos quatro deles: `pthread_create`, `pthread_exit`, `pthread_join` e `thread_yield`, mas em geral existem outros.

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

```

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

#define NUMBER_OF_THREADS 10

void *print_hello_world(void *tid)
{
    /* Esta função imprime o identificador do thread e sai. */
    printf("Hello World. Greetings from thread %d\n", tid);
    pthread_exit(NULL);
}

int main(int argc, char *argv[])
{
    /* O programa principal cria 10 threads e sai. */
    pthread_t threads[NUMBER_OF_THREADS];
    int status, i;

    for(i=0; i < NUMBER_OF_THREADS; i++) {
        printf("Main here. Creating thread %d\n", i);
        status = pthread_create(&threads[i], NULL, print_hello_world, (void *)i);

        if (status != 0) {
            printf("Oops. pthread_create returned error code %d\n", status);
            exit(-1);
        }
    }
    exit(NULL);
}

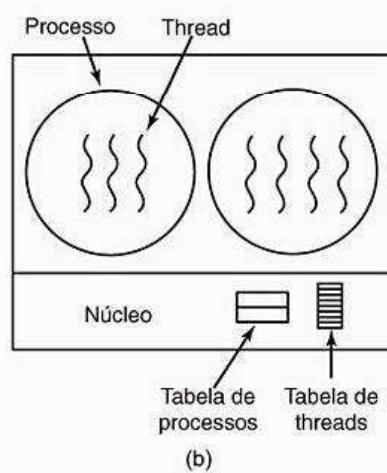
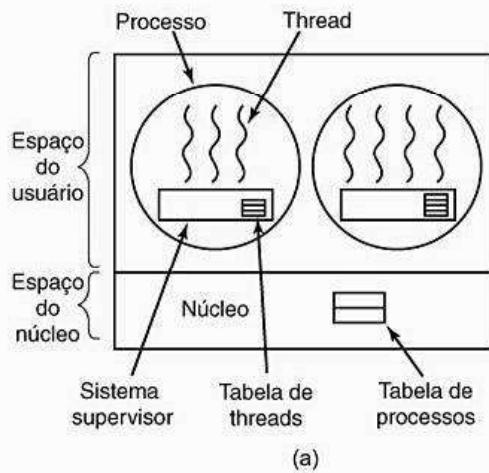
```

**Figura 2.10** Um exemplo de programa usando threads.

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

Quando um thread faz algo que possa bloqueá-lo localmente — por exemplo, espera que um outro thread em seu processo termine algum trabalho —, ele chama uma rotina do sistema de tempo de execução. Essa rotina veri-

fica se o thread deve entrar no estado bloqueado. Em caso afirmativo, ele armazena os registradores do thread (isto é, seus próprios) na tabela de threads, busca na tabela por um thread pronto para executar e recarrega os registradores da máquina com os novos valores salvos do thread. Logo que o ponteiro de pilha e o contador de programa forem alternados, o novo thread reviverá automaticamente. Se a máquina tiver uma instrução que salve todos os regis-



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

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

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

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

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

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

Outra alternativa surge quando se pode antever se uma chamada bloqueará. Em algumas versões do UNIX, há uma chamada de sistema, a *select*, que permite a quem chama, saber se um futuro *read* bloqueará. Quando essa chamada está presente, a rotina de biblioteca *read* pode ser

substituída por outra que antes chama *select* e que depois só chama *read* se isso for seguro (isto é, se não causar bloqueio). Se a chamada *read* causar bloqueio, a chamada não será feita. Em vez disso, um outro thread é executado. Da próxima vez que assumir o controle, o sistema de tempo de execução poderá verificar novamente se a chamada *read* é segura. Esse método requer que se reescrevam partes da biblioteca de chamadas de sistema, é ineficiente e deselegante, mas há poucas alternativas. O código que envolve a chamada de sistema para fazer a verificação é chamado de **jaqueta** (*jacket*) ou **wrapper**.

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

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

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

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

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

### 2.2.5 | Implementação de threads no núcleo

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

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

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

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

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

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

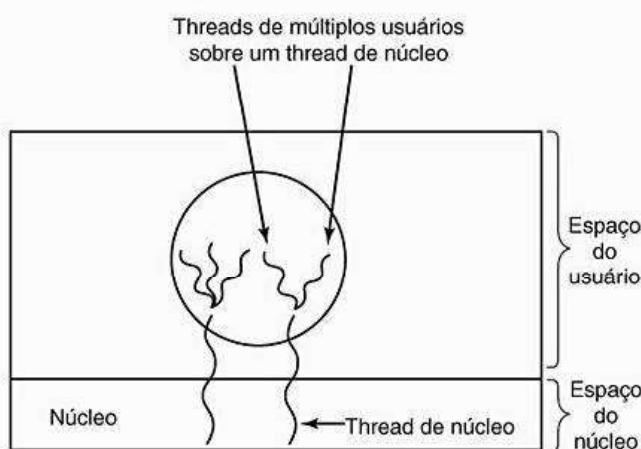
Embora os threads de núcleo resolvam alguns problemas, eles não resolvem todos. Por exemplo, o que acontece quando um processo multithread é bifurcado? O novo processo tem tantos threads quanto o anterior ou tem apenas um? Em muitos casos, a melhor escolha depende do que o processo está planejando fazer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

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

### 2.2.6 | Implementações híbridas

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

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



**Figura 2.12** Multiplexando threads de usuários sobre threads de núcleo.

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

### 2.2.7 | Ativações do escalonador

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

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

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

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

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

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

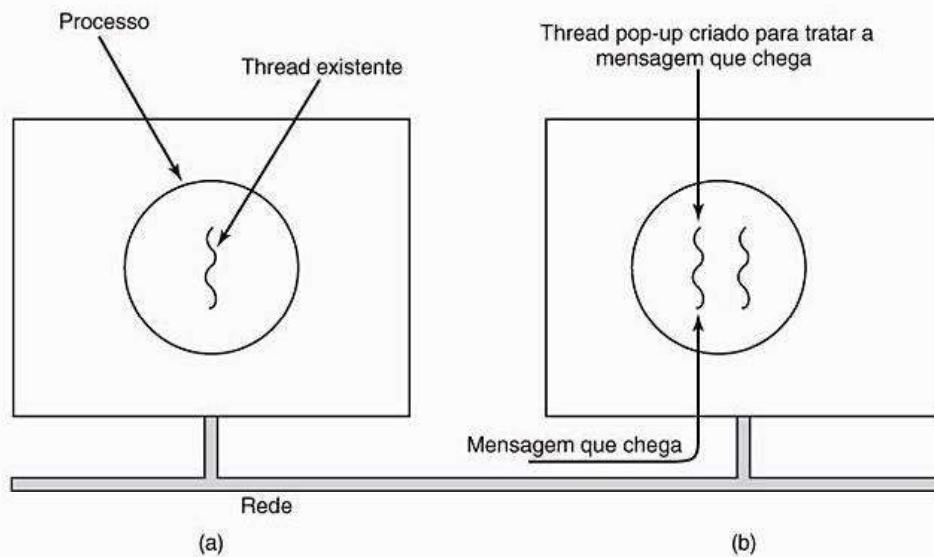
Quando ocorre uma interrupção de hardware enquanto um thread de usuário estiver executando, a CPU interrompida vai para o modo núcleo. Se a interrupção for causada por um evento que não é de interesse do processo interrompido — como a finalização de uma E/S de outro processo —, quando o manipulador da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

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

### 2.2.8 | Threads pop-up

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

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



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

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

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

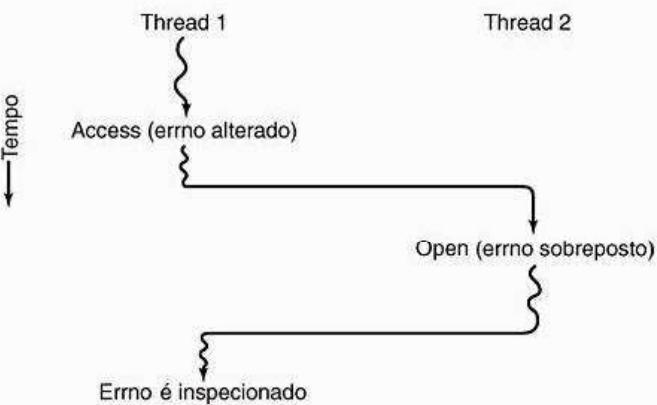
usar qualquer variável global), mas outros threads deveriam deixá-las logicamente isoladas.

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

### 2.2.9 Convertendo o código monothread em código multithread

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

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



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

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

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

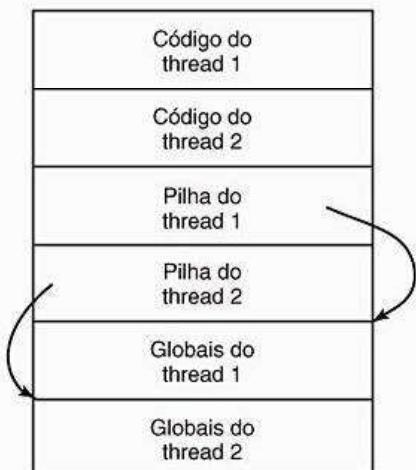
De outra maneira, novas rotinas de biblioteca podem ser introduzidas para criar, alterar e ler essas variáveis globais com abrangência restrita ao thread. A primeira chamada pode ser algo como:

```
create_global("bufptr");
```

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

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

```
set_global("bufptr", &buf);
```



**Figura 2.15** Threads podem ter variáveis globais individuais.

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

```
bufptr = read_global("bufptr");
```

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

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

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

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

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

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

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

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

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

## 2.3 Comunicação entre processos

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

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

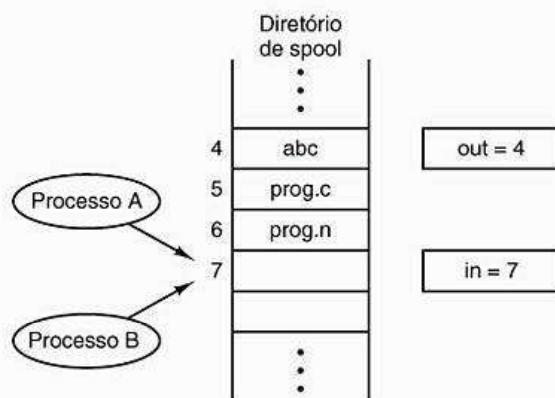
É importante mencionar também que dois desses tópicos se aplicam igualmente bem aos threads. O primeiro — passar informação — é fácil, já que threads compartilham um espaço de endereçamento comum (threads em espaços de endereçamento diferentes e que precisam se comunicar são assuntos da comunicação entre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

### 2.3.1 Condições de corrida

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

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

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



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

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

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

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

### 2.3.2 | Regiões críticas

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

projeto de qualquer sistema operacional e um assunto que estudaremos em detalhes nas próximas seções.

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

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

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

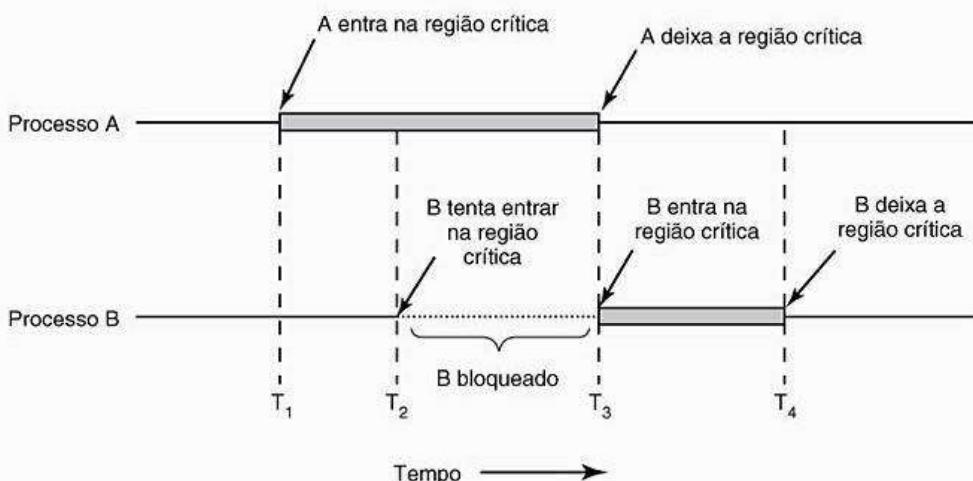
Em um sentido abstrato, o comportamento que queremos é mostrado na Figura 2.17. Ali, o processo *A* entra em sua região crítica no tempo  $T_1$ . Um pouco depois, no tempo  $T_2$ , o processo *B* tenta entrar em sua região crítica, mas falha porque outro processo já está em sua região crítica, e é permitido que apenas um por vez o faça. Consequentemente, *B* fica temporariamente suspenso até que, no tempo  $T_3$ , *A* deixe sua região crítica, permitindo que *B* entre imediatamente. Por fim, *B* sai (em  $T_4$ ) e voltamos à situação original, sem processos em suas regiões críticas.

### 2.3.3 | Exclusão mútua com espera ociosa

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

#### Desabilitando interrupções

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



**Figura 2.17** Exclusão mútua usando regiões críticas.

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

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

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

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

### Variáveis do tipo trava (lock)

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

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

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

### Chaveamento obrigatório

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

```

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

(a)

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

(b)

```

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

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

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

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

De repente, o processo 0 termina sua região não crítica e volta ao início de seu laço. Infelizmente, a ele não será permitido entrar em sua região crítica agora, pois a variável *turn* está em 1 e o processo 1 está ocupado com sua região

não crítica. O processo 0 fica suspenso em seu laço while até que o processo 1 coloque a variável *turn* em 0. Em outras palavras, chavear a vez não é uma boa ideia quando um dos processos for muito mais lento que o outro.

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

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

### Solução de Peterson

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

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

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

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

```

#define FALSE 0
#define TRUE 1
#define N    2           /* número de processos */

int turn;                /* de quem é a vez? */
int interested[N];       /* todos os valores 0 (FALSE) */

void enter_region(int process); /* processo é 0 ou 1 */
{
    int other;            /* número de outro processo */

    other = 1 - process;
    interested[process] = TRUE;
    turn = process;      /* o oposto do processo */
    while (turn == process && interested[other] == TRUE) /* comando nulo */;
}

void leave_region(int process) /* processo: quem está saindo */
{
    interested[process] = FALSE; /* indica a saída da região crítica */
}

```

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

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

### A instrução TSL

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

TSL RX,LOCK

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

É importante notar que impedir o barramento de memória é muito diferente de desabilitar interrupções. Desabilitar interrupções e depois executar a leitura de uma palavra na memória seguida pela escrita não impede que um

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

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

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

<b>enter_region:</b>	
TSL REGISTER,LOCK	copia lock para o registrador e põe lock em 1
CMP REGISTER,#0	lock valia zero?
JNE enter_region	se fosse diferente de zero, lock estaria ligado, portanto, continue no laço de repetição
RET	retorna a quem chamou; entrou na região crítica
 <b>leave_region:</b>	
MOVE LOCK,#0	coloque 0 em lock
RET	retorna a quem chamou

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

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

Uma instrução alternativa à TSL é XCHG<sup>1</sup>, que troca os conteúdos de duas posições atomicamente, por exemplo, um registro e uma palavra de memória. O código é mostrado na Figura 2.21 e, como pode ser visto, é basicamente o mesmo da solução com TSL. Todas as CPUs Intel x86 usam instruções XCHG para sincronização de baixo nível.

### 2.3.4 | Dormir e acordar

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

Esse método não só gasta tempo de CPU, mas também pode ter efeitos inesperados. Considere um computador com dois processos: *H*, com alta prioridade, e *L*, com baixa prioridade. As regras de escalonamento são tais que

*H* é executado sempre que estiver no estado pronto. Em certo momento, com *L* em sua região crítica, *H* torna-se pronto para executar (por exemplo, termina uma operação de E/S). Agora *H* inicia uma espera ocupada, mas, como *L* nunca é escalonado enquanto *H* está executando, *L* nunca tem a oportunidade de deixar sua região crítica e, assim, *H* fica em um laço infinito. Essa situação algumas vezes é referida como o **problema da inversão de prioridade**.

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

### O problema do produtor-consumidor

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

<b>enter_region:</b>	
MOVE REGISTER,#1	insira 1 no registrador
XCHG REGISTER,LOCK	substitua os conteúdos do registrador e a variação de lock
CMP REGISTER,#0	lock valia zero?
JNE enter_region	se fosse diferente de zero, lock estaria ligado, portanto continue no laço de repetição
RET	retorna a quem chamou; entrou na região crítica
 <b>leave_region:</b>	
MOVE LOCK,#0	coloque 0 em lock
RET	retorna a quem chamou

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

1. Do inglês *exchange* = troca (N.R.T.).

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

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

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

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

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

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

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

```
#define N 100
int count = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        if (count == N) sleep();
        insert_item(item);
        count = count + 1;
        if (count == 1) wakeup(consumer);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        if (count == 0) sleep();
        item = remove_item();
        count = count - 1;
        if (count == N - 1) wakeup(producer);
        consume_item(item);
    }
}
```

/\* número de lugares no buffer \*/  
/\* número de itens no buffer \*/

/\* repita para sempre \*/  
/\* gera o próximo item \*/  
/\* se o buffer estiver cheio, vá dormir \*/  
/\* ponha um item no buffer \*/  
/\* incremente o contador de itens no buffer \*/  
/\* o buffer estava vazio? \*/

/\* repita para sempre \*/  
/\* se o buffer estiver cheio, vá dormir \*/  
/\* retire o item do buffer \*/  
/\* decresça de um o contador de itens no buffer \*/  
/\* o buffer estava cheio? \*/  
/\* imprima o item \*/

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

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

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

### 2.3.5 | Semáforos

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

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

A operação up incrementa o valor de um dado semáforo. Se um ou mais processos estivessem dormindo naquele semáforo, incapacitados de terminar uma operação down anterior, um deles seria escolhido pelo sistema (por exemplo, aleatoriamente) e seria dada a permissão para terminar seu down. Portanto, depois de um up em um semáforo com processos dormindo nele, o semáforo permanecerá 0, mas haverá um processo a menos dormindo nele. A operação

de incrementar o semáforo e acordar um processo também é indivisível. Um processo nunca é bloqueado a partir de um up — assim como, no modelo anterior, um processo nunca é bloqueado fazendo um wakeup.

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

### Resolvendo o problema produtor-consumidor usando semáforos

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

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

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

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

```

#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}

void consumer(void)
{
    int item;

    while (TRUE) {
        down(&full);
        down(&mutex);
        item = remove_item();
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}

```

/\* número de lugares no buffer \*/  
 /\* semáforos são um tipo especial de int \*/  
 /\* controla o acesso à região crítica \*/  
 /\* conta os lugares vazios no buffer \*/  
 /\* conta os lugares preenchidos no buffer \*/

/\* TRUE é a constante 1 \*/  
 /\* gera algo para pôr no buffer \*/  
 /\* decresce o contador empty \*/  
 /\* entra na região crítica \*/  
 /\* põe novo item no buffer \*/  
 /\* sai da região crítica \*/  
 /\* incrementa o contador de lugares preenchidos \*/

/\* laço infinito \*/  
 /\* decresce o contador full \*/  
 /\* entra na região crítica \*/  
 /\* pega item do buffer \*/  
 /\* sai da região crítica \*/  
 /\* incrementa o contador de lugares vazios \*/  
 /\* faz algo com o item \*/

**Figura 2.23** O problema produtor-consumidor usando semáforos.

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

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

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

### 2.3.6 | Mutexes

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

Um **mutex** é uma variável que pode estar em um dos dois estados seguintes: desimpedido ou impedido. Consequentemente, somente 1 bit é necessário para representá-lo, mas, na prática, muitas vezes se usa um inteiro, com 0 para

desimpedido e qualquer outro valor para impedido. Duas rotinas são usadas com mutexes. Quando um thread (ou processo) precisa ter acesso a uma região crítica, ele chama *mutex\_lock*. Se o mutex estiver desimpedido (indicando que a região crítica está disponível), a chamada prosseguirá e o thread que chamou *mutex\_lock* ficará livre para entrar na região crítica.

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

Por serem muito simples, os mutexes podem ser implementados facilmente no espaço de usuário, se houver uma instrução TSL ou XCHG disponível. Os códigos de *mutex\_lock* e *mutex\_unlock*, para que sejam usados com um pacote de threads de usuário, são mostrados na Figura 2.24. A solução com XCHG é essencialmente a mesma.

O código do *mutex\_lock* é similar ao código do *enter\_region* da Figura 2.20, mas com uma diferença fundamental. Quando falha ao entrar na região crítica, o *enter\_region* continua testando repetidamente a variável de trava (espera ociosa). Ao final, o tempo de CPU se esgota e algum outro processo é escalonado para executar. Cedo ou tarde o processo que detém a trava é executado e o libera.

Com threads (de usuário), a situação é diferente porque não há relógio que pare os threads que estiverem executando há muito tempo. Consequentemente, um thread que tentar obter a variável de trava pela espera ociosa ficará em um laço infinito e nunca conseguirá obter essa variável, pois ele nunca permitirá que qualquer outro thread execute e libere a variável de trava.

Eis a diferença entre *enter\_region* e *mutex\_lock*: quando falha em verificar a variável de trava, *mutex\_lock* chama *thread\_yield* para que abra mão da CPU em favor de outro thread. Consequentemente, não há espera ocupada. Quando executar na próxima vez, o thread verificará a variável de trava novamente.

O *thread\_yield* é muito rápido, pois é apenas uma chamada do escalonador de threads no espaço do usuário. Como consequência, nem *mutex\_lock* nem *mutex\_unlock* requerem

qualquer chamada ao núcleo. Usando-as, threads de usuário podem sincronizar totalmente dentro do espaço de usuário, com rotinas que exigem somente algumas instruções.

O sistema mutex que descrevemos anteriormente é um conjunto mínimo de chamadas. Para todo software há sempre uma exigência por aperfeiçoamentos, e o caso das primitivas de sincronização não é exceção. Por exemplo, algumas vezes um pacote de thread oferece uma chamada *mutex\_trylock* que obtém a variável de trava ou retorna um código de falha sem bloquear. Essa chamada dá ao thread a flexibilidade de decidir o que fazer se houver alternativas a apenas esperar.

Até agora não tratamos de um determinado tópico, mas é melhor explicitá-lo. Com o pacote de threads no espaço de usuário, não há problema de existirem múltiplos threads com acesso ao mesmo mutex, já que todos os threads operam em um espaço de endereçamento comum. Contudo, como a maioria das soluções anteriores — como o algoritmo de Peterson e os semáforos —, há uma hipótese, não comentada, de que múltiplos processos tenham acesso a pelo menos alguma memória compartilhada, talvez somente a uma única palavra na memória, mas pelo menos alguma. Se os processos possuírem espaços de endereçamento disjuntos, conforme temos dito inconsistentemente, como eles poderiam compartilhar a variável *turn* no algoritmo de Peterson, ou os semáforos, ou um buffer comum?

Há duas respostas. Primeiro, algumas das estruturas de dados compartilhadas, como os semáforos, podem ser armazenadas no núcleo e somente ter seu acesso disponível via chamadas de sistema. Esse método elimina o problema. Em segundo, a maioria dos sistemas operacionais modernos (incluindo UNIX e Windows) oferece meios para que processos compartilhem alguma parte de seus espaços de endereçamento com outros processos. Desse modo, buffers e outras estruturas de dados podem ser compartilhados. Em um caso extremo, em que nada mais é possível, pode-se usar um arquivo compartilhado.

Se dois ou mais processos compartilharem a maior parte ou a totalidade do espaço de endereçamento, a distinção entre processos e threads torna-se algo difusa, mas ainda presente. Dois processos que compartilham um espaço de

<b>mutex_lock:</b>	
TSL REGISTER,MUTEX	copia mutex para o registrador e atribui a ele o valor 1
CMP REGISTER,#0	o mutex era zero?
JZE ok	se era zero, o mutex estava desimpedido, portanto retorne
CALL thread_yield	o mutex está ocupado; escalone um outro thread
JMP mutex_lock	tente novamente mais tarde
ok: RET	retorna a quem chamou; entrou na região crítica
<b>mutex_unlock:</b>	
MOVE MUTEX,#0	coloca 0 em mutex
RET	retorna a quem chamou

Figura 2.24 Implementação do *mutex\_lock* e do *mutex\_unlock*.

endereçamento comum ainda têm arquivos abertos diferentes, temporizadores de alarme e outras propriedades por processo, enquanto os threads dentro de um único processo compartilham todas as propriedades. É sempre válida a afirmação de que múltiplos processos que compartilham um espaço de endereçamento comum nunca têm a eficiência de threads de usuário, pois o núcleo estará profundamente envolvido no gerenciamento desses processos.

### Mutexes em pthreads

Os Pthreads fornecem várias funções que podem ser usadas para sincronizar threads. O mecanismo básico usa uma variável mutex, que pode ser travada ou destravada, para proteger cada região crítica. Um thread que queira entrar em uma região crítica primeiro tenta travar o mutex associado. Se o mutex estiver destravado, o thread pode entrar imediatamente e uma trava é estabelecida atomicamente, evitando que outros threads entrem. Se o mutex já estiver travado, o thread que chama é bloqueado até que ele seja destravado. Se múltiplos threads estão esperando pelo mesmo mutex, quando ele for destravado, apenas um deles é autorizado a continuar e travá-lo novamente. Essas travas não são obrigatórias. Cabe ao programador assegurar que os threads os utilizem corretamente.

As principais chamadas relacionadas a mutexes são mostradas na Tabela 2.6. Como era de se esperar, eles podem ser criados e destruídos. As chamadas para executar essas operações são *pthread\_mutex\_init* e *pthread\_mutex\_destroy*, respectivamente. Eles também podem ser travados por *pthread\_mutex\_lock* — que tenta conquistar a trava e é bloqueado, se o mutex já estiver impedido. Também há a opção de tentar travar um mutex e falhar com um código de erro em vez de bloqueá-lo, se ele já estiver bloqueado. Essa chamada é *pthread\_mutex\_trylock*. Essa chamada permite que um thread entre em espera caso seja necessário. Por fim, *pthread\_mutex\_unlock* ocupa um mutex e libera exatamente um thread se um ou mais estiverem esperando por ele. Os mutexes também podem ter atributos, mas esses são usados apenas para objetivos especializados.

Além dos mutexes, os Pthreads oferecem um segundo mecanismo de sincronização: **variáveis de condição**. Os mutexes são úteis para permitir ou bloquear o acesso a

Chamada de thread	Descrição
<i>pthread_mutex_init</i>	Cria um mutex
<i>pthread_mutex_destroy</i>	Destrói um mutex existente
<i>pthread_mutex_lock</i>	Conquista uma trava ou bloqueio
<i>pthread_mutex_trylock</i>	Conquista uma trava ou falha
<i>pthread_mutex_unlock</i>	Libera uma trava

**Tabela 2.6** Algumas chamadas de Pthreads relacionadas a mutexes.

uma região crítica. As variáveis de condição permitem que os threads bloqueiem em virtude de alguma condição não satisfeita. Quase sempre os dois métodos são usados juntos. Examinemos agora em maiores detalhes a interação de threads, mutexes e variáveis de condição.

Considere novamente o exemplo simples da situação produtor-consumidor: um thread coloca coisas em um buffer e outro as retira. Se o produtor descobrir que não há mais espaço disponível no buffer, ele deve bloquear até que algum se torne disponível. Os mutexes permitem fazer a verificação automaticamente sem interferência de outros threads, mas, diante da descoberta de que o buffer está cheio, o produtor precisa de uma maneira para bloquear e despertar mais tarde. É isso que as variáveis de condição permitem.

Algumas das chamadas relacionadas às variáveis de condição são mostradas na Tabela 2.7. Como você provavelmente já esperava, há chamadas para criar e destruir variáveis de condição. Elas podem ter atributos e há várias chamadas para administrá-las (não mostradas). As operações principais de variáveis de condição são *pthread\_cond\_wait* e *pthread\_cond\_signal*. A primeira bloqueia o thread que chama até que algum outro thread sinalize (usando a última chamada). É claro que as razões para bloquear e esperar não são parte do protocolo de espera e sinalização. O thread bloqueado muitas vezes está esperando que o thread que sinaliza faça algum trabalho, libere algum recurso ou execute alguma outra atividade. Somente depois disso o thread que bloqueia pode prosseguir. As variáveis de condição permitem que essa espera e bloqueio sejam feitos atomicamente. A chamada *pthread\_cond\_broadcast* é usada quando potencialmente há múltiplos threads bloqueados e esperando pelo mesmo sinal.

Variáveis de condição e mutexes são sempre usados em conjunto. O padrão é um thread travar um mutex e então esperar por uma variável condicional quando não puder obter o que precisa. Eventualmente, outro thread sinalizará e ele pode continuar. O *pthread\_cond\_wait* chama atomicamente e destrava atomicamente o mutex que está controlando. Por essa razão, o mutex é um dos parâmetros.

Chamada de thread	Descrição
<i>pthread_cond_init</i>	Cria uma variável de condição
<i>pthread_cond_destroy</i>	Destrói uma variável de condição
<i>pthread_cond_wait</i>	Bloqueio esperando por um sinal
<i>pthread_cond_signal</i>	Sinaliza para outro thread e o desperta
<i>pthread_cond_broadcast</i>	Sinaliza para múltiplos threads e desperta todos eles

**Tabela 2.7** Algumas chamadas de Pthreads relacionadas a variáveis de condição.

É importante observar também que as variáveis de condição (à diferença dos semáforos) não têm memória. Se um sinal é enviado para uma variável de condição pela qual nenhum thread está esperando, o sinal é perdido. Os programadores devem ser cuidadosos para não perder sinais.

Como exemplo do modo como mutexes e variáveis de condição são usados, a Figura 2.25 mostra um problema de consumidor–produtor muito simples com um único buffer. Quando o produtor tiver enchedo o buffer, ele deve esperar até que o consumidor o esvazie antes de gerar o próximo item. De modo semelhante, quando o consumidor tiver removido um item, ele deve esperar até que o produtor te-

nha gerado outro. Embora seja muito simples, esse exemplo ilustra os mecanismos básicos. O comando que coloca um thread para dormir sempre deveria verificar a condição para assegurar que ela seja satisfeita antes de prosseguir, visto que o thread poderia ter sido despertado em virtude de um sinal do UNIX ou por alguma outra razão.

### 2.3.7 | Monitores

Com o uso de semáforos e mutexes, a comunicação entre processos parece fácil, não é? Você ainda não viu nada! Observe com bastante atenção a ordem dos downs antes de

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;
int buffer = 0;

void *producer(void *ptr)
{
    int i;

    for (i= 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* obtém acesso exclusivo ao buffer */
        while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
        buffer = i; /* põe item no buffer */
        pthread_cond_signal(&condc); /* acorda consumidor */
        pthread_mutex_unlock(&the_mutex);/* libera acesso ao buffer */
    }
    pthread_exit(0);
}

void *consumer(void *ptr)
{
    int i;

    for (i = 1; i <= MAX; i++) {
        pthread_mutex_lock(&the_mutex); /* obtém acesso exclusivo ao buffer */
        while (buffer == 0 ) pthread_cond_wait(&condc, &the_mutex);
        buffer = 0; /* retire o item do buffer */
        pthread_cond_signal(&condp); /* acorda o produtor */
        pthread_mutex_unlock(&the_mutex);/* libera acesso ao buffer */
    }
    pthread_exit(0);
}

int main(int argc, char **argv)
{
    pthread_t pro, con;
    pthread_mutex_init(&the_mutex, 0);
    pthread_cond_init(&condc, 0);
    pthread_cond_init(&condp, 0);
    pthread_create(&con, 0, consumer, 0);
    pthread_create(&pro, 0, producer, 0);
    pthread_join(pro, 0);
    pthread_join(con, 0);
    pthread_cond_destroy(&condc);
    pthread_cond_destroy(&condp);
    pthread_mutex_destroy(&the_mutex);
}
```

**Figura 2.25** Usando threads para resolver o problema produtor-consumidor.

inserir ou remover os itens do buffer na Figura 2.23. Suponha que os dois downs, no código do produtor, estivessem invertidos, de modo que o *mutex* seria decrescido antes de *vazio* em vez de depois dele. Se o buffer estivesse completamente cheio, o produtor seria bloqueado com *mutex* em 0. Consequentemente, na vez seguinte em que o consumidor tentasse ter acesso ao buffer, faria um down no *mutex*, agora em 0, e seria bloqueado também. Ambos os processos permaneceriam eternamente bloqueados e nunca mais funcionariam. Essa situação infeliz é chamada de impasse (*deadlock*). Estudaremos os impasses em detalhes no Capítulo 6.

Esse problema foi levantado para mostrar o cuidado que se deve ter no uso de semáforos. Um erro sutil pode pôr tudo a perder. É como programar em linguagem assembly — ou até pior, pois os erros são condições de corrida, impasses e outros modos de comportamento imprevisível e irreprodutível.

Para facilitar a escrita correta de programas, Hoare (1974) e Brinch Hansen (1973) propuseram uma unidade básica de sincronização de alto nível chamada **monitor**. As propostas deles eram um pouco diferentes, conforme veremos a seguir. Um monitor é uma coleção de rotinas, variáveis e estruturas de dados, tudo isso agrupado em um tipo especial de módulo ou pacote. Os processos podem chamar as rotinas em um monitor quando quiserem, mas não podem ter acesso direto às estruturas internas de dados ao monitor a partir de rotinas declaradas fora dele. A Figura 2.26 ilustra um monitor escrito em uma linguagem imaginária, a Pascal Pidgin. C não pode ser usada aqui porque os monitores são um conceito de *linguagem* e C não os tem.

Os monitores apresentam uma propriedade importante que os torna úteis para realizar a exclusão mútua: somente um processo pode estar ativo em um monitor em um dado momento. O monitor é uma construção da linguagem de programação e, portanto, os compiladores sabem que eles são especiais e, por isso, tratam as chamadas a rotinas do monitor de modo diferente de outras chamadas de procedimento. Em geral, quando um processo chama uma rotina

```
monitor example
  integer i;
  condition c;

  procedure producer();
    begin
      end;
    procedure consumer();
    begin
      end;
  end;
end monitor;
```

Figura 2.26 Um monitor.

do monitor, algumas das primeiras instruções da rotina verificarão se qualquer outro processo está atualmente ativo dentro do monitor. Se estiver, o processo que chamou será suspenso até que o outro processo deixe o monitor. Se nenhum outro processo estiver usando o monitor, o processo que chamou poderá entrar.

Cabe ao compilador implementar a exclusão mútua nas entradas do monitor, mas um modo comum é usar um *mutex* ou um semáforo binário. Como é o compilador e não o programador que providencia a exclusão mútua, é muito menos provável que algo dê errado. De qualquer maneira, quem codifica o monitor não precisa saber como o compilador implementou a exclusão mútua. Basta saber que, convertendo todas as regiões críticas em rotinas do monitor, dois processos nunca executarão suas regiões críticas ao mesmo tempo.

Embora monitores ofereçam um modo fácil de fazer a exclusão mútua (como vimos anteriormente), isso não é o bastante. É preciso também um modo de bloquear processos quando não puderem continuar. No problema produtor-consumidor, é muito fácil colocar todos os testes de buffer cheio e buffer vazio nas rotinas do monitor, mas como o produtor seria bloqueado quando ele encontrasse o buffer cheio?

A solução está na introdução de **variáveis condicionais**, com duas operações sobre elas: *wait* e *signal*. Quando uma rotina do monitor descobre que não pode prosseguir (por exemplo, o produtor percebe que o buffer está cheio), executa um *wait* sobre alguma variável condicional — por exemplo, *cheio*. Essa ação resulta no bloqueio do processo que está chamando. Ela também permite que outro processo anteriormente proibido de entrar no monitor agora entre. Vimos variáveis de condição e essas operações no contexto de Pthreads anteriormente.

Esse outro processo — por exemplo, o consumidor — pode acordar seu parceiro adormecido a partir da emissão de um *signal* para a variável condicional que seu parceiro está esperando. Para evitar que dois processos permaneçam no monitor ao mesmo tempo, precisamos de uma regra que determine o que acontece depois de um *signal*. Hoare propôs deixar o processo recém-acordado executar, suspendendo o outro. Brinch Hansen sugeriu uma maneira astuta de resolver o problema: exigir que um processo que emitir um *signal* saia do monitor imediatamente. Em outras palavras, um comando *signal* só poderá aparecer como o último comando de uma rotina do monitor. Usaremos a proposta de Brinch Hansen porque ela é conceitualmente mais simples e também mais fácil de implementar. Se um *signal* é emitido sobre uma variável condicional pela qual vários processos estejam esperando, somente um deles, determinado pelo escalonador do sistema, é despertado.

É preciso mencionar que há uma terceira solução que não foi proposta por Hoare nem por Brinch Hansen. Essa solução deixa o emissor do sinal prosseguir sua execução e permite ao processo em espera começar a executar somente depois que o emissor do sinal tenha saído do monitor.

Variáveis condicionais não são contadores. Elas não acumulam sinais para usá-los depois, como fazem os semáforos. Assim, se uma variável condicional for sinalizada sem ninguém estar esperando pelo sinal, este ficará perdido para sempre. Em outras palavras, wait deve vir antes do signal. Essa regra torna a implementação muito mais simples. Na prática não é um problema, pois é fácil manter o controle do estado de cada processo com variáveis, se for necessário. Um processo capaz de emitir um signal pode perceber, a partir da verificação das variáveis, que essa operação não é necessária.

Um esqueleto do problema produtor-consumidor com monitores é mostrado na Figura 2.27 em uma linguagem imaginária, a Pascal Pidgin. A vantagem de usar Pascal Pidgin é que ela é pura, simples e segue exatamente o modelo de Hoare/Brinch Hansen.

Você pode estar pensando que as operações wait e signal são parecidas com as operações sleep e wakeup, que vimos anteriormente ao tratar das condições de corrida fatais.

```

monitor ProducerConsumer
  condition full, empty;
  integer count,
  procedure insert(item:integer);
  begin
    if count=N then wait(full);
    insert_item(item);
    count:=count+1;
    if count=1 then signal(empty)
  end;
  function remove:integer;
  begin
    if count=0 then wait(empty);
    remove=remove_item;
    count:=count-1;
    if count=N-1 then signal(full)
  end;
  count:=0;
end monitor;
procedure producer;
begin
  while true do
    begin
      item=produce_item;
      ProducerConsumer.insert(item)
    end
  end;
procedure consumer;
begin
  while true do
    begin
      item=ProducerConsumer.remove;
      consume_item(item)
    end
  end;
end;

```

**Figura 2.27** Um esqueleto do problema produtor-consumidor com monitores. Somente uma rotina está ativa por vez no monitor. O buffer tem  $N$  lugares.

Elas são muito similares, mas apresentam uma diferença fundamental: sleep e wakeup falharam porque, enquanto um processo estava tentando ir dormir, o outro tentava acordá-lo. Com monitores, isso não acontece. A exclusão mútua automática das rotinas do monitor garante, por exemplo, que, se o produtor dentro de uma rotina do monitor descobrir que o buffer está cheio, esse produtor será capaz de terminar a operação wait sem se preocupar com a possibilidade de o escalonador chamar para o consumidor um pouco antes de wait terminar. O consumidor nem mesmo será permitido dentro do monitor até que wait tenha terminado e o produtor tenha sido marcado como não mais executável.

Embora a Pascal Pidgin seja uma linguagem imaginária, algumas linguagens de programação reais também dão suporte a monitores, embora nem sempre conforme o projetado por Hoare e Brinch Hansen. Uma dessas linguagens é Java. Esta é uma linguagem orientada a objetos que dão suporte a threads de usuário e também permite que métodos (rotinas) sejam agrupados em classes. Adicionando-se a palavra-chave synchronized à declaração de um método, Java garante que, uma vez iniciado qualquer thread executando aquele método, a nenhum outro thread será permitido executar qualquer outro método synchronized naquela classe.

Uma solução para o problema produtor-consumidor com base no uso de monitores em Java é mostrada na Figura 2.28. A solução é constituída de quatro classes. A classe mais externa, *ProducerConsumer*, cria e inicia dois threads, *p* e *c*. A segunda e a terceira classe, *producer* e *consumer*, respectivamente, contêm o código para o produtor e o consumidor. Por fim, a classe *our\_monitor* é o monitor; ela contém dois threads sincronizados que são usados, na verdade, para inserir elementos no buffer compartilhado e tirá-los de lá. Diferentemente dos exemplos anteriores, agora mostramos o código completo para *insert* e *remove*.

Os threads produtor e consumidor são funcionalmente idênticos a seus correspondentes em todos os nossos exemplos anteriores. O produtor possui um laço infinito que gera dados e os põe no buffer comum. O consumidor tem igualmente um laço infinito, que tira dados do buffer comum e faz algo útil com ele.

A parte interessante desse programa é a classe *our\_monitor*, que contém o buffer, as variáveis de administração e dois métodos sincronizados. Quando está ativo dentro do *insert*, o produtor sabe com certeza que o consumidor não pode estar ativo dentro do *remove*, tornando seguras as atualizações das variáveis e do buffer sem o temor das condições de corrida. A variável *count* controla o número de itens que estão no buffer. Ela pode assumir qualquer valor entre 0 e  $N-1$  inclusive. A variável *lo* aponta para um lugar do buffer que contém o próximo item a ser buscado. Da mesma maneira, *hi* aponta para um lugar do buffer onde o próximo item será colocado. É permitido que *lo* = *hi*, o que

```

public class ProducerConsumer {
    static final int N = 100      // constante com o tamanho do buffer
    static producer p = new producer(); // instância de um novo thread produtor
    static consumer c = new consumer(); // instância de um novo thread consumidor
    static our_monitor mon = new our_monitor(); // instância de um novo monitor

    public static void main(String args[]) {
        p.start(); // inicia o thread produtor
        c.start(); // inicia o thread consumidor
    }

    static class producer extends Thread {
        public void run() { // o método run contém o código do thread
            int item;
            while (true) { // laço do produtor
                item = produce_item();
                mon.insert(item);
            }
        }
        private int produce_item() { ... } // realmente produz
    }

    static class consumer extends Thread {
        public void run() { método run contém o código do thread
            int item;
            while (true) { // laço do consumidor
                item = mon.remove();
                consume_item(item);
            }
        }
        private void consume_item(int item) { ... } // realmente consome
    }

    static class our_monitor { // este é o monitor
        private int buffer[] = new int[N];
        private int count = 0, lo = 0, hi = 0; // contadores e índices

        public synchronized void insert(int val) {
            if (count == N) go_to_sleep(); // se o buffer estiver cheio, vá dormir
            buffer [hi] = val; // insere um item no buffer
            hi = (hi + 1) % N; // lugar para colocar o próximo item
            count = count + 1; // mais um item no buffer agora
            if (count == 1) notify(); // se o consumidor estava dormindo, acorde-o
        }

        public synchronized int remove() {
            int val;
            if (count == 0) go_to_sleep(); // se o buffer estiver vazio, vá dormir
            val = buffer [lo]; // busca um item no buffer
            lo = (lo + 1) % N; // lugar de onde buscar o próximo item
            count = count - 1; // um item a menos no buffer
            if (count == N - 1) notify(); // se o produtor estava dormindo, acorde-o
            return val;
        }

        private void go_to_sleep() { try{wait();} catch(InterruptedException exc) {}}
    }
}

```

**Figura 2.28** Uma solução para o problema produtor-consumidor em Java.

significa que 0 item ou  $N$  itens estão no buffer. O valor de  $count$  indica qual desses casos ocorre.

Métodos sincronizados em Java são essencialmente diferentes dos monitores clássicos: Java não tem variáveis condicionais. Em vez disso, ela oferece dois métodos, *wait* e *notify*, equivalentes a *sleep* e *wakeup*, exceto que, quando usados dentro de métodos sincronizados, não estão sujeitos às condições de corrida. Teoricamente, o método *wait* pode ser interrompido — que é o papel do código que o envolve.

Java requer que o tratamento de exceções seja explícito. Para nosso propósito, imagine apenas que *go\_to\_sleep* seja o caminho para ir dormir.

Tornando automática a exclusão mútua das regiões críticas, os monitores deixam a programação paralela muito menos sujeita a erros que com semáforos. Ainda assim eles também têm alguns problemas. Não é à toa que nossos dois exemplos de monitores estavam escritos em Pascal Pidgin e em Java — em vez de C, como os outros exemplos

deste livro. Conforme foi mencionado anteriormente, os monitores são um conceito de linguagem de programação. O compilador deve reconhecê-los e organizá-los para a exclusão mútua de alguma maneira. C, Pascal e a maioria das outras linguagens não possuem monitores; portanto, não é razoável esperar que seus compiladores imponham alguma regra de exclusão mútua. Na verdade, como poderia o compilador saber até mesmo quais rotinas estavam nos monitores e quais não estavam?

Essas mesmas linguagens não apresentam semáforos, mas incluí-los é fácil: tudo o que você precisa fazer é adicionar à biblioteca duas pequenas rotinas, em código de linguagem assembly, a fim de emitir as chamadas de sistema up e down. Os compiladores nem sequer precisam saber que elas existem. Obviamente, os sistemas operacionais precisam ser informados sobre os semáforos, mas, caso se tenha um sistema operacional baseado em semáforos, é possível ainda escrever os programas de usuário para ele em C ou C++ (ou até mesmo em linguagem assembly, se você for masoquista o suficiente). Para os monitores, você precisa de uma linguagem que os tenha construído.

Outro problema com monitores e também com semáforos é que eles foram projetados para resolver o problema da exclusão mútua em uma ou mais CPUs, todas com acesso a uma memória comum. Pondo os semáforos na memória compartilhada e protegendo-os com as instruções TSL, ou XCHG, podemos evitar disputas. Quando vamos para um sistema distribuído formado por múltiplas CPUs, cada qual com sua própria memória privada e conectada por uma rede local, essas primitivas tornam-se inaplicáveis. A conclusão é que os semáforos são de nível muito baixo e os monitores não são úteis, exceto para algumas linguagens de programação. Além disso, nenhuma dessas primitivas permite troca de informações entre as máquinas. Algo diferente se faz necessário.

### 2.3.8 | Troca de mensagens

Esse algo diferente é a **troca de mensagens** (*message passing*). Esse método de comunicação entre processos usa duas primitivas, send e receive, que, assim como os semáforos mas diferentemente dos monitores, são chamadas de sistema e não construções de linguagem. Dessa maneira, elas podem ser facilmente colocadas em rotinas de biblioteca, como

```
send(destination, &message);
```

e

```
receive(source, &message);
```

A primeira chamada envia uma mensagem para um dado destino; a segunda recebe uma mensagem de uma dada origem (ou de uma origem qualquer, se o receptor não se importar). Se nenhuma mensagem estiver disponível, o receptor poderá ficar bloqueado até que alguma mensagem chegue. Como alternativa, ele pode retornar imediatamente acompanhado de um código de erro.

### Projeto de sistemas de troca de mensagens

Sistemas de troca de mensagens apresentam muitos problemas complexos e dificuldades de projeto que não ocorrem com semáforos ou monitores — especialmente se os processos comunicantes estiverem em máquinas diferentes conectadas por uma rede. Por exemplo, as mensagens podem ser perdidas pela rede. Para se prevenir contra mensagens perdidas, o emissor e o receptor podem combinar que, assim que uma mensagem tenha sido recebida, o receptor enviará de volta uma mensagem especial de **confirmação de recebimento** (*acknowledgement*). Se o emissor não tiver recebido a confirmação de recebimento dentro de um certo intervalo de tempo, ele retransmitirá a mensagem.

Agora, pense no que acontece se a própria mensagem for recebida corretamente, mas a confirmação de recebimento tiver sido perdida. O emissor retransmitirá a mensagem e, portanto, o receptor a receberá duas vezes. É fundamental que o receptor seja capaz de distinguir entre uma mensagem nova e a retransmissão de uma mensagem antiga. Normalmente, esse problema é resolvido mediante a colocação de números em uma sequência consecutiva em cada mensagem original. Se o receptor obtém uma mensagem que carregue o mesmo número sequencial de uma mensagem anterior, ele sabe que a mensagem é uma duplicata que pode ser ignorada. A comunicação bem-sucedida diante de trocas de mensagens não confiáveis é uma importante parte do estudo sobre redes de computadores. Para mais informações, veja Tanenbaum (1996).

Os sistemas de mensagens também precisam lidar com a questão dos nomes dos processos, para que o processo especificado em uma chamada send ou receive não seja ambíguo. A **autenticação** também é um tópico de sistemas de mensagens: como o cliente pode saber que está se comunicando com o servidor de arquivos real e não com um impostor?

Na outra ponta do espectro, há ainda tópicos de projeto que são importantes quando o emissor e o receptor estão na mesma máquina. Um desses tópicos é o desempenho. Copiar mensagens de um processo para outro é sempre mais lento que realizar uma operação de semáforo ou entrar em um monitor. Muito se tem feito para tornar a troca de mensagens eficiente. Cheriton (1984), por exemplo, sugeriu um tamanho de mensagem limitado, que caiba nos registradores das máquinas, para então serem realizadas as trocas de mensagens com o uso dos registradores.

### O problema produtor-consumidor com troca de mensagens

Agora vejamos como o problema produtor-consumidor pode ser resolvido com a troca de mensagens e sem qualquer memória compartilhada. Uma solução possível é mostrada na Figura 2.29. Partimos do pressuposto de que todas as mensagens são do mesmo tamanho e que as

```

#define N 100                                /* número de lugares no buffer */

void producer(void)
{
    int item;
    message m;                            /* buffer de mensagens */

    while (TRUE) {
        item = produce_item();
        receive(consumer, &m);
        build_message(&m, item);
        send(consumer, &m);                /* gera alguma coisa para colocar no buffer */
                                                /* espera que uma mensagem vazia chegue */
                                                /* monta uma mensagem para enviar */
                                                /* envia item para consumidor */
    }
}

void consumer(void)
{
    int item, i;
    message m;

    for (i = 0; i < N; i++) send(producer, &m); /* envia N mensagens vazias */
    while (TRUE) {
        receive(producer, &m);           /* pega mensagem contendo item */
        item = extract_item(&m);         /* extrai o item da mensagem */
        send(producer, &m);              /* envia a mensagem vazia como resposta */
        consume_item(item);             /* faz alguma coisa com o item */
    }
}

```

**Figura 2.29** O problema produtor-consumidor com  $N$  mensagens.

mensagens enviadas, mas ainda não recebidas, são armazenadas automaticamente pelo sistema operacional. Nessa solução, é usado um total de  $N$  mensagens, analogamente aos  $N$  lugares no buffer em uma memória compartilhada. O consumidor começa enviando  $N$  mensagens vazias para o produtor. Se tiver algum item para fornecer ao consumidor, o produtor pegará uma mensagem vazia e enviará de volta uma mensagem cheia. Desse modo, o número total de mensagens no sistema permanece constante com o decorrer do tempo, e assim elas podem ser armazenadas em uma quantidade de memória previamente conhecida.

Se o produtor trabalhar mais rápido que o consumidor, todas as mensagens serão preenchidas, à espera do consumidor; o produtor será bloqueado, aguardando que uma mensagem vazia volte. Se o consumidor trabalhar mais rápido, então acontecerá o inverso: todas as mensagens estarão vazias esperando que o produtor as preencha; o consumidor será bloqueado, esperando por uma mensagem cheia.

Há muitas variações possíveis do mecanismo de troca de mensagens. Para começar, observemos como as mensagens são endereçadas. Um meio para isso é atribuir a cada processo um endereço único e fazer as mensagens serem endereçadas aos processos. Um outro modo é inventar uma nova estrutura de dados, chamada **caixa postal**. Uma caixa postal é um local para armazenar temporariamente um certo número de mensagens, normalmente especificado quando ela é criada. Quando as caixas postais são usadas,

os parâmetros de endereço nas chamadas `send` e `receive` são as caixas postais, não os processos. Ao tentar enviar para uma caixa postal que esteja cheia, um processo é suspenso até que uma mensagem seja removida daquela caixa postal e dê lugar a uma nova.

Para o problema produtor-consumidor, tanto o produtor quanto o consumidor criariam caixas postais suficientemente grandes para conter  $N$  mensagens. O produtor enviria mensagens contendo dados à caixa postal do consumidor e este mandaria mensagens vazias para a caixa postal do produtor. O mecanismo de buffer das caixas postais é bastante simples: a caixa postal de destino contém mensagens enviadas ao processo de destino, mas ainda não aceitas.

O outro extremo das caixas postais é eliminar todo o armazenamento temporário. Quando se opta por esse caminho, se o `send` é emitido antes do `receive`, o processo emissor permanece bloqueado até que ocorra o `receive`, momento no qual a mensagem pode ser copiada diretamente do emissor para o receptor, sem armazenamento intermedio. Da mesma maneira, se o `receive` é emitido antes, o receptor é bloqueado até que ocorra um `send`. Essa estratégia é mais conhecida como **rendezvous**<sup>2</sup>. Ela é mais fácil de implementar que um esquema de armazenamento de mensagens, mas é menos flexível, pois o emissor e o receptor são forçados a executar de maneira interdependente.

A troca de mensagens é bastante usada em sistemas de programação paralela. Um sistema de troca de men-

2. Expressão em francês para ‘encontro marcado’ (N.T.).

sagens bem conhecido, por exemplo, é o **MPI** (*message-passing interface — interface de troca de mensagem*), amplamente usado em computação científica. Para mais informações sobre ele, veja, por exemplo, Gropp et al. (1994) e Snir et al. (1996).

### 2.3.9 | Barreiras

Nosso último mecanismo de sincronização é dirigido aos grupos de processos em vez de situações que envolvem dois processos do tipo produtor-consumidor. Algumas aplicações são divididas em fases e têm como regra que nenhum processo pode avançar para a próxima fase até que todos os processos estejam prontos a fazê-lo. Isso pode ser conseguido por meio da colocação de uma **barreira** no final de cada fase. Quando alcança a barreira, um processo permanece bloqueado até que todos os processos alcancem a barreira. A operação de uma barreira é ilustrada na Figura 2.30.

Na Figura 2.30(a), vemos quatro processos chegando a uma barreira, o que significa que eles estão apenas computando e ainda não atingiram o final da fase atual. Depois de um tempo, o primeiro processo termina toda a computação atribuída a ele para a primeira fase. Ele então executa a primitiva barrier, em geral por intermédio da chamada a uma rotina de biblioteca. O processo é, então, suspenso. Mais tarde, a primeira fase é terminada por um segundo e depois por um terceiro processo, que também executam a primitiva

barrier. Essa situação é ilustrada na Figura 2.30(b). Por fim, quando o último processo, C, atinge a barreira, todos os processos são liberados, conforme ilustrado na Figura 2.30(c).

Como exemplo de uma situação que requer barreiras, considere um problema típico de relaxamento, da física ou da engenharia. Há em geral uma matriz que contém alguns valores iniciais. Os valores podem representar temperaturas em vários pontos de uma placa de metal. O objetivo pode ser calcular quanto tempo leva para que o efeito de uma chama localizada em um canto se propague por toda a placa.

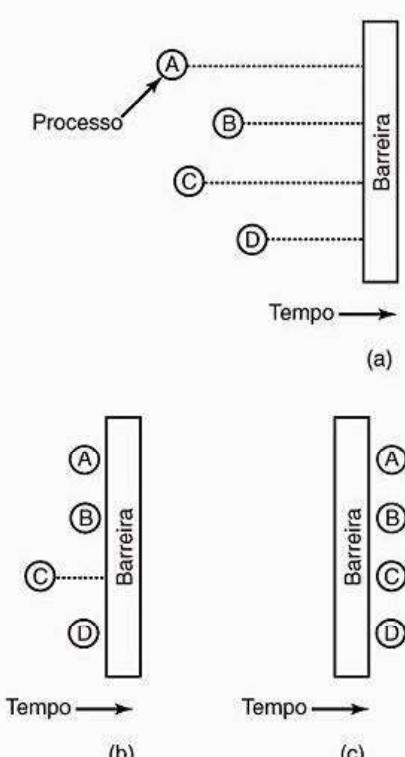
Começando com os valores atuais, uma transformação é aplicada à matriz para obter uma segunda versão da matriz — por exemplo, aplicando-se as leis da termodinâmica para verificar todas as temperaturas em um instante  $T$  mais tarde. O processo é, então, repetido várias vezes e fornece as temperaturas nos pontos de amostragem como uma função do tempo, à medida que a placa é aquecida. O algoritmo produz, portanto, uma série de matrizes ao longo do tempo.

Agora, imagine que a matriz seja muito grande (digamos, um milhão por um milhão), exigindo o uso de processamento paralelo (possivelmente em um sistema multiprocessador) para aumentar a velocidade do cálculo. Processos diferentes trabalham com diferentes partes da matriz, calculando os elementos da nova matriz a partir dos valores anteriores e de acordo com as leis da física. Contudo, um processo só pode começar uma iteração  $n + 1$  quando a iteração  $n$  terminar, isto é, quando todos os processos terminarem seus trabalhos atuais. O meio de chegar a esse objetivo é programar cada processo de maneira que ele execute uma operação barrier depois que terminar sua parte da iteração. Quando todos tiverem feito sua parte, a nova matriz (a entrada para a próxima iteração) estará pronta e todos os processos serão simultaneamente liberados para inicializar a próxima iteração.

## 2.4 Escalonamento

Quando um computador é multiprogramado, ele muitas vezes tem múltiplos processos ou threads que competem pela CPU ao mesmo tempo. Essa situação ocorre sempre que dois ou mais processos estão simultaneamente no estado pronto. Se somente uma CPU se encontrar disponível, deverá ser feita uma escolha de qual processo executará em seguida. A parte do sistema operacional que faz a escolha é chamada de **escalonador**, e o algoritmo que ele usa é o **algoritmo de escalonamento**. Esses tópicos formam o assunto das próximas seções.

Muitos dos problemas que se aplicam ao escalonamento de processos também são válidos para o escalonamento de threads, embora haja diferenças. Quando o núcleo gerencia threads, o escalonamento normalmente é feito por thread, dando pouca ou nenhuma atenção ao processo ao qual o thread pertence. Inicialmente nos concentraremos em questões de escalonamento que se aplicam tanto a processos como a threads. Em seguida estudaremos especifica-



**Figura 2.30** Uso de uma barreira. (a) Processos se aproximando de uma barreira. (b) Todos os processos, exceto um, estão bloqueados pela barreira. (c) Quando o último processo chega à barreira, todos passam por ela.

mente o escalonamento de threads e alguns dos problemas exclusivos que suscita. Lidaremos com chips multinúcleo no Capítulo 8.

### 2.4.1 | Introdução ao escalonamento

De volta aos velhos tempos dos sistemas em lote, com a entrada na forma de imagens de cartões em uma fita magnética, o algoritmo de escalonamento era simples: apenas execute a próxima tarefa que está na fita. Com os sistemas multiprogramados, o algoritmo de escalonamento tornou-se mais complexo porque, em geral, havia vários usuários esperando por um serviço. Alguns computadores de grande porte ainda combinam serviços em lote e de tempo compartilhado, exigindo assim que o escalonador decida se uma tarefa em lote ou um usuário interativo em um terminal deve ser atendido. (Atente para o seguinte: uma tarefa em lote pode ser uma requisição para executar uma sucessão de vários programas, mas, para esta seção, vamos supor que seja uma requisição para executar um único programa.) Como o tempo de CPU é um recurso escasso nessas máquinas, um bom escalonador pode fazer uma grande diferença no desempenho observado e na satisfação do usuário. Consequentemente, muito se fez tendo em vista desenvolver algoritmos de escalonamento inteligentes e eficientes.

Com o advento dos computadores pessoais, a situação mudou de duas maneiras. Primeiro, na maior parte do tempo existe apenas um processo ativo. É improvável que um usuário esteja, simultaneamente, entrando com um documento em um processador de textos e compilando um programa em segundo plano. Quando o usuário digita um comando para o processador de textos, o escalonador não precisa trabalhar muito para perceber qual processo executar — o processador de textos é o único candidato.

Em segundo lugar, os computadores, com o passar dos anos, ficaram tão mais rápidos que a CPU raramente chegará a ser um recurso escasso. A maioria dos programas para computadores pessoais é limitada pela velocidade com que o usuário pode entrar dados (digitando ou clicando), e não pela taxa na qual a CPU é capaz de processá-los. Até os compiladores — grandes consumidores de ciclos de CPU no passado — atualmente levam, no máximo, alguns segundos. Mesmo quando dois programas estiverem executando de modo simultâneo — como um processador de textos e uma planilha —, dificilmente importará qual começa primeiro, já que o usuário estará esperando, provavelmente, que ambos terminem. Como consequência, o escalonamento não é tão importante em PCs simples. Claro, há aplicações que praticamente esgotam a CPU: exibir uma hora de vídeo de alta resolução pode exigir processamento de imagens de altíssima capacidade para lidar com cada um dos 108 mil quadros em NTSC (90 mil no PAL), mas essas aplicações são exceções, não a regra.

Quando nos concentramos em servidores e estações de trabalho de alto desempenho em rede, a situação muda.

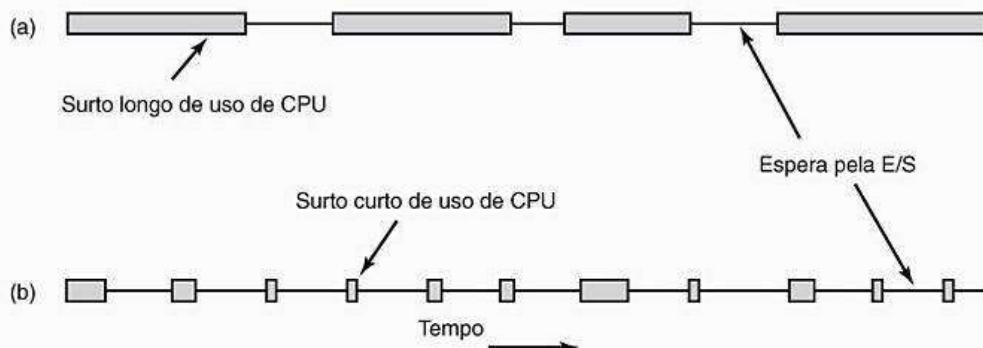
Nesse caso, é comum haver múltiplos processos competindo pela CPU, e, portanto, o escalonamento torna-se importante novamente. Por exemplo, quando a CPU precisar decidir entre executar um processo que reúne estatísticas diárias e um que atende às solicitações dos usuários, estes ficarão muito mais satisfeitos se o último tiver precedência na CPU.

Além de escolher o processo certo para executar, o escalonador também deve se preocupar em fazer um uso eficiente da CPU, pois chavear processos é muito custoso. De início, deve ocorrer um chaveamento do modo de usuário para o modo núcleo. Depois, o estado atual do processo deve ser salvo, armazenando-se inclusive seus registradores na tabela de processos, para que possam ser recarregados posteriormente. Em muitos sistemas, o mapa de memória (por exemplo, os bits de referência à memória na tabela de páginas) também deve ser salvo. Em seguida, um novo processo precisa ser selecionado pela execução do algoritmo de escalonamento. Depois disso, a MMU (*memory management unit* — unidade de gerenciamento de memória) tem de ser recarregada com o mapa de memória do novo processo. Por fim, o novo processo precisa ser iniciado. Além disso tudo, o chaveamento do processo normalmente invalida toda a memória cache, forçando-a a ser dinamicamente recarregada da memória principal por duas vezes (ao entrar no núcleo e ao sair dele). De modo geral, realizar muitos chaveamentos de processos por segundo pode comprometer uma grande quantidade do tempo de CPU; portanto, todo cuidado é pouco.

### Comportamento do processo

Quase todos os processos alternam surtos de computação com requisições de E/S (de disco), conforme mostra a Figura 2.31. Em geral, a CPU executa indefinidamente e então é feita uma chamada de sistema para ler de um arquivo ou escrever nele. Quando a chamada de sistema termina, a CPU computa novamente até que ela requisite ou tenha de escrever mais dados, e assim continua. Perceba que algumas atividades de E/S contam como computação. Por exemplo, quando a CPU copia bits para uma RAM de vídeo a fim de atualizar a tela, ela está computando, não fazendo E/S, pois a CPU se encontra em uso. E/S, nesse sentido, é o que ocorre quando um processo entra no estado bloqueado esperando que um dispositivo externo termine o que está fazendo.

O que é importante observar na Figura 2.31 é que alguns processos, como os da Figura 2.31(a), gastam a maior parte do tempo computando, enquanto outros, como os da Figura 2.31(b), passam a maior parte de seu tempo esperando E/S. Os primeiros são chamados **limitados pela CPU** (*compute-bound* ou *CPU-bound*); os últimos são os **limitados pela E/S** (*I/O-bound*). Os processos limitados pela CPU apresentam, em geral, longos surtos de uso da CPU e espórdicas esperas por E/S; já os processos limitados por E/S têm pequenos surtos de uso da CPU e esperas frequentes



**Figura 2.31** Usos de surtos de CPU se alternam com períodos de espera por E/S. (a) Um processo orientado à CPU. (b) Um processo orientado à E/S.

por E/S. Note que o fator principal é o tamanho do surto de CPU, não o tamanho do surto de E/S. Os processos orientados à E/S são assim chamados porque, entre uma requisição e outra por E/S, eles não realizam muita computação, não porque tenham requisições por E/S especialmente demoradas. O tempo para a leitura de um bloco de disco é sempre o mesmo, independentemente do quanto demore processar os dados que chegam depois.

Convém observar que, à medida que as CPUs se tornam mais rápidas, os processos tendem a ficar mais limitados por E/S. Esse efeito ocorre porque as CPUs estão ficando muito mais rápidas que os discos. Como consequência, o escalonamento de processos orientados à E/S deverá ser um assunto mais importante no futuro. A ideia básica é que, se um processo orientado à E/S quiser executar, essa oportunidade deve ser rapidamente dada a ele, pois assim ele executará suas requisições de disco, mantendo o disco ocupado. Como vimos na Figura 2.4, quando os processos são orientados à E/S, são necessários alguns deles para manter a CPU totalmente ocupada.

### Quando escalonar

Um tópico fundamental, relacionado ao escalonamento, é o momento certo de tomar as decisões de escalonar. É claro que há uma variedade de situações nas quais o escalonamento é necessário. Primeiro, quando se cria um novo processo, é necessário tomar uma decisão entre executar o processo pai ou o processo filho. Como ambos os processos estão no estado pronto, essa é uma decisão normal de escalonamento e pode levar à escolha de um ou de outro — isto é, o escalonador pode escolher legitimamente executar o pai ou o filho.

Em segundo lugar, uma decisão de escalonamento deve ser tomada ao término de um processo. Como o processo não pode executar mais (já que ele não existe mais), algum outro processo deve ser escolhido entre os processos prontos. Se nenhum processo estiver pronto, é executado um processo ocioso gerado pelo sistema.

Em terceiro lugar, quando um processo bloqueia para E/S, sobre um semáforo ou por alguma outra razão, outro

processo precisa ser selecionado para executar. O motivo do bloqueio pode, algumas vezes, influenciar na escolha. Por exemplo, se *A* for um processo importante e estiver esperando *B* sair de sua região crítica, deixar *B* executar em seguida permitirá que ele saia de sua região crítica e, portanto, permite que *A* continue. O problema, contudo, é que geralmente o escalonador não possui a informação necessária para considerar essa dependência.

Em quarto lugar, quando ocorre uma interrupção de E/S, pode-se tomar uma decisão de escalonamento. Se a interrupção vem de um dispositivo de E/S que acabou de fazer seu trabalho, o processo que estava bloqueado, esperando pela E/S, pode agora ficar pronto para execução. É o escalonador quem decide se executa o processo que acabou de ficar pronto, o processo que estava executando no momento da interrupção ou algum terceiro processo.

Se um hardware de relógio fornece interrupções periódicas a 50 Hz, 60 Hz ou alguma outra frequência, uma decisão de escalonamento pode ser tomada a cada interrupção de relógio ou a cada  $k$ -ésima interrupção de relógio. Os algoritmos de escalonamento podem ser divididos em duas categorias quanto ao modo como tratam essas interrupções. Um algoritmo de escalonamento **não preemptivo** escolhe um processo para executar e, então, o deixa executar até que seja bloqueado (à espera de E/S ou de um outro processo) ou até que ele voluntariamente libere a CPU. Mesmo que ele execute por horas, não será compulsoriamente suspenso. Na verdade, nenhuma decisão de escalonamento é tomada durante as interrupções de relógio. Depois que o processamento da interrupção de relógio termina, o processo que estava executando antes da interrupção prossegue até acabar, a menos que um processo de prioridade mais alta esteja esperando por um tempo de espera agora satisfeito.

Por outro lado, um algoritmo de escalonamento **preemptivo** escolhe um processo e o deixa em execução por um tempo máximo fixado. Se ainda estiver executando ao final desse intervalo de tempo, o processo será suspenso e o escalonador escolherá outro processo para executar (se houver algum disponível). O escalonamento preemptivo

requer a existência de uma interrupção de relógio ao fim do intervalo de tempo para que o controle sobre a CPU seja devolvido ao escalonador. Se não houver relógio disponível, o escalonamento não preemptivo será a única opção.

### Categorias de algoritmos de escalonamento

É claro que, para ambientes diferentes, são necessários diferentes algoritmos de escalonamento. Essa situação ocorre porque diferentes áreas de aplicação (e diferentes tipos de sistemas operacionais) têm objetivos diferentes. Em outras palavras, o que deve ser otimizado pelo escalonador não é o mesmo para todos os sistemas. Três ambientes merecem distinção:

1. Lote.
2. Interativo.
3. Tempo real.

Os sistemas em lote ainda são amplamente utilizados pelas empresas para folhas de pagamento, estoque, contas a receber, contas a pagar, cálculo de juros (em bancos), processamento de pedidos de indenização (em companhias de seguros) e outras tarefas periódicas. Nos sistemas em lote não há, em seus terminais, usuários esperando impacientes por uma resposta rápida. Consequentemente, os algoritmos não preemptivos ou preemptivos com longo intervalo de tempo para cada processo são, em geral, aceitáveis. Essa tática reduz os chaveamentos entre processos e assim melhora o desempenho. Na verdade, os algoritmos de lote são bastante comuns e muitas vezes aplicáveis a outras situações também, o que torna importante estudá-los, até para pessoas que não estejam envolvidas em computação central corporativa.

Em um ambiente com usuários interativos, a preempção é essencial para evitar que um processo se aposse da CPU e, com isso, negue serviço aos outros. Mesmo que nenhum processo execute intencionalmente para sempre, uma fa-

lha em um programa pode levar um processo a impedir indefinidamente que todos os outros executem. A preempção é necessária para impedir esse comportamento. Os servidores também caem nessa categoria, visto que normalmente servem a usuários (remotos) múltiplos, todos muito apressados.

Em sistemas com restrições de tempo real, a preempção é, estranhamente, algumas vezes desnecessária, pois os processos sabem que não podem executar por longos períodos e, em geral, fazem seus trabalhos e bloqueiam rapidamente. A diferença com relação aos sistemas interativos é que os sistemas de tempo real executam apenas programas que visam ao progresso da aplicação. Já os sistemas interativos são de propósito geral e podem executar programas arbitrários não cooperativos ou até mal-intencionados.

### Objetivos do algoritmo de escalonamento

Para projetar um algoritmo de escalonamento, é necessário ter alguma ideia do que um bom algoritmo deve fazer. Alguns objetivos dependem do ambiente (lote, interativo ou tempo real), mas há também aqueles que são desejáveis para todos os casos. Alguns objetivos são relacionados na Tabela 2.8. Discutiremos isso logo a seguir.

Em qualquer circunstância, justiça é algo importante. Processos semelhantes devem ter serviços semelhantes. Não é justo dar mais tempo de CPU a um processo do que a outro equivalente. É claro que categorias diferentes de processos podem ser tratadas de modo muito diverso. Pense no controle de segurança e na folha de pagamento de um centro de computação de uma usina nuclear.

De alguma maneira relacionada à justiça está o cumprimento das políticas do sistema. Se a política local estabelece que os processos do controle de segurança executam quando quiserem, mesmo que isso signifique que a folha de pagamento atrasse 30 segundos, o escalonador deve assegurar que essa política seja cumprida.

#### Todos os sistemas

- Justiça — dar a cada processo uma porção justa da CPU
- Aplicação da política — verificar se a política estabelecida é cumprida
- Equilíbrio — manter ocupadas todas as partes do sistema

#### Sistemas em lote

- Vazão (*throughput*) — maximizar o número de tarefas por hora
- Tempo de retorno — minimizar o tempo entre a submissão e o término
- Utilização de CPU — manter a CPU ocupada o tempo todo

#### Sistemas interativos

- Tempo de resposta — responder rapidamente às requisições
- Proporcionalidade — satisfazer às expectativas dos usuários

#### Sistemas de tempo real

- Cumprimento dos prazos — evitar a perda de dados
- Previsibilidade — evitar a degradação da qualidade em sistemas multimídia

I Tabela 2.8 Alguns objetivos do algoritmo de escalonamento sob diferentes circunstâncias.

Outro objetivo geral é manter, quando possível, todas as partes do sistema ocupadas. Se a CPU e os demais dispositivos de E/S puderem ser mantidos em execução o tempo todo, mais trabalho por segundo será feito do que se algum dos componentes estiver ocioso. Em um sistema em lote, por exemplo, o escalonador tem o controle de quais tarefas são trazidos para a memória para executar. É melhor ter juntos na memória alguns processos limitados pela CPU e outros limitados por E/S do que carregar todas as tarefas limitadas pela CPU primeiro e quando terminarem, carregar e executar todas as tarefas limitadas por E/S. Se a última estratégia for usada, quando os processos orientados à CPU estiverem executando, disputarão a CPU e, assim, o disco ficará ocioso. Em seguida, quando as tarefas limitadas por E/S executarem, disputarão o disco e a CPU se encontrará ociosa. É melhor manter o sistema todo executando de uma vez formulando-se cuidadosamente essa mistura de processos.

Os gerentes de grandes centros de computação — que executam muitas tarefas em lote — observam, em geral, três métricas para verificar se os sistemas deles estão executando bem ou não: vazão, tempo de retorno e utilização da CPU. **Vazão** é o número de tarefas por hora que o sistema termina. Considerando tudo o que foi discutido, terminar 50 tarefas por hora é melhor do que terminar 40 no mesmo período. O **tempo de retorno** é estatisticamente o tempo médio do momento em que uma tarefa em lote é submetido até o momento em que ele é terminado. Ele indica quanto tempo, em média, o usuário tem de esperar pelo fim de um trabalho. Aqui a regra é: quanto menor, melhor.

Um algoritmo de escalonamento que maximize a vazão pode não necessariamente minimizar o tempo de retorno. Por exemplo, dada uma mistura de tarefas curtas e longas, um escalonador que execute sempre tarefas curtas e nunca tarefas longas pode conseguir uma excelente vazão (muitas tarefas curtas por hora), mas à custa de um enorme tempo de retorno para as tarefas longas. Se as tarefas curtas mantiverem uma taxa de chegada constante, as tarefas longas poderão nunca executar, tornando o tempo médio de retorno infinito, embora atingindo alta vazão.

A utilização da CPU também é muitas vezes usada como parâmetro em sistemas em lote. Na verdade, esse não é um bom parâmetro. O que realmente interessa é quantas tarefas por hora saem do sistema (vazão) e quanto tempo leva para receber o resultado do trabalho (tempo de retorno). Tomar a utilização da CPU como medida é o mesmo que avaliar um carro pelo número de giros que seu motor dá a cada hora. Por outro lado, saber quando a utilização da CPU está se aproximando de 100 por cento é útil para identificar o momento de obter mais potência para o computador.

Para sistemas interativos, aplicam-se objetivos diferentes. O que importa é minimizar o **tempo de resposta**, isto

é, o tempo entre a emissão de um comando e a obtenção do resultado. Em um computador pessoal, no qual está sendo executado um processo em segundo plano (por exemplo, lendo e armazenando mensagens de correio eletrônico da rede), uma requisição de usuário, para inicializar um programa ou abrir um arquivo, deveria ter precedência sobre o trabalho em segundo plano. Atender antes a todas as requisições interativas será considerado um bom serviço.

Uma questão relacionada a esse tópico é a chamada **proporcionalidade**. Os usuários têm uma intuição (mas muitas vezes errada) de quanto tempo as coisas devem durar. Quando uma requisição tida como complexa demora, os usuários aceitam isso, mas, quando a demora ocorre com uma requisição considerada simples, os usuários ficam irritados. Por exemplo, se, ao clicar em um ícone que inicia o envio de um fax, são necessários 60 segundos para concluir-lo, o usuário provavelmente encarárás isso como inevitável porque não espera que um fax seja enviado em cinco segundos.

Por outro lado, quando o usuário clicar em um ícone para interromper a conexão telefônica após o envio do fax, ele tem expectativas diferentes. Se não tiver sido concluído após 30 segundos, o usuário provavelmente começará a reclamar e, após 60 segundos, estará espumando de raiva. Esse comportamento é causado pela inevitável comparação que o usuário faz que realizar uma chamada telefônica e passar um fax deveria demorar muito mais do que desligar o telefone. Em alguns casos (como esse), o escalonador não pode fazer nada com relação ao tempo de resposta, mas em outros casos sim, especialmente naqueles em que o atraso é decorrente de uma má escolha da ordem dos processos.

Sistemas de tempo real têm propriedades diferentes dos sistemas interativos e, portanto, objetivos diferentes. Eles são caracterizados por prazos que devem — ou pelo menos deveriam — ser cumpridos. Por exemplo, em um computador encarregado de controlar um dispositivo que produz dados a uma taxa constante, uma falha ao executar o processo de coleta de dados em tempo hábil pode resultar na perda de dados. Assim, a principal exigência de um sistema de tempo real é cumprir todos os prazos (ou a maior parte deles).

Em alguns sistemas de tempo real, especialmente naqueles que envolvem multimídia, a previsibilidade é importante. Deixar de cumprir um prazo ocasional não é fatal, mas, se o processo de áudio, por exemplo, executar erraticamente, a qualidade do som vai deteriorar rápido. O vídeo também é um problema, mas o ouvido é muito mais sensível a atrasos que a visão. Para evitar esse problema, o escalonamento de processos deve ser altamente previsível e regular. Ainda neste capítulo, estudaremos os algoritmos de escalonamento em lote e interativos, mas adiaremos a maior parte de nosso estudo sobre o escalonamento em tempo real para o Capítulo 7, que trata de sistemas operacionais multimídia.

### 2.4.2 | Escalonamento em sistemas em lote

É chegada a hora de passar dos tópicos gerais de escalonamento para os algoritmos específicos desse processo. Nesta seção, estudaremos os algoritmos usados em sistemas em lote. Em seguida, veremos sistemas interativos e de tempo real. Convém ressaltar que alguns algoritmos são usados tanto em sistemas em lote quanto em sistemas interativos. Esses serão estudados depois. Agora, o foco será mantido sobre algoritmos adequados somente a sistemas em lote.

#### Primeiro a chegar, primeiro a ser servido

Provavelmente o mais simples algoritmo de escalonamento seja o não preemptivo **primeiro a chegar, primeiro a ser servido** (*first come, first served — FCFS*). Com esse algoritmo, a CPU é atribuída aos processos na ordem em que eles a requisitam. Basicamente, há uma fila única de processos prontos. Quando a primeira tarefa entra no sistema, logo quando chega de manhã, é iniciado imediatamente e autorizado a executar por quanto tempo queira. Ele não é interrompido porque está sendo executado há muito tempo. À medida que chegam as outras tarefas, elas são encaminhadas para o fim da fila. Quando o processo em execução é bloqueado, o próximo processo na fila é o próximo a executar. Quando um processo bloqueado fica pronto — assim como uma tarefa que acabou de chegar —, ele é posto no final da fila.

A grande vantagem desse algoritmo é que ele é fácil de entender e igualmente fácil de programar. É também um algoritmo justo, assim como é justo destinar escassos ingressos para eventos esportivos ou musicais para as pessoas que estejam dispostas a ficar na fila desde as 2 da madrugada. Com esse algoritmo, uma única lista encadeada controla todos os processos prontos. Adicionar uma nova tarefa ou um processo desbloqueado requer apenas a inserção dele no final da fila. O que poderia ser mais simples de entender e implementar?

Infelizmente, o algoritmo *primeiro a chegar, primeiro a ser servido* apresenta uma grande desvantagem. Imagine um processo orientado à computação que execute durante um segundo por vez e muitos outros processos limitados por E/S que usem pouco tempo de CPU, mas que precisem realizar, cada um, mil leituras de disco antes de terminar. O processo orientado à computação executa por um segundo e então lê um bloco de disco (bloqueia). Com esse processo bloqueado à espera de E/S, todos os outros processos limitados por E/S executam e iniciam as leituras de disco. Quando o processo orientado à computação obtém seu bloco de dados (desbloqueia), ele executa por mais um segundo, seguido novamente por todos os processos limitados por E/S, em uma rápida sucessão.

O resultado líquido é que cada um dos processos limitados por E/S lê um bloco por segundo e, portanto, demorará mil segundos para terminar. Com um algoritmo

de escalonamento que causasse a preempção do processo orientado à computação a cada dez milissegundos (em vez de a cada um segundo), os processos de E/S terminariam em dez segundos, e não em mil segundos, sem atrasar tanto o processo orientado à computação.

#### Tarefa mais curta primeiro

Vejamos um outro algoritmo em lote não preemptivo que supõe como previamente conhecidos todos os tempos de execução. Em uma companhia de seguros, por exemplo, as pessoas podem prever, com bastante precisão, quanto tempo será necessário para executar um lote de mil solicitações, já que um trabalho similar é feito todos os dias. Quando várias tarefas igualmente importantes estiverem postadas na fila de entrada à espera de serem iniciados, o escalonador escolhe a **tarefa mais curto primeiro** (*shortest job first*). Veja a Figura 2.32. Nela encontramos quatro tarefas — A, B, C e D — com seus respectivos tempos de execução — 8, 4, 4 e 4 minutos, respectivamente. Ao executá-los nessa ordem, o tempo de retorno para A é de oito minutos, para B é de 12 minutos, para C é de 16 minutos e para D é de 20 minutos, o que resulta em uma média de 14 minutos.

Consideremos agora a execução desses quatro tarefas a partir do algoritmo *tarefa mais curta primeiro*, conforme ilustrado na Figura 2.32(b). Os tempos de retorno são agora 4, 8, 12 e 20 minutos, com uma média de 11 minutos. A *tarefa mais curta primeiro* parece ótimo, não? Considere o caso de quatro tarefas, com tempos de execução a, b, c e d, respectivamente. A primeira tarefa termina no tempo a, o segundo termina no tempo  $a + b$  e assim por diante. O tempo médio de retorno é  $(4a + 3b + 2c + d)/4$ . É claro que a contribui mais para a média que os outros tempos; portanto, ele deveria ser a tarefa mais curto, com o b depois, então o c e, por fim, o d — sendo este o mais demorado e que afeta somente seu próprio tempo de retorno. O mesmo argumento se aplica igualmente bem a qualquer número de tarefas.

Convém observar que a *tarefa mais curta primeiro* é adequado somente para situações em que todas as tarefas estejam disponíveis simultaneamente. Como um contraexemplo,

8	4	4	4
A	B	C	D

(a)

4	4	4	8
B	C	D	A

(b)

**Figura 2.32** Um exemplo do escalonamento *tarefa mais curta primeiro*. (a) Execução de quatro tarefas na ordem original. (b) Execução na ordem *tarefa mais curta primeiro*.

considere cinco tarefas, de *A* a *E*, com tempos de execução 2, 4, 1, 1 e 1, respectivamente. Seus tempos de chegada são 0, 0, 3, 3 e 3. De início, somente *A* ou *B* podem ser escolhidas, já que os outras três tarefas ainda não chegaram. Usando a *tarefa mais curto primeiro*, executaremos as tarefas na ordem *A, B, C, D, E* para um tempo médio de espera de 4,6. Contudo, executá-los na ordem *B, C, D, E, A* implica um tempo médio de espera de 4,4.

### Próximo de menor tempo restante

Uma versão preemptiva da *tarefa mais curta primeiro* é o **próximo de menor tempo restante** (*shortest remaining time next*). Com esse algoritmo, o escalonador sempre escolhe o processo cujo tempo de execução restante seja o menor. Novamente, o tempo de execução deve ser previamente conhecido. Quando chega uma nova tarefa, seu tempo total é comparado ao tempo restante do processo em curso. Se, para terminar, a nova tarefa precisar de menos tempo que o processo corrente, então esse será suspenso e a nova tarefa será iniciado. Esse esquema permite que novas tarefas curtos obtenham um bom desempenho.

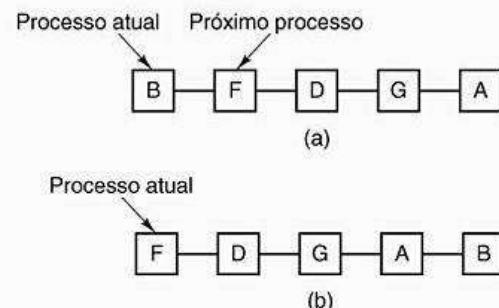
### 2.4.3 Escalonamento em sistemas interativos

Vejamos então alguns algoritmos aplicados a sistemas interativos. Eles são comuns em computadores pessoais, servidores e outros tipos de sistemas também.

#### Escalonamento por chaveamento circular (*round-robin*)

Um dos algoritmos mais antigos, simples, justos e amplamente usados é o **circular**. A cada processo é atribuído um intervalo de tempo, o seu **quantum**, no qual ele é permitido executar. Se, ao final do quantum, o processo ainda estiver executando, a CPU sofrerá preempção e será dada a outro processo. Se o processo foi bloqueado ou terminou antes que o quantum tenha decorrido, a CPU é chaveada para outro processo. O escalonamento circular é fácil de implementar. O escalonador só precisa manter uma lista de processos executáveis, conforme mostra a Figura 2.33(a). Quando o processo usa todo o seu quantum, ele é colocado no final da lista, como mostra a Figura 2.33(b).

O que interessa para o escalonamento circular é o tamanho do quantum. O chaveamento de um processo para outro requer uma certa quantidade de tempo para sua administração — salvar e carregar registradores e mapas de memória, atualizar várias listas e tabelas, carregar e descartar a memória cache etc. Suponha que esse **chaveamento de processo** — ou **chaveamento de contexto**, como é algumas vezes chamado — dure 1 ms, incluindo o chaveamento dos mapas de memória, descarga e recarga da cache etc. Suponha também que o quantum seja de 4 ms. Com esses parâmetros, depois de fazer 4 ms de trabalho útil, a CPU



**Figura 2.33** Escalonamento circular (*round robin*). (a) Lista de processos executáveis. (b) Lista de processos executáveis depois que *B* usou todo o seu quantum.

terá de gastar (ou melhor, desperdiçar) 1 ms para chavear o processo. Nesse exemplo, 20 por cento do tempo de CPU será gasto em administração, o que sem dúvida é demais.

Para melhorar a eficiência da CPU, poderíamos estabelecer o valor do quantum em, digamos, 100 ms. Agora, o tempo gasto é de apenas 1 por cento. No entanto, considere o que pode acontecer em um sistema de tempo compartilhado se 50 solicitações forem feitas dentro de um curto intervalo de tempo e com grande variação nas necessidades de CPU. Cinquenta processos serão colocados na lista de processos executáveis. Se a CPU estiver ociosa, o primeiro dos processos inicializará imediatamente, o segundo não poderá inicializar enquanto não se passarem 100 ms e assim por diante. O último azarado pode ter de esperar cinco segundos antes de ter uma oportunidade — supondo que todos os outros usem inteiramente seus quanta<sup>3</sup>. A maioria dos usuários verá como problema uma resposta se um pequeno comando demorar cinco segundos. Essa situação é especialmente ruim se alguma das solicitações próximas ao fim da fila exigir apenas alguns milissegundos de tempo da CPU. Com um quantum curto, os usuários teriam obtido o melhor serviço.

Outro fator é o seguinte: se o quantum for maior que o surto médio de CPU, a preempção raramente ocorrerá. Na verdade, a maior parte dos processos bloqueará antes que o quantum acabe, causando um chaveamento de processo. Eliminar a preempção melhora o desempenho porque o chaveamento de processo somente ocorre quando é logicamente necessário, isto é, quando um processo bloqueia e não é mais capaz de continuar.

A conclusão pode ser formulada assim: adotar um quantum muito curto causa muitos chaveamentos de processo e reduz a eficiência da CPU, mas um quantum muito longo pode gerar uma resposta pobre às requisições interativas curtas. Um quantum em torno de 20 ms a 50 ms é bastante razoável.

#### Escalonamento por prioridades

O escalonamento circular pressupõe que todos os processos sejam igualmente importantes. É frequente as pes-

3. Quanta = plural de quantum (N. R. T.).

soas que possuem e operam computadores multiusuário pensarem de modo diferente sobre o assunto. Em uma universidade, por exemplo, uma ordem hierárquica seria encabeçada pelo reitor, e então viriam os professores, os secretários, os porteiros e finalmente os estudantes. Da necessidade de se considerarem fatores externos resulta o **escalonamento por prioridades**. A ideia básica é simples: a cada processo é atribuída uma prioridade, e ao processo executável com a prioridade mais alta é permitido executar.

Mesmo em um PC com um único proprietário, pode haver múltiplos processos, alguns mais importantes que outros. Por exemplo, a um processo daemon, que envia mensagens de correio eletrônico em segundo plano, deve ser atribuída uma prioridade mais baixa que a um processo que exibe um vídeo na tela em tempo real.

Para evitar que processos de alta prioridade executem indefinidamente, o escalonador pode reduzir a prioridade do processo em execução a cada tique de relógio (isto é, a cada interrupção de relógio). Se isso fizer com que sua prioridade caia abaixo da prioridade do próximo processo com prioridade mais alta, então ocorrerá um chaveamento de processo. Outra possibilidade é atribuir a cada processo um quantum máximo no qual ele pode executar. Quando esse quantum estiver esgotado, será dada a oportunidade para que o próximo processo com prioridade mais alta execute.

Prioridades podem ser atribuídas aos processos estática ou dinamicamente. Em um computador militar, os processos iniciados por generais podem partir com prioridade em 100; os processos iniciados por coronéis, em 90; os de maiores, em 80; os de capitães, em 70; os de tenentes, em 60, e assim por diante. De outra forma, em um centro de computação comercial, trabalhos de alta prioridade podem custar cem dólares por uma hora; um de prioridade média, 75 dólares por hora; e os de prioridade baixa, 50 dólares pelo mesmo período. O sistema UNIX tem um comando, *nice*, que permite que um usuário reduza voluntariamente a prioridade de seu processo e, assim, seja gentil com os outros usuários. Usuários nunca o utilizam.

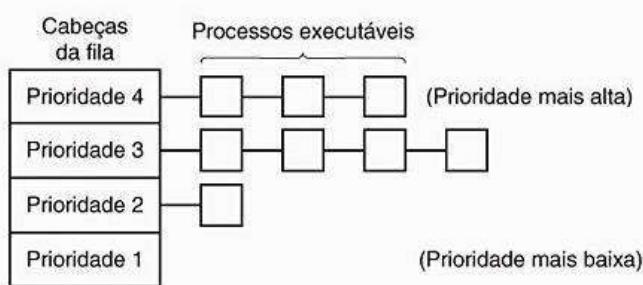
O sistema também pode atribuir dinamicamente as prioridades para atingir certos objetivos. Por exemplo, alguns processos são altamente orientados à E/S e gastam a maior parte de seu tempo esperando que uma E/S termine. Se um processo como esse quisesse a CPU, deveria recebê-la imediatamente, para deixá-lo inicializar sua próxima requisição de E/S, a qual poderia então continuar em paralelo com outro processo que estivesse realmente computando. Fazer o processo orientado à E/S esperar um longo tempo pela CPU significa tê-lo ocupando a memória por tempo demais desnecessariamente. Um algoritmo simples e que funciona bem para processos orientados à E/S é atribuir  $1/f$  à prioridade, sendo  $f$  a fração do último quantum que o processo usou. Um processo que tivesse utilizado somente 1 ms de seu quantum de 50 ms obteria priorida-

de 50, enquanto um processo que executa 25 ms antes de bloquear teria prioridade 2, e um processo que houvesse consumido todo o quantum teria prioridade 1.

Muitas vezes é conveniente agrupar processos em classes de prioridade e usar o escalonamento por prioridades entre as classes — contudo, dentro de cada classe, usar o escalonamento circular. A Figura 2.34 mostra um sistema com quatro classes de prioridade. O algoritmo de escalonamento é o seguinte: enquanto houver processos executáveis na classe de prioridade 4, execute apenas um por quantum usando escalonamento circular e nunca perca tempo com as classes de baixa prioridade. Se a classe de prioridade 4 estiver vazia (sem processos para executar), então execute os processos da classe 3 em chaveamento circular. Se as classes 4 e 3 estiverem ambas vazias, então execute a classe 2 em chaveamento circular, e assim por diante. Se as prioridades não forem ocasionalmente ajustadas, as classes de prioridade mais baixas poderão todas morrer de fome.

### Filas múltiplas

Um dos primeiros escalonadores por prioridades foi implementado no CTSS, o sistema compatível de tempo compartilhado do MIT que operava no IBM 7094 (Corbató et al., 1962). O CTSS tinha um problema: o chaveamento de processo era muito lento porque o 7094 só podia manter na memória um processo por vez. Cada chaveamento significava trocar todo o processo, ou seja, enviá-lo para o disco e ler outro do disco. Os projetistas do CTSS logo perceberam que era mais eficiente dar, de vez em quando, um quantum grande para os processos limitados pela CPU do que fornecer frequentemente um quantum pequeno (para reduzir as operações de troca entre o disco e a memória). Por outro lado, dar a todos os processos um quantum grande significaria ter um tempo de resposta inadequado, conforme vimos. A solução, então, foi definir classes de prioridade. Os processos na classe de prioridade mais alta eram executados por um quantum. Os processos na classe seguinte de prioridade mais alta executavam por dois quanta. Os processos na próxima classe executavam por quatro quanta e assim por diante. Se um processo utilizasse todos os seus quanta, seria movido para uma classe inferior.



**Figura 2.34** Um algoritmo de escalonamento com quatro classes de prioridade.

Como exemplo, imagine um processo que precisasse computar continuamente por 100 quanta. Inicialmente, a ele seria dado um quantum, e ele então seria levado da memória para o disco (troca para o disco). Na vez seguinte ele teria dois quanta antes de ocorrer a troca para o disco. As próximas execuções obteriam 4, 8, 16, 32 e 64 quanta, embora ele tivesse usado apenas 37 dos últimos 64 quanta para realizar seu trabalho. Seriam necessárias somente sete trocas entre a memória e o disco (incluindo a carga inicial), em vez de cem para um algoritmo puramente circular. Além disso, à medida que o processo se aprofundasse mais nas filas de prioridade, ele seria cada vez menos frequentemente executado, liberando a CPU para processos interativos e rápidos.

Foi então adotada a seguinte política para impedir uma longa punição a um processo que, quando iniciado pela primeira vez, precisasse executar por um longo intervalo de tempo, mas que depois se tornasse interativo. Se fosse digitado um <Enter> em um terminal, o processo pertencente àquele terminal era movido para a classe de prioridade mais alta, na suposição de que ele estivesse prestes a se tornar interativo. Certo dia, algum usuário com um processo pesadamente limitado pela CPU descobriu que, sentando ao terminal e digitando <Entra> de maneira aleatória e a intervalos de poucos segundos, poderia fazer maravilhas por seu tempo de resposta. Ele contou isso para todos os seus amigos. Moral da história: conseguir acertar na prática é muito mais difícil que acertar na teoria.

Muitos outros algoritmos foram usados para atribuir processos a classes de prioridade. Por exemplo, o influente sistema XDS 940 (Lampson, 1968), construído em Berkeley, possuía quatro classes de prioridade: terminal, E/S, quantum curto e quantum longo. Quando um processo que estivesse esperando pela entrada de um terminal finalmente acordasse, ele iria para classe de prioridade mais alta (terminal). Quando um processo bloqueado pelo disco ficasse pronto, ele iria para a segunda classe. Se, enquanto um processo ainda estivesse executando, seu quantum acabasse, seria inicialmente alocado na terceira classe. Contudo, se um processo terminasse seu quantum várias vezes sem ser bloqueado pelo terminal ou por outra E/S, iria para a última fila. Muitos outros sistemas usam algo semelhante para favorecer os usuários interativos mais do que os processos em segundo plano.

### Próximo processo mais curto (*shortest process next*)

Como a tarefa mais curta primeiro sempre resulta no mínimo tempo médio de resposta para sistemas em lote, seria bom se ele também pudesse ser usado para processos interativos. Até certo ponto, isso é possível. Processos interativos geralmente seguem o padrão de esperar por comando, executar comando, esperar por comando, executar comando e assim por diante. Se vissemos a execução de cada comando como uma ‘tarefa’ isolado, então poderíamos minimizar o tempo de resposta geral executando a tarefa mais

curta primeiro. O único problema é saber qual dos processos atualmente executáveis é o mais curto.

Uma saída é realizar uma estimativa com base no comportamento passado e, então, executar o processo cujo tempo de execução estimado seja o menor. Suponha que o tempo estimado por comando para algum terminal seja  $T_0$  e que sua próxima execução seja medida como  $T_1$ . Podríamos atualizar nossa estimativa tomando uma soma ponderada desses dois números, isto é,  $aT_0 + (1 - a)T_1$ . Pela escolha de  $a$ , podemos decidir se o processo de estimativa esquecerá rapidamente as execuções anteriores ou se lembrará delas por um longo tempo. Com  $a = 1/2$ , obtemos estimativas sucessivas de

$$T_0, T_0/2 + T_1/2, T_0/4 + T_1/4 + T_2/2, T_0/8 + T_1/8 + T_2/4 + T_3/2$$

Depois de três novas execuções, o peso de  $T_0$  na nova estimativa caiu para 1/8.

A técnica de estimar o valor seguinte da série, tomando a média ponderada do valor sendo medido e a estimativa anterior, é algumas vezes chamada de **aging** (envelhecimento). Essa técnica é aplicável a muitas situações nas quais é preciso uma previsão baseada nos valores anteriores. Aging é especialmente fácil de implementar quando  $a = 1/2$ . Basta apenas adicionar o novo valor à estimativa atual e dividir a soma por 2 (deslocando 1 bit à direita).

### Escalonamento garantido

Um método completamente diferente de lidar com o escalonamento é fazer promessas reais sobre o desempenho aos usuários e, então, satisfazê-los. Uma promessa realista e fácil de cumprir é esta: se houver  $n$  usuários conectados enquanto você estiver trabalhando, você receberá cerca de  $1/n$  de CPU. De modo semelhante, em um sistema monousuário com  $n$  processos em execução, todos iguais, cada um deve receber  $1/n$  ciclos de CPU. Isso parece suficientemente justo.

Para fazer valer essa promessa, o sistema deve manter o controle da quantidade de CPU que cada processo recebe desde sua criação. Ele então calcula a quantidade de CPU destinada a cada um ou simplesmente o tempo desde a criação dividido por  $n$ . Como a quantidade de tempo de CPU que cada processo realmente teve é também conhecida, torna-se fácil calcular a taxa entre o tempo de CPU de fato consumido e o tempo de CPU destinado a cada processo. Uma taxa de 0,5 significa que um processo teve somente a metade do que ele deveria ter, e uma taxa de 2,0 significa que um processo teve duas vezes mais do que lhe foi destinado. O algoritmo então executará o processo com a taxa mais baixa, até que sua taxa cresça e se aproxime da de seu competidor.

### Escalonamento por loteria

Fazer promessas aos usuários e satisfazê-los é uma boa ideia, porém difícil de implementar. Contudo, um outro algoritmo pode ser usado com resultados similarmente previsí-

veis, mas de implementação muito mais simples. É o chamado **escalonamento por loteria** (Waldspurger e Weihl, 1994).

A ideia básica é dar bilhetes de loteria aos processos, cujos prêmios são vários recursos do sistema, como tempo de CPU. Se houver uma decisão de escalonamento, um bilhete de loteria será escolhido aleatoriamente e o processo que tem o bilhete conseguirá o recurso. Quando aplicado ao escalonamento de CPU, o sistema pode fazer um sorteio 50 vezes por segundo e, portanto, cada vencedor terá 20 ms de tempo de CPU como prêmio.

Parafraseando George Orwell: "Todos os processos são iguais, mas alguns são mais iguais que os outros". Aos processos mais importantes podem ser atribuídos bilhetes extras para aumentar suas probabilidades de vitória. Se houver cem bilhetes extras e um processo detiver 20 deles, esse processo terá uma chance de 20 por cento de vencer cada loteria. Ao longo da execução, ele obterá 20 por cento da CPU. Diferentemente de um escalonador por prioridades, no qual é muito difícil estabelecer o que de fato significa uma prioridade 40, aqui há uma regra clara: um processo que detenha uma fração  $f$  dos bilhetes obterá em torno de uma fração  $f$  do recurso em questão.

O escalonamento por loteria tem várias propriedades interessantes. Por exemplo, se aparece um novo processo e a ele são atribuídos alguns bilhetes, já no próximo sorteio da loteria sua probabilidade de vencer será proporcional ao número de bilhetes que ele tiver. Em outras palavras, o escalonamento por loteria é altamente responsivo.

Os processos cooperativos podem trocar bilhetes entre si, se assim desejarem. Por exemplo, quando um processo cliente envia uma mensagem para um processo servidor e, então, é bloqueado, ele pode dar todos os seus bilhetes ao servidor, para que aumentem as probabilidades de o servidor executar logo. Quando o servidor termina, retorna os bilhetes para o cliente executar novamente. Na verdade, na ausência de clientes, os servidores nem precisam de bilhetes.

O escalonamento por loteria pode ser usado para resolver problemas difíceis de solucionar a partir de outros métodos. Um exemplo é o de um servidor de vídeo, no qual vários processos alimentam o fluxo de vídeo de seus clientes, mas em diferentes taxas de apresentação dos quadros. Suponha que os processos precisem de taxas em 10, 20 e 25 quadros/s. Alocando a esses processos dez, 20 e 25 bilhetes, respectivamente, eles vão automaticamente dividir a CPU aproximadamente na proporção correta, que é 10:20:25.

#### **Escalonamento por fração justa (fair-share)**

Até agora temos partido do pressuposto de que cada processo é escalonado por si próprio, sem nos preocuparmos com quem é seu dono. Como resultado, se o usuário 1 inicia nove processos e o usuário 2 inicia um processo, com chaveamento circular ou com prioridades iguais, o usuário 1 obterá 90 por cento da CPU e o usuário 2 terá somente 10 por cento dela.

Para evitar isso, alguns sistemas consideram a propriedade do processo antes de escaloná-lo. Nesse modelo, a cada usuário é alocada uma fração da CPU, e o escalonador escolhe os processos de modo que garanta essa fração. Assim, se dois usuários tiverem 50 por cento da CPU prometida a cada um deles, cada um obterá os 50 por cento, não importando quantos processos eles tenham gerado.

Como exemplo, imagine um sistema com dois usuários, cada qual com 50 por cento da CPU prometida a ele. O usuário 1 tem quatro processos, *A, B, C e D*, e o usuário 2 tem somente um processo, *E*. Se for usado o escalonamento circular, uma sequência possível de escalonamento que cumpra todas as exigências será a seguinte:

A E B E C E D E A E B E C E D E...

Por outro lado, se ao usuário 1 se destinar duas vezes mais tempo de CPU que para o usuário 2, poderemos obter:

A B E C D E A B E C D E...

É claro que existem inúmeras outras possibilidades, igualmente passíveis de serem exploradas, dependendo da noção de justiça.

#### **2.4.4 | Escalonamento em sistemas de tempo real**

Um sistema de **tempo real** é aquele no qual o tempo tem uma função essencial. Em geral, um ou mais dispositivos físicos externos ao computador geram estímulos, e o computador deve reagir apropriadamente a eles dentro de um dado intervalo de tempo. Por exemplo, o computador em um CD player obtém os bits que chegam do drive e precisa convertê-los em música em um intervalo de tempo muito curto. Se o cálculo que ele fizer for muito demorado, a música soará diferente. Outros exemplos de sistemas de tempo real incluem: monitoração de pacientes em unidades de terapia intensiva de hospitais, piloto automático de aeronaves e robôs de controle em fábricas automatizadas. Em todos esses casos, ter a resposta certa, mas tardia, é tão ruim quanto não ter nada.

Sistemas de tempo real são em geral categorizados como **tempo real crítico**, isto é, há prazos absolutos que devem ser cumpridos ou, então, como **tempo real não crítico**, no qual o descumprimento ocasional de um prazo é indesejável, contudo tolerável. Em ambos os casos, o comportamento de tempo real é implementado dividindo-se o programa em vários processos cujo comportamento é previamente conhecido. De modo geral, esses processos têm vida curta e podem executar em bem menos de um segundo. Quando é detectado um evento externo, o trabalho do escalonador é escalonar os processos de tal maneira que todos os prazos sejam cumpridos.

Os eventos aos quais um sistema de tempo real pode precisar responder podem ser categorizados ainda como **periódicos** (ocorrem em intervalos regulares) ou **aperiódicos**.

**dicos** (acontecem de modo imprevisível). Um sistema pode ter de responder a múltiplos fluxos de eventos periódicos. Dependendo de quanto tempo cada evento requeira para processar, talvez nem seja possível tratar de todos. Por exemplo, se houver  $m$  eventos periódicos e o evento  $i$  ocorrer com período  $P_i$  e requerer  $C_i$  segundos de CPU para tratar cada evento, então a carga poderá ser tratada somente se

$$\sum_{i=1}^m \frac{C_i}{P_i} \leq 1$$

Um sistema de tempo real que satisfaça esse critério é chamado de **escalonável**.

Como exemplo, considere um sistema de tempo real não crítico com três eventos periódicos, com períodos de 100, 200 e 500 ms, respectivamente. Se esses eventos requererem 50, 30 e 100 ms de tempo de CPU por evento, nessa ordem, o sistema é escalonável porque  $0,5 + 0,15 + 0,2 < 1$ . Se um quarto evento, com período de 1 s, for adicionado, o sistema permanecerá escalonável desde que esse evento não precise de mais de 150 ms do tempo de CPU por evento. Está implícita nesse cálculo a hipótese de que o custo extra do chaveamento de contexto é tão pequeno que pode ser desprezado.

Os algoritmos de escalonamento de tempo real podem ser estáticos ou dinâmicos. Os primeiros tomam suas decisões de escalonamento antes de o sistema começar a executar. Os últimos o fazem em tempo de execução. O escalonamento estático só funciona quando há prévia informação perfeita disponível sobre o trabalho necessário a ser feito e os prazos que devem ser cumpridos. Os algoritmos de escalonamento dinâmico não apresentam essas restrições. Adiaremos nosso estudo sobre algoritmos específicos para quando tratarmos de sistemas multimídia de tempo real no Capítulo 7.

#### 2.4.5 | Política versus mecanismo

Até agora, temos presumido tacitamente que todos os processos no sistema pertencem a usuários diferentes e estão, portanto, competindo pela CPU. Embora isso muitas vezes seja verdade, um processo pode ter muitos filhos executando sob seu controle — por exemplo, um processo de um sistema de gerenciamento de bancos de dados. Cada filho pode estar atendendo a uma requisição diferente ou ter uma função específica para realizar (análise sintática de consultas, acesso a disco etc.). É totalmente possível que o processo principal tenha uma ideia clara de quais de seus filhos sejam os mais importantes (ou tenham tempo crítico) e quais sejam os menos importantes. Infelizmente, nenhum dos escalonadores discutidos anteriormente aceita qualquer entrada proveniente de processos do usuário sobre decisões de escalonamento. Como resultado, o escalonador raramente faz a melhor escolha.

A solução para esse problema é separar o **mecanismo de escalonamento da política de escalonamento**, um

princípio estabelecido há muito tempo (Levin et al., 1975). Isso significa que o algoritmo de escalonamento é de algum modo parametrizado, mas os parâmetros podem ser preenchidos pelos processos dos usuários. Consideremos novamente o exemplo do banco de dados. Suponha que o núcleo use um algoritmo de escalonamento por prioridades, mas que disponibilize uma chamada de sistema na qual um processo seja capaz de configurar (e alterar) as prioridades e seus filhos. Desse modo, o pai pode controlar em detalhes como seus filhos são escalonados, mesmo que ele próprio não faça o escalonamento. Nesse exemplo, o mecanismo de escalonamento está no núcleo, mas a política é estabelecida por um processo de usuário.

#### 2.4.6 | Escalonamento de threads

Quando cada um dentre vários processos tem múltiplos threads, ocorrem dois níveis de paralelismo: processos e threads. O escalonamento nesses sistemas pode diferir de modo substancial, dependendo de os threads serem de usuário ou de núcleo (ou ambos).

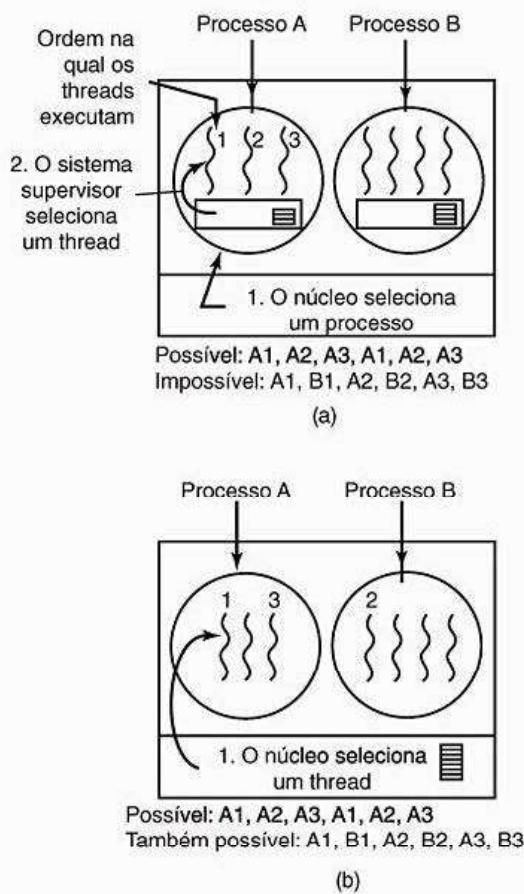
Consideremos primeiro os threads de usuário. Como o núcleo não sabe da existência de threads, ele opera como sempre faz, escolhendo um processo — por exemplo,  $A$  — e dando-lhe o controle de seu quantum. O escalonador do thread em  $A$  decide qual thread deve executar — por exemplo,  $A1$ . Como não há interrupções de relógio para multiprogramar threads, esse thread pode continuar executando enquanto quiser. Se ele usar todo o quantum do processo, o núcleo selecionará um outro processo para executar.

Quando o processo  $A$  finalmente voltar a executar, o thread  $A1$  permanecerá executando. Ele continuará a consumir todo o tempo de  $A$  até que termine. Contudo, seu comportamento antissocial não afetará outros processos: eles obterão aquilo que o escalonador considerar uma fração apropriada, não importando o que esteja acontecendo dentro do processo  $A$ .

Agora, imagine que os threads de  $A$  tenham relativamente pouco trabalho a fazer por surto de CPU — por exemplo, 5 ms de trabalho para um quantum de 50 ms. Consequentemente, cada um executa por um pouquinho de tempo e então cede a CPU de volta para o escalonador de thread. Isso pode levar à sequência  $A1, A2, A3, A1, A2, A3, A1, A2, A3, A1$ , antes de o núcleo chavear para o processo  $B$ . Essa situação é ilustrada na Figura 2.35(a).

O algoritmo de escalonamento usado pelo sistema de tempo de execução pode ser qualquer um dos que acabam de ser descritos. Na prática, o escalonamento circular e o escalonamento por prioridades são os mais comuns. A única limitação é a ausência de uma interrupção de relógio para interromper um thread que esteja executando há muito tempo.

Agora, considere a situação com os threads de núcleo. Nesse caso, o núcleo escolhe um thread para executar. Ele não precisa levar em conta a qual processo o thread pertence, mas, se quiser, pode considerar esse fato. Ao thread é



**Figura 2.35** (a) Escalonamento possível de threads de usuário com um quantum de processo de 50 ms e threads que executam 5 ms por surto de CPU. (b) Escalonamento possível de threads de núcleo com as mesmas características de (a).

dado um quantum, e ele será compulsoriamente suspenso se exceder o quantum. Com um quantum de 50 ms, mas com threads que bloqueiam depois de 5 ms, a ordem dos threads por um período de 30 ms pode ser A1, B1, A2, B2, A3, B3, algo impossível de conseguir com esses parâmetros e com threads de usuário. Essa situação é parcialmente mostrada na Figura 2.35(b).

Uma diferença importante entre os threads de usuário e os threads de núcleo é o desempenho. O chaveamento de thread com threads de usuário usa poucas instruções de máquina. Para os threads de núcleo, o chaveamento requer um chaveamento completo do contexto, com alteração do mapa de memória e invalidação da cache — o que significa uma demora várias ordens de magnitude. Por outro lado, para os threads de núcleo, um thread bloqueado pela E/S não suspende o processo inteiro, como ocorre nos threads de usuário.

Como o núcleo sabe que o chaveamento de um thread no processo A para um thread no processo B custa mais do que executar um segundo thread no processo A (pois terá de mudar o mapa de memória e invalidar a memória

cache), ele pode considerar essa informação quando tomar uma decisão. Por exemplo, dados dois threads igualmente importantes, sendo que um deles pertence ao mesmo processo de um thread que acabou de ser bloqueado e o outro pertence a um processo diferente, a preferência poderia ser dada ao primeiro.

Outro fator importante é que os threads de usuário podem utilizar um escalonador de thread específico para uma aplicação. Considere, por exemplo, o servidor da Web da Figura 2.6. Suponha que um thread operário tenha acabado de ser bloqueado e que o thread despachante e dois threads operários estejam prontos. Qual deveria executar? O sistema de tempo de execução — que sabe o que cada thread faz — pode facilmente escolher o despachante como o próximo thread a executar, para que este coloque outro operário para executar. Essa estratégia maximiza a quantidade de paralelismo em um ambiente no qual os operários frequentemente são bloqueados pela E/S de disco. Já no caso dos threads de núcleo, este nunca saberia o que cada thread fez (embora a eles pudessem ser atribuídas diferentes prioridades). Contudo, em geral os escalonadores de threads específicos para uma aplicação são capazes de ajustar uma aplicação melhor do que o núcleo pode fazê-lo.

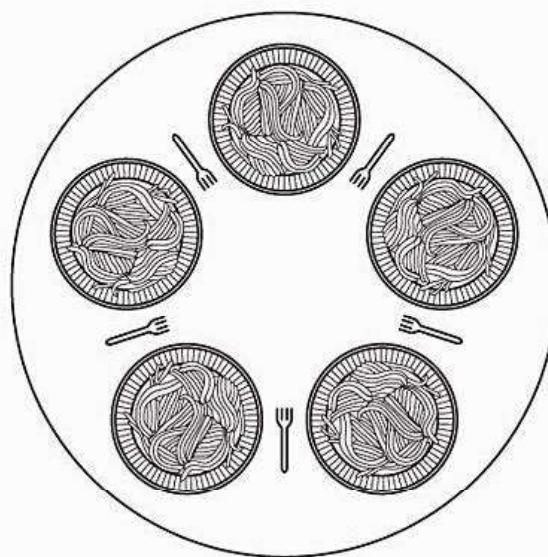
## 2.5 Problemas clássicos de IPC

A literatura sobre sistemas operacionais está repleta de problemas interessantes que têm sido amplamente discutidos e analisados a partir de vários métodos de sincronização. Nas próximas seções examinaremos três desses problemas mais comuns.

### 2.5.1 O problema do jantar dos filósofos

Em 1965, Dijkstra formulou e resolveu um problema de sincronização que ele chamou de **problema do jantar dos filósofos**. Desde então, cada um que inventasse mais uma primitiva de sincronização via-se obrigado a demonstrar até que ponto essa nova primitiva era maravilhosa, mostrando com que elegância ela resolvia o problema do jantar dos filósofos. O problema pode ser explicado de maneira muito simples. Cinco filósofos estão sentados em torno de uma mesa circular. Cada filósofo tem um prato de espaguete. O espaguete está tão escorregadio que um filósofo precisa de dois garfos para comê-lo. Entre cada par de pratos está um garfo. O diagrama da mesa é ilustrado na Figura 2.36.

A vida de um filósofo consiste em alternar períodos de comer e pensar. (Trata-se apenas de uma abstração, mesmo para os filósofos; as outras atividades são irrelevantes ao problema.) Quando uma filósofa fica com fome, ela tenta pegar os garfos à sua direita e à sua esquerda, um de cada vez, em qualquer ordem. Se conseguir pegar dois garfos, ela comerá durante um determinado tempo e, então, colocará os garfos na mesa novamente e continuará a pen-



**Figura 2.36** Hora do almoço no Departamento de Filosofia.

sar. A questão fundamental é: você consegue escrever um programa para cada filósofo que faça o que deve fazer e nunca trave? (Observe que a necessidade de ter dois garfos é artificial; talvez fosse melhor mudar de comida italiana para comida chinesa, substituindo o espaguete por arroz e os garfos por pauzinhos.)

A Figura 2.37 mostra a solução óbvia. A rotina *take\_fork* espera até que o garfo específico esteja disponível e então o pega. Infelizmente, a solução óbvia está errada. Suponha que todos os cinco filósofos resolvam usar seus garfos simultaneamente. Nenhum deles será capaz de pegar o garfo que estiver a sua direita e, assim, ocorrerá um impasse.

Podemos fazer modificações para que o programa, depois de pegar o garfo esquerdo, verifique se o garfo direito está disponível. Se não estiver, o filósofo devolverá o garfo esquerdo à mesa, esperará por algum tempo e então repetirá todo o processo. Mas essa proposta também falha, embora por uma razão diferente. Com um pouco de azar, todos os filósofos poderiam começar o algoritmo simultâ-

neamente; pegando seus garfos esquerdos e, vendo que seus garfos direitos não estariam disponíveis, devolveriam seus garfos esquerdos, esperariam, de novo pegariam seus garfos esquerdos simultaneamente, e assim permaneceriaiam para sempre. Uma situação como essa — na qual todos os programas continuam executando indefinidamente, mas falham ao tentar progredir — é chamada de **inanição** (*starvation*). (E é assim chamada mesmo quando o problema não ocorre em um restaurante italiano ou chinês.)

Mas então você poderia pensar que, se os filósofos esperassem por um tempo aleatório, em vez de esperarem por um tempo fixo depois de falharem ao pegar o garfo do lado direito, a probabilidade de tudo continuar intertravado, mesmo que por uma hora, seria muito pequena. Essa observação é verdadeira, e, em quase todas as aplicações, tentar de novo mais tarde é uma abordagem adotada. Por exemplo, na popular rede local Ethernet, se dois computadores enviam um pacote ao mesmo tempo (levando a uma colisão de pacotes), cada um espera por um tempo aleatório antes de tentar novamente; na prática, essa solução funciona bem. Contudo, para algumas aplicações seria preferível uma solução que sempre fosse válida e não falhasse por causa de uma série improvável de números aleatórios. Pense, por exemplo, no controle de segurança em uma usina de energia nuclear.

Um aperfeiçoamento da solução mostrada na Figura 2.37 que não apresenta impasse nem inanição é proteger os cinco comandos que seguem a chamada *think* com um semáforo binário. Antes de começar a pegar garfos, um filósofo faria um *down* no *mutex*. Depois de trocar os garfos, ele faria um *up* no *mutex*. Do ponto de vista teórico, essa solução é adequada. Do ponto de vista prático, ela apresenta um problema de desempenho: somente um filósofo por vez pode comer a qualquer instante. Com cinco garfos disponíveis, seria possível permitir que dois filósofos comessem ao mesmo tempo.

A solução apresentada na Figura 2.38 é livre de impasse e permite o máximo paralelismo a um número arbitrário de filósofos. Ela usa um arranjo, *estado*, para controlar se

```
#define N 5
/* número de filósofos */

void philosopher(int i)
{
    while (TRUE) {
        think();
        take_fork(i);
        take_fork((i+1) % N);
        eat();
        put_fork(i);
        put_fork((i+1) % N);
    }
}

/* i: número do filósofo, de 0 a 4 */
/* o filósofo está pensando */
/* pega o garfo esquerdo */
/* pega o garfo direito; % é o operador módulo */
/* hummm! Espaguete */
/* devolve o garfo esquerdo à mesa */
/* devolve o garfo direito à mesa */
```

**Figura 2.37** Uma solução errada para o problema do jantar dos filósofos.

```

#define N      5          /* número de filósofos */
#define LEFT   (i+N-1)%N  /* número do vizinho à esquerda de i */
#define RIGHT  (i+1)%N   /* número do vizinho à direita de i */
#define THINKING 0        /* o filósofo está pensando */
#define HUNGRY   1        /* o filósofo está tentando pegar garfos */
#define EATING   2        /* o filósofo está comendo */
typedef int semaphore;
int state[N];
semaphore mutex = 1;
semaphore s[N];

void philosopher(int i)
{
    while (TRUE) {
        think();
        take_forks(i);
        eat();
        put_forks(i);
    }
}

void take_forks(int i)
{
    down(&mutex);
    state[i] = HUNGRY;
    test(i);
    up(&mutex);
    down(&s[i]);
}

void put_forks(i)
{
    down(&mutex);
    state[i] = THINKING;
    test(LEFT);
    test(RIGHT);
    up(&mutex);
}

void test(i)/* i: o número do filósofo, de 0 a N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}

```

**Figura 2.38** Uma solução para o problema do jantar dos filósofos.

um filósofo está comendo, pensando ou faminto (tentando pegar garfos). Um filósofo só pode mudar para o estado ‘comendo’ se nenhum dos vizinhos estiver comendo. Os vizinhos do filósofo  $i$  são definidos pelas macros  $LEFT$  e  $RIGHT$ . Em outras palavras, se  $i$  for 2,  $LEFT$  será 1 e  $RIGHT$  será 3.

O programa usa um arranjo de semáforos, um por filósofo; assim, filósofos famintos podem ser bloqueados se os garfos necessários estiverem ocupados. Observe que cada processo executa a rotina *philosopher* como seu código principal, mas as outras rotinas — *take\_forks*, *put\_forks* e *test* — são rotinas ordinárias, e não processos separados.

### 2.5.2 | O problema dos leitores e escritores

O problema do jantar dos filósofos é útil para modelar processos que competem pelo acesso exclusivo a um número limitado de recursos, como dispositivos de E/S. Outro problema famoso é o caso dos leitores e escritores (Courtois et al., 1971), que modela o acesso a uma base de dados. Imagine, por exemplo, um sistema de reserva de linhas aéreas, com muitos processos em competição, querendo ler e escrever. É aceitável que múltiplos processos leiam a base de dados ao mesmo tempo, mas, se um processo estiver atualizando (escrevendo) na base de dados, nenhum outro

processo pode ter acesso ao banco de dados, nem mesmo os leitores. A questão é: como programar os leitores e os escritores? Uma solução é mostrada na Figura 2.39.

A partir dessa solução, para obter o acesso à base de dados, o primeiro leitor faz um down no semáforo *db*. Os leitores subsequentes meramente incrementam um contador, *rc*. Conforme saem, os leitores decrementam o contador de 1 e o último leitor a sair faz um up no semáforo, permitindo que um eventual escritor bloqueado entre.

A solução apresentada aqui contém implicitamente uma decisão sutil que vale a pena comentar. Suponha que, enquanto um leitor está usando a base de dados, um outro leitor chegue. Como ter dois leitores ao mesmo tempo não é um problema, o segundo leitor é admitido. Leitores adicionais também podem ser admitidos se chegarem.

Agora imagine que apareça um escritor. Este não pode ser admitido na base de dados, pois escritores devem ter acesso exclusivo. O escritor é, então, suspenso. Leitores adicionais chegam. Enquanto houver pelo menos um leitor ativo, leitores subsequentes serão admitidos. Como consequência dessa estratégia, enquanto houver um fluxo estável de leitores chegando, todos entrarão assim que chegarem. O escritor permanecerá suspenso até que nenhum leitor esteja presente. Se um novo leitor chegar — digamos,

a cada dois segundos — e cada leitor levar cinco segundos para fazer seu trabalho, o escritor nunca entrará.

Para evitar essa situação, o programa poderia ser escrito de modo um pouco diferente: se um leitor chegar quando um escritor estiver esperando, o leitor será suspenso logo depois do escritor, em vez de ser admitido de imediato. Dessa maneira, um escritor, para terminar, precisa esperar por leitores que estavam ativos quando ele chegou, mas não por leitores que chegaram depois dele. A desvantagem dessa solução é que se consegue menos concorrência e, portanto, um desempenho menor. Courtois et al. apresentam uma solução que dá prioridade aos escritores. Para mais detalhes, consulte o artigo.

## 2.6 Pesquisas em processos e threads

No Capítulo 1, estudamos algumas das pesquisas atuais em estrutura de sistemas operacionais. Neste capítulo e nos subsequentes, examinaremos pesquisas mais específicas, iniciando com os processos. Aos poucos, torna-se claro que alguns assuntos são muito mais estáveis que outros. A maioria das pesquisas tende a ser sobre tópicos novos, em vez daqueles que nos rodeiam há décadas.

```

typedef int semaphore;
semaphore mutex = 1;
semaphore db = 1;
int rc = 0;

void reader(void)
{
    while (TRUE) {
        down(&mutex);
        rc = rc + 1;
        if (rc == 1) down(&db);
        up(&mutex);
        read_data_base();
        down(&mutex);
        rc = rc - 1;
        if (rc == 0) up(&db);
        up(&mutex);
        use_data_read();
    }
}

void writer(void)
{
    while (TRUE) {
        think_up_data();
        down(&db);
        write_data_base();
        up(&db);
    }
}

```

**Figura 2.39** Uma solução para o problema dos leitores e escritores.

O conceito de um processo é um exemplo de algo muito bem estabelecido. Quase todo sistema tem alguma noção de um processo como um recipiente para agrupar recursos relacionados, como um espaço de endereçamento, threads, arquivos abertos, permissões de proteção etc. Sistemas diferentes fazem esse agrupamento de maneira um pouco diferente, mas trata-se apenas de diferenças de engenharia. A ideia básica não é tão controversa e há pouca pesquisa nova sobre o tema processos.

Os threads são uma ideia mais nova que os processos, mas tem havido bastantes reflexões sobre eles. Além disso, ocasionalmente aparecem artigos sobre threads tratando, por exemplo, de aglomerados de multiprocessadores (Tam et al., 2007) ou do dimensionamento do número de threads em um processo para cem mil (Von Behren et al., 2003).

A sincronização de processos está muito mais definida agora, mas de vez em quando ainda há artigos, como aqueles sobre processamento concorrente sem locks (Fraser e Harris, 2007) ou sincronização no modo não bloqueante em sistemas de tempo real (Hothmuth e Haertig, 2001).

O escalonamento (tanto uniprocessador quanto multiprocessador) ainda é um tópico recente e caro a alguns pesquisadores. Alguns tópicos sendo pesquisados incluem escalonamento de dispositivos móveis em termos de eficiência energética (Yuan e Nahrstedt, 2006), escalonamento com tecnologia hyperthreading (Bulpin e Pratt, 2005), modos de reduzir a ociosidade da CPU (Eggert e Touch, 2005) e escalonamento de sistemas de tempo virtual (Neh et al., 2001). Contudo, poucos projetistas de sistemas operacionais andam desesperados pela falta de um algoritmo decente para o escalonamento de threads — portanto, parece que esse tipo de pesquisa é mais um desejo do que uma necessidade de pesquisadores. De modo geral, processos, threads e escalonamento não são mais tópicos de pesquisa tão procurados como antes. A pesquisa avançou.

## 2.7 Resumo

Para ocultar os efeitos das interrupções, os sistemas operacionais oferecem um modelo conceitual que consiste em processos sequenciais executando em paralelo. Os processos podem ser criados e terminados dinamicamente. Cada processo tem seu próprio espaço de endereçamento.

Para algumas aplicações, é útil ter múltiplos threads de controle dentro de um único processo. Esses threads são escalonados independentemente e cada um tem sua própria pilha, mas todos os threads em um processo compartilham um espaço de endereçamento comum. Threads podem ser implementados no espaço do usuário ou no núcleo.

Os processos podem se comunicar uns com os outros por meio de primitivas de comunicação entre processos, como semáforos, monitores ou mensagens. Essas unidades básicas são usadas para assegurar que dois processos nunca estarão em suas regiões críticas ao mesmo tempo — uma

situação que levaria ao caos. Um processo pode estar executando, ser executável ou bloqueado e alterar o estado quando ele ou um outro processo executa uma das unidades básicas de comunicação entre processos. A comunicação interthread é semelhante.

As primitivas de comunicação entre processos podem ser usadas para resolver problemas como o produtor-consumidor, o jantar dos filósofos e o leitor-escritor. Mesmo com essas primitivas, devem-se tomar cuidados para evitar erros e impasses.

Muitos algoritmos de escalonamento têm sido estudados. Alguns deles são usados principalmente em sistemas em lote, como a *tarefa mais curta primeiro*. Outros são comuns aos sistemas em lote e aos sistemas interativos — como o escalonamento circular, o escalonamento por prioridades, as filas múltiplas, o escalonamento garantido, o escalonamento por loteria e o escalonamento por fração justa. Alguns sistemas fazem uma separação entre o mecanismo de escalonamento e a política de escalonamento, o que permite aos usuários um controle sobre o algoritmo de escalonamento.

## Problemas

- Na Figura 2.2, são mostrados três estados de processos. Na teoria, com três estados poderia haver seis transições, duas para cada estado. Contudo, somente quatro transições são mostradas. Há alguma circunstância na qual uma delas ou ambas as transições não ilustradas possam ocorrer?
- Suponha que você seja o projetista de uma arquitetura de computador avançada que fez o chaveamento entre processos por hardware em vez de usar interrupções. De que informação a CPU precisaria? Descreva como o processo de chaveamento por hardware poderia funcionar.
- Em todos os computadores atuais, pelo menos uma parte dos manipuladores de interrupção (interrupt handlers) é escrita em linguagem assembly. Por quê?
- Quando uma interrupção ou uma chamada de sistema transfere o controle para o sistema operacional, geralmente é usada uma área da pilha do núcleo separada da pilha do processo interrompido. Por quê?
- Tarefas múltiplas podem ser executadas paralelamente e terminar mais rápido do que se tivessem sido executados sucessivamente. Suponha que duas tarefas, cada uma precisando de dez minutos do tempo da CPU, começarem simultaneamente. De quanto tempo o último precisará para terminar se elas forem executados sucessivamente? Quanto tempo se forem executadas paralelamente? Suponha 50 por cento de espera de E/S.
- No texto, foi estabelecido que o modelo da Figura 2.8(a) não era adequado para um servidor de arquivos que usasse uma cache na memória. Por que não? Cada processo poderia ter sua própria cache?
- Se um processo multithread bifurcar, há problemas se o filho copia todos os threads do pai. Suponha que um dos threads originais estivesse esperando por uma entrada do

teclado. Agora dois threads estão esperando pela entrada do teclado, um em cada processo. Esse problema pode ocorrer em processos de thread único?

8. Na Figura 2.6, é mostrado um servidor da Web multithread. Se o único modo de ler a partir de um arquivo for o bloqueio normal da chamada de sistema `read`, você acha que threads de usuário ou de núcleo estão sendo usados para o servidor da Web? Por quê?
9. No texto, descrevemos um servidor da Web multithread, mostrando por que ele é melhor que um servidor de thread único e um servidor de máquina de estados finitos. Há alguma circunstância na qual um servidor de thread único poderia ser melhor? Dê um exemplo.
10. Na Tabela 2.4, o conjunto de registradores é relacionado como um item por thread, e não por processo. Por quê? (Afinal, a máquina tem somente um conjunto de registradores.)
11. O que faria um thread desistir voluntariamente da CPU chamando `thread_yield`? (Afinal, como não há interrupção periódica de relógio, ele pode nunca mais obter a CPU de volta.)
12. Um thread pode sofrer preempção por uma interrupção de relógio? Em caso afirmativo, sob quais circunstâncias? Do contrário, por que não?
13. Neste problema, você deve comparar a leitura de um arquivo usando um servidor de arquivos monothread e um servidor multithread. São necessários 15 ms para obter uma requisição de trabalho, despachá-la e fazer o restante do processamento necessário, presumindo que os dados essenciais estejam na cache de blocos. Se for necessária uma operação de disco — como ocorre em um terço das vezes —, será preciso um tempo adicional de 75 ms, durante o qual o thread dorme. Quantas requisições/segundo o servidor pode tratar se for monothread? E se for multithread?
14. Qual a maior vantagem de implementar threads no espaço do usuário? Qual é a maior desvantagem?
15. Na Figura 2.10, as criações de threads e as mensagens impressas pelos threads são intercaladas aleatoriamente. Há algum modo de impor que a ordem seja estritamente thread 1 criado, thread 1 imprime mensagem, thread 1 sai, thread 2 criado, thread 2 imprime a mensagem, thread 2 sai e assim por diante? Em caso de resposta afirmativa, qual é esse modo? Em caso de resposta negativa, por que não?
16. Na discussão sobre variáveis globais em threads, usamos uma rotina `create_global` para alocar memória a um ponteiro para a variável, em vez de alocar diretamente a própria variável. Isso é essencial ou as rotinas poderiam funcionar muito bem apenas com os próprios valores?
17. Considere um sistema no qual threads são implementados inteiramente no espaço do usuário, sendo que o sistema de tempo de execução sofre uma interrupção de relógio a cada segundo. Suponha que uma interrupção de relógio ocorra enquanto algum thread estiver executando no sistema de tempo de execução. Que problema poderia ocorrer? O que você sugere para resolvê-lo?
18. Suponha que um sistema operacional não tenha uma chamada de sistema como a `select` para verificar previamente se é seguro ler um arquivo, um pipe ou algum dispositivo, mas ele permite que “alarm clocks” sejam setados, os quais interrompem chamadas de sistema bloqueadas. É possível implementar um pacote de threads, no espaço de usuário, sob essas condições? Comente.
19. O problema de inversão de prioridades discutido na Seção 2.3.4 pode acontecer com threads de usuário? Por quê?
20. Na Seção 2.3.4, foi descrita uma situação com um processo de alta prioridade,  $H$ , e um de baixa prioridade,  $L$ , que levava  $H$  a um laço infinito. O mesmo problema ocorreria se fosse usado o escalonamento circular em vez do escalonamento por prioridades? Comente.
21. Em um sistema com threads, quando são utilizados threads de usuário, há uma pilha por thread ou uma pilha por processo? E quando se usam threads de núcleo? Explique.
22. Quando um computador está sendo desenvolvido, ele é antes simulado por um programa que executa uma instrução por vez. Mesmo os multiprocessadores são simulados de modo estritamente sequencial. É possível que ocorra uma condição de corrida quando não há eventos simultâneos como nessas simulações?
23. A solução de espera ociosa usando a variável `turn` (Figura 2.18) funciona quando os dois processos estão executando em um multiprocessador de memória compartilhada, isto é, duas CPUs compartilhando uma memória comum?
24. A solução de Peterson para o problema da exclusão mútua, mostrado na Figura 2.19, funciona quando o escalonamento do processo for preemptivo? E quando o escalonamento não for preemptivo?
25. Faça um esboço de como um sistema operacional capaz de desabilitar interrupções poderia implementar semáforos.
26. Mostre como os semáforos contadores (isto é, os semáforos que podem conter um valor arbitrário) podem ser implementados usando somente semáforos binários e simples instruções de máquina.
27. Se um sistema tem somente dois processos, tem sentido usar uma barreira para sincronizá-los? Por quê?
28. Dois threads podem, no mesmo processo, sincronizar a partir do uso de um semáforo de núcleo se os threads forem implementados pelo núcleo? E se os threads fossem implementados no espaço do usuário? (Suponha que nenhum thread em qualquer outro processo tenha acesso ao semáforo.) Comente suas respostas.
29. Sincronização com monitores usa variáveis de condição e duas operações especiais, `wait` e `signal`. Uma forma mais geral de sincronização seria ter uma única primitiva, `waituntil`, que possuísse um predicado booleano como parâmetro. Assim, alguém poderia dizer, por exemplo,

$$\text{waituntil } x < 0 \text{ ou } y + z < n$$

A primitiva `signal` não seria mais necessária. Esse esquema é claramente mais geral que o proposto por Hoare ou Brinch Hansen, mas não é utilizado. Por quê? Dica: pense na implementação.

- 30.** Um restaurante de fast-food tem quatro tipos de empregados: (1) anotadores de pedido, que anotam o pedido dos clientes; (2) cozinheiros, que preparam a comida; (3) embaladores, que colocam a comida nas sacolas; e (4) caixas, que entregam as sacolas para os clientes e recebem o dinheiro deles. Cada empregado pode ser observado como um processo sequencial de comunicação. Qual forma de comunicação entre processos eles usariam? Relacione esse modelo aos processos no UNIX?
- 31.** Imagine um sistema por troca de mensagens que use caixas postais. Quando se envia para uma caixa postal cheia ou tenta-se receber de uma caixa postal vazia, um processo não bloqueia. Na verdade, ele obtém um código de erro. O processo responde ao código de erro apenas tentando novamente, sucessivamente, até que ele consiga. Esse esquema leva a condições de corrida?
- 32.** Os computadores CDC 6600 podiam lidar simultaneamente com até dez processos de E/S, usando uma forma interessante de escalonamento circular chamada **compartilhamento de processador**. Um chaveamento de processo ocorria depois de cada instrução; assim, a instrução 1 vinha do processo 1, a instrução 2 vinha do processo 2 e assim por diante. O chaveamento do processo era feito por um hardware especial e a sobrecarga era zero. Se um processo precisasse de  $T$  segundos para terminar sua execução, na ausência de competição, quanto tempo seria necessário se o compartilhamento do processador fosse usado com  $n$  processos?
- 33.** Seria possível estabelecer uma medida sobre o quanto um processo é limitado pela CPU ou limitados por E/S analisando o código-fonte? Como isso poderia ser determinado em tempo de execução?
- 34.** Na seção ‘Quando escalonar’, foi mencionado que, algumas vezes, o escalonamento poderia ser melhorado se um processo importante fosse passível de desempenhar um papel, ao ser bloqueado, na seleção do próximo processo a executar. Pense em uma situação na qual isso poderia ser usado e explique como.
- 35.** As medidas de um certo sistema mostram que o processo médio executa por um tempo  $T$  antes de ser bloqueado para E/S. Um chaveamento de processos requer um tempo  $S$  efetivamente gasto (sobrecarga). Para o escalonamento circular com um quantum  $Q$ , dê uma fórmula para a eficiência da CPU em cada um dos seguintes casos:
- $Q = \infty$ .
  - $Q > T$ .
  - $S < Q < T$ .
  - $Q = S$ .
  - $Q$  próximo de 0.
- 36.** Cinco tarefas estão esperando para serem executadas. Seus tempos de execução previstos são 9, 6, 3, 5 e X. Em que ordem elas deveriam ser executadas para minimizar o tempo médio de resposta? (Sua resposta dependerá de X.)
- 37.** Cinco tarefas em lote, A a E, chegam a um centro de computação quase ao mesmo tempo. Elas têm tempos de execução estimados em 10, 6, 2, 4 e 8 minutos. Suas prioridades (externamente determinadas) são 3, 5, 2, 1 e 4, respectivamente, sendo 5 a prioridade mais alta. Para cada um dos seguintes algoritmos de escalonamento, determine o tempo médio de ida e volta. Ignore a sobrecarga de chaveamento de processos.
- Circular.
  - Escalonamento por prioridades.
  - Primeiro a chegar, primeiro a ser servido* (execute na ordem 10, 6, 2, 4, 8).
  - Tarefa mais curta primeiro*.
- Para (a), presuma que o sistema é multiprogramado e que cada tarefa obtenha sua fração justa da CPU. Para os itens (b) a (d), considere a execução de somente uma tarefa por vez, até que termine. Todas as tarefas são completamente limitadas pela CPU.
- 38.** Um processo executando no CTSS precisa de 30 quanta para terminar. Quantas vezes ocorrerá uma troca para a memória, incluindo a primeira vez (antes de executar qualquer coisa)?
- 39.** Você tem ideia de como impedir que o sistema de prioridade do CTSS seja enganado digitando-se a tecla <Entrar> aleatoriamente?
- 40.** O algoritmo do envelhecimento (*aging*) com  $a = 1/2$  está sendo usado para prever tempos de execução. As quatro execuções anteriores, da primeira à mais recente, são 40, 20, 40 e 15 ms. Qual é a previsão da próxima execução?
- 41.** Um sistema de tempo real tem quatro eventos periódicos com períodos de 50, 100, 200 e 250 ms cada. Suponha que os quatro eventos requeiram 35, 20, 10 e  $x$  ms de tempo de CPU, respectivamente. Qual é o maior valor de  $x$  para que o sistema seja escalonável?
- 42.** Explique por que o escalonamento em dois níveis é bastante usado.
- 43.** Um sistema de tempo real precisa controlar duas chamadas de voz, cada uma delas executada a cada 5 ms e consumindo 1 ms do tempo da CPU por surto, além de um vídeo de 25 quadros/s, e cada quadro requer 20 ms do tempo da CPU. Esse sistema pode ser escalonado?
- 44.** Considere um sistema no qual se deseja separar a política e o mecanismo para o algoritmo de escalonamento dos threads de núcleo. Proponha um meio de chegar a esse objetivo.
- 45.** Na solução para o problema do jantar dos filósofos (Figura 2.38), por que é atribuído *HUNGRY* à variável de estado na rotina *take\_forks*?
- 46.** Observe a rotina *put\_forks* da Figura 2.38. Suponha que à variável *state[i]* fosse atribuída *THINKING* depois das duas chamadas de *test*, e não *antes*. Como isso poderia afetar a solução?
- 47.** O problema dos leitores e escritores pode ser formulado de várias maneiras, dependendo de quando cada categoria de processos pode ser iniciada. Descreva, detalhadamente, três variações diferentes do problema, cada uma favorecendo (ou não) alguma categoria de processos. Para

cada variação, especifique o que acontece quando um leitor ou um escritor fica pronto para ter acesso ao banco de dados e o que ocorre quando um processo acaba de usar o banco de dados.

- 48.** Escreva um script do shell que produza um arquivo de números sequenciais lendo-se o último número no arquivo, adicionando-se 1 a ele e, então, anexando-o ao arquivo. Execute uma instância do script em background (segundo plano) e outra em foreground (primeiro plano), cada uma realizando acessos ao mesmo arquivo. Quanto tempo transcorre antes de se manifestar uma condição de disputa? Qual é a região crítica? Modifique o script para impedir a disputa. (*Dica:* use

`In file file.lock`

para proteger o arquivo de dados.)

- 49.** Imagine um sistema operacional que permita semáforos. Implemente um sistema de mensagens. Escreva as rotinas para enviar e receber mensagens.
- 50.** Resolva o problema do jantar dos filósofos usando monitores em vez de semáforos.
- 51.** Suponha que uma universidade, para mostrar como é politicamente correta, aplique a doutrina da Suprema Corte dos Estados Unidos, “Separado mas igual é inherentemente desigual”, para gênero e raça, pondo fim a sua prática de longa data de banheiros no campus segregados por gênero.

Contudo, como uma concessão à tradição, ela decreta que, quando uma mulher estiver no banheiro, outra mulher poderá entrar, mas um homem não e vice-versa. Um sinal com um marcador deslizante, na porta de cada banheiro, indica em qual dos três estados o banheiro se encontra:

Vazio

Com mulher

Com homem

Escreva, em sua linguagem de programação favorita, as seguintes rotinas: *mulher\_quer\_entrar*, *homem\_quer\_entrar*, *mulher\_sai*, *homem\_sai*. Você pode usar os contadores e as técnicas de sincronização que quiser.

- 52.** Reescreva o programa da Figura 2.18 para tratar mais de dois processos.
- 53.** Escreva um problema produtor-consumidor que use threads e compartilhe um buffer comum. Contudo, não use semáforos ou qualquer outra primitiva de sincronização para proteger a estrutura de dados compartilhada. Apenas deixe cada thread ter acesso a eles quando quiser. Use `sleep` e `wakeup` para tratar as condições de buffer cheio e buffer vazio. Veja quanto tempo leva até ocorrer uma condição de disputa fatal. Por exemplo, você pode ter o produtor imprimindo um número a cada intervalo de tempo. Não imprima mais do que um número a cada minuto, porque a E/S poderia afetar as condições de corrida.