ANDREW S.
TANENBAUM
HERBERT
BOS

SISTEMAS OPERACIONAIS MODERNOS

4ª EDIÇÃO

Pearson

CAPÍTULO 2

PROCESSOS E THREADS

Estamos prestes a embarcar agora em um estudo detalhado de 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 o mais depende desse conceito, e o projetista (e estudante) do sistema operacional deve ter uma compreensão profunda do que é um processo o mais cedo possível.

Processos são uma das mais antigas e importantes abstrações que os sistemas operacionais proporcionam. Eles dão suporte à possibilidade de haver operações (pseudo) concorrentes mesmo quando há apenas uma CPU disponível, transformando uma única CPU em múltiplas CPUs virtuais. Sem a abstração de processo, a computação moderna não poderia existir. Neste capítulo, examinaremos detalhadamente os processos e seus "primos", os threads.

2.1 Processos

Todos os computadores modernos frequentemente realizam várias tarefas ao mesmo tempo. As pessoas acostumadas a trabalhar com computadores talvez não estejam totalmente cientes desse fato, então alguns exemplos podem esclarecer este ponto. Primeiro, considere um servidor da web, em que solicitações de páginas da web chegam de toda parte. Quando uma solicitação chega, o servidor confere para ver se a página requisitada está em cache. Se estiver, ela é enviada de volta; se não, uma solicitação de acesso ao disco é iniciada para buscá-la. No entanto, do ponto de vista da CPU, as solicitações de acesso ao disco levam uma eternidade. Enquanto espera que uma solicitação de acesso ao disco seja concluída, muitas outras solicitações podem chegar. Se há múltiplos discos presentes, algumas ou todas as solicitações mais recentes podem ser enviadas para os outros discos muito antes de a primeira solicitação ter sido concluída. Está claro que algum método é necessário para modelar e controlar essa concorrência. Processos (e especialmente threads) podem ajudar nisso.

Agora considere um PC de usuário. Quando o sistema é inicializado, muitos processos são secretamente iniciados, quase sempre desconhecidos para o usuário. Por exemplo, um processo pode ser inicializado para esperar pela chegada de e-mails. Outro pode ser executado em prol do programa antivírus para conferir periodicamente se há novas definições de vírus disponíveis. Além disso, processos explícitos de usuários podem ser executados, imprimindo arquivos e salvando as fotos do usuário em um pen-drive, tudo isso enquanto o usuário está navegando na Web. Toda essa atividade tem de ser gerenciada, e um sistema de multiprogramação que dê suporte a múltiplos processos é muito útil nesse caso.

Em qualquer sistema de multiprogramação, a CPU muda de um processo para outro rapidamente, executando cada um por dezenas ou centenas de milissegundos. Enquanto, estritamente falando, em qualquer dado instante a CPU está executando apenas um processo, no curso de 1s ela pode trabalhar em vários deles, dando a ilusão do paralelismo. Às vezes, as pessoas falam em **pseudoparalelismo** neste contexto, para diferenciar do verdadeiro paralelismo de hardware dos sistemas multiprocessadores (que têm duas ou mais CPUs compartilhando a mesma memória física). Ter controle sobre múltiplas atividades em paralelo é algo difícil para as pessoas realizarem. Portanto, projetistas de sistemas

operacionais através dos anos desenvolveram um modelo conceitual (processos sequenciais) que torna o paralelismo algo mais fácil de lidar. Esse modelo, seus usos e algumas das suas consequências compõem o assunto deste capítulo.

2.1.1 O modelo de processo

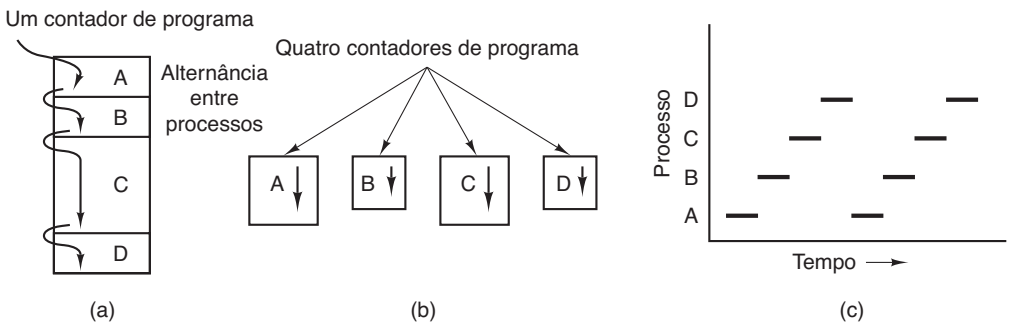
Nesse modelo, todos os softwares executáveis no computador, às vezes incluindo o sistema operacional, são organizados em uma série de **processos sequenciais**, ou, simplesmente, **processos**. Um processo é apenas uma instância de um programa em execução, incluindo os valores atuais do contador do programa, registradores e variáveis. Conceitualmente, cada processo tem sua própria CPU virtual. Na verdade, a CPU real troca a todo momento de processo em processo, mas, para compreender o sistema, é muito mais fácil pensar a respeito de uma coleção de processos sendo executados em (pseudo) paralelo do que tentar acompanhar como a CPU troca de um programa para o outro. Esse mecanismo de trocas rápidas é chamado de **multiprogramação**, como vimos no Capítulo 1.

Na Figura 2.1(a) vemos um computador multiprogramando quatro programas na memória. Na Figura 2.1(b) vemos quatro processos, cada um com seu próprio fluxo de controle (isto é, seu próprio contador de programa lógico) e sendo executado independente dos outros. É claro que há apenas um contador de programa físico, de maneira que, quando cada processo é executado, o seu contador de programa lógico é carregado para o contador de programa real. No momento em que ele é concluído, o contador de programa físico é salvo no contador de programa lógico do processo na memória. Na Figura 2.1(c) vemos que, analisados durante um intervalo longo o suficiente, todos os processos tiveram progresso, mas a qualquer dado instante apenas um está sendo de fato executado.

Neste capítulo, presumiremos que há apenas uma CPU. Cada vez mais, no entanto, essa suposição não é verdadeira, tendo em vista que os chips novos são muitas vezes *multinúcleos* (*multicore*), com dois, quatro ou mais núcleos. Examinaremos os chips multinúcleos e multiprocessadores em geral no Capítulo 8, mas, por ora, é mais simples pensar em apenas uma CPU de cada vez. Então quando dizemos que uma CPU pode na realidade executar apenas um processo de cada vez, se há dois núcleos (ou CPUs) cada um deles pode ser executado apenas um processo de cada vez.

Com o chaveamento rápido da CPU entre os processos, a taxa pela qual um processo realiza a sua computação não será uniforme e provavelmente nem reproduzível se os mesmos processos forem executados outra vez. Desse modo, processos não devem ser programados com suposições predefinidas sobre a temporização. Considere, por exemplo, um processo de áudio que toca música para acompanhar um vídeo de alta qualidade executado por outro dispositivo. Como o áudio deve começar um pouco depois do que o vídeo, ele sinaliza ao servidor do vídeo para começar a execução, e então realiza um laço ocioso 10.000 vezes antes de executar o áudio. Se o laço for um temporizador confiável, tudo vai correr bem, mas se a CPU decidir trocar para outro processo durante o laço ocioso, o processo de áudio pode não ser executado de novo até que os quadros de vídeo correspondentes já tenham vindo e ido embora, e o vídeo e o áudio ficarão irritantemente fora de sincronia. Quando um processo tem exigências de tempo real, críticas como essa, isto é, eventos particulares, *têm* de ocorrer dentro de um número específico de milissegundos e medidas especiais precisam ser tomadas para assegurar que elas ocorram. Em geral, no entanto, a maioria dos processos não é afetada pela multiprogramação

FIGURA 2.1 (a) Multiprogramação de quatro programas. (b) Modelo conceitual de quatro processos sequenciais independentes. (c) Apenas um programa está ativo de cada vez.



subjacente da CPU ou as velocidades relativas de processos diferentes.

A diferença entre um processo e um programa é sutil, mas absolutamente crucial. Uma analogia poderá ajudá-lo aqui: considere um cientista de computação que gosta de cozinhar e está preparando um bolo de aniversário para sua filha mais nova. Ele tem uma receita de um bolo de aniversário e uma cozinha bem estocada com todas as provisões: farinha, ovos, açúcar, extrato de baunilha etc. Nessa analogia, a receita é o programa, isto é, o algoritmo expresso em uma notação adequada, o cientista de computação é o processador (CPU) e os ingredientes do bolo são os dados de entrada. O processo é a atividade consistindo na leitura da receita, busca de ingredientes e preparo do bolo por nosso cientista.

Agora imagine que o filho do cientista de computação aparece correndo chorando, dizendo que foi picado por uma abelha. O cientista de computação registra onde ele estava na receita (o estado do processo atual é salvo), pega um livro de primeiros socorros e começa a seguir as orientações. Aqui vemos o processador sendo trocado de um processo (preparo do bolo) para um processo mais prioritário (prestar cuidado médico), cada um tendo um programa diferente (receita *versus* livro de primeiros socorros). Quando a picada de abelha tiver sido cuidada, o cientista de computação volta para o seu bolo, continuando do ponto onde ele havia parado.

A ideia fundamental aqui é que um processo é uma atividade de algum tipo. Ela tem um programa, uma entrada, uma saída e um estado. Um único processador pode ser compartilhado entre vários processos, com algum algoritmo de escalonamento sendo usado para determinar quando parar o trabalho em um processo e servir outro. Em comparação, um programa é algo que pode ser armazenado em disco sem fazer nada.

Vale a pena observar que se um programa está sendo executado duas vezes, é contado como dois processos. Por exemplo, muitas vezes 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 programa não importa, eles são processos distintos. O sistema operacional pode ser capaz de compartilhar o código entre eles de maneira que apenas uma cópia esteja na memória, mas isso é um detalhe técnico que não muda a situação conceitual de dois processos sendo executados.

2.1.2 Criação de processos

Sistemas operacionais precisam de alguma maneira para criar processos. Em sistemas muito simples, ou em sistemas projetados para executar apenas uma única aplicação (por exemplo, o controlador em um forno micro-ondas), pode ser possível ter todos os processos que serão em algum momento necessários quando o sistema for ligado. Em sistemas para fins gerais, no entanto, alguma maneira é necessária para criar e terminar processos, na medida do necessário, durante a operação. Vamos examinar agora algumas das questões.

Quatro eventos principais fazem com que os processos sejam criados:

1. Inicialização do sistema.
2. Execução de uma chamada de sistema de criação de processo por um processo em execução.
3. Solicitação de um usuário para criar um novo processo.
4. Início de uma tarefa em lote.

Quando um sistema operacional é inicializado, em geral uma série de processos é criada. Alguns desses processos são de primeiro plano, isto é, processos que interagem com usuários (humanos) e realizam trabalho para eles. Outros operam no segundo plano e não estão associados com usuários em particular, mas em vez disso têm alguma função específica. Por exemplo, um processo de segundo plano pode ser projetado para aceitar e-mails, ficando inativo a maior parte do dia, mas subitamente entrando em ação quando chega um e-mail. Outro processo de segundo plano pode ser projetado para aceitar solicitações de páginas da web hospedadas naquela máquina, despertando quando uma solicitação chega para servir àquele pedido. Processos que ficam em segundo plano para lidar com algumas atividades, como e-mail, páginas da web, notícias, impressão e assim por diante, são chamados de **daemons**. Grandes sistemas comumente têm dúzias deles: no UNIX,[1] o programa *ps* pode ser usado para listar os processos em execução; no Windows, o gerenciador de tarefas pode ser usado.

Além dos processos criados durante a inicialização do sistema, novos processos podem ser criados depois também. Muitas vezes, um processo em execução emitirá chamadas de sistema para criar um ou mais processos novos para ajudá-lo em seu trabalho. Criar processos novos é particularmente útil quando o trabalho a ser feito pode ser facilmente formulado em termos de vários

[1] Neste capítulo, o UNIX deve ser interpretado como incluindo quase todos os sistemas baseados em POSIX, incluindo Linux, FreeBSD, OS X, Solaris etc., e, até certo ponto, Android e iOS também. (N. A.)

processos relacionados, mas de outra forma interagindo de maneira independente. Por exemplo, se uma grande quantidade de dados está sendo buscada através de uma rede para processamento subsequente, pode ser conveniente criar um processo para buscar os dados e colocá-los em um local compartilhado de memória enquanto um segundo processo remove os itens de dados e os processa. Em um multiprocessador, permitir que cada processo execute em uma CPU diferente também pode fazer com que a tarefa seja realizada mais rápido.

Em sistemas interativos, os usuários podem começar um programa digitando um comando ou clicando duas vezes sobre um ícone. Cada uma dessas ações inicia um novo processo e executa nele o programa selecionado. Em sistemas UNIX baseados em comandos que executam X, o novo processo ocupa a janela na qual ele foi iniciado. No Windows, quando um processo é iniciado, ele não tem uma janela, mas ele pode criar uma (ou mais), e a maioria o faz. Em ambos os sistemas, os usuários têm múltiplas janelas abertas de uma vez, cada uma executando algum processo. Utilizando o mouse, o usuário pode selecionar uma janela e interagir com o processo, por exemplo, fornecendo a entrada quando necessário.

A última situação na qual processos são criados aplica-se somente aos sistemas em lote encontrados em grandes computadores. Pense no gerenciamento de estoque ao fim de um dia em uma cadeia de lojas, nesse caso usuários podem submeter tarefas em lote ao sistema (possivelmente de maneira remota). Quando o sistema operacional decide que ele tem os recursos para executar outra tarefa, ele cria um novo processo e executa a próxima tarefa a partir da fila de entrada nele.

Tecnicamente, em todos esses casos, um novo processo é criado por outro já existente executando uma chamada de sistema de criação de processo. Esse outro processo pode ser um processo de usuário sendo executado, um processo de sistema invocado do teclado ou mouse, ou um processo gerenciador de lotes. O que esse processo faz é executar uma chamada de sistema para criar o novo processo. Essa chamada de sistema diz ao sistema operacional para criar um novo processo e indica, direta ou indiretamente, qual programa executar nele.

No UNIX, há apenas uma chamada de sistema para criar um novo processo: fork. Essa chamada cria um clone exato do processo que a chamou. Após a 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 então executa 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, por exemplo, *sort*, para o shell, este se bifurca gerando um processo filho, e o processo filho executa *sort*. O objetivo desse processo em dois passos é permitir que o processo filho manipule seus descritores de arquivos depois da fork, mas antes da execve, a fim de conseguir o redirecionamento de entrada padrão, saída padrão e erro padrão.

No Windows, em comparação, uma única chamada de função Win32, CreateProcess, lida tanto com a criação do processo, quanto com o carga do programa correto no novo processo. Essa chamada tem 10 parâmetros, que incluem o programa a ser executado, os parâmetros de linha de comando para alimentar aquele programa, vários atributos de segurança, bits que controlam se os arquivos abertos são herdados, informações sobre prioridades, uma especificação da janela a ser criada para o processo (se houver alguma) e um ponteiro para uma estrutura na qual as informações sobre o processo recentemente criado é retornada para quem o chamou. Além do CreateProcess, Win32 tem mais ou menos 100 outras funções para gerenciar e sincronizar processos e tópicos relacionados.

Tanto no sistema UNIX quanto no Windows, após um processo ser criado, o pai e o filho têm os seus próprios espaços de endereços distintos. Se um dos dois processos muda uma palavra no seu espaço de endereço, a mudança não é visível para o outro processo. No UNIX, o espaço de endereço inicial do filho é uma *cópia* do espaço de endereço do pai, mas há definitivamente dois espaços de endereços distintos envolvidos; nenhuma memória para escrita é compartilhada. Algumas implementações UNIX compartilham o programa de texto entre as duas, tendo em vista que isso não pode ser modificado. Alternativamente, o filho pode compartilhar toda a memória do pai, mas nesse caso, a memória é compartilhada no sistema **copy-on-write (cópia-na-escrita)**, o que significa que sempre que qualquer uma das duas quiser modificar parte da memória, aquele pedaço da memória é explicitamente copiado primeiro para certificar-se de que a modificação ocorra em uma área de memória privada. Novamente, nenhuma memória que pode ser escrita é compartilhada. É possível, no entanto, que um processo recentemente criado compartilhe de alguns dos outros recursos do seu criador, como arquivos abertos. No Windows, os espaços de endereços do pai e do filho são diferentes desde o início.

2.1.3 Término de processos

Após um processo ter sido criado, ele começa a ser executado e realiza qualquer que seja o seu trabalho. No

entanto, nada dura para sempre, nem mesmo os processos. Cedo ou tarde, o novo processo terminará, normalmente devido a uma das condições a seguir:

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

A maioria dos processos termina por terem realizado o seu trabalho. Quando um compilador termina de traduzir o programa dado a ele, o compilador executa uma chamada para dizer ao sistema operacional que ele terminou. Essa chamada é exit em UNIX e ExitProcess no Windows. Programas baseados em tela também dão suporte ao término voluntário. Processadores de texto, visualizadores da internet e programas similares sempre têm um ícone ou item no menu em que o usuário pode clicar para dizer ao processo para remover quaisquer arquivos temporários que ele tenha aberto e então concluí-lo.

A segunda razão para o término é a que o processo descobre um erro fatal. Por exemplo, se um usuário digita o comando

```
cc foo.c
```

para compilar o programa *foo.c* e não existe esse arquivo, o compilador simplesmente anuncia esse fato e termina a execução. Processos interativos com base em tela geralmente não fecham quando parâmetros ruins são dados. Em vez disso, eles abrem uma caixa de diálogo e pedem ao usuário para tentar de novo.

A terceira razão para o término é um erro causado pelo processo, muitas vezes decorrente de um erro de programa. Exemplos incluem executar uma instrução ilegal, referenciar uma memória não existente, ou dividir por zero. Em alguns sistemas (por exemplo, UNIX), um processo pode dizer ao sistema operacional que ele gostaria de lidar sozinho com determinados erros, nesse caso o processo é sinalizado (interrompido), em vez de terminado quando ocorrer um dos erros.

A quarta razão pela qual um processo pode ser finalizado ocorre quando o processo executa uma chamada de sistema dizendo ao sistema operacional para matar outro processo. Em UNIX, essa chamada é kill. A função Win32 correspondente é TerminateProcess. Em ambos os casos, o processo que mata o outro processo precisa da autorização necessária para fazê-lo. Em alguns sistemas, quando um processo é finalizado, seja voluntariamente ou de outra maneira, todos os processos que ele criou são de imediato mortos também. No entanto, nem o UNIX, tampouco 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 a ser associados de certas maneiras. O processo filho pode em si criar mais processos, formando uma hierarquia de processos. Observe que, diferentemente das plantas e dos animais que usam a reprodução sexual, um processo tem apenas um pai (mas zero, um, dois ou mais filhos). Então um processo lembra mais uma hidra do que, digamos, uma vaca.

Em UNIX, um processo e todos os seus filhos e demais descendentes formam juntos um grupo de processos. Quando um usuário envia um sinal do teclado, o sinal é entregue a todos os membros do grupo de processos associados com o teclado no momento (em geral todos os processos ativos que foram criados na janela atual). Individualmente, cada processo pode pegar o sinal, ignorá-lo, ou assumir a ação predefinida, que é ser morto pelo sinal.

Como outro exemplo de onde a hierarquia de processos tem um papel fundamental, vamos examinar como o UNIX se inicializa logo após o computador ser ligado. Um processo especial, chamado *init*, está presente na imagem de inicialização do sistema. Quando começa a ser executado, ele lê um arquivo dizendo quantos terminais existem, então ele se bifurca em um novo processo para cada terminal. Esses processos esperam que alguém se conecte. Se uma conexão é bem-sucedida, o processo de conexão executa um shell para aceitar os comandos. Esses comandos podem iniciar mais processos e assim por diante. Desse modo, todos os processos no sistema inteiro pertencem a uma única árvore, com *init* em sua raiz.

Em comparação, o Windows não tem conceito de uma hierarquia de processos. Todos os processos são iguais. O único indício de uma hierarquia ocorre quando um processo é criado e o pai recebe um identificador especial (chamado de **handle**) que ele pode usar para controlar o filho. No entanto, ele é livre para passar esse identificador para algum outro processo, desse modo invalidando a hierarquia. Processos em 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, processos muitas vezes precisam interagir entre si. Um processo pode gerar alguma saída que outro processo usa como entrada. No comando shell

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

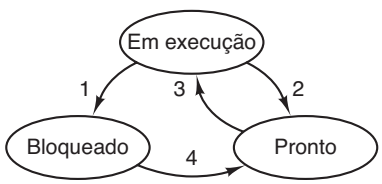
o primeiro processo, executando *cat*, gera como saída a concatenação dos três arquivos. O segundo processo, executando *grep*, seleciona todas as linhas contendo a palavra "tree". Dependendo das velocidades relativas dos dois processos (que dependem tanto da complexidade relativa dos programas, quanto do tempo de CPU que cada um teve), pode acontecer que *grep* esteja pronto para ser executado, mas não haja entrada esperando por ele. Ele deve então ser bloqueado até que alguma entrada esteja disponível.

Quando um processo bloqueia, ele o faz porque logicamente não pode continuar, em geral porque está esperando pela entrada que ainda não está disponível. Também é possível que um processo que esteja conceitualmente pronto e capaz de executar seja bloqueado porque o sistema operacional decidiu alocar a CPU para outro processo por um tempo. Essas duas condições são completamente diferentes. No primeiro caso, a suspensão é inerente ao problema (você não pode processar a linha de comando do usuário até que ela tenha sido digitada). No segundo caso, trata-se de uma tecnicalidade do sistema (não há CPUs suficientes para dar a cada processo seu próprio processador privado). Na Figura 2.2 vemos um diagrama de estado mostrando os três estados nos quais um processo pode se encontrar:

1. Em execução (realmente usando a CPU naquele instante).
2. Pronto (executável, temporariamente parado para deixar outro processo ser executado).
3. Bloqueado (incapaz de ser executado até que algum evento externo aconteça).

Claro, os primeiros dois estados são similares. Em ambos os casos, o processo está disposto a ser executado, apenas no segundo temporariamente não há uma CPU disponível para ele. O terceiro estado é fundamentalmente diferente dos dois primeiros, pois o processo não pode ser executado, mesmo que a CPU esteja ociosa e não tenha nada mais a fazer.

FIGURA 2.2 Um processo pode estar nos estados em execução, bloqueado ou pronto. Transições entre esses estados ocorrem como mostrado.



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

Como apresentado na Figura 2.2, quatro transições são possíveis entre esses três estados. A transição 1 ocorre quando o sistema operacional descobre que um processo não pode continuar agora. Em alguns sistemas o processo pode executar uma chamada de sistema, como em pause, para entrar em um estado bloqueado. Em outros, incluindo UNIX, quando um processo lê de um pipe ou de um arquivo especial (por exemplo, um terminal) e não há uma entrada disponível, o processo é automaticamente bloqueado.

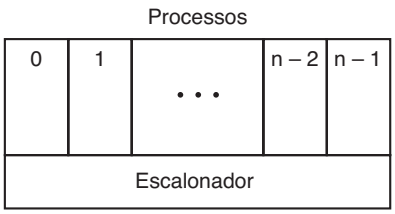
As transições 2 e 3 são causadas pelo escalonador de processos, uma parte do sistema operacional, sem o processo nem saber a respeito delas. A transição 2 ocorre quando o escalonador decide que o processo em andamento foi executado por tempo suficiente, e é o momento de deixar outro processo ter algum tempo de CPU. A transição 3 ocorre quando todos os outros processos tiveram sua parcela justa e está na hora de o primeiro processo chegar à CPU para ser executado novamente. O escalonamento, isto é, decidir qual processo deve ser executado, quando e por quanto tempo, é um assunto importante; nós o examinaremos mais adiante neste capítulo. Muitos algoritmos foram desenvolvidos para tentar equilibrar as demandas concorrentes de eficiência para o sistema como um todo e justiça para os processos individuais. Estudaremos algumas delas ainda neste capítulo.

A transição 4 se verifica quando o evento externo pelo qual um processo estava esperando (como a chegada de alguma entrada) acontece. Se nenhum outro processo estiver sendo executado naquele instante, a transição 3 será desencadeada e o processo começará a ser executado. Caso contrário, ele talvez tenha de esperar no estado de *pronto* por um intervalo curto até que a CPU esteja disponível e chegue sua vez.

Usando o modelo de processo, torna-se muito mais fácil pensar sobre o que está acontecendo dentro do sistema. Alguns dos processos executam programas que levam adiante comandos digitados pelo usuário. Outros processos são parte do sistema e lidam com tarefas como levar adiante solicitações para 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 uma decisão para parar de executar o processo atual e executa o processo de disco, que foi bloqueado esperando por essa interrupção. Assim, em vez de pensar a respeito de interrupções, podemos pensar sobre os processos de usuários, processos de disco, processos terminais e assim por diante, que bloqueiam quando estão esperando que algo aconteça. Quando o disco foi lido ou o caractere digitado, o processo esperando por ele é desbloqueado e está disponível para ser executado novamente.

Essa visão dá origem ao modelo mostrado na Figura 2.3. Nele, o nível mais baixo do sistema operacional é o escalonador, com uma variedade de processos acima dele. Todo o tratamento de interrupções e detalhes sobre o início e parada de processos estão ocultos naquilo que é chamado aqui de escalonador, que, na verdade, não tem muito código. O resto do sistema operacional é bem estruturado na forma de processos. No entanto, poucos sistemas reais são tão bem estruturados como esse.

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



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 um deles. (Alguns autores chamam essas entradas de **blocos de controle de processo**.) Essas entradas contêm informações importantes sobre o estado do processo, incluindo o seu contador de programa, ponteiro de pilha, alocação de memória, estado dos arquivos abertos, informação sobre sua contabilidade e escalonamento e tudo o mais que deva ser salvo quando o processo é trocado do estado *em execução* para *pronto* ou *bloqueado*, de maneira que ele possa ser reiniciado mais tarde como se nunca tivesse sido parado.

A Figura 2.4 mostra alguns dos campos fundamentais em um sistema típico: os campos na primeira coluna relacionam-se ao gerenciamento de processo. Os outros dois relacionam-se ao gerenciamento de memória e de arquivos, respectivamente. Deve-se observar que precisamente quais campos cada tabela de processo tem é algo altamente dependente do sistema, mas esse número dá uma ideia geral dos tipos de informações necessárias.

Agora que examinamos a tabela de processo, é possível explicar um pouco mais sobre como a ilusão de múltiplos processos sequenciais é mantida em uma (ou cada) CPU. Associada com cada classe de E/S há um local (geralmente em um local fixo próximo da parte inferior da memória) chamado de **vetor de interrupção**. Ele contém o endereço da rotina de serviço de interrupção. Suponha que o processo do usuário 3 esteja sendo executado quando ocorre uma interrupção de disco. O contador de programa do processo do usuário 3, palavra de estado de programa, e, às vezes, um ou mais registradores são colocados na pilha (atual) pelo hardware de interrupção. O computador, então, desvia a execução

FIGURA 2.4 Alguns dos campos de uma entrada típica na tabela de processos.

Gerenciamento de processo	Gerenciamento de memória	Gerenciamento de arquivo
Registros	Ponteiro para informações sobre o segmento de texto	Diretório-raiz
Contador de programa		Diretório de trabalho
Palavra de estado do programa	Ponteiro para informações sobre o segmento de dados	Descritores de arquivo
Ponteiro da pilha		ID do usuário
Estado do processo	Ponteiro para informações sobre o segmento de pilha	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		

para o endereço especificado no vetor de interrupção. Isso é tudo o que o hardware faz. Daqui em diante, é papel do software, em particular, realizar a rotina do serviço de interrupção.

Todas as interrupções começam salvando os registradores, muitas vezes na entrada da tabela de processo para o processo atual. Então a informação empurrada para a pilha pela interrupção é removida e o ponteiro de pilha é configurado para apontar para uma pilha temporária usada pelo tratador de processos. Ações como salvar os registradores e configurar o ponteiro da pilha não podem ser expressas em linguagens de alto nível, como C, por isso elas são desempenhadas por uma pequena rotina de linguagem de montagem, normalmente a mesma para todas as interrupções, já que o trabalho de salvar os registros é idêntico, não importa qual seja a causa da interrupção.

Quando essa rotina é concluída, ela chama uma rotina C para fazer o resto do trabalho para esse tipo específico de interrupção. (Presumimos que o sistema operacional seja escrito em C, a escolha mais comum para todos os sistemas operacionais reais). Quando o trabalho tiver sido concluído, possivelmente deixando algum processo agora pronto, o escalonador é chamado para ver qual é o próximo processo a ser executado. Depois disso, o controle é passado de volta ao código de linguagem de montagem para carregar os registradores e mapa de memória para o processo agora atual e iniciar a sua execução. O tratamento e o escalonamento de interrupção estão resumidos na Figura 2.5. Vale a pena observar que os detalhes variam de alguma maneira de sistema para sistema.

FIGURA 2.5 O esqueleto do que o nível mais baixo do sistema operacional faz quando ocorre uma interrupção.

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 vetor de interrupções 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.

Um processo pode ser interrompido milhares de vezes durante sua execução, mas a ideia fundamental é que, após cada interrupção, o processo retorne precisamente para o mesmo estado em que se encontrava antes de ser interrompido.

2.1.7 Modelando a multiprogramação

Quando a multiprogramação é usada, a utilização da CPU pode ser aperfeiçoada. Colocando a questão de maneira direta, se o processo médio realiza computações apenas 20% do tempo em que está na memória, então com cinco processos ao mesmo tempo na memória, a CPU deve estar ocupada o tempo inteiro. Entretanto, esse modelo é irrealisticamente otimista, tendo em vista que ele presume de modo tácito que todos os cinco processos jamais estarão esperando por uma E/S ao mesmo tempo.

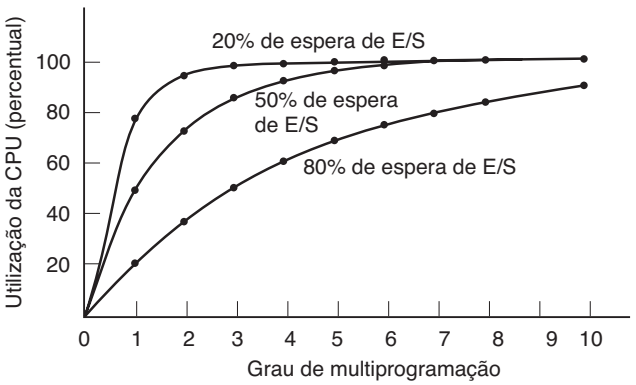
Um modelo melhor é examinar o uso da CPU a partir de um ponto de vista probabilístico. Suponha que um processo passe uma fração p de seu tempo esperando que os dispositivos de E/S sejam concluídos. Com n processos na memória ao mesmo tempo, a probabilidade de que todos os processos n estejam esperando para E/S (caso em que a CPU estará ociosa) é p^n. A utilização da CPU é então dada pela fórmula

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

A Figura 2.6 mostra a utilização da CPU como uma função de n, que é chamada de **grau de multiprogramação**.

Segundo a figura, fica claro que se os processos passam 80% do tempo esperando por dispositivos de E/S, pelo menos 10 processos devem estar na memória ao mesmo tempo para que a CPU desperdice menos de 10%. Quando você percebe que um processo interativo esperando por um usuário para digitar algo em um terminal (ou clicar em um ícone) está no estado de espera

FIGURA 2.6 Utilização da CPU como uma função do número de processos na memória.



de E/S, deve ficar claro que tempos de espera de E/S de 80% ou mais não são incomuns. Porém mesmo em servidores, processos executando muitas operações de E/S em disco muitas vezes terão essa percentagem ou mais.

Levando em consideração a precisão, deve ser destacado que o modelo probabilístico descrito há pouco é apenas uma aproximação. Ele presume implicitamente que todos os *n* processos são independentes, significando que é bastante aceitável para um sistema com cinco processos na memória ter três em execução e dois esperando. Mas com uma única CPU, não podemos ter três processos sendo executados ao mesmo tempo, portanto o processo que ficar pronto enquanto a CPU está ocupada terá de esperar. Então, os processos não são independentes. Um modelo mais preciso pode ser construído usando a teoria das filas, mas o ponto que estamos sustentando — a multiprogramação deixa que os processos usem a CPU quando ela estaria em outras circunstâncias ociosa — ainda é válido, mesmo que as verdadeiras curvas da Figura 2.6 sejam ligeiramente diferentes daquelas mostradas na imagem.

Embora o modelo da Figura 2.6 seja bastante simples, ele pode ser usado para realizar previsões específicas, embora aproximadas, a respeito do desempenho da CPU. Suponha, por exemplo, que um computador tenha 8 GB de memória, com o sistema operacional e suas tabelas ocupando 2 GB e cada programa de usuário também ocupando 2 GB. Esses tamanhos permitem que três programas de usuários estejam na memória simultaneamente. Com uma espera de E/S média de 80%, temos uma utilização de CPU (ignorando a sobrecarga do sistema operacional) de $1 - 0{,}8^3$ ou em torno de 49%. Acrescentar outros 8 GB de memória permite que o sistema aumente seu grau de multiprogramação de três para sete, aumentando desse modo a utilização da CPU para 79%. Em outras palavras, os 8 GB adicionais aumentarão a utilização da CPU em 30%.

Acrescentar outros 8 GB ainda aumentaria a utilização da CPU apenas de 79% para 91%, desse modo elevando a utilização da CPU em apenas 12% a mais. Usando esse modelo, o proprietário do computador pode decidir que a primeira adição foi um bom investimento, mas a segunda, não.

2.2 Threads

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

2.2.1 Utilização de threads

Por que alguém iria querer ter um tipo de processo dentro de um processo? Na realidade, há várias razões para a existência desses miniprocessos, chamados **threads**. Vamos examinar agora algumas delas. A principal razão para se ter threads é que em muitas aplicações múltiplas atividades estão ocorrendo simultaneamente e algumas delas podem bloquear de tempos em tempos. Ao decompormos uma aplicação dessas em múltiplos threads sequenciais que são executados em quase paralelo, o modelo de programação torna-se mais simples.

Já vimos esse argumento antes. É precisamente o argumento para se ter processos. Em vez de pensar a respeito de interrupções, temporizadores e chaveamentos de contextos, podemos pensar a respeito de processos em paralelo. Apenas agora com os threads acrescentamos um novo elemento: a capacidade para as entidades em paralelo compartilharem um espaço de endereçamento e todos os seus dados entre si. Essa capacidade é essencial para determinadas aplicações, razão pela qual ter múltiplos processos (com seus espaços de endereçamento em separado) não funcionará.

Um segundo argumento para a existência dos threads é que como eles são mais leves do que os processos, eles são mais fáceis (isto é, mais rápidos) para criar e destruir do que os processos. Em muitos sistemas, criar um thread é algo de 10 a 100 vezes mais rápido do que criar um processo. Quando o número necessário de threads muda dinâmica e rapidamente, é útil se contar com essa propriedade.

Uma terceira razão para a existência de threads também é o argumento do desempenho. O uso de threads não resulta em um ganho de desempenho quando todos eles são limitados pela CPU, mas quando há uma computação substancial e também E/S substancial, contar com threads permite que essas atividades se sobreponham, acelerando desse modo a aplicação.

Por fim, threads são úteis em sistemas com múltiplas CPUs, onde o paralelismo real é possível. Voltaremos a essa questão no Capítulo 8.

É mais fácil ver por que os threads são úteis observando alguns exemplos concretos. Como um primeiro

exemplo, considere um processador de texto. Processadores de texto em geral exibem o documento que está sendo criado em uma tela formatada exatamente como aparecerá na página impressa. Em particular, todas as quebras de linha e quebras de página estão em suas posições finais e corretas, de maneira que o usuário pode inspecioná-las e mudar o documento se necessário (por exemplo, eliminar viúvas e órfãos — linhas incompletas no início e no fim das páginas, que são consideradas esteticamente desagradáveis).

Suponha que o usuário esteja escrevendo um livro. Do ponto de vista de um autor, é mais fácil manter o livro inteiro como um único arquivo para tornar mais fácil buscar por tópicos, realizar substituições globais e assim por diante. Como alternativa, cada capítulo pode ser um arquivo em separado. No entanto, ter cada seção e subseção como um arquivo em separado é um verdadeiro inconveniente quando mudanças globais precisam ser feitas para o livro inteiro, visto que centenas de arquivos precisam ser individualmente editados, um de cada vez. Por exemplo, se o padrão xxxx proposto é aprovado um pouco antes de o livro ser levado para impressão, todas as ocorrências de "Padrão provisório xxxx" têm de ser modificadas para "Padrão xxxx" no último minuto. Se o livro inteiro for um arquivo, em geral um único comando pode realizar todas as substituições. Em comparação, se o livro estiver dividido em mais de 300 arquivos, cada um deve ser editado separadamente.

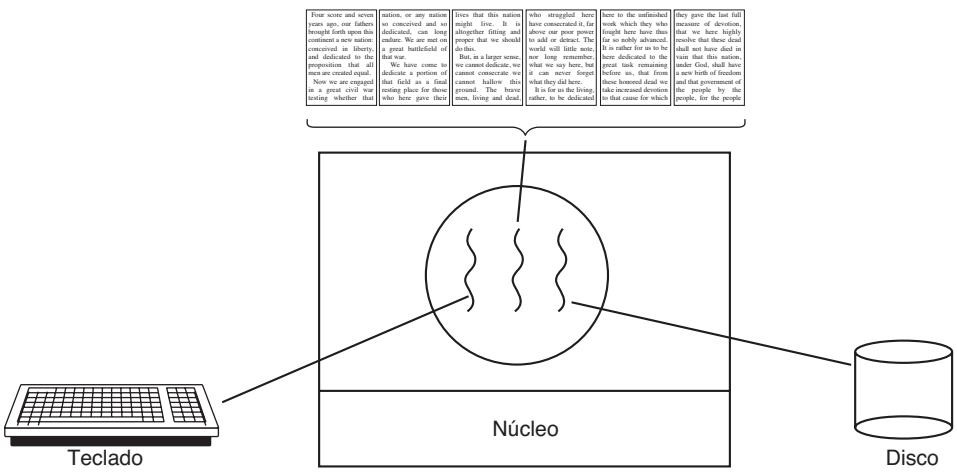
Agora considere o que acontece quando o usuário subitamente apaga uma frase da página 1 de um livro de 800 páginas. Após conferir a página modificada para assegurar-se de que está corrigida, ele agora quer fazer outra mudança na página 600 e digita um comando dizendo ao processador de texto para ir até aquela página (possivelmente procurando por uma frase ocorrendo apenas ali). O processador de texto agora é forçado a reformatar o livro inteiro até a página 600, algo difícil, pois ele não sabe qual será a primeira linha da página 600 até ter processado todas as páginas anteriores. Pode haver um atraso substancial antes que a página 600 seja exibida, resultando em um usuário infeliz.

Threads podem ajudar aqui. Suponha que o processador de texto seja escrito como um programa com dois threads. Um thread interage com o usuário e o outro lida com a reformatação em segundo plano. Tão logo a frase é apagada da página 1, o thread interativo diz ao de reformatação para reformatar o livro inteiro. Enquanto isso, o thread interativo continua a ouvir o teclado e o mouse e responde a comandos simples como rolar a página 1 enquanto o outro thread está trabalhando com afinco no segundo plano. Com um pouco de sorte, a reformatação será concluída antes que o usuário peça para ver a página 600, então ela pode ser exibida instantaneamente.

Enquanto estamos nesse exemplo, por que não acrescentar um terceiro thread? Muitos processadores de texto têm a capacidade de salvar automaticamente o arquivo inteiro para o disco em intervalos de poucos minutos para proteger o usuário contra o perigo de perder um dia de trabalho caso o programa ou o sistema trave ou falte luz. O terceiro thread pode fazer *backups* de disco sem interferir nos outros dois. A situação com os três threads é mostrada na Figura 2.7.

Se o programa tivesse apenas um thread, então sempre que um *backup* de disco fosse iniciado, comandos do teclado e do mouse seriam ignorados até que o *backup* tivesse sido concluído. O usuário certamente perceberia isso como um desempenho lento. Como alternativa,

FIGURA 2.7 Um processador de texto com três threads.



eventos do teclado e do mouse poderiam interromper o *backup* do disco, permitindo um bom desempenho, mas levando a um modelo de programação complexo orientado à interrupção. Com três threads, o modelo de programação é muito mais simples: o primeiro thread apenas interage com o usuário, o segundo reformata o documento quando solicitado, o terceiro escreve os conteúdos da RAM para o disco periodicamente.

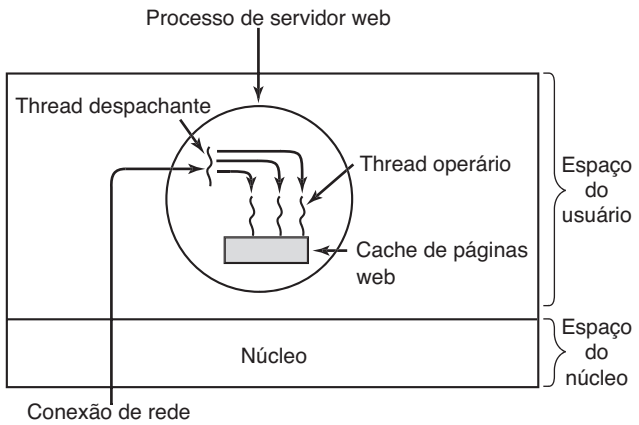
Deve ficar claro que ter três processos em separado não funcionaria aqui, pois todos os três threads precisam operar no documento. Ao existirem três threads em vez de três processos, eles compartilham de uma memória comum e desse modo têm acesso ao documento que está sendo editado. Com três processos isso seria impossível.

Uma situação análoga existe com muitos outros programas interativos. Por exemplo, uma planilha eletrônica é um programa que permite a um usuário manter uma matriz, na qual alguns elementos são dados fornecidos pelo usuário e outros são calculados com base nos dados de entrada usando fórmulas potencialmente complexas. Quando um usuário muda um elemento, muitos outros precisam ser recalculados. Ao ter um thread de segundo plano para o recálculo, o thread interativo pode permitir ao usuário fazer mudanças adicionais enquanto o cálculo está sendo realizado. De modo similar, um terceiro thread pode cuidar sozinho dos backups periódicos para o disco.

Agora considere mais um exemplo onde os threads são úteis: um servidor para um website. Solicitações para páginas chegam e a página solicitada é enviada de volta para o cliente. Na maioria dos websites, algumas páginas são mais acessadas do que outras. Por exemplo, a página principal da Sony é acessada muito mais do que uma página mais profunda na árvore contendo as especificações técnicas de alguma câmera em particular. Servidores da web usam esse fato para melhorar o desempenho mantendo uma coleção de páginas intensamente usadas na memória principal para eliminar a necessidade de ir até o disco para buscá-las. Essa coleção é chamada de **cache** e é usada em muitos outros contextos também. Vimos caches de CPU no Capítulo 1, por exemplo.

Uma maneira de organizar o servidor da web é mostrada na Figura 2.8(a). Aqui, um thread, o **despachante**, lê as requisições de trabalho que chegam da rede. Após examinar a solicitação, ele escolhe um **thread operário** ocioso (isto é, bloqueado) e passa para ele a solicitação, possivelmente escrevendo um ponteiro para a mensagem em uma palavra especial associada com cada thread. O despachante então acorda o operário adormecido, movendo-o do estado bloqueado para o estado pronto.

FIGURA 2.8 Um servidor web multithread.



Quando o operário desperta, ele verifica se a solicitação pode ser satisfeita a partir do cache da página da web, ao qual todos os threads têm acesso. Se não puder, ele começa uma operação read para conseguir a página do disco e é bloqueado até a operação de disco ser concluída. Quando o thread é bloqueado na operação de disco, outro thread é escolhido para ser executado, talvez o despachante, a fim de adquirir mais trabalho, ou possivelmente outro operário esteja pronto para ser executado agora.

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 aceita uma solicitação de um despachante e confere a cache da web para ver se a página está presente. Se estiver, ela é devolvida ao cliente, e o operário é bloqueado aguardando por uma nova solicitação. Se não estiver, ele pega a página do disco, retorna-a ao cliente e é bloqueado esperando por uma nova solicitação.

Um esquema aproximado do código é dado na Figura 2.9. Aqui, como no resto deste livro, *TRUE* é presumido que seja a constante 1. Do mesmo modo, *buf* e *page* são estruturas apropriadas para manter uma solicitação de trabalho e uma página da web, respectivamente.

Considere como o servidor web teria de ser escrito na ausência de threads. Uma possibilidade é fazê-lo operar um único thread. O laço principal do servidor web recebe uma solicitação, examina-a e a conduz até sua conclusão antes de receber a próxima. Enquanto espera pelo disco, o servidor está ocioso e não processa nenhum outro pedido chegando. Se o servidor web estiver sendo executado em uma máquina dedicada, como é o caso no geral, a CPU estará simplesmente ociosa

FIGURA 2.9 Um esquema aproximado do código para a Figura 2.8. (a) Thread despachante. (b) Thread operário.

```
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)

enquanto o servidor estiver esperando pelo disco. O resultado final é que muito menos solicitações por segundo poderão ser processadas. Assim, threads ganham um desempenho considerável, mas cada thread é programado sequencialmente, como de costume.

Até o momento, vimos dois projetos possíveis: um servidor web multithread e um servidor web com um único thread. Suponha que múltiplos threads não estejam disponíveis, mas que os projetistas de sistemas consideram inaceitável a perda de desempenho decorrente do único thread. Se uma versão da chamada de sistema read sem bloqueios estiver disponível, uma terceira abordagem é possível. Quando uma solicitação chegar, o único thread a examina. Se ela puder ser satisfeita a partir da cache, ótimo, se não, uma operação de disco sem bloqueios é inicializada.

O servidor registra o estado da solicitação atual em uma tabela e então lida com o próximo evento. O próximo evento pode ser uma solicitação para um novo trabalho ou uma resposta do disco sobre uma operação anterior. Se for um novo trabalho, esse trabalho é iniciado. Se for uma resposta do disco, a informação relevante é buscada da tabela e a resposta processada. Com um sistema de E/S de disco sem bloqueios, uma resposta provavelmente terá de assumir a forma de um sinal ou interrupção.

Nesse projeto, o modelo de "processo sequencial" que tínhamos nos primeiros dois casos é perdido. O estado da computação deve ser explicitamente salvo e restaurado na tabela toda vez que o servidor chaveia do trabalho de uma solicitação para outra. Na realidade, estamos simulando os threads e suas pilhas do jeito mais difícil. Um projeto como esse, no qual cada computação tem um estado salvo e existe algum conjunto de eventos que pode ocorrer para mudar o estado, é chamado de **máquina de estados finitos**. Esse conceito é amplamente usado na ciência de computação.

Deve estar claro agora o que os threads têm a oferecer. Eles tornam possível reter a ideia de processos sequenciais que fazem chamadas bloqueantes (por exemplo, para E/S de disco) e ainda assim alcançar o paralelismo. Chamadas de sistema bloqueantes tornam a programação mais fácil, e o paralelismo melhora o desempenho. O servidor de thread único retém a simplicidade das chamadas de sistema bloqueantes, mas abre mão do desempenho. A terceira abordagem alcança um alto desempenho por meio do paralelismo, mas usa chamadas não bloqueantes e interrupções, e assim é difícil de programar. Esses modelos são resumidos na Figura 2.10.

Um terceiro exemplo em que threads são úteis encontra-se nas aplicações que precisam processar grandes quantidades de dados. Uma abordagem normal é ler em um bloco de dados, processá-lo e então escrevê-lo de novo. O problema aqui é que se houver apenas a disponibilidade de chamadas de sistema bloqueantes, o processo é bloqueado enquanto os dados estão chegando e saindo. Ter uma CPU ociosa quando há muita computação a ser feita é um claro desperdício e deve ser evitado se possível.

Threads oferecem uma solução: o processo poderia ser estruturado com um thread de entrada, um de processamento e um de saída. O thread de entrada lê dados para um buffer de entrada; o thread de processamento pega os dados do buffer de entrada, processa-os e coloca os resultados no buffer de saída; e o thread de saída escreve esses resultados de volta para o disco. Dessa maneira, entrada, saída e processamento podem estar todos acontecendo ao mesmo tempo. É claro que esse modelo funciona somente se uma chamada de sistema bloqueia apenas o thread de chamada, não o processo inteiro.

FIGURA 2.10 Três maneiras de construir um servidor.

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

2.2.2 O modelo de thread clássico

Agora que vimos por que os threads podem ser úteis e como eles podem ser usados, vamos investigar a ideia um pouco mais de perto. O modelo de processo é baseado em dois conceitos independentes: agrupamento de recursos e execução. Às vezes é útil separá-los; é onde os threads entram. Primeiro, examinaremos o modelo de thread clássico; depois disso veremos o modelo de thread Linux, que torna indistintas as diferenças entre processos e threads.

Uma maneira de se ver um processo é que ele é um modo para agrupar recursos relacionados. Um processo tem um espaço de endereçamento contendo o código e os dados do programa, assim como outros recursos. Esses recursos podem incluir arquivos abertos, processos filhos, alarmes pendentes, tratadores de sinais, informação sobre contabilidade e mais. Ao colocá-los juntos na forma de um processo, eles podem ser gerenciados com mais facilidade.

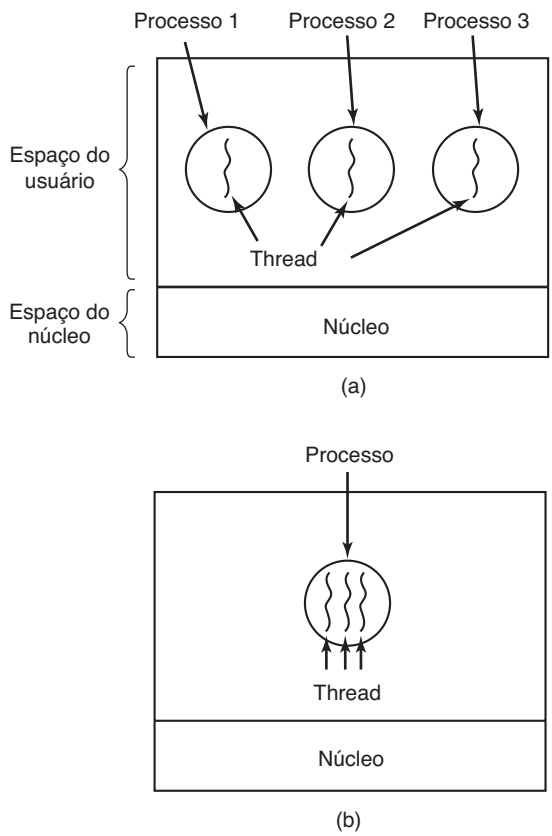
O outro conceito que um processo tem é de uma linha (*thread*) de execução, normalmente abreviado para apenas **thread**. O thread tem um contador de programa que controla qual instrução deve ser executada em seguida. Ele tem registradores, que armazenam suas variáveis de trabalho atuais. Tem uma pilha, que contém o histórico de execução, com uma estrutura para cada rotina chamada, mas ainda não retornada. Embora um thread deva executar em algum processo, 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 execução na CPU.

O que os threads acrescentam para o modelo de processo é permitir que ocorram múltiplas execuções no mesmo ambiente, com um alto grau de independência uma da outra. Ter múltiplos threads executando em paralelo em um processo equivale a ter múltiplos processos executando em paralelo em um computador. No primeiro caso, os threads compartilham um espaço de endereçamento e outros recursos. No segundo caso, os processos compartilham memórias físicas, discos, impressoras e outros recursos. Como threads têm algumas das propriedades dos processos, às vezes eles são chamados de **processos leves**. O termo **multithread** também é usado para descrever a situação de permitir múltiplos threads no mesmo processo. Como vimos no Capítulo 1, algumas CPUs têm suporte de hardware direto para multithread e permitem que chaveamentos de threads aconteçam em uma escala de tempo de nanossegundos.

Na Figura 2.11(a) vemos três processos tradicionais. Cada processo tem seu próprio espaço de endereçamento e um único thread de controle. Em comparação, na Figura 2.11(b) vemos um único processo com três threads de controle. Embora em ambos os casos tenhamos três threads, na Figura 2.11(a) cada um deles opera em um espaço de endereçamento diferente, enquanto na Figura 2.11(b) todos os três compartilham o mesmo espaço de endereçamento.

Quando um processo multithread é executado em um sistema de CPU única, os threads se revezam executando. Na Figura 2.1, vimos como funciona a multiprogramação de processos. Ao chavear entre múltiplos processos, o sistema passa a ilusão de processos sequenciais executando em paralelo. O multithread funciona da mesma maneira. A CPU chaveia rapidamente entre os threads, dando a ilusão de que eles estão executando em paralelo, embora em uma CPU mais lenta do que a real. Em um processo limitado pela CPU com três threads, eles pareceriam executar em paralelo, cada um em uma CPU com um terço da velocidade da CPU real.

FIGURA 2.11 (a) Três processos, cada um com um thread. (b) Um processo com três threads.



Threads diferentes em um processo não são tão independentes quanto processos diferentes. Todos os threads têm exatamente o mesmo espaço de endereçamento, o que significa que eles também compartilham as mesmas variáveis globais. Tendo em vista que todo thread pode acessar todo espaço de endereçamento de memória dentro do espaço de endereçamento do processo, um thread pode ler, escrever, ou mesmo apagar a pilha de outro thread. Não há proteção entre threads, porque (1) é impossível e (2) não seria necessário. Ao contrário de processos distintos, que podem ser de usuários diferentes e que podem ser hostis uns com os outros, um processo é sempre propriedade de um único usuário, que presumivelmente criou múltiplos threads de maneira que eles possam cooperar, não lutar. Além de compartilhar um espaço de endereçamento, todos os threads podem compartilhar o mesmo conjunto de arquivos abertos, processos filhos, alarmes e sinais, e assim por diante, como mostrado na Figura 2.12. Assim, a organização da Figura 2.11(a) seria usada quando os três processos forem essencialmente não relacionados, enquanto a Figura 2.11(b) seria apropriada quando os três threads fizerem na realidade parte do mesmo trabalho e estiverem cooperando uns com os outros de maneira ativa e próxima.

Na Figura 2.12, os itens na primeira coluna são propriedades de processos, não threads de propriedades. Por exemplo, se um thread abre um arquivo, esse arquivo fica visível aos outros threads no processo e eles podem ler e escrever nele. Isso é lógico, já que o processo, e não o thread, é a unidade de gerenciamento de recursos. Se cada thread tivesse o seu próprio espaço de endereçamento, arquivos abertos, alarmes pendentes e assim por diante, seria um processo em separado. O que estamos tentando alcançar com o conceito de thread é a capacidade para múltiplos threads de execução de compartilhar um conjunto de recursos de maneira que possam trabalhar juntos intimamente para desempenhar alguma tarefa.

Como um processo tradicional (isto é, um processo com apenas um thread), um thread pode estar em qualquer um de vários estados: em execução, bloqueado, pronto, ou concluído. Um thread em execução tem a CPU naquele momento e está ativo. Em comparação, um thread bloqueado está esperando por algum evento para desbloqueá-lo. Por exemplo, quando um thread realiza uma chamada de sistema para ler do teclado, ele está bloqueado até que uma entrada seja digitada. Um thread pode bloquear esperando por algum evento externo acontecer ou por algum outro thread para desbloqueá-lo. Um thread pronto está programado para ser executado e o será tão logo chegue a sua vez. As transições entre estados de thread são as mesmas que aquelas entre estados de processos e estão ilustradas na Figura 2.2.

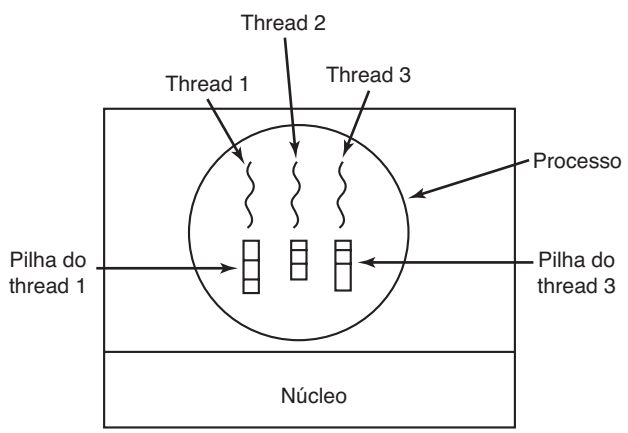
É importante perceber que cada thread tem a sua própria pilha, como ilustrado na Figura 2.13. Cada pilha do thread contém uma estrutura para cada rotina chamada, mas ainda não retornada. Essa estrutura contém as variáveis locais da rotina e o endereço de retorno para usar quando a chamada de rotina for encerrada. Por exemplo, se a rotina *X* chama a rotina *Y* e *Y* chama a rotina *Z*, então enquanto *Z* está executando, as estruturas para *X, Y* e *Z* estarão todas na pilha. Cada thread geralmente chamará rotinas diferentes e desse modo terá uma história de execução diferente. Essa é a razão pela qual cada thread precisa da sua própria pilha.

Quando o multithreading está presente, os processos normalmente começam com um único thread presente. Esse thread tem a capacidade de criar novos, chamando

FIGURA 2.12 A primeira coluna lista alguns itens compartilhados por todos os threads em um processo. A segunda lista alguns itens específicos a cada thread.

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 tratadores de sinais	
Informação de contabilidade	

FIGURA 2.13 Cada thread tem a sua própria pilha.



uma rotina de biblioteca como *thread_create*. Um parâmetro para *thread_create* especifica o nome de uma rotina para o novo thread executar. Não é necessário (ou mesmo possível) especificar algo sobre o espaço de endereçamento do novo thread, tendo em vista que ele automaticamente é executado no espaço de endereçamento do thread em criação. Às vezes, threads são hierárquicos, com uma relação pai-filho, mas muitas vezes não existe uma relação dessa natureza, e todos os threads são iguais. Com ou sem uma relação hierárquica, normalmente é devolvido ao thread em criação um identificador de thread que nomeia o novo thread.

Quando um thread tiver terminado o trabalho, pode concluir sua execução chamando uma rotina de biblioteca, como *thread_exit*. Ele então desaparece e não é mais escalonável. Em alguns sistemas, um thread pode esperar pela saída de um thread (específico) chamando uma rotina, por exemplo, *thread_join*. Essa rotina bloqueia o thread que executou a chamada até que um thread (específico) tenha terminado. Nesse sentido, a criação e a conclusão de threads é muito semelhante à criação e ao término de processos, com mais ou menos as mesmas opções.

Outra chamada de thread comum é *thread_yield*, que permite que um thread abra mão voluntariamente da CPU para deixar outro thread ser executado. Uma chamada dessas é importante porque não há uma interrupção de relógio para realmente forçar a multiprogramação como há com os processos. Desse modo, é importante que os threads sejam educados e voluntariamente entreguem a CPU de tempos em tempos para dar aos outros threads uma chance de serem executados. Outras chamadas permitem que um thread espere por outro thread para concluir algum trabalho, para um thread anunciar que terminou alguma tarefa e assim por diante.

Embora threads sejam úteis na maioria das vezes, eles também introduzem uma série de complicações no modelo de programação. Para começo de conversa, considere os efeitos da chamada fork de sistema UNIX. Se o processo pai tem múltiplos threads, o filho não deveria tê-los também? Do contrário, é possível que o processo não funcione adequadamente, tendo em vista que todos eles talvez sejam essenciais.

No entanto, se o processo filho possuir tantos threads quanto o pai, o que acontece se um thread no pai estava bloqueado em uma chamada read de um teclado? Dois threads estão agora bloqueados no teclado, um no pai e outro no filho? Quando uma linha é digitada, ambos os threads recebem uma cópia? Apenas o pai? Apenas o filho? O mesmo problema existe com conexões de rede abertas.

Outra classe de problemas está relacionada ao fato de que threads compartilham muitas estruturas de dados. O que acontece se um thread fecha um arquivo enquanto outro ainda está lendo dele? Suponha que um thread observe que há pouca memória e comece a alocar mais memória. No meio do caminho há um chaveamento de threads, e o novo também observa que há pouca memória e também começa a alocar mais memória. A memória provavelmente será alocada duas vezes. Esses problemas podem ser solucionados com algum esforço, mas os programas de multithread devem ser pensados e projetados com cuidado para funcionarem corretamente.

2.2.3 Threads POSIX

Para possibilitar que se escrevam 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 **Pthreads**. A maioria dos sistemas UNIX dá suporte a ele. O padrão define mais de 60 chamadas de função, um número grande demais para ser visto aqui. Em vez disso, descreveremos apenas algumas das principais para dar uma ideia de como funcionam. As chamadas que descreveremos a seguir estão listadas na Figura 2.14.

Todos os threads têm determinadas propriedades. Cada um tem um identificador, um conjunto de registradores (incluindo o contador de programa), e um conjunto de atributos, que são armazenados em uma estrutura. Os atributos incluem tamanho da pilha, parâmetros de escalonamento e outros itens necessários para usar o thread.

FIGURA 2.14 Algumas das chamadas de função do Pthreads.

Chamada de thread	Descrição
Pthread_create	Cria um novo thread
Pthread_exit	Conclui a chamada de thread
Pthread_join	Espera que um thread específico seja abandonado
Pthread_yield	Libera a CPU para que outro thread seja executado
Pthread_attr_init	Cria e inicializa uma estrutura de atributos do thread
Pthread_attr_destroy	Remove uma estrutura de atributos do thread

Um novo thread é criado usando a chamada *pthread_create*. O identificador de um thread recentemente criado é retornado como o valor da função. Essa chamada é intencionalmente muito parecida com a chamada de sistema fork (exceto pelos parâmetros), com o identificador de thread desempenhando o papel do PID (número de processo), em especial para identificar threads referenciados em outras chamadas.

Quando um thread tiver acabado o trabalho para o qual foi designado, ele pode terminar chamando *pthread_exit*. Essa chamada para o thread e libera sua pilha.

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

Às vezes acontece de um thread não estar logicamente bloqueado, mas sente que já foi executado tempo suficiente e quer dar a outro thread a chance de ser executado. Ele pode atingir essa meta chamando *pthread_yield*. Essa chamada não existe para processos, pois o pressuposto aqui é que os processos são altamente competitivos e cada um quer o tempo de CPU que conseguir obter. No entanto, já que os threads de um processo estão trabalhando juntos e seu código é invariavelmente escrito pelo mesmo programador, às vezes o programador quer que eles se deem outra chance.

As duas chamadas seguintes de thread lidam com atributos. *Pthread_attr_init* cria a estrutura de atributos associada com um thread e o inicializa com os valores padrão. Esses valores (como a prioridade) podem ser modificados manipulando campos na estrutura de atributos.

Por fim, *pthread_attr_destroy* remove a estrutura de atributos de um thread, liberando a sua memória. Essa chamada não afeta os threads que a usam; eles continuam a existir.

FIGURA 2.15 Um exemplo de programa usando threads.

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

#define NUMBER_OF_THREADS     10

void *print_hello_world(void *tid)
{
        /* Esta funcao imprime o identificador do thread e sai. */
        printf("Ola mundo. Boas vindas do 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("Metodo Main. Criando 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);
}
```

Para ter uma ideia melhor de como o Pthreads funciona, considere o exemplo simples da Figura 2.15. Aqui o principal programa realiza *NUMBER_OF_THREADS* iterações, criando um novo thread em cada iteração, após anunciar sua intenção. Se a criação do thread fracassa, ele imprime uma mensagem de erro e então termina. Após criar todos os threads, o programa principal termina.

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

As chamadas Pthreads descritas anteriormente não são as únicas. Examinaremos algumas das outras após discutir a sincronização de processos e threads.

2.2.4 Implementando threads no espaço do usuário

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

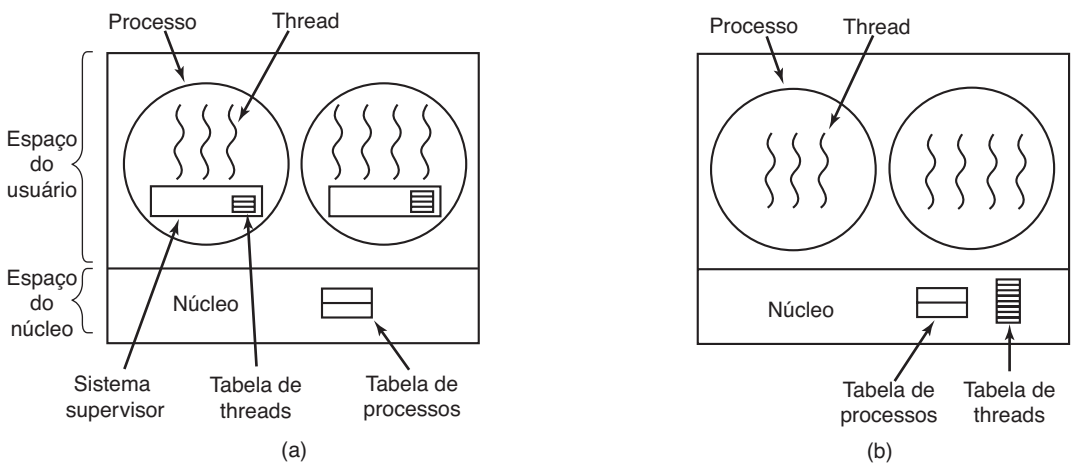
O primeiro método é colocar o pacote de threads inteiramente no espaço do usuário. O núcleo não sabe nada a respeito deles. Até onde o núcleo sabe, ele está gerenciando processos comuns de um único thread. A primeira vantagem, e mais óbvia, é que o pacote de threads no nível do usuário pode ser implementado em um sistema operacional que não dá suporte aos threads. Todos os sistemas operacionais costumavam cair nessa categoria, e mesmo agora alguns ainda caem. Com essa abordagem, threads são implementados por uma biblioteca.

Todas essas implementações têm a mesma estrutura geral, ilustrada na Figura 2.16(a). Os threads executam em cima de um sistema de tempo de execução, que é uma coleção de rotinas que gerencia os threads. Já vimos quatro desses: *pthread_create*, *pthread_exit*, *pthread_join* e *pthread_yield*, mas geralmente há mais.

Quando os threads são gerenciados no espaço do usuário, cada processo precisa da sua própria **tabela de threads** privada para controlá-los naquele processo. Ela é análoga à tabela de processo do núcleo, exceto por controlar apenas as propriedades de cada thread, como o contador de programa, o ponteiro de pilha, registradores, 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 informação necessária para reiniciá-lo é armazenada na tabela de threads, exatamente da mesma maneira que o núcleo armazena informações sobre processos na tabela de processos.

Quando um thread faz algo que possa bloqueá-lo localmente, por exemplo, esperar que outro thread em seu processo termine um trabalho, ele chama uma rotina do sistema de tempo de execução. Essa rotina verifica se o thread precisa ser colocado no estado bloqueado. Caso isso seja necessário, ela armazena os registradores do thread (isto é, os seus próprios) na tabela de threads, procura na tabela por um thread pronto para ser executado, e recarrega os registradores da máquina com os valores salvos do novo thread. Tão logo o ponteiro de pilha e o contador de programa tenham sido trocados, o novo thread ressurge para a vida novamente de maneira automática. Se a máquina porventura tiver uma instrução para

FIGURA 2.16 (a) Um pacote de threads no espaço do usuário. (b) Um pacote de threads gerenciado pelo núcleo.



armazenar todos os registradores e outra para carregá-los, todo o chaveamento do thread poderá ser feito com apenas um punhado de instruções. Realizar um chaveamento de thread como esse é pelo menos uma ordem de magnitude — talvez mais — mais rápida do que desviar o controle para o núcleo, além de ser um forte argumento a favor de pacotes de threads de usuário.

No entanto, há uma diferença fundamental com os processos. Quando um thread decide parar de executar — por exemplo, quando ele chama *thread_yield* — o código de *thread_yield* pode salvar as informações do thread na própria tabela de thread. Além disso, ele pode então chamar o escalonador de threads para pegar outro thread para executar. A rotina que salva o estado do thread e o escalonador são apenas rotinas locais, de maneira que invocá-las é algo muito mais eficiente do que fazer uma chamada de núcleo. Entre outras coisas, nenhuma armadilha (*trap*) é necessária, nenhum chaveamento de contexto é necessário, a cache de memória não precisa ser esvaziada e assim por diante. Isso torna o escalonamento de thread muito rápido.

Threads de usuário também têm outras vantagens. Eles permitem que cada processo tenha seu próprio algoritmo de escalonamento customizado. Para algumas aplicações, por exemplo, aquelas com um thread coletor de lixo, é uma vantagem não ter de se preocupar com um thread ser parado em um momento inconveniente. Eles também escalam melhor, já que threads de núcleo sempre exigem algum espaço de tabela e de pilha no núcleo, o que pode ser um problema se houver um número muito grande de threads.

Apesar do seu melhor desempenho, pacotes de threads de usuário têm alguns problemas importantes. Primeiro, o problema de como chamadas de sistema bloqueantes são implementadas. Suponha que um thread leia de um teclado antes que quaisquer teclas tenham sido acionadas. Deixar que o thread realmente faça a chamada de sistema é algo inaceitável, visto que isso parará todos os threads. Uma das principais razões para ter threads era permitir que cada um utilizasse chamadas com bloqueio, enquanto evitaria que um thread bloqueado afetasse os outros. Com chamadas de sistema bloqueantes, é difícil ver como essa meta pode ser alcançada prontamente.

As chamadas de sistema poderiam ser todas modificadas para que não bloqueassem (por exemplo, um read no teclado retornaria apenas 0 byte se nenhum caractere já estivesse no buffer), mas exigir mudanças para o sistema operacional não é algo atraente. Além disso, um argumento para threads de usuário era precisamente que eles podiam ser executados com sistemas operacionais *existentes*. Ainda, mudar a semântica de read exigirá mudanças para muitos programas de usuários.

Mais uma alternativa encontra-se disponível no caso de ser possível dizer antecipadamente se uma chamada será bloqueada. Na maioria das versões de UNIX, existe uma chamada de sistema, select, que permite a quem chama saber se um read futuro será bloqueado. Quando essa chamada está presente, a rotina de biblioteca *read* pode ser substituída por uma nova que primeiro faz uma chamada select e, então, faz a chamada read somente se ela for segura (isto é, não for bloqueada). Se a chamada read for bloqueada, a chamada não é feita. Em vez disso, outro thread é executado. Da próxima vez que o sistema de tempo de execução assumir o controle, ele pode conferir de novo se a read está então segura. Essa abordagem exige reescrever partes da biblioteca de chamadas de sistema, e é algo ineficiente e deselegante, mas há pouca escolha. O código colocado em torno da chamada de sistema para fazer a verificação é chamado de **jacket** ou **wrapper**.

Um problema de certa maneira análogo às chamadas de sistema bloqueantes é o problema das faltas de páginas. Nós as estudaremos no Capítulo 3. Por ora, basta dizer que os computadores podem ser configurados de tal maneira que nem todo o programa está na memória principal ao mesmo tempo. Se o programa chama ou salta para uma instrução que não esteja na memória, ocorre uma falta de página e o sistema operacional buscará a instrução perdida (e suas vizinhas) do disco. Isso é chamado de uma falta de página. O processo é bloqueado enquanto as instruções necessárias estão sendo localizadas e lidas. Se um thread causa uma falta de página, o núcleo, desconhecendo até a existência dos threads, naturalmente bloqueia o processo inteiro até que o disco de E/S esteja completo, embora outros threads possam ser executados.

Outro problema com pacotes de threads de usuário é que se um thread começa a ser executado, nenhum outro naquele processo será executado a não ser que o primeiro thread voluntariamente abra mão da CPU. Dentro de um único processo, não há interrupções de relógio, impossibilitando escalonar processos pelo esquema de escalonamento circular (dando a vez ao outro). A menos que um thread entre voluntariamente no sistema de tempo de execução, o escalonador jamais terá uma chance.

Uma solução possível para o problema de threads sendo executados infinitamente é obrigar o sistema de tempo de execução a solicitar um sinal de relógio

(interrupção) a cada segundo para dar a ele o controle, mas isso, também, é algo grosseiro e confuso para o programa. Interrupções periódicas de relógio em uma frequência mais alta nem sempre são possíveis, e mesmo que fossem, a sobrecarga total poderia ser substancial. Além disso, um thread talvez precise também de uma interrupção de relógio, interferindo com o uso do relógio pelo sistema de tempo de execução.

Outro — e o mais devastador — argumento contra threads de usuário é que os programadores geralmente desejam threads precisamente em aplicações nas quais eles são bloqueados com frequência, por exemplo, em um servidor web com múltiplos threads. Esses threads estão constantemente fazendo chamadas de sistema. Uma vez que tenha ocorrido uma armadilha para o núcleo a fim de executar uma chamada de sistema, não daria muito mais trabalho para o núcleo trocar o thread sendo executado, se ele estiver bloqueado, o núcleo fazendo isso elimina a necessidade de sempre realizar chamadas de sistema select que verificam se as chamadas de sistema read são seguras. Para aplicações que são em sua essência inteiramente limitadas pela CPU e raramente são bloqueadas, qual o sentido de usar threads? Ninguém proporia seriamente computar os primeiros *n* números primos ou jogar xadrez usando threads, porque não há nada a ser ganho com isso.

2.2.5 Implementando threads no núcleo

Agora vamos considerar que o núcleo saiba sobre os threads e os gerencie. Não é necessário um sistema de tempo de execução em cada um, como mostrado na Figura 2.16(b). Também não há uma tabela de thread em cada processo. Em vez disso, o núcleo tem uma tabela que controla todos os threads no sistema. Quando um thread quer criar um novo ou destruir um existente, ele faz uma chamada de núcleo, que então faz a criação ou a destruição atualizando a tabela de threads do núcleo.

A tabela de threads do núcleo contém os registradores, estado e outras informações de cada thread. A informação é a mesma com os threads de usuário, mas agora mantidas no núcleo em vez de no espaço do usuário (dentro do sistema de tempo de execução). Essa informação é um subconjunto das informações que os núcleos tradicionais mantêm a respeito dos seus processos de thread único, isto é, o estado de processo. Além disso, o núcleo também mantém a tabela de processos tradicional para controlar os processos.

Todas as chamadas que poderiam bloquear um thread são implementadas como chamadas de sistema, a um custo consideravelmente maior do que uma chamada para uma rotina de sistema de tempo de execução. Quando um thread é bloqueado, o núcleo tem a opção de executar outro thread do mesmo processo (se um estiver pronto) ou algum de um processo diferente. Com threads de usuário, o sistema de tempo de execução segue executando threads a partir do seu próprio processo até o núcleo assumir a CPU dele (ou não houver mais threads prontos para serem executados).

Em decorrência do custo relativamente maior de se criar e destruir threads no núcleo, alguns sistemas assumem uma abordagem ambientalmente correta e reciclam seus threads. Quando um thread é destruído, ele é marcado como não executável, mas suas estruturas de dados de núcleo não são afetadas de outra maneira. Depois, quando um novo thread precisa ser criado, um antigo é reativado, evitando parte da sobrecarga. A reciclagem de threads também é possível para threads de usuário, mas tendo em vista que o overhead de gerenciamento de threads é muito menor, há menos incentivo para fazer isso.

Threads de núcleo não exigem quaisquer chamadas de sistema novas e não bloqueantes. Além disso, se um thread em um processo provoca uma falta de página, o núcleo pode facilmente conferir para ver se o processo tem quaisquer outros threads executáveis e, se assim for, executar um deles enquanto espera que a página exigida seja trazida do disco. Sua principal desvantagem é que o custo de uma chamada de sistema é substancial, então se as operações de thread (criação, término etc.) forem frequentes, ocorrerá uma sobrecarga muito maior.

Embora threads de núcleo solucionem alguns problemas, eles não resolvem todos eles. Por exemplo, o que acontece quando um processo com múltiplos threads é bifurcado? O novo processo tem tantos threads quanto o antigo, ou possui apenas um? Em muitos casos, a melhor escolha depende do que o processo está planejando fazer em seguida. Se ele for chamar exec para começar um novo programa, provavelmente um thread é a escolha correta, mas se ele continuar a executar, reproduzir todos os threads talvez seja o melhor.

Outra questão são os sinais. Lembre-se de que os sinais são enviados para os processos, não para os threads, pelo menos no modelo clássico. Quando um sinal chega, qual thread deve cuidar dele? Threads poderiam talvez registrar seu interesse em determinados sinais, de maneira que, ao chegar um sinal, ele seria dado para o thread que disse querê-lo. Mas o que acontece se dois ou mais threads registraram interesse para o mesmo sinal? Esses são apenas dois dos problemas que os threads introduzem, mas há outros.

2.2.6 Implementações híbridas

Várias maneiras foram investigadas para tentar combinar as vantagens de threads de usuário com threads de núcleo. Uma maneira é usar threads de núcleo e então multiplexar os de usuário em alguns ou todos eles, como mostrado na Figura 2.17. Quando essa abordagem é usada, o programador pode determinar quantos threads de núcleo usar e quantos threads de usuário multiplexar para cada um. Esse modelo proporciona o máximo em flexibilidade.

Com essa abordagem, o núcleo está consciente *apenas* dos threads de núcleo e os escalona. Alguns desses threads podem ter, em cima deles, múltiplos threads de usuário multiplexados, os quais são criados, destruídos e escalonados exatamente como threads de usuário em um processo executado em um sistema operacional sem capacidade de múltiplos threads. Nesse modelo, cada thread de núcleo tem algum conjunto de threads de usuário que se revezam para usá-lo.

2.2.7 Ativações pelo escalonador

Embora threads de núcleo sejam melhores do que threads de usuário em certos aspectos-chave, eles são também indiscutivelmente mais lentos. Como consequência, pesquisadores procuraram maneiras de melhorar a situação sem abrir mão de suas boas propriedades. A seguir descreveremos uma abordagem desenvolvida por Anderson et al. (1992), chamada **ativações pelo escalonador**. Um trabalho relacionado é discutido por Edler et al. (1988) e Scott et al. (1990).

A meta do trabalho da ativação pelo escalonador é imitar a funcionalidade dos threads de núcleo, mas com melhor desempenho e maior flexibilidade normalmente associados aos pacotes de threads implementados no espaço do usuário. Em particular, threads de usuário não deveriam ter de fazer chamadas de sistema especiais sem bloqueio ou conferir antecipadamente se é seguro fazer determinadas chamadas de sistema. Mesmo assim, quando um thread é bloqueado em uma chamada de sistema ou uma página falha, deve ser possível executar outros threads dentro do mesmo processo, se algum estiver pronto.

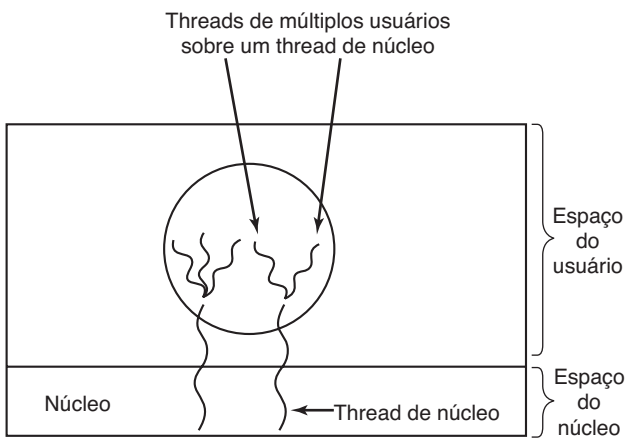
A eficiência é alcançada evitando-se transições desnecessárias entre espaço do usuário e do núcleo. Se um thread é bloqueado esperando por outro para fazer algo, por exemplo, não há razão para envolver o núcleo, poupando assim a sobrecarga da transição núcleo-usuário. O sistema de tempo de execução pode bloquear o thread de sincronização e escalonar sozinho um novo.

Quando ativações pelo escalonador são usadas, o núcleo designa um determinado número de processadores virtuais para cada processo e deixa o sistema de tempo de execução (no espaço do usuário) alocar threads para eles. Esse mecanismo também pode ser usado em um multiprocessador onde os processadores virtuais podem ser CPUs reais. O número de processadores virtuais alocados para um processo é de início um, mas o processo pode pedir mais e também pode devolver processadores que não precisa mais. O núcleo também pode retomar processadores virtuais já alocados a fim de colocá-los em processos mais necessitados.

A ideia básica para o funcionamento desse esquema é a seguinte: quando sabe que um thread foi bloqueado (por exemplo, tendo executado uma chamada de sistema bloqueante ou causado uma falta de página), o núcleo notifica o sistema de tempo de execução do processo, passando como parâmetros na pilha o número do thread em questão e uma descrição do evento que ocorreu. A notificação acontece quando o núcleo ativa o sistema de tempo de execução em um endereço inicial conhecido, de maneira mais ou menos análoga a um sinal em UNIX. Esse mecanismo é chamado **upcall**.

Uma vez ativado, o sistema de tempo de execução pode reescalonar os seus threads, tipicamente marcando o thread atual como bloqueado e tomando outro da lista pronta, configurando seus registradores e reinicializando-o. Depois, quando o núcleo fica sabendo que o thread original pode executar de novo (por exemplo, o pipe do qual ele estava tentando ler agora contém dados, ou a página sobre a qual ocorreu uma falta foi trazida do disco), ele faz outro upcall para informar o sistema de tempo de execução. O sistema de tempo de execução pode então

FIGURA 2.17 Multiplexando threads de usuário em threads de núcleo.



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 troca para o modo núcleo. Se a interrupção for causada por um evento que não é de interesse do processo interrompido, como a conclusão da E/S de outro processo, quando o tratador da interrupção terminar, ele coloca o thread de interrupção de volta no estado de antes da interrupção. Se, no entanto, o processo está interessado na interrupção, como a chegada de uma página necessitada por um dos threads do processo, o thread interrompido não é reinicializado. Em vez disso, ele é suspenso, e o sistema de tempo de execução é inicializado naquela CPU virtual, com o estado do thread interrompido na pilha. Então cabe ao sistema de tempo de execução decidir qual thread escalonar naquela CPU: o thread interrompido, o recentemente pronto ou uma terceira escolha.

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

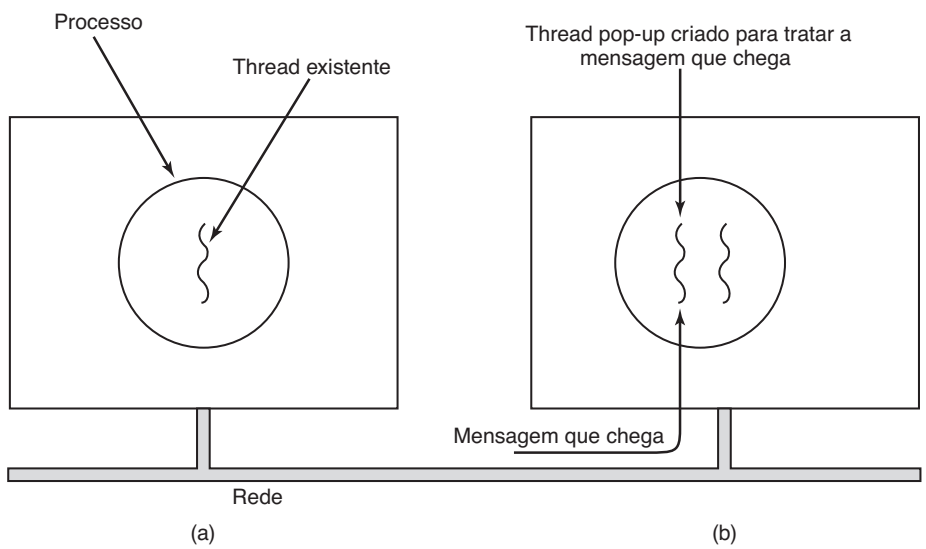
2.2.8 Threads pop-up

Threads costumam ser úteis em sistemas distribuídos. Um exemplo importante é como mensagens que chegam — por exemplo, requisições de serviços — são tratadas. A abordagem tradicional é ter um processo ou thread que esteja bloqueado em uma chamada de sistema receive esperando pela mensagem que chega. Quando uma mensagem chega, ela é aceita, aberta, seu conteúdo examinado e processada.

No entanto, uma abordagem completamente diferente também é possível, na qual a chegada de uma mensagem faz o sistema criar um novo thread para lidar com a mensagem. Esse thread é chamado de **thread pop-up** e está ilustrado na Figura 2.18. Uma vantagem fundamental de threads pop-up é que como são novos, eles não têm história alguma — registradores, pilha, o que quer que seja — que devem ser restaurados. Cada um começa fresco e cada um é idêntico a todos os outros. Isso possibilita a criação de tais threads rapidamente. O thread novo recebe a mensagem que chega para processar. O resultado da utilização de threads pop-up é que a latência entre a chegada da mensagem e o começo do processamento pode ser encurtada.

Algum planejamento prévio é necessário quando threads pop-up são usados. Por exemplo, em qual processo o thread é executado? Se o sistema dá suporte a threads sendo executados no contexto núcleo, o thread pode ser executado ali (razão pela qual não mostramos o núcleo na Figura 2.18). Executar um thread pop-up no espaço núcleo normalmente é mais fácil e mais rápido do que colocá-lo no espaço do usuário. Também, um thread pop-up no espaço núcleo consegue facilmente acessar todas as tabelas do núcleo e os dispositivos de E/S, que podem ser necessários para o processamento de interrupções. Por outro lado, um thread de núcleo

FIGURA 2.18 Criação de um thread novo quando a mensagem chega. (a) Antes da chegada da mensagem. (b) Depois da chegada da mensagem.



com erros pode causar mais danos que um de usuário com erros. Por exemplo, se ele for executado por tempo demais e não liberar a CPU, dados que chegam podem ser perdidos para sempre.

2.2.9 Convertendo código de um thread em código multithread

Muitos programas existentes foram escritos para processos monothread. Convertê-los para multithreading é muito mais complicado do que pode parecer em um primeiro momento. A seguir examinaremos apenas algumas das armadilhas.

Para começo de conversa, o código de um thread em geral consiste em múltiplas rotinas, exatamente como um processo. Essas rotinas podem ter variáveis locais, variáveis globais e parâmetros. Variáveis locais e de parâmetros não causam problema algum, mas variáveis que são globais para um thread, mas não globais para o programa inteiro, são um problema. Essas são variáveis que são globais no sentido de que muitos procedimentos dentro do thread as usam (como poderiam usar qualquer variável global), mas outros threads devem logicamente deixá-las sozinhas.

Como exemplo, considere a variável *errno* mantida pelo UNIX. Quando um processo (ou thread) faz uma chamada de sistema que falha, o código de erro é colocado em *errno*. Na Figura 2.19, o thread 1 executa a chamada de sistema access para descobrir se ela tem permissão para acessar determinado arquivo. O sistema operacional retorna a resposta na variável global *errno*. Após o controle ter retornado para o thread 1, mas antes de ele ter uma chance de ler *errno*, o escalonador decide que o thread 1 teve tempo de CPU suficiente para o momento e decide trocar para o thread 2. O thread 2 executa uma chamada open que falha, o que faz que *errno* seja sobrescrito e o código de acesso do thread 1 seja perdido para sempre. Quando o thread 1 inicia posteriormente, ele terá o valor errado e comportar-se-á incorretamente.

Várias soluções para esse problema são possíveis. Uma é proibir completamente as variáveis globais. Por mais válido que esse ideal possa ser, ele entra em conflito com grande parte dos softwares existentes. Outra é designar a cada thread as suas próprias variáveis globais privadas, como mostrado na Figura 2.20. Dessa maneira, cada thread tem a sua própria cópia privada de *errno* e outras variáveis globais, de modo que os conflitos são evitados. Na realidade, essa decisão cria um novo nível de escopo, variáveis visíveis a todas as rotinas de um thread (mas não para outros threads), além dos níveis de escopo existentes de variáveis visíveis apenas para uma rotina e variáveis visíveis em toda parte no programa.

No entanto, acessar as variáveis globais privadas é um pouco complicado já que a maioria das linguagens de programação tem uma maneira de expressar variáveis locais e variáveis globais, mas não formas intermediárias. É possível alocar um pedaço da memória para as globais e passá-lo para cada rotina no thread como um parâmetro extra. Embora dificilmente você possa considerá-la uma solução elegante, ela funciona.

Alternativamente, novas rotinas de biblioteca podem ser introduzidas para criar, alterar e ler essas variáveis globais restritas ao thread. A primeira chamada pode parecer assim:

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

FIGURA 2.19 Conflitos entre threads sobre o uso de uma variável global.

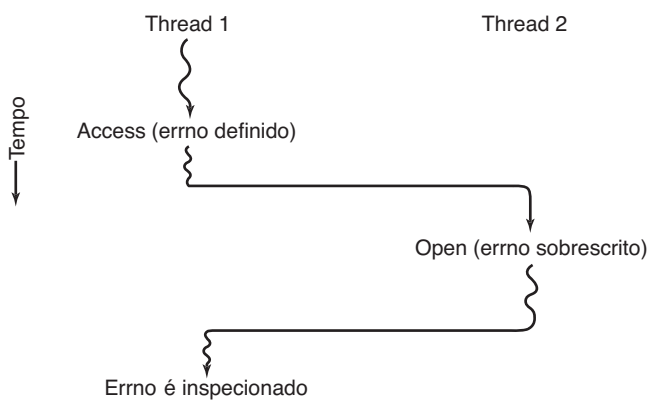
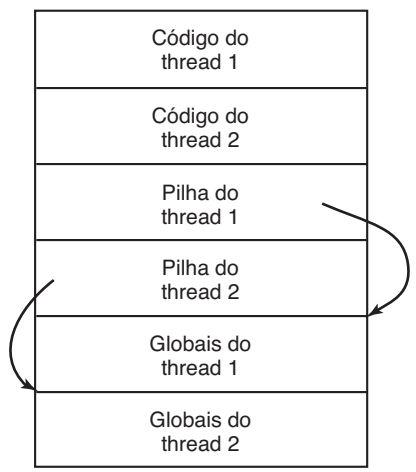


FIGURA 2.20 Threads podem ter variáveis globais individuais.



Ela aloca memória para um ponteiro chamado *bufptr* no heap ou em uma área de armazenamento especial reservada para o thread que emitiu a chamada. Não importa onde a memória esteja alocada, apenas o thread que emitiu a chamada tem acesso à variável global. Se outro thread criar uma variável global com o mesmo nome, ele obterá uma porção de memória que não entrará em conflito com a existente.

Duas chamadas são necessárias para acessar variáveis globais: uma para escrevê-las e a outra para lê-las. Para escrever, algo como

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

bastará. Ela armazena o valor de um ponteiro na porção de memória previamente criada pela chamada para *create_global*. Para ler uma variável global, a chamada pode ser algo como

```
bufptr = read_global("bufptr").
```

Ela retorna o endereço armazenado na variável global, de maneira que os seus dados possam ser acessados.

O próximo problema em transformar um programa de um único thread em um programa com múltiplos threads é que muitas rotinas de biblioteca não são reentrantes. Isto é, elas não foram projetadas para ter uma segunda chamada feita para uma rotina enquanto uma anterior ainda não tiver sido concluída. Por exemplo, o envio de uma mensagem através de uma rede pode ser programado com a montagem da mensagem em um buffer fixo dentro da biblioteca, seguido de um chaveamento para enviá-la. O que acontece se um thread montou a sua mensagem no buffer, então, uma interrupção de relógio força um chaveamento para um segundo thread que imediatamente sobrescreve o buffer com sua própria mensagem?

Similarmente, rotinas de alocação de memória como *malloc* no UNIX, mantêm tabelas cruciais sobre o uso de memória, por exemplo, uma lista encadeada de pedaços de memória disponíveis. Enquanto *malloc* está ocupada atualizando essas listas, elas podem temporariamente estar em um estado inconsistente, com ponteiros que apontam para lugar nenhum. Se um chaveamento de threads ocorrer enquanto as tabelas forem inconsistentes e uma nova chamada entrar de um thread diferente, um ponteiro inválido poderá ser usado, levando à queda do programa. Consertar todos esses problemas efetivamente significa reescrever toda a biblioteca. Fazê-lo é uma atividade não trivial com uma possibilidade real de ocorrer a introdução de erros sutis.

Uma solução diferente é fornecer a cada rotina uma proteção que altera um bit para indicar que a biblioteca está sendo usada. Qualquer tentativa de outro thread usar uma rotina de biblioteca enquanto a chamada anterior ainda não tiver sido concluída é bloqueada. Embora essa abordagem possa ser colocada em funcionamento, ela elimina muito o paralelismo em potencial.

Em seguida, considere os sinais. Alguns são logicamente específicos a threads, enquanto outros não. Por exemplo, se um thread chama alarm, faz sentido para o sinal resultante ir até o thread que fez a chamada. No entanto, quando threads são implementados inteiramente no espaço de usuário, o núcleo não tem nem ideia a respeito dos threads e mal pode dirigir o sinal para o thread certo. Uma complicação adicional ocorre se um processo puder ter apenas um alarme pendente por vez e vários threads chamam alarm independentemente.

Outros sinais, como uma interrupção de teclado, não são específicos aos threads. Quem deveria pegá-los? Um thread designado? Todos os threads? Um thread pop-up recentemente criado? Além disso, o que acontece se um thread mudar os tratadores de sinal sem contar para os outros threads? E o que acontece se um thread quiser pegar um sinal em particular (digamos, o usuário digitando CTRL-C), e outro thread quiser esse sinal para concluir o processo? Essa situação pode surgir se um ou mais threads executarem rotinas de biblioteca-padrão e outros forem escritos por usuários. Claramente, esses desejos são incompatíveis. Em geral, sinais são suficientemente difíceis para gerenciar em um ambiente de um thread único. Ir para um ambiente de múltiplos threads não torna a situação mais fácil de lidar.

Um último problema introduzido pelos threads é o gerenciamento de pilha. Em muitos sistemas, quando a pilha de um processo transborda, o núcleo apenas fornece àquele processo mais pilha automaticamente. Quando um processo tem múltiplos threads, ele também tem múltiplas pilhas. Se o núcleo não tem ciência de todas essas pilhas, ele não pode fazê-las crescer automaticamente por causa de uma falha de pilha. Na realidade, ele pode nem se dar conta de que uma falha de memória está relacionada com o crescimento da pilha de algum thread.

Esses problemas certamente não são insuperáveis, mas mostram que apenas introduzir threads em um sistema existente sem uma alteração bastante substancial do sistema não vai funcionar mesmo. No mínimo, as semânticas das chamadas de sistema talvez precisem ser redefinidas e as bibliotecas reescritas. E todas essas coisas devem ser feitas de maneira a permanecerem compatíveis com programas já existentes para o caso limitante de um processo com apenas um thread. Para informações adicionais sobre threads, ver Hauser et al. (1993), Marsh et al. (1991) e Rodrigues et al. (2010).

2.3 Comunicação entre processos

Processos quase sempre precisam comunicar-se com outros processos. Por exemplo, em um pipeline do interpretador de comandos, a saída do primeiro processo tem de ser passada para o segundo, e assim por diante até o fim da linha. Então, há uma necessidade por comunicação entre os processos, de preferência de uma maneira bem estruturada sem usar interrupções. Nas seções a seguir, examinaremos algumas das questões relacionadas com essa **comunicação entre processos** (*interprocess communication* — **IPC**).

De maneira bastante resumida, há três questões aqui. A primeira acaba de ser mencionada: como um processo pode passar informações para outro. A segunda tem a ver com certificar-se de que dois ou mais processos não se atrapalhem, por exemplo, dois processos em um sistema de reserva de uma companhia aérea cada um tentando ficar com o último assento em um avião para um cliente diferente. A terceira diz respeito ao sequenciamento adequado quando dependências estão presentes: se o processo *A* produz dados e o processo *B* os imprime, *B* tem de esperar até que *A* tenha produzido alguns dados antes de começar a imprimir. Examinaremos todas as três questões começando na próxima seção.

Também é importante mencionar que duas dessas questões aplicam-se igualmente bem aos threads. A primeira — passar informações — é fácil para os threads, já que eles compartilham de um espaço de endereçamento comum (threads em espaços de endereçamento diferentes que precisam comunicar-se são questões relativas à comunicação entre processos). No entanto, as outras duas — manter um afastado do outro e o sequenciamento correto — aplicam-se igualmente bem aos threads. A seguir discutiremos o problema no contexto de processos, mas, por favor, mantenha em mente que os mesmos problemas e soluções também se aplicam aos threads.

2.3.1 Condições de corrida

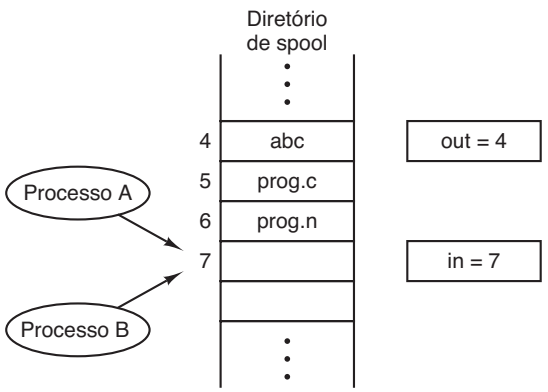
Em alguns sistemas operacionais, processos que estão trabalhando juntos podem compartilhar de alguma memória comum que cada um pode ler e escrever. A memória compartilhada pode encontrar-se na memória principal (possivelmente em uma estrutura de dados de núcleo) ou ser um arquivo compartilhado; o local da memória compartilhada não muda a natureza da comunicação ou os problemas que surgem. Para ver como a comunicação entre processos funciona na prática, vamos considerar um exemplo simples, mas comum: um spool de impressão. Quando um processo quer imprimir um arquivo, ele entra com o nome do arquivo em um **diretório de spool** especial. Outro processo, o **daemon de impressão**, confere periodicamente para ver se há quaisquer arquivos a serem impressos, e se houver, ele os imprime e então remove seus nomes do diretório.

Imagine que nosso diretório de spool tem um número muito grande de vagas, numeradas 0, 1, 2, ..., cada uma capaz de conter um nome de arquivo. Também imagine que há duas variáveis compartilhadas, *out*, que apontam para o próximo arquivo a ser impresso, e *in*, que aponta para a próxima vaga livre no diretório. Essas duas variáveis podem muito bem ser mantidas em um arquivo de duas palavras disponível para todos os processos. Em determinado instante, as vagas 0 a 3 estarão vazias (os arquivos já foram impressos) e as vagas 4 a 6 estarão cheias (com os nomes dos arquivos na fila para impressão). De maneira mais ou menos simultânea, os processos *A* e *B* decidem que querem colocar um arquivo na fila para impressão. Essa situação é mostrada na Figura 2.21.

Nas jurisdições onde a Lei de Murphy[2] for aplicável, pode ocorrer o seguinte: o processo *A* lê *in* e armazena o valor, 7, em uma variável local chamada *next_free_slot*. Logo em seguida uma interrupção de relógio ocorre e a CPU decide que o processo *A* executou por tempo suficiente, então, ele troca para o processo *B*. O processo *B* também lê *in* e recebe um 7. Ele, também, o armazena em *sua* variável local *next_free_slot*. Nesse instante, ambos os processos acreditam que a próxima vaga disponível é 7.

O processo *B* agora continua a executar. Ele armazena o nome do seu arquivo na vaga 7 e atualiza *in* para ser um 8. Então ele segue em frente para fazer outras coisas.

FIGURA 2.21 Dois processos querem acessar a memória compartilhada ao mesmo tempo.



[2] Se algo pode dar errado, certamente vai dar. (N. do A.)

Por fim, o processo *A* executa novamente, começando do ponto onde ele parou. Ele olha para *next_free_slot*, encontra um 7 ali e escreve seu nome de arquivo na vaga 7, apagando o nome que o processo *B* recém-colocou ali. Então calcula *next_free_slot* + 1, que é 8, e configura *in* para 8. O diretório de spool está agora internamente consistente, então o daemon de impressão não observará nada errado, mas o processo *B* jamais receberá qualquer saída. O usuário *B* ficará em torno da impressora por anos, aguardando esperançoso por uma saída que nunca virá. Situações como essa, em que dois ou mais processos estão lendo ou escrevendo alguns dados compartilhados e o resultado final depende de quem executa precisamente e quando, são chamadas de **condições de corrida**. A depuração de programas contendo condições de corrida não é nem um pouco divertida. Os resultados da maioria dos testes não encontram nada, mas de vez em quando algo esquisito e inexplicável acontece. Infelizmente, com o incremento do paralelismo pelo maior número de núcleos, as condições de corrida estão se tornando mais comuns.

2.3.2 Regiões críticas

Como evitar as condições de corrida? A chave para evitar problemas aqui e em muitas outras situações envolvendo memória compartilhada, arquivos compartilhados e tudo o mais compartilhado é encontrar alguma maneira de proibir mais de um processo de ler e escrever os dados compartilhados ao mesmo tempo. Colocando a questão em outras palavras, o que precisamos é de **exclusão mútua**, isto é, alguma maneira de se certificar de que se um processo está usando um arquivo ou variável compartilhados, os outros serão impedidos de realizar a mesma coisa. A dificuldade mencionada ocorreu porque o processo *B* começou usando uma das variáveis compartilhadas antes de o processo *A* ter terminado com ela. A escolha das operações primitivas apropriadas para alcançar a exclusão mútua é uma questão de projeto fundamental em qualquer sistema operacional, e um assunto que examinaremos detalhadamente nas seções a seguir.

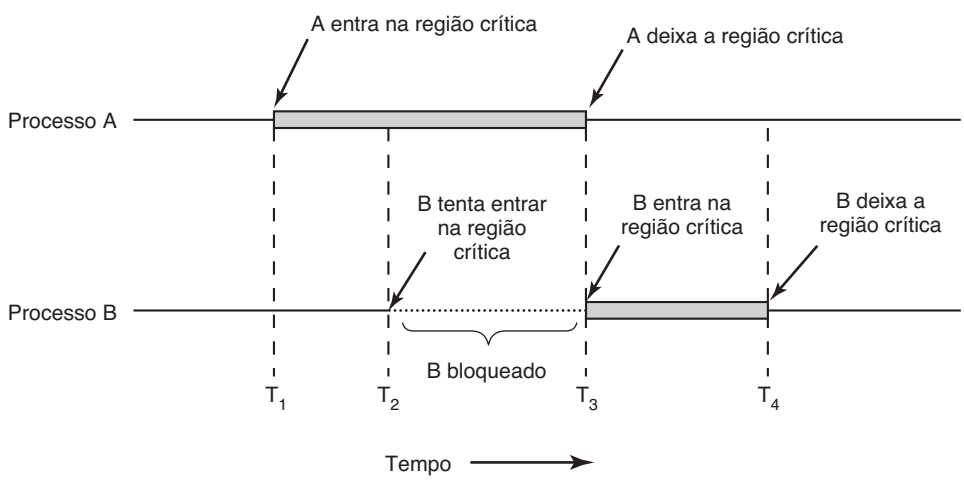
O problema de evitar condições de corrida também pode ser formulado de uma maneira abstrata. Durante parte do tempo, um processo está ocupado realizando computações internas e outras coisas que não levam a condições de corrida. No entanto, às vezes um processo tem de acessar uma memória compartilhada ou arquivos, ou realizar outras tarefas críticas que podem levar a corridas. Essa parte do programa onde a memória compartilhada é acessada é chamada de **região crítica** ou **seção crítica**. Se conseguíssemos arranjar as coisas de maneira que jamais dois processos estivessem em suas regiões críticas ao mesmo tempo, poderíamos evitar as corridas.

Embora essa exigência evite as condições de corrida, ela não é suficiente para garantir que processos em paralelo cooperem de modo correto e eficiente usando dados compartilhados. Precisamos que quatro condições se mantenham para chegar a uma boa solução:

1. Dois processos jamais podem estar simultaneamente dentro de suas regiões críticas.
2. Nenhuma suposição pode ser feita a respeito de velocidades ou do número de CPUs.
3. Nenhum processo executando fora de sua região crítica pode bloquear qualquer processo.
4. Nenhum processo deve ser obrigado a esperar eternamente para entrar em sua região crítica.

Em um sentido abstrato, o comportamento que queremos é mostrado na Figura 2.22. Aqui o processo *A*

FIGURA 2.22 Exclusão mútua usando regiões críticas.



entra na sua região crítica no tempo T_1. Um pouco mais tarde, no tempo T_2, o processo B tenta entrar em sua região crítica, mas não consegue porque outro processo já está em sua região crítica e só permitimos um de cada vez. Em consequência, B é temporariamente suspenso até o tempo T_3, quando A deixa sua região crítica, permitindo que B entre de imediato. Por fim, B sai (em T_4) e estamos de volta à situação original sem nenhum processo em suas regiões críticas.

2.3.3 Exclusão mútua com espera ocupada

Nesta seção examinaremos várias propostas para realizar a exclusão mútua, de maneira que enquanto um processo está ocupado atualizando a memória compartilhada em sua região crítica, nenhum outro entrará na sua região crítica para causar problemas.

Desabilitando interrupções

Em um sistema de processador único, a solução mais simples é fazer que cada processo desabilite todas as interrupções logo após entrar em sua região crítica e as reabilitar um momento antes de partir. Com as interrupções desabilitadas, nenhuma interrupção de relógio poderá ocorrer. Afinal de contas, a CPU só é chaveada de processo em processo em consequência de uma interrupção de relógio ou outra, e com as interrupções desligadas, a CPU não será chaveada para outro processo. Então, assim que um processo tiver desabilitado as interrupções, ele poderá examinar e atualizar a memória compartilhada sem medo de que qualquer outro processo interfira.

Em geral, essa abordagem é pouco atraente, pois não é prudente dar aos processos de usuário o poder de desligar interrupções. E se um deles desligasse uma interrupção e nunca mais a ligasse de volta? Isso poderia ser o fim do sistema. Além disso, se o sistema é um multiprocessador (com duas ou mais CPUs), desabilitar interrupções afeta somente a CPU que executou a instrução disable. As outras continuarão executando e podem acessar a memória compartilhada.

Por outro lado, não raro, é conveniente para o próprio núcleo desabilitar interrupções por algumas instruções enquanto está atualizando variáveis ou especialmente listas. Se uma interrupção acontece enquanto a lista de processos prontos está, por exemplo, no estado inconsistente, condições de corrida podem ocorrer. A conclusão é: desabilitar interrupções é muitas vezes uma técnica útil dentro do próprio sistema operacional, mas não é apropriada como um mecanismo de exclusão mútua geral para processos de usuário.

A possibilidade de alcançar a exclusão mútua desabilitando interrupções — mesmo dentro do núcleo — está se tornando menor a cada dia por causa do número cada vez maior de chips multinúcleo mesmo em PCs populares. Dois núcleos já são comuns, quatro estão presentes em muitas máquinas, e oito, 16, ou 32 não ficam muito atrás. Em um sistema multinúcleo (isto é, sistema de multiprocessador) desabilitar as interrupções de uma CPU não evita que outras CPUs interfiram com as operações que a primeira está realizando. Em consequência, esquemas mais sofisticados são necessários.

Variáveis do tipo trava

Como uma segunda tentativa, vamos procurar por uma solução de software. Considere ter uma única variável (de trava) compartilhada, inicialmente 0. Quando um processo quer entrar em sua região crítica, ele primeiro testa a trava. Se a trava é 0, o processo a configura para 1 e entra na região crítica. Se a trava já é 1, o processo apenas espera até que ela se torne 0. Desse modo, um 0 significa que nenhum processo está na região crítica, e um 1 significa que algum processo está em sua região crítica.

Infelizmente, essa ideia contém exatamente a mesma falha fatal que vimos no diretório de spool. Suponha que um processo lê a trava e vê que ela é 0. Antes que ele possa configurar a trava para 1, outro processo está escalonado, executa e configura a trava para 1. Quando o primeiro processo executa de novo, ele também configurará a trava para 1, e dois processos estarão nas suas regiões críticas ao mesmo tempo.

Agora talvez você pense que poderíamos dar um jeito nesse problema primeiro lendo o valor da trava, então, conferindo-a outra vez um instante antes de armazená-la, mas isso na realidade não ajuda. A corrida agora ocorre se o segundo processo modificar a trava logo após o primeiro ter terminado a sua segunda verificação.

Alternância explícita

Uma terceira abordagem para o problema da exclusão mútua é mostrada na Figura 2.23. Esse fragmento de programa, como quase todos os outros neste livro, é escrito em C. C foi escolhido aqui porque sistemas operacionais reais são virtualmente sempre escritos em C (ou às vezes C++), mas dificilmente em linguagens como Java, Python, ou Haskell. C é poderoso,

eficiente e previsível, características críticas para se escrever sistemas operacionais. Java, por exemplo, não é previsível, porque pode ficar sem memória em um momento crítico e precisar invocar o coletor de lixo para recuperar memória em um momento realmente inoportuno. Isso não pode acontecer em C, pois não há coleta de lixo em C. Uma comparação quantitativa de C, C++, Java e quatro outras linguagens é dada por Prechelt (2000).

Na Figura 2.23, a variável do tipo inteiro *turn*, inicialmente 0, serve para controlar de quem é a vez de entrar na região crítica e examinar ou atualizar a memória compartilhada. Inicialmente, o processo 0 inspeciona *turn*, descobre que ele é 0 e entra na sua região crítica. O processo 1 também encontra lá o valor 0 e, portanto, espera em um laço fechado testando continuamente *turn* para ver quando ele vira 1. Testar continuamente uma variável até que algum valor apareça é chamado de **espera ocupada**. Em geral ela deve ser evitada, já que desperdiça tempo da CPU. Apenas quando há uma expectativa razoável de que a espera será curta, a espera ocupada é usada. Uma trava que usa a espera ocupada é chamada de **trava giratória** (*spin lock*).

Quando o processo 0 deixa a região crítica, ele configura *turn* para 1, a fim de permitir que o processo 1 entre em sua região crítica. Suponha que o processo 1 termine sua região rapidamente, de modo que ambos os processos estejam em suas regiões não críticas, com *turn* configurado para 0. Agora o processo 0 executa todo seu laço rapidamente, deixando sua região crítica e configurando *turn* para 1. Nesse ponto, *turn* é 1 e ambos os processos estão sendo executados em suas regiões não críticas.

De repente, o processo 0 termina sua região não crítica e volta para o topo do seu laço. Infelizmente, não lhe é permitido entrar em sua região crítica agora, pois *turn* é 1 e o processo 1 está ocupado com sua região não crítica. Ele espera em seu laço while até que o processo 1 configura *turn* para 0. Ou seja, chavear a vez não é uma boa ideia quando um dos processos é muito mais lento que o outro.

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

Na realidade, essa solução exige que os dois processos alternem-se estritamente na entrada em suas regiões críticas para, por exemplo, enviar seus arquivos para o spool. Apesar de evitar todas as corridas, esse algoritmo não é realmente um sério candidato a uma solução, pois viola a condição 3.

FIGURA 2.23 Uma solução proposta para o problema da região crítica. (a) Processo 0. (b) Processo 1. Em ambos os casos, certifique-se de observar os pontos e vírgulas concluindo os comandos while.

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

(a)

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

(b)

Solução de Peterson

Ao combinar a ideia de alternar a vez com a ideia das variáveis de trava e de advertência, um matemático holandês, T. Dekker, foi o primeiro a desenvolver uma solução de software para o problema da exclusão mútua que não exige uma alternância explícita. Para uma discussão do algoritmo de Dekker, ver Dijkstra (1965).

Em 1981, G. L. Peterson descobriu uma maneira muito mais simples de realizar a exclusão mútua, tornando assim a solução de Dekker obsoleta. O algoritmo de Peterson é mostrado na Figura 2.24. Esse algoritmo consiste em duas rotinas escritas em ANSI C, o que significa que os protótipos de função devem ser fornecidos para todas as funções definidas e usadas. No entanto, a fim de poupar espaço, não mostraremos os protótipos aqui ou posteriormente.

Antes de usar as variáveis compartilhadas (isto é, antes de entrar na região crítica), cada processo chama *enter_region* com seu próprio número de processo, 0 ou 1, como parâmetro. Essa chamada fará que ele espere, se necessário, até que seja seguro entrar. Após haver terminado com as variáveis compartilhadas, o processo chama *leave_region* para indicar que ele terminou e para permitir que outros processos entrem, se assim desejarem.

FIGURA 2.24 A solução de Peterson para realizar a exclusão mútua.

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

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

void enter_region(int process);         /* processo e 0 ou 1 */
{
    int other;                          /* numero do outro processo */

    other = 1 – process;                /* o oposto do processo */
    interested[process] = TRUE;         /* mostra que voce esta interessado */
    turn = process;                     /* altera o valor de turn */
    while (turn == process && interested[other] == TRUE) /* comando nulo */ ;
}

void leave_region(int process)          /* processo: quem esta saindo */
{
    interested[process] = FALSE;        /* indica a saida da regiao critica */
}
```

Vamos ver como essa solução funciona. Inicialmente, nenhum processo está na sua região crítica. Agora o processo 0 chama *enter_region*. Ele indica o seu interesse alterando o valor de seu elemento de arranjo e alterando *turn* para 0. Como o processo 1 não está interessado, *enter_region* retorna imediatamente. Se o processo 1 fizer agora uma chamada para *enter_region*, ele esperará ali até que *interested*[0] mude para *FALSE*, um evento que acontece apenas quando o processo 0 chamar *leave_region* para deixar a região crítica.

Agora considere o caso em que ambos os processos chamam *enter_region* quase simultaneamente. Ambos armazenarão seu número de processo em *turn*. O último a armazenar é o que conta; o primeiro é sobrescrito e perdido. Suponha que o processo 1 armazene por último, então *turn* é 1. Quando ambos os processos chegam ao comando while, o processo 0 o executa zero vez e entra em sua região crítica. O processo 1 permance no laço e não entra em sua região crítica até que o processo 0 deixe a sua.

A instrução TSL

Agora vamos examinar uma proposta que exige um pouco de ajuda do hardware. Alguns computadores, especialmente aqueles projetados com múltiplos processadores em mente, têm uma instrução como

```
TSL RX,LOCK
```

(*Test and Set Lock* — teste e configure trava) que funciona da seguinte forma: ele lê o conteúdo da palavra *lock* da memória para o registrador RX e então armazena um valor diferente de zero no endereço de memória *lock*. As operações de leitura e armazenamento da palavra são seguramente indivisíveis — nenhum outro processador pode acessar a palavra na memória até que a instrução tenha terminado. A CPU executando a instrução TSL impede o acesso ao barramento de memória para proibir que outras CPUs acessem a memória até ela terminar.

É importante observar que impedir o barramento de memória é algo muito diferente de desabilitar interrupções. Desabilitar interrupções e então realizar a leitura de uma palavra na memória seguida pela escrita não evita que um segundo processador no barramento acesse a palavra entre a leitura e a escrita. Na realidade, desabilitar interrupções no processador 1 não exerce efeito algum sobre o processador 2. A única maneira de manter o processador 2 fora da memória até o processador 1 ter terminado é travar o barramento, o que exige um equipamento de hardware especial (basicamente, uma linha de barramento que assegura que o barramento está travado e indisponível para outros processadores fora aquele que o travar).

Para usar a instrução TSL, usaremos uma variável compartilhada, *lock*, para coordenar o acesso à memória compartilhada. Quando *lock* está em 0, qualquer

processo pode configurá-lo para 1 usando a instrução TSL e, então, ler ou escrever a memória compartilhada. Quando terminado, o processo configura *lock* de volta para 0 usando uma instrução move comum.

Como essa instrução pode ser usada para evitar que dois processos entrem simultaneamente em suas regiões críticas? A solução é dada na Figura 2.25. Nela, uma sub-rotina de quatro instruções é mostrada em uma linguagem de montagem fictícia (mas típica). A primeira instrução copia o valor antigo de *lock* para o registrador e, então, configura *lock* para 1. Assim, o valor antigo é comparado a 0. Se ele não for zero, a trava já foi configurada, de maneira que o programa simplesmente volta para o início e o testa novamente. Cedo ou tarde ele se tornará 0 (quando o processo atualmente em sua região crítica tiver terminado com sua própria região crítica), e a sub-rotina retornar, com a trava configurada. Destravá-la é algo bastante simples: o programa apenas armazena um 0 em *lock*; não são necessárias instruções de sincronização especiais.

Uma solução para o problema da região crítica agora é simples. Antes de entrar em sua região crítica, um processo chama *enter_region*, que fica em espera ocupada até a trava estar livre; então ele adquire a trava e retorna. Após deixar a região crítica, o processo chama *leave_region*, que armazena um 0 em *lock*. Assim como com todas as soluções baseadas em regiões críticas, os processos precisam chamar *enter_region* e *leave_region* nos momentos corretos para que o método funcione. Se um processo trapaceia, a exclusão mútua fracassará. Em outras palavras, regiões críticas funcionam somente se os processos cooperarem.

Uma instrução alternativa para TSL é XCHG, que troca os conteúdos de duas posições atomicamente; por exemplo, um registrador e uma palavra de memória. O código é mostrado na Figura 2.26, e, como podemos ver, é essencialmente o mesmo que a solução com TSL. Todas as CPUs Intel x86 usam a instrução XCHG para a sincronização de baixo nível.

2.3.4 Dormir e acordar

Tanto a solução de Peterson, quanto as soluções usando TSL ou XCHG estão corretas, mas ambas têm o defeito de necessitar da espera ocupada. Na essência, o que

FIGURA 2.25 Entrando e saindo de uma região crítica usando a instrução TSL.

```
enter_region:
        TSL REGISTER,LOCK       | copia lock para o registrador e poe lock em 1
        CMP REGISTER,#0         | lock valia zero?
        JNE enter_region        | se fosse diferente de zero, lock estaria ligado; portanto,
                                  continue no laco de repeticao
        RET                     | retorna a quem chamou; entrou na regiao critica


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

FIGURA 2.26 Entrando e saindo de uma região crítica usando a instrução XCHG.

```
enter_region:
        MOVE REGISTER,#1        | insira 1 no registrador
        XCHG REGISTER,LOCK      | substitua os conteudos do registrador e a variacao de lock
        CMP REGISTER,#0         | lock valia zero?
        JNE enter_region        | se fosse diferente de zero, lock estaria ligado; portanto
                                  continue no laco de repeticao
        RET                     | retorna a quem chamou; entrou na regiao critica


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

essas soluções fazem é o seguinte: quando um processo quer entrar em sua região crítica, ele confere para ver se a entrada é permitida. Se não for, o processo apenas esperará em um laço apertado até que isso seja permitido.

Não apenas essa abordagem desperdiça tempo da CPU, como 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 colocadas de tal forma que *H* é executado sempre que ele estiver em um estado pronto. Em um determinado momento, com *L* em sua região crítica, *H* torna-se pronto para executar (por exemplo, completa uma operação de E/S). *H* agora começa a espera ocupada, mas tendo em vista que *L* nunca é escalonado enquanto *H* estiver executando, *L* jamais recebe a chance de deixar a sua região crítica, de maneira que *H* segue em um laço infinito. Essa situação às vezes é referida como **problema da inversão de prioridade**.

Agora vamos examinar algumas primitivas de comunicação entre processos que bloqueiam em vez de desperdiçar tempo da CPU quando eles não são autorizados a entrar nas 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 chamou bloqueie, isto é, seja suspenso até que outro processo o desperte. A chamada wakeup tem um parâmetro, o processo a ser desperto. Alternativamente, tanto sleep quanto wakeup cada um tem um parâmetro, um endereço de memória usado para parear sleeps com wakeups.

O problema do produtor-consumidor

Como um exemplo de como essas primitivas podem ser usadas, vamos considerar o problema **produtor-consumidor** (também conhecido como problema do **buffer limitado**). Dois processos compartilham de um buffer de tamanho fixo comum. Um deles, o produtor, insere informações no buffer, e o outro, o consumidor, as retira dele. (Também é possível generalizar o problema para ter *m* produtores e *n* consumidores, mas consideraremos apenas o caso de um produtor e um consumidor, porque esse pressuposto simplifica as soluções).

O problema surge quando o produtor quer colocar um item novo no buffer, mas ele já está cheio. A solução é o produtor ir dormir, para ser desperto quando o consumidor tiver removido um ou mais itens. De modo similar, se o consumidor quer remover um item do buffer e vê que este está vazio, ele vai dormir até o produtor colocar algo no buffer e despertá-lo.

Essa abordagem soa suficientemente simples, mas leva aos mesmos tipos de condições de corrida que vimos anteriormente com o diretório de spool. Para controlar o número de itens no buffer, precisaremos de uma variável, *count*. Se o número máximo de itens que o buffer pode conter é *N*, o código do produtor primeiro testará para ver se *count* é *N*. Se ele for, o produtor vai dormir; se não for, o produtor acrescentará um item e incrementará *count*.

O código do consumidor é similar: primeiro testar *count* para ver se ele é 0. Se for, vai dormir; se não for, remove um item e decresce o contador. Cada um dos processos também testa para ver se o outro deve ser desperto e, se assim for, despertá-lo. O código para ambos, produtor e consumidor, é mostrado na Figura 2.27.

Para expressar chamadas de sistema como sleep e wakeup em C, nós as mostraremos como chamadas para rotinas de biblioteca. Elas não fazem 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, lidam com o controle da inserção e retirada de itens do buffer.

Agora voltemos às condições da corrida. Elas podem ocorrer porque o acesso a *count* não é restrito. Como consequência, a situação a seguir poderia eventualmente ocorrer. O buffer está vazio e o consumidor acabou de ler *count* para ver se é 0. Nesse instante, o escalonador decide parar de executar o consumidor temporariamente e começar a executar o produtor. O produtor insere um item no buffer, incrementa *count* e nota que ele agora está em 1. Ponderando que *count* era apenas 0, e assim o consumidor deve estar dormindo, o produtor chama wakeup para despertar o consumidor.

Infelizmente, o consumidor ainda não está logicamente dormindo, então o sinal de despertar é perdido. Quando o consumidor executa em seguida, ele testará o valor de *count* que ele leu antes, descobrirá que ele é 0 e irá dormir. Cedo ou tarde o produtor preencherá o buffer e vai dormir também. Ambos dormirão para sempre.

A essência do problema aqui é que um chamado de despertar enviado para um processo que (ainda) não está dormindo é perdido. Se não fosse perdido, tudo o mais funcionaria. Uma solução rápida é modificar as regras para acrescentar ao quadro um **bit de espera pelo sinal de acordar**. Quando um sinal de despertar é enviado para um processo que ainda está desperto, esse bit é configurado. Depois, quando o processo tentar adormecer, se o bit de espera pelo sinal de acordar estiver ligado, ele será desligado, mas o processo permanecerá desperto. O bit de espera pelo sinal de acordar é um cofrinho para armazenar sinais de despertar. O consumidor limpa o bit de espera pelo sinal de acordar em toda iteração do laço.

FIGURA 2.27 O problema do produtor-consumidor com uma condição de corrida fatal.

```
#define N 100                                   /* numero de lugares no buffer */
int count = 0;                                  /* numero de itens no buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* repita para sempre */
          item = produce_item( );               /* gera o proximo item */
          if (count == N) sleep( );             /* se o buffer estiver cheio, va dormir */
          insert_item(item);                    /* ponha um item no buffer */
          count = count + 1;                    /* incremente o contador de itens no buffer */
          if (count == 1) wakeup(consumer);     /* o buffer estava vazio? */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* repita para sempre */
          if (count == 0) sleep( );             /* se o buffer estiver cheio, va dormir */
          item = remove_item( );                /* retire o item do buffer */
          count = count – 1;                    /* descresca de um contador de itens no buffer */
          if (count == N – 1) wakeup(producer); /* o buffer estava cheio? */
          consume_item(item);                   /* imprima o item */
     }
}
```

Embora o bit de espera pelo sinal de acordar salve a situação nesse exemplo simples, é fácil construir exemplos com três ou mais processos nos quais o bit de espera pelo sinal de acordar é insuficiente. Poderíamos fazer outra simulação e acrescentar um segundo bit de espera pelo sinal de acordar, ou talvez 8 ou 32 deles, mas em princípio o problema ainda está ali.

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. Em sua proposta, um novo tipo de variável, que ele chamava de **semáforo**, foi introduzido. Um semáforo podia ter o valor 0, indicando que nenhum sinal de despertar fora salvo, ou algum valor positivo se um ou mais sinais de acordar estivessem pendentes.

Dijkstra propôs ter duas operações nos semáforos, hoje normalmente chamadas de down e up (generalizações de sleep e wakeup, respectivamente). A operação down em um semáforo confere para ver se o valor é maior do que 0. Se for, ele decrementará o valor (isto é, gasta um sinal de acordar armazenado) e apenas continua. Se o valor for 0, o processo é colocado para dormir sem completar o down para o momento. Conferir o valor, modificá-lo e possivelmente dormir são feitos como uma única **ação atômica** indivisível. É garantido que uma vez que a operação de semáforo tenha começado, nenhum outro processo pode acessar o semáforo até que a operação tenha sido concluída ou bloqueada. Essa atomicidade é absolutamente essencial para solucionar problemas de sincronização e evitar condições de corrida. Ações atômicas, nas quais um grupo de operações relacionadas são todas realizadas sem interrupção ou não são executadas em absoluto, são extremamente importantes em muitas outras áreas da ciência de computação também.

A operação up incrementa o valor de um determinado semáforo. Se um ou mais processos estiverem dormindo naquele semáforo, incapaz de completar uma operação down anterior, um deles é escolhido pelo sistema (por exemplo, ao acaso) e é autorizado a completar seu down. Desse modo, após um up com processos dormindo em um semáforo, ele ainda estará em 0, mas haverá menos processos dormindo nele. A operação de incrementar o semáforo e despertar um processo também é indivisível. Nenhum processo é bloqueado realizando um up, assim como nenhum processo é bloqueado realizando um wakeup no modelo anterior.

Como uma nota, no estudo original de Dijkstra, ele usou os nomes P e V em vez de down e up, respectivamente. Como essas letras não têm significância mnemônica para pessoas que não falam holandês e apenas marginal para aquelas que o falam — *Proberen* (tentar) e *Verhogen* (levantar, erguer) — usaremos os termos down e up em vez disso. Esses mecanismos foram introduzidos pela primeira vez na linguagem de programação Algol 68.

Solucionando o problema produtor-consumidor usando semáforos

Semáforos solucionam o problema do sinal de acordar perdido, como mostrado na Figura 2.28. Para fazê-los funcionar corretamente, é essencial que eles sejam implementados de uma maneira indivisível. A maneira

FIGURA 2.28 O problema do produtor-consumidor usando semáforos.

```
#define N 100                                   /* numero de lugares no buffer */
typedef int semaphore;                          /* semaforos sao um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso a regiao critica */
semaphore empty = N;                            /* conta os lugares vazios no buffer */
semaphore full = 0;                             /* conta os lugares preenchidos no buffer */

void producer(void)
{
    int item;

    while (TRUE) {                              /* TRUE e a constante 1 */
        item = produce_item( );                 /* gera algo para por no buffer */
        down(&empty);                           /* decresce o contador empty */
        down(&mutex);                           /* entra na regiao critica */
        insert_item(item);                      /* poe novo item no buffer */
        up(&mutex);                             /* sai da regiao critica */
        up(&full);                              /* incrementa o contador de lugares preenchidos */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                              /* laco infinito */
        down(&full);                            /* decresce o contador full */
        down(&mutex);                           /* entra na regiao critica */
        item = remove_item( );                  /* pega item do buffer */
        up(&mutex);                             /* sai da regiao critica */
        up(&empty);                             /* incrementa o contador de lugares vazios */
        consume_item(item);                     /* faz algo com o item */
    }
}
```

normal é implementar up e down como chamadas de sistema, com o sistema operacional desabilitando brevemente todas as interrupções enquanto ele estiver testando o semáforo, atualizando-o e colocando o processo para dormir, se necessário. Como todas essas ações exigem apenas algumas instruções, nenhum dano resulta ao desabilitar as interrupções. Se múltiplas CPUs estiverem sendo usadas, cada semáforo deverá ser protegido por uma variável de trava, com as instruções TSL ou XCHG usadas para certificar-se de que apenas uma CPU de cada vez examina o semáforo.

Certifique-se de que você compreendeu que usar TSL ou XCHG para evitar que várias CPUs acessem o semáforo ao mesmo tempo é bastante diferente do produtor ou consumidor em espera ocupada para que o outro esvazie ou encha o buffer. A operação de semáforo levará apenas alguns microssegundos, enquanto o produtor ou o consumidor podem levar tempos arbitrariamente longos.

Essa solução usa três semáforos: um chamado *full* para contar o número de vagas que estão cheias, outro chamado *empty* para contar o número de vagas que estão vazias e mais um chamado *mutex* para se certificar de que o produtor e o consumidor não acessem o buffer ao mesmo tempo. Inicialmente, *full* é 0, *empty* é igual ao número de vagas no buffer e *mutex* é 1. Semáforos que são inicializados para 1 e usados por dois ou mais processos para assegurar que apenas um deles consiga entrar em sua região crítica de cada vez são chamados de **semáforos binários**. Se cada processo realiza um down um pouco antes de entrar em sua região crítica e um up logo depois de deixá-la, a exclusão mútua é garantida.

Agora que temos uma boa primitiva de comunicação entre processos à disposição, vamos voltar e examinar a sequência de interrupção da Figura 2.5 novamente. Em um sistema usando semáforos, a maneira natural de esconder interrupções é ter um semáforo inicialmente configurado para 0, associado com cada dispositivo de E/S. Logo após inicializar um dispositivo de E/S, o processo de gerenciamento realiza um down no semáforo associado, desse modo bloqueando-o imediatamente. Quando a interrupção chega, o tratamento de interrupção então realiza um up nesse modelo, o que deixa o processo relevante pronto para executar de novo. Nesse modelo, o passo 5 na Figura 2.5 consiste em realizar um up no semáforo do dispositivo, assim no passo 6 o escalonador será capaz de executar o gerenciador do dispositivo. É claro que se vários processos estão agora prontos, o escalonador pode escolher executar um processo mais importante ainda em seguida. Examinaremos alguns dos algoritmos usados para escalonamento mais tarde neste capítulo.

No exemplo da Figura 2.28, na realidade, usamos semáforos de duas maneiras diferentes. Essa diferença é importante o suficiente para ser destacada. O semáforo *mutex* é usado para exclusão mútua. Ele é projetado para garantir que apenas um processo de cada vez esteja lendo ou escrevendo no buffer e em variáveis associadas. Essa exclusão mútua é necessária para evitar o caos. Na próxima seção, estudaremos a exclusão mútua e como consegui-la.

O outro uso dos semáforos é para a **sincronização**. Os semáforos *full* e *empty* são necessários para garantir que determinadas sequências ocorram ou não. Nesse caso, eles asseguram que o produtor pare de executar quando o buffer estiver cheio, e que o consumidor pare de executar quando ele estiver vazio. Esse uso é diferente da exclusão mútua.

2.3.6 Mutexes

Quando a capacidade do semáforo de fazer contagem não é necessária, uma versão simplificada, chamada mutex, às vezes é usada. Mutexes são bons somente para gerenciar a exclusão mútua de algum recurso ou trecho de código compartilhados. Eles são fáceis e eficientes de implementar, o que os torna especialmente úteis em pacotes de threads que são implementados inteiramente no espaço do usuário.

Um **mutex** é uma variável compartilhada que pode estar em um de dois estados: destravado ou travado. Em consequência, apenas 1 bit é necessário para representá-lo, mas na prática muitas vezes um inteiro é usado, com 0 significando destravado e todos os outros valores significando travado. Duas rotinas são usadas com mutexes. Quando um thread (ou processo) precisa de acesso a uma região crítica, ele chama *mutex_lock*. Se o mutex estiver destravado naquele momento (significando que a região crítica está disponível), a chamada seguirá e o thread que chamou estará livre para entrar na região crítica.

Por outro lado, se o mutex já estiver travado, o thread que chamou será bloqueado até que o thread na região crítica tenha concluído e chame *mutex_unlock*. Se múltiplos threads estiverem bloqueados no mutex, um deles será escolhido ao acaso e liberado para adquirir a trava.

Como mutexes são muito simples, eles podem ser facilmente implementados no espaço do usuário, desde que uma instrução TSL ou XCHG esteja disponível. O código para *mutex_lock* e *mutex_unlock* para uso com um pacote de threads de usuário são mostrados na

Figura 2.29. A solução com XCHG é essencialmente a mesma.

O código de *mutex_lock* é similar ao código de *enter_region* da Figura 2.25, mas com uma diferença crucial: quando *enter_region* falha em entrar na região crítica, ele segue testando a trava repetidamente (espera ocupada); por fim, o tempo de CPU termina e outro processo é escalonado para executar. Cedo ou tarde, o processo que detém a trava é executado e a libera.

Com threads (de usuário) a situação é diferente, pois não há um relógio que pare threads que tenham sido executados por tempo demais. Em consequência, um thread que tenta adquirir uma trava através da espera ocupada, ficará em um laço para sempre e nunca adquirirá a trava, pois ela jamais permitirá que outro thread execute e a libere.

É aí que entra a diferença entre *enter_region* e *mutex_lock*. Quando o segundo falha em adquirir uma trava, ele chama *thread_yield* para abrir mão da CPU para outro thread. Em consequência, não há espera ocupada. Quando o thread executa da vez seguinte, ele testa a trava novamente.

Tendo em vista que *thread_yield* é apenas uma chamada para o escalonador de threads no espaço de usuário, ela é muito rápida. Em consequência, nem *mutex_lock*, tampouco *mutex_unlock* exigem quaisquer chamadas de núcleo. Usando-os, threads de usuário podem sincronizar inteiramente no espaço do usuário usando rotinas que exigem apenas um punhado de instruções.

O sistema mutex que descrevemos é um conjunto mínimo de chamadas. Com todos os softwares há sempre uma demanda por mais inovações e as primitivas de sincronização não são exceção. Por exemplo, às vezes um pacote de thread oferece uma chamada *mutex_trylock* que adquire a trava ou retorna um código de falha, mas sem bloquear. Essa chamada dá ao thread a flexibilidade para decidir o que fazer em seguida, se houver alternativas para apenas esperar.

Há uma questão sutil que até agora tratamos superficialmente, mas que vale a pena ser destacada: com um pacote de threads de espaço de usuário não há um problema com múltiplos threads terem acesso ao mesmo mutex, já que todos os threads operam em um espaço de endereçamento comum; no entanto, com todas as soluções anteriores, como o algoritmo de Peterson e os semáforos, há uma suposição velada de que os múltiplos processos têm acesso a pelo menos alguma memória compartilhada, talvez apenas uma palavra, mas têm acesso a algo. Se os processos têm espaços de endereçamento disjuntos, como dissemos consistentemente, como eles podem compartilhar a variável *turn* no algoritmo de Peterson, ou semáforos, ou um buffer comum?

Há duas respostas: primeiro, algumas das estruturas de dados compartilhadas, como os semáforos, podem ser armazenadas no núcleo e acessadas somente por chamadas de sistema — essa abordagem elimina o problema; segundo, a maioria dos sistemas operacionais modernos (incluindo UNIX e Windows) oferece uma maneira para os processos compartilharem alguma porção do seu espaço de endereçamento com outros. Dessa maneira, buffers e outras estruturas de dados podem ser compartilhados. No pior caso, em que nada mais é possível, um arquivo compartilhado pode ser usado.

Se dois ou mais processos compartilham a maior parte ou todo o seu espaço de endereçamento, a distinção entre processos e threads torna-se confusa, mas segue de certa forma presente. Dois processos que compartilham um espaço de endereçamento comum ainda têm arquivos abertos diferentes, temporizadores de alarme e outras propriedades por processo, enquanto os threads dentro de um único processo os compartilham.

FIGURA 2.29 Implementação de *mutex_lock* e *mutex_unlock*.

```
mutex_lock:
    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 esta ocupado; escalone um outro thread
    JMP mutex_lock          | tente novamente
ok: RET                     | retorna a quem chamou; entrou na regiao critica

mutex_unlock:
    MOVE MUTEX,#0           | coloca 0 em mutex
    RET                     | retorna a quem chamou
```

E é sempre verdade que múltiplos processos compartilhando um espaço de endereçamento comum nunca têm a eficiência de threads de usuário, já que o núcleo está profundamente envolvido em seu gerenciamento.

Futexes

Com o paralelismo cada vez maior, a sincronização eficiente e o travamento são muito importantes para o desempenho. Travas giratórias são rápidas se a espera for curta, mas desperdiçam ciclos de CPU se não for o caso. Se houver muita contenção, logo é mais eficiente bloquear o processo e deixar o núcleo desbloqueá-lo apenas quando a trava estiver liberada. Infelizmente, isso tem o problema inverso: funciona bem sob uma contenção pesada, mas trocar continuamente para o núcleo fica caro se houver pouca contenção. Para piorar a situação, talvez não seja fácil prever o montante de contenção pela trava.

Uma solução interessante que tenta combinar o melhor dos dois mundos é conhecida como **futex** (ou *fast userspace mutex* — mutex rápido de espaço usuário). Um futex é uma inovação do Linux que implementa travamento básico (de maneira muito semelhante com mutex), mas evita adentrar o núcleo, a não ser que ele realmente tenha de fazê-lo. Tendo em vista que chavear para o núcleo e voltar é algo bastante caro, fazer isso melhora o desempenho consideravelmente. Um futex consiste em duas partes: um serviço de núcleo e uma biblioteca de usuário. O serviço de núcleo fornece uma "fila de espera" que permite que múltiplos processos esperem em uma trava. Eles não executarão, a não ser que o núcleo explicitamente os desbloqueie. Para um processo ser colocado na fila de espera é necessária uma chamada de sistema (cara) e deve ser evitado. Na ausência da contenção, portanto, o futex funciona completamente no espaço do usuário. Especificamente, os processos compartilham uma variável de trava comum — um nome bacana para um número inteiro de 32 bits alinhados que serve como trava. Suponha que a trava seja de início 1 — que consideramos que signifique a trava estar livre. Um thread pega a trava realizando um "decremento e teste" atômico (funções atômicas no Linux consistem de código de montagem em linha revestido por funções C e são definidas em arquivos de cabeçalho). Em seguida, o thread inspeciona o resultado para ver se a trava está ou não liberada. Se ela não estiver no estado travado, tudo vai bem e o nosso thread foi bem-sucedido em pegar a trava. No entanto, se ela estiver nas mãos de outro thread, o nosso tem de esperar. Nesse caso, a biblioteca futex não utiliza a trava giratória, mas usa uma chamada de sistema para colocar o thread na fila de espera no núcleo. Esperamos que o custo da troca para o núcleo seja agora justificado, porque o thread foi bloqueado de qualquer maneira. Quando um thread tiver terminado com a trava, ele a libera com um "incremento e teste" atômico e confere o resultado para ver se algum processo ainda está bloqueado na fila de espera do núcleo. Se isso ocorrer, ele avisará o núcleo que ele pode desbloquear um ou mais desses processos. Se não houver contenção, o núcleo não estará envolvido de maneira alguma.

Mutexes em pthreads

Pthreads proporcionam uma série de funções que podem ser usadas para sincronizar threads. O mecanismo básico usa uma variável mutex, que pode ser travada ou destravada, para guardar cada região crítica. Um thread desejando entrar em uma região crítica tenta primeiro travar o mutex associado. Se o mutex estiver destravado, o thread pode entrar imediatamente e a trava é atomicamente configurada, evitando que outros threads entrem. Se o mutex já estiver travado, o thread que chamou é bloqueado até ser desbloqueado. Se múltiplos threads estão esperando no mesmo mutex, quando ele está destravado, apenas um deles é autorizado a continuar e travá-lo novamente. Essas travas não são obrigatórias. Cabe ao programador assegurar que os threads as usem corretamente.

As principais chamadas relacionadas com os mutexes estão mostradas na Figura 2.30. Como esperado, mutexes podem ser criados e destruídos. As chamadas para realizar essas operações são *pthread_mutex_init* e *pthread_mutex_destroy*, respectivamente. Eles também podem ser travados — por *pthread_mutex_lock* — que tenta adquirir a trava e é bloqueado se o mutex já estiver travado. Há também uma 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*. Ela permite que um thread de fato realize a espera ocupada, se isso for em algum momento necessário. Finalmente, *pthread_mutex_unlock* destrava um mutex e libera exatamente um thread, se um ou mais estiver esperando por ele. Mutexes também podem ter atributos, mas esses são usados somente para fins especializados.

FIGURA 2.30 Algumas chamadas de Pthreads relacionadas a mutexes.

Chamada de thread	Descrição
Pthread_mutex_init	Cria um mutex
Pthread_mutex_destroy	Destrói um mutex existente
Pthread_mutex_lock	Obtém uma trava ou é bloqueado
Pthread_mutex_trylock	Obtém uma trava ou falha
Pthread_mutex_unlock	Libera uma trava

Além dos mutexes, pthreads oferecem um segundo mecanismo de sincronização: **variáveis de condição.** Mutexes são bons para permitir ou bloquear acesso à região crítica. Variáveis de condição permitem que threads sejam bloqueados devido a alguma condição não estar sendo atendida. Quase sempre os dois métodos são usados juntos. Vamos agora examinar a interação de threads, mutexes e variáveis de condição um pouco mais detalhadamente.

Como um exemplo simples, considere o cenário produtor-consumidor novamente: um thread coloca as coisas em um buffer e outro as tira. Se o produtor descobre que não há mais posições livres disponíveis no buffer, ele tem de ser bloqueado até uma se tornar disponível. Mutexes tornam possível realizar a conferência atomicamente sem interferência de outros threads, mas tendo descoberto que o buffer está cheio, o produtor precisa de uma maneira para bloquear e ser despertado mais tarde. É isso que as variáveis de condição permitem.

As chamadas mais importantes relacionadas com as variáveis de condição são mostradas na Figura 2.31. Como você provavelmente esperaria, há chamadas para criar e destruir as variáveis de condição. Elas podem ter atributos e há várias chamadas para gerenciá-las (não mostradas). As operações principais das variáveis de condição são *pthread_cond_wait* e *pthread_cond_signal*. O primeiro bloqueia o thread que chamou até que algum outro thread sinalize (usando a última chamada). As razões para bloquear e esperar não são parte do protocolo de espera e sinalização, é claro. O thread que é bloqueado muitas vezes está esperando pelo que sinaliza para realizar algum trabalho, liberar algum recurso ou desempenhar alguma outra atividade. Apenas então o thread que bloqueia continua. As variáveis de condição permitem que essa espera e bloqueio sejam feitos atomicamente. A chamada *pthread_cond_broadcast* é usada quando há múltiplos threads todos potencialmente bloqueados e esperando pelo mesmo sinal.

FIGURA 2.31 Algumas das chamadas de Pthreads relacionadas com variáveis de condição.

Chamada de thread	Descrição
Pthread_cond_init	Cria uma variável de condição
Pthread_cond_destroy	Destrói uma variável de condição
Pthread_cond_wait	É bloqueado esperando por um sinal
Pthread_cond_signal	Sinaliza para outro thread e o desperta
Pthread_cond_broadcast	Sinaliza para múltiplos threads e desperta todos eles

Variáveis de condição e mutexes são sempre usados juntos. O padrão é um thread travar um mutex e, então, esperar em uma variável condicional quando ele não consegue o que precisa. Por fim, outro thread vai sinalizá-lo, e ele pode continuar. A chamada *pthread_cond_wait* destrava atomicamente o mutex que o está segurando. Por essa razão, o mutex é um dos parâmetros.

Também vale a pena observar que variáveis de condição (diferentemente de semáforos) não têm memória. Se um sinal é enviado sobre o qual não há um thread esperando, o sinal é perdido. Programadores têm de ser cuidadosos para não perder sinais.

Como um exemplo de como mutexes e variáveis de condição são usados, a Figura 2.32 mostra um problema produtor-consumidor muito simples com um único buffer. Quando o produtor preencher o buffer, ele deve esperar até que o consumidor o esvazie antes de produzir o próximo item. Similarmente, quando o consumidor remover um item, ele deve esperar até que o produtor tenha produzido outro. Embora muito simples, esse exemplo ilustra mecanismos básicos. O comando que coloca um thread para dormir deve sempre checar a condição para se certificar de que ela seja satisfeita antes de continuar, visto que o thread poderia ter sido despertado por causa de um sinal UNIX ou alguma outra razão.

2.3.7 Monitores

Com semáforos e mutexes a comunicação entre processos parece fácil, certo? Errado. Observe de perto a ordem dos downs antes de inserir ou remover itens do buffer na Figura 2.28. Suponha que os dois downs no código do produtor fossem invertidos, de maneira que *mutex* tenha sido decrescido antes de *empty* em vez de depois dele. Se o buffer estivesse completamente cheio, o produtor seria bloqueado, com *mutex* configurado para 0. Em consequência,

FIGURA 2.32 Usando threads para solucionar o problema produtor-consumidor.

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

#define MAX 1000000000                          /* quantos numeros produzir */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                    /* usado para sinalizacao */
int buffer = 0;                                 /* buffer usado entre produtor e consumidor */

void *producer(void *ptr)                       /* dados do produtor */
{       int i;

        for (i= 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);   /* obtem acesso exclusivo ao buffer */
                while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
                buffer = i;                       /*coloca 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)                       /* dados do consumidor */
{       int i;

        for (i = 1; i <= MAX; i++) {
                pthread_mutex_lock(&the_mutex);   /* obtem acesso exclusivo ao buffer */
                while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
                buffer = 0;                       /* retira 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);
}
```

da próxima vez que o consumidor tentasse acessar o buffer, ele faria um down em *mutex*, agora 0, e seria bloqueado também. Ambos os processos ficariam bloqueados para sempre e nenhum trabalho seria mais realizado. Essa situação infeliz é chamada de impasse (*deadlock*). Estudaremos impasses detalhadamente no Capítulo 6.

Esse problema é destacado para mostrar o quão cuidadoso você precisa ser quando usa semáforos. Um erro sutil e tudo para completamente. É como programar em linguagem de montagem, mas pior, pois os erros são condições de corrida, impasses e outras formas de comportamento imprevisível e irreproduzível.

Para facilitar a escrita de programas corretos, Brinch Hansen (1973) e Hoare (1974) propuseram uma primitiva de sincronização de nível mais alto chamada **monitor**. Suas propostas diferiam ligeiramente, como descrito mais adiante. Um monitor é uma coleção de rotinas, variáveis e estruturas de dados que são reunidas em um tipo especial de módulo ou pacote. Processos podem chamar as rotinas em um monitor sempre que eles quiserem, mas eles não podem acessar diretamente as estruturas de dados internos do monitor a partir de rotinas declaradas fora dele. A Figura 2.33 ilustra um monitor escrito em uma linguagem imaginária, Pidgin Pascal. C não pode ser usado aqui, porque os monitores são um conceito de *linguagem* e C não os tem.

Os monitores têm uma propriedade importante que os torna úteis para realizar a exclusão mútua: apenas um processo pode estar ativo em um monitor em qualquer dado instante. Monitores são uma construção da linguagem de programação, então o compilador sabe que eles são especiais e podem lidar com chamadas para rotinas de monitor diferentemente de outras chamadas de rotina. Tipicamente, quando um processo chama uma rotina do monitor, as primeiras instruções conferirão para ver se qualquer outro processo está ativo no momento dentro do monitor. Se isso ocorrer, o processo que chamou será suspenso até que o outro processo tenha deixado o monitor. Se nenhum outro processo está usando o monitor, o processo que chamou pode entrar.

FIGURA 2.33 Um monitor.

```
monitor example
    integer i;
    condition c;

    procedure producer ( );
    .
    .
    .
    end;

    procedure consumer ( );
    .       .       .
    end;
end monitor;
```

Cabe ao compilador implementar a exclusão mútua nas entradas do monitor, mas uma maneira comum é usar um mutex ou um semáforo binário. Como o compilador, não o programador, está arranjando a exclusão mútua, é muito menos provável que algo dê errado. De qualquer maneira, a pessoa escrevendo o monitor não precisa ter ciência de como o compilador arranja a exclusão mútua. Basta saber que ao transformar todas as regiões críticas em rotinas de monitores, dois processos jamais executarão suas regiões críticas ao mesmo tempo.

Embora os monitores proporcionem uma maneira fácil de alcançar a exclusão mútua, como vimos anteriormente, isso não é suficiente. Também precisamos de uma maneira para os processos bloquearem quando não puderem prosseguir. No problema do produtor-consumidor, é bastante fácil colocar todos os testes de buffer cheio e buffer vazio nas rotinas de monitor, mas como o produtor seria bloqueado quando encontrasse o buffer cheio?

A solução encontra-se na introdução de **variáveis de condição**, junto com duas operações, wait e signal. Quando uma rotina de monitor descobre que não pode continuar (por exemplo, o produtor encontra o buffer cheio), ela realiza um wait em alguma variável de condição, digamos, *full*. Essa ação provoca o bloqueio do processo que está chamando. Ele também permite outro processo que tenha sido previamente proibido de entrar no monitor a entrar agora. Vimos variáveis de condição e essas operações no contexto de Pthreads anteriormente.

Nesse outro processo, o consumidor, por exemplo, pode despertar o parceiro adormecido realizando um signal na variável de condição que seu parceiro está esperando. Para evitar ter dois processos ativos no monitor ao mesmo tempo, precisamos de uma regra dizendo o que acontece depois de um signal. Hoare propôs deixar o processo recentemente desperto executar, suspendendo o outro. Brinch Hansen propôs uma saída inteligente para o problema, exigindo que um processo realizando um signal *deva* sair do monitor imediatamente. Em outras palavras, um comando signal pode aparecer apenas como o comando final em uma rotina de monitor. Usaremos a proposta de Brinch Hansen, porque ela é conceitualmente mais simples e também mais fácil de implementar. Se um signal for realizado em uma variável de condição em que vários processos estejam esperando, apenas um deles, determinado pelo escalonador do sistema, será revivido.

Como nota, vale mencionar que há também uma terceira solução, não proposta por Hoare, tampouco por

Hansen, que é deixar o emissor do sinal continuar a executar e permitir que o processo em espera comece a ser executado apenas depois de o emissor do sinal ter deixado o monitor.

Variáveis de condição não são contadores. Elas não acumulam sinais para uso posterior da maneira que os semáforos fazem. Desse modo, se uma variável de condição for sinalizada sem ninguém estar esperando pelo sinal, este será perdido para sempre. Em outras palavras, wait precisa vir antes de signal. Essa regra torna a implementação muito mais simples. Na prática, não é um problema, pois é fácil controlar o estado de cada processo com variáveis, se necessário. Um processo que poderia de outra forma realizar um signal pode ver que essa operação não é necessária ao observar as variáveis.

Um esqueleto do problema produtor-consumidor com monitores é dado na Figura 2.34 em uma linguagem imaginária, Pidgin Pascal. A vantagem de usar a Pidgin Pascal é que ela é pura e simples e segue exatamente o modelo Hoare/Brinch Hansen.

Talvez você esteja pensando que as operações wait e signal parecem similares a sleep e wakeup, que vimos antes e tinham condições de corrida fatais. Bem, elas *são* muito similares, mas com uma diferença crucial: sleep e wakeup fracassaram porque enquanto um processo estava tentando dormir, o outro tentava despertá-lo. Com monitores, isso não pode acontecer. A exclusão mútua automática nas rotinas de monitor garante que se, digamos, o produtor dentro de uma rotina de monitor descobrir que o buffer está cheio, ele será capaz de completar a operação wait sem ter de se preocupar com a possibilidade de que o escalonador possa trocar para o consumidor um instante antes de wait ser concluída. O consumidor não será nem deixado entrar no monitor até que wait seja concluído e o produtor seja marcado como não mais executável.

Embora Pidgin Pascal seja uma linguagem imaginária, algumas linguagens de programação reais também dão suporte a monitores, embora nem sempre na forma projetada por Hoare e Brinch Hansen. Java é uma dessas linguagens. Java é uma linguagem orientada a objetos que dá suporte a threads de usuário e também permite que métodos (rotinas) sejam agrupados juntos em classes. Ao acrescentar a palavra-chave synchronized a uma declaração de método, Java garante que uma vez que qualquer thread tenha começado a executar aquele método, não será permitido a nenhum outro thread executar qualquer outro método synchronized daquele objeto. Sem synchronized, não há garantias sobre essa intercalação.

FIGURA 2.34 Um esqueleto do problema produtor-consumidor com monitores. Somente uma rotina do monitor está ativa por vez. O buffer tem *N* vagas.

```
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 = 1then signal (empty)
    end;

    functionremove: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;
```

Uma solução para o problema produtor-consumidor usando monitores em Java é dada na Figura 2.35. Nossa solução tem quatro classes: a classe exterior, *ProducerConsumer*, cria e inicia dois threads, *p* e *c*; a segunda e a terceira classes, *producer* e *consumer*, respectivamente, contêm o que código para o produtor e o consumidor; e, por fim, a classe *our_monitor*, que é o monitor, e ela contém dois threads sincronizados que são usados para realmente inserir itens no buffer compartilhado e removê-los. Diferentemente dos exemplos anteriores, aqui temos o código completo de *insert* e *remove*.

FIGURA 2-35 Uma solução para o problema produtor-consumidor em Java.

```
public class ProducerConsumer {
    static final int N = 100        // constante contendo o tamanho do buffer
    static producer p = new producer( );   // instancia de um novo thread produtor
    static consumer c = new consumer( ); // instancia de um novo thread consumidor
    static our_monitor mon = new our_monitor( );  // instancia 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 metodo run contem o codigo do thread
            int item;
            while (true) {    // laco do produtor
                item = produce_item( );
                mon.insert(item);
            }
        }
        private int produce_item( ) { ... }     // realmente produz
    }

    static class consumer extends Thread {
        public void run( )  {metodo run contem o codigo do thread
            int item;
            while (true) {     // laco do consumidor
                item = mon.remove( );
                consume_item (item);
            }
        }
        private void consume_item(int item) { ... }// realmente consome
    }
    static class our_monitor { // este e o monitor
        private int buffer[ ] = new int[N];
        private int count = 0, lo = 0, hi = 0; // contadores e indices

        public synchronized void insert(int val) {
            if (count == N) go_to_sleep( );   // se o buffer estiver cheio, va dormir
            buffer [hi] = val; // insere um item no buffer
            hi = (hi + 1) % N;        // lugar para colocar o proximo 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, va dormir
            val = buffer [lo];  // busca um item no buffer
            lo = (lo + 1) % N;       // lugar de onde buscar o proximo 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) {};}
    }
}
```

Os threads do produtor e do consumidor são funcionalmente idênticos a seus correspondentes em todos os nossos exemplos anteriores. O produtor tem um laço infinito gerando dados e colocando-os no buffer comum. O consumidor tem um laço igualmente infinito tirando dados do buffer comum e fazendo algo divertido 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 o produtor está ativo dentro de *insert*, ele sabe com certeza que o consumidor não pode estar ativo dentro de *remove*, tornando seguro atualizar as variáveis e o buffer sem medo das condições de corrida. A variável *count* controla quantos itens estão no buffer. Ela pode assumir qualquer valor de 0 até e incluindo $N - 1$. A variável *lo* é o índice da vaga do buffer onde o próximo item deve ser buscado. Similarmente, *hi* é o índice da vaga do buffer onde o próximo item deve ser colocado. É permitido que $lo = hi$, o que significa que 0 item ou N itens estão no buffer. O valor de *count* diz qual caso é válido.

Métodos sincronizados em Java diferem dos monitores clássicos em um ponto essencial: Java não tem variáveis de condição inseridas. Em vez disso, ela oferece duas rotinas, *wait* e *notify*, que são o equivalente a *sleep* e *wakeup*, exceto que, ao serem usadas dentro de métodos sincronizados, elas não são sujeitas a condições de corrida. Na teoria, o método *wait* pode ser interrompido, que é o papel do código que o envolve. Java exige que o tratamento de exceções seja claro. Para nosso propósito, apenas imagine que *go_to_sleep* seja a maneira para ir dormir.

Ao tornar automática a exclusão mútua de regiões críticas, os monitores tornam a programação paralela muito menos propensa a erros do que o uso de semáforos. Mesmo assim, eles também têm alguns problemas. Não é à toa que nossos dois exemplos de monitores estavam escritos em Pidgin Pascal em vez de C, como são os outros exemplos neste livro. Como já dissemos, monitores são um conceito de linguagem de programação. O compilador deve reconhecê-los e arranjá-los para a exclusão mútua de alguma maneira ou outra. C, Pascal e a maioria das outras linguagens não têm monitores, de maneira que não é razoável esperar que seus compiladores imponham quaisquer regras de exclusão mútua. Na realidade, como o compilador poderia saber quais rotinas estavam nos monitores e quais não estavam?

Essas mesmas linguagens tampouco têm semáforos, mas acrescentar semáforos é fácil: tudo o que você precisa fazer é acrescentar suas rotinas de código de montagem curtas à biblioteca para emitir as chamadas up e down. Os compiladores não precisam nem saber que elas existem. É claro que os sistemas operacionais precisam saber a respeito dos semáforos, mas pelo menos se você tem um sistema operacional baseado em semáforo, ainda pode escrever os programas de usuário para ele em C ou C++ (ou mesmo linguagem de montagem se você for masoquista o bastante). Com monitores, você precisa de uma linguagem que os tenha incorporados.

Outro problema com monitores, e também com semáforos, é que eles foram projetados para solucionar o problema da exclusão mútua em uma ou mais CPUs, todas com acesso a uma memória comum. Podemos evitar as corridas ao colocar os semáforos na memória compartilhada e protegê-los com instruções TSL ou XCHG. Quando movemos para um sistema distribuído consistindo em múltiplas CPUs, cada uma com sua própria memória privada e conectada por uma rede de área local, essas primitivas tornam-se inaplicáveis. A conclusão é que os semáforos são de um nível baixo demais e os monitores não são utilizáveis, exceto em algumas poucas linguagens de programação. Além disso, nenhuma das primitivas permite a troca de informações entre máquinas. Algo mais é necessário.

2.3.8 Troca de mensagens

Esse algo mais é a **troca de mensagens**. Esse método de comunicação entre processos usa duas primitivas, send e receive, que, como semáforos e diferentemente dos monitores, são chamadas de sistema em vez de construções de linguagem. Como tais, elas podem ser facilmente colocadas em rotinas de biblioteca, como

```
send(destination, &message);
```

e

```
receive(source, &message);
```

A primeira chamada envia uma mensagem para determinado destino e a segunda recebe uma mensagem de uma dada fonte (ou de *ANY* — qualquer —, se o receptor não se importar). Se nenhuma mensagem estiver disponível, o receptor pode bloquear até que uma chegue. Alternativamente, ele pode retornar imediatamente com um código de erro.

Questões de projeto para sistemas de troca de mensagens

Sistemas de troca de mensagens têm muitos problemas e questões de projeto que não surgem com semáforos ou com monitores, especialmente se os processos de comunicação são de máquinas diferentes conectadas por uma rede. Por exemplo, mensagens podem ser perdidas pela rede. Para proteger-se contra mensagens perdidas, o emissor e o receptor podem combinar que tão logo uma mensagem tenha sido recebida, o receptor enviará uma mensagem especial de **confirmação de recebimento** de volta. Se o emissor não tiver recebido a confirmação de recebimento em dado intervalo, ele retransmite a mensagem.

Agora considere o que acontece se a mensagem é recebida corretamente, mas a confirmação de recebimento enviada de volta para o emissor for perdida. O emissor retransmitirá a mensagem, assim o receptor a receberá duas vezes. É essencial que o receptor seja capaz de distinguir uma nova mensagem da retransmissão de uma antiga. Normalmente, esse problema é solucionado colocando os números em uma sequência consecutiva em cada mensagem original. Se o receptor receber uma mensagem trazendo o mesmo número de sequência que a anterior, ele saberá que a mensagem é uma cópia que pode ser ignorada. A comunicação bem-sucedida diante de trocas de mensagens não confiáveis é uma parte importante do estudo de redes de computadores. Para mais informações, ver Tanenbaum e Wetherall (2010).

Sistemas de mensagens também têm de lidar com a questão de como processos são nomeados, de maneira que o processo especificado em uma chamada send ou receive não seja ambíguo. A **autenticação** também é uma questão nos sistemas de mensagens: como o cliente pode saber se está se comunicando com o servidor de arquivos real, e não com um impostor?

Na outra ponta do espectro, há também questões de projeto que são importantes quando o emissor e o receptor estão na mesma máquina. Uma delas é o desempenho. Copiar mensagens de um processo para o outro é sempre algo mais lento do que realizar uma operação de semáforo ou entrar em um monitor. Muito trabalho foi dispendido em tornar eficiente a transmissão da mensagem.

O problema produtor-consumidor com a troca de mensagens

Agora vamos ver como o problema do produtor-consumidor pode ser solucionado com a troca de mensagens e nenhuma memória compartilhada. Uma solução é dada na Figura 2.36. Supomos que todas as mensagens são do mesmo tamanho e que as enviadas, mas ainda não recebidas são colocadas no buffer automaticamente pelo sistema operacional. Nessa solução, um total de N mensagens é usado, de maneira análoga às N vagas em um buffer de memória compartilhada. O consumidor começa enviando N mensagens vazias para o produtor. Sempre que o produtor tem um item para dar ao consumidor, ele pega uma mensagem vazia e envia de volta uma cheia. Assim, o número total de mensagens no sistema segue constante pelo tempo, de maneira que elas podem ser armazenadas em um determinado montante de memória previamente conhecido.

Se o produtor trabalhar mais rápido que o consumidor, todas as mensagens terminarão cheias, esperando pelo consumidor; o produtor será bloqueado, esperando por uma vazia voltar. Se o consumidor trabalhar mais rápido, então o inverso acontecerá: todas as mensagens estarão vazias esperando pelo produtor para enchê-las; o consumidor será bloqueado, esperando por uma mensagem cheia.

Muitas variações são possíveis com a troca de mensagens. Para começo de conversa, vamos examinar como elas são endereçadas. Uma maneira é designar a cada processo um endereço único e fazer que as mensagens sejam endereçadas aos processos. Uma maneira diferente é inventar uma nova estrutura de dados, chamada **caixa postal**. Uma caixa postal é um local para armazenar um dado número de mensagens, tipicamente especificado quando a caixa postal é criada. Quando caixas postais são usadas, os parâmetros de endereço nas chamadas send e receive são caixas postais, não processos. Quando um processo tenta enviar uma mensagem para uma caixa postal que está cheia, ele é suspenso até que uma mensagem seja removida daquela caixa postal, abrindo espaço para uma nova mensagem.

Para o problema produtor-consumidor, tanto o produtor quanto o consumidor criariam caixas postais grandes o suficiente para conter N mensagens. O produtor enviaria mensagens contendo dados reais para a caixa postal do consumidor. Quando caixas postais são usadas, o mecanismo de buffer é claro: a caixa postal de destino armazena mensagens que foram enviadas para o processo de destino, mas que ainda não foram aceitas.

O outro extremo de ter caixas postais é eliminar todo o armazenamento. Quando essa abordagem é escolhida, se o send for realizado antes do receive, o processo de envio é bloqueado até receive acontecer, momento

FIGURA 2.36 O problema produtor-consumidor com *N* mensagens.

```
#define N 100                                   /* numero de lugares no buffer */

void producer(void)
{
     int item;
     message m;                                 /* buffer de mensagens */

     while (TRUE) {
          item = produce_item( );               /* gera alguma coisa para colocar no buffer */
          receive(consumer, &m);                /* espera que uma mensagem vazia chegue */
          build_message(&m, item);              /* monta uma mensagem para enviar */
          send(consumer, &m);                   /* 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 */
     }
}
```

em que a mensagem pode ser copiada diretamente do emissor para o receptor, sem armazenamento. De modo similar, se o receive é realizado primeiro, o receptor é bloqueado até que um send aconteça. Essa estratégia é muitas vezes conhecida como **rendezvous** (encontro marcado). Ela é mais fácil de implementar do que um esquema de mensagem armazenada, mas é menos flexível, já que o emissor e o receptor são forçados a ser executados de maneira sincronizada.

A troca de mensagens é comumente usada em sistemas de programação paralela. Um sistema de troca de mensagens bem conhecido, por exemplo, é o **MPI** (**message passing interface** — interface de troca de mensagem). Ele é amplamente usado para a computação científica. Para mais informações sobre ele, veja Gropp et al. (1994) e Snirt et al. (1996).

2.3.9 Barreiras

Nosso último mecanismo de sincronização é dirigido a 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 deve prosseguir para a fase seguinte até que todos os processos estejam prontos para isso. Tal comportamento pode ser conseguido colocando uma **barreira** no fim de cada fase. Quando um processo atinge a barreira, ele é bloqueado até que todos os processos tenham atingido a barreira. Isso permite que grupos de processos sincronizem. A operação de barreira está ilustrada na Figura 2.37.

Na Figura 2.37(a) vemos quatro processos aproximando-se de uma barreira. O que isso significa é que eles estão apenas computando e não chegaram ao fim da fase atual ainda. Após um tempo, o primeiro processo termina toda a computação exigida dele durante a primeira fase, então, ele executa a primitiva barrier, geralmente chamando um procedimento de biblioteca. O processo é então suspenso. Um pouco mais tarde, um segundo e então um terceiro processo terminam a primeira fase e também executam a primitiva barrier. Essa situação está ilustrada na Figura 2.37(b). Por fim, quando o último processo, *C*, atinge a barreira, todos os processos são liberados, como mostrado na Figura 2.37(c).

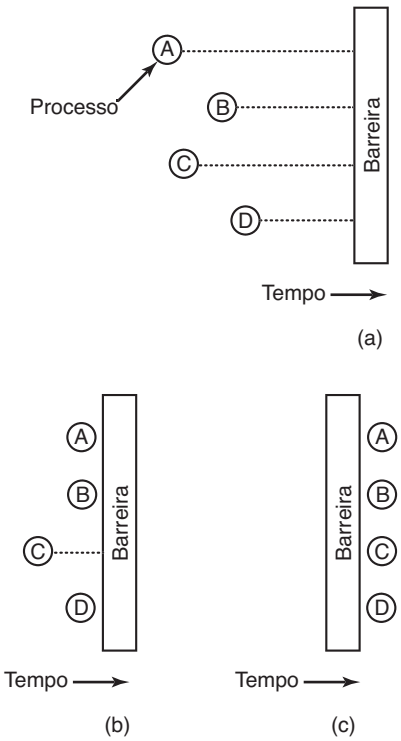


FIGURA 2.37 Uso de uma barreira. (a) Processos aproximando-se de uma barreira. (b) Todos os processos, exceto um, bloqueados na barreira. (c) Quando o último processo chega à barreira, todos podem passar.

Como exemplo de um problema exigindo barreiras, considere um problema típico de relaxação na física ou engenharia. Há tipicamente uma matriz que contém alguns valores iniciais. Os valores podem representar temperaturas em vários pontos em uma lâmina de metal. A ideia pode ser calcular quanto tempo leva para o efeito de uma chama colocada em um canto propagar-se através da lâmina.

Começando com os valores atuais, uma transformação é aplicada à matriz para conseguir a segunda versão; por exemplo, aplicando as leis da termodinâmica para ver quais serão todas as temperaturas posteriormente a ΔT. Então o processo é repetido várias vezes, fornecendo as temperaturas nos pontos de amostra como uma função do tempo à medida que a lâmina aquece. O algoritmo produz uma sequência de matrizes ao longo do tempo, cada uma para um determinado ponto no tempo.

Agora imagine que a matriz é muito grande (por exemplo, 1 milhão por 1 milhão), de maneira que processos paralelos sejam necessários (possivelmente em um multiprocessador) para acelerar o cálculo. Processos diferentes funcionam em partes diferentes da matriz, calculando os novos elementos de matriz a partir dos valores anteriores de acordo com as leis da física. No entanto, nenhum processo pode começar na iteração $n + 1$ até que a iteração n esteja completa, isto é, até que todos os processos tenham terminado seu trabalho atual. A maneira de se alcançar essa meta é programar cada processo para executar uma operação barrier após ele ter terminado sua parte da iteração atual. Quando todos tiverem terminado, a nova matriz (a entrada para a próxima iteração) será terminada, e todos os processos serão liberados simultaneamente para começar a próxima iteração.

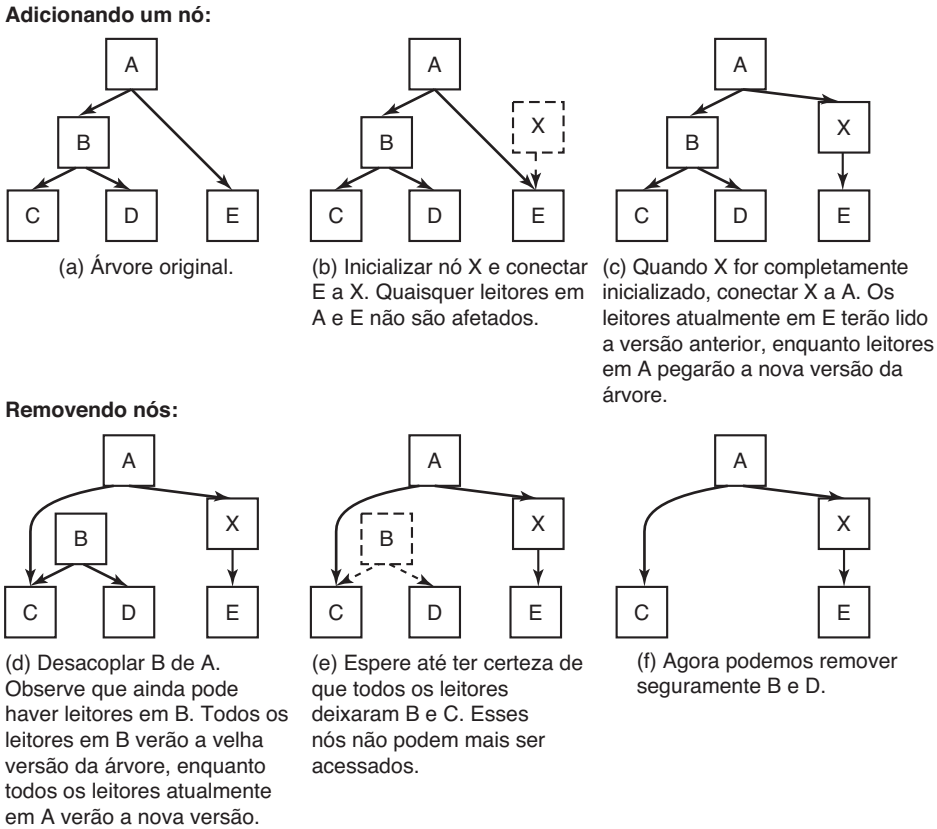
2.3.10 Evitando travas: leitura-cópia-atualização

As travas mais rápidas não são travas de maneira alguma. A questão é se podemos permitir acessos de leitura e escrita concorrentes a estruturas de dados compartilhados sem usar travas. Em geral, a resposta é claramente não. Imagine o processo A ordenando um conjunto de números, enquanto o processo B está calculando a média. Como A desloca os valores para lá e para cá através do conjunto, B pode encontrar alguns valores várias vezes e outros jamais. O resultado poderia ser qualquer um, mas seria quase certamente errado.

Em alguns casos, no entanto, podemos permitir que um escritor atualize uma estrutura de dados mesmo que outros processos ainda a estejam usando. O truque é assegurar que cada leitor leia a versão anterior dos dados, ou a nova, mas não alguma combinação esquisita da anterior e da nova. Como ilustração, considere a árvore mostrada na Figura 2.38. Leitores percorrem a árvore da raiz até suas folhas. Na metade superior da figura, um novo nó X é acrescentado. Para fazê-lo, "deixamos" o nó configurado antes de torná-lo visível na árvore: inicializamos todos os valores no nó X, incluindo seus ponteiros filhos. Então, com uma escrita atômica, tornamos X um filho de A. Nenhum leitor jamais lerá uma versão inconsistente. Na metade inferior da figura, subsequentemente removemos B e D. Primeiro, fazemos que o ponteiro filho esquerdo de A aponte para C. Todos os leitores que estavam em A continuarão com o nó C e nunca verão B ou D. Em outras palavras, eles verão apenas a nova versão. Da mesma maneira, todos os leitores atualmente em B ou D continuarão seguindo os ponteiros estruturais de dados originais e verão a versão anterior. Tudo fica bem assim, e jamais precisamos travar qualquer coisa. A principal razão por que a remoção de B e D funciona sem travar a estrutura de dados é que **RCU (Ready-Copy-Update** — leitura-cópia-atualização) desacopla as fases de *remoção* e *recuperação* da atualização.

É claro que há um problema. Enquanto não tivermos certeza de não haver mais leitores de B ou D, não podemos realmente liberá-los. Mas quanto tempo devemos esperar? Um minuto? Dez? Temos de esperar até que o último leitor

FIGURA 2.38 *Leitura-cópia-atualização*: inserindo um nó na árvore e então removendo um galho — tudo sem travas.



(a) Árvore original.

(b) Inicializar nó X e conectar E a X. Quaisquer leitores em A e E não são afetados.

(c) Quando X for completamente inicializado, conectar X a A. Os leitores atualmente em E terão lido a versão anterior, enquanto leitores em A pegarão a nova versão da árvore.

(d) Desacoplar B de A. Observe que ainda pode haver leitores em B. Todos os leitores em B verão a velha versão da árvore, enquanto todos os leitores atualmente em A verão a nova versão.

(e) Espere até ter certeza de que todos os leitores deixaram B e C. Esses nós não podem mais ser acessados.

(f) Agora podemos remover seguramente B e D.

tenha deixado esses nós. RCU determina cuidadosamente o tempo máximo que um leitor pode armazenar uma referência para a estrutura de dados. Após esse período, ele pode recuperar seguramente a memória. Especificamente, leitores acessam a estrutura de dados no que é conhecido como uma **seção crítica do lado do leitor** que pode conter qualquer código, desde que não bloqueie ou adormeça. Nesse caso, sabemos o tempo máximo que precisamos esperar. Especificamente, definimos um **período de graça** como qualquer período no qual sabemos que cada thread está do lado de fora da seção de leitura crítica pelo menos uma vez. Tudo ficará bem se esperarmos por um período que seja pelo menos igual ao período de graça antes da recuperação. Como não é permitido ao código na seção crítica de leitura bloquear ou adormecer, um critério simples é esperar até que todos os threads tenham realizado um chaveamento de contexto.

2.4 Escalonamento

Quando um computador é multiprogramado, ele frequentemente tem múltiplos processos ou threads competindo pela CPU ao mesmo tempo. Essa situação ocorre sempre que dois ou mais deles estão simultaneamente no estado pronto. Se apenas uma CPU está disponível, uma escolha precisa ser feita sobre qual processo será executado em seguida. A parte do sistema operacional que faz a escolha é chamada de **escalonador**, e o algoritmo que ele usa é chamado de **algoritmo de escalonamento**. Esses tópicos formam o assunto a ser tratado nas seções a seguir.

Muitas das mesmas questões que se aplicam ao escalonamento de processos também se aplicam ao escalonamento de threads, embora algumas sejam diferentes. Quando o núcleo gerencia threads, o escalonamento é geralmente feito por thread, com pouca ou nenhuma consideração sobre o processo ao qual o thread pertence. De início nos concentraremos nas questões de escalonamento que se aplicam a ambos, processos e threads. Depois, examinaremos explicitamente o escalonamento de threads e algumas das questões exclusivas que ele gera. Abordaremos os chips multinúcleo no Capítulo 8.

2.4.1 Introdução ao escalonamento

Nos 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 o próximo trabalho na fita. Com os sistemas de multiprogramação, ele tornou-se mais complexo porque geralmente havia múltiplos usuários esperando pelo serviço. Alguns computadores de grande porte ainda combinam serviço em lote e de compartilhamento de tempo, exigindo que o escalonador decida se um trabalho em lote ou um usuário interativo em um terminal deve ir em seguida. (Como uma nota, um trabalho em lote pode ser uma solicitação para executar múltiplos programas em sucessão, mas para esta seção presumiremos apenas que se trata de uma solicitação 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 percebido e satisfação do usuário. Em consequência, uma grande quantidade de trabalho foi dedicada ao desenvolvimento de algoritmos de escalonamento inteligentes e eficientes.

Com o advento dos computadores pessoais, a situação mudou de duas maneiras. Primeiro, na maior parte do tempo há apenas um processo ativo. É improvável que um usuário preparando um documento em um processador de texto esteja simultaneamente compilando um programa em segundo plano. Quando o usuário digita um comando ao processador de texto, o escalonador não precisa fazer muito esforço para descobrir qual processo executar — o processador de texto é o único candidato.

Segundo, computadores tornaram-se tão rápidos com o passar dos anos que a CPU dificilmente ainda é um recurso escasso. A maioria dos programas para computadores pessoais é limitada pela taxa na qual o usuário pode apresentar a entrada (digitando ou clicando), não pela taxa na qual a CPU pode processá-la. Mesmo as compilações, um importante sorvedouro de ciclos de CPUs no passado, levam apenas alguns segundos hoje. Mesmo quando dois programas estão de fato sendo executados ao mesmo tempo, como um processador de texto e uma planilha, dificilmente importa qual deles vai primeiro, pois o usuário provavelmente está esperando que ambos terminem. Como consequência, o escalonamento não importa muito em PCs simples. É claro que há aplicações que praticamente devoram a CPU viva. Por exemplo, reproduzir uma hora de vídeo de alta resolução enquanto se ajustam as cores em cada um dos 107.892 quadros (em NTSC) ou 90.000 quadros (em PAL) exige uma potência computacional de nível industrial. No entanto, aplicações similares são a exceção em vez de a regra.

Quando voltamos aos servidores em rede, a situação muda consideravelmente. Aqui múltiplos processos muitas vezes competem pela CPU, de maneira que o escalonamento importa outra vez. Por exemplo, quando a CPU tem de escolher entre executar um processo que reúne as estatísticas diárias e um que serve a solicitações de usuários, estes ficarão muito mais contentes se o segundo receber a primeira chance de acessar a CPU.

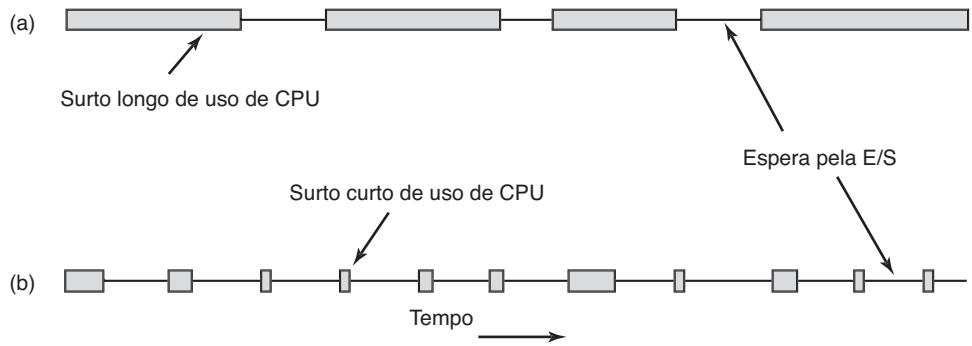
O argumento da "abundância de recursos" também não se sustenta em muitos dispositivos móveis, como smartphones (exceto talvez os modelos mais potentes) e nós em redes de sensores. Além disso, já que a duração da bateria é uma das restrições mais importantes nesses dispositivos, alguns escalonadores tentam otimizar o consumo de energia.

Além de escolher o processo certo a ser executado, o escalonador também tem de se preocupar em fazer um uso eficiente da CPU, pois o chaveamento de processos é algo caro. Para começo de conversa, uma troca do modo usuário para o modo núcleo precisa ocorrer. Então o estado do processo atual precisa ser salvo, incluindo armazenar os seus registros na tabela de processos para que eles possam ser recarregados mais tarde. Em alguns sistemas, o mapa de memória (por exemplo, os bits de referência à memória na tabela de páginas) precisa ser salvo da mesma maneira. Em seguida, um novo processo precisa ser selecionado executando o algoritmo de escalonamento. Após isso, a MMU (memory management unit — unidade de gerenciamento de memória) precisa ser recarregada com o mapa de memória do novo processo. Por fim, o novo processo precisa ser inicializado. Além de tudo isso, a troca de processo pode invalidar o cache de memória e as tabelas relacionadas, forçando-o a ser dinamicamente recarregado da memória principal duas vezes (ao entrar no núcleo e ao deixá-lo). De modo geral, realizar muitas trocas de processos por segundo pode consumir um montante substancial do tempo da CPU, então, recomenda-se cautela.

Comportamento de processos

Quase todos os processos alternam surtos de computação com solicitações de E/S (disco ou rede), como mostrado na Figura 2.39. Muitas vezes, a CPU executa por um tempo sem parar, então uma chamada de sistema é feita para ler de um arquivo ou escrever para um arquivo. Quando a chamada de sistema é concluída, a CPU calcula novamente até que ela precisa de mais dados ou tem de escrever mais dados, e assim por diante. Observe que algumas atividades de E/S contam como computação. Por exemplo, quando a CPU copia bits para uma RAM de vídeo para atualizar a tela, ela está

FIGURA 2.39 Surtos de uso da CPU alternam-se com períodos de espera por E/S. (a) Um processo limitado pela CPU. (b) Um processo limitado pela E/S.



computando, não realizando E/S, pois a CPU está em uso. E/S nesse sentido é quando um processo entra no estado bloqueado esperando por um dispositivo externo para concluir o seu trabalho.

A questão importante a ser observada a respeito da Figura 2.39 é que alguns processos, como o mostrado na Figura 2.39(a), passam a maior do tempo computando, enquanto outros, como o mostrado na Figura 2.39(b), passam a maior parte do tempo esperando pela E/S. Os primeiros são chamados **limitados pela computação** ou **limitados pela CPU**; os segundos são chamados **limitados pela E/S**. Processos limitados pela CPU geralmente têm longos surtos de CPU e então esporádicas esperas de E/S, enquanto os processos limitados pela E/S têm surtos de CPU curtos e esperas de E/S frequentes. Observe que o fator chave é o comprimento do surto da CPU, não o comprimento do surto da E/S. Processos limitados pela E/S são limitados pela E/S porque eles não computam muito entre solicitações de E/S, não por terem tais solicitações especialmente demoradas. Eles levam o mesmo tempo para emitir o pedido de hardware para ler um bloco de disco, independentemente de quanto tempo levam para processar os dados após eles chegarem.

Vale a pena observar que, à medida que as CPUs ficam mais rápidas, os processos tendem a ficar mais limitados pela E/S. Esse efeito ocorre porque as CPUs estão melhorando muito mais rápido que os discos. Em consequência, é provável que o escalonamento de processos limitados pela E/S torne-se um assunto mais importante no futuro. A ideia básica aqui é que se um processo limitado pela E/S quiser executar, ele deve receber uma chance rapidamente para que possa emitir sua solicitação de disco e manter o disco ocupado. Como vimos na Figura 2.6, quando os processos são limitados pela E/S, são necessários diversos deles para manter a CPU completamente ocupada.

Quando escalonar

Uma questão fundamental relacionada com o escalonamento é quando tomar decisões de escalonamento. Na realidade, há uma série de situações nas quais o escalonamento é necessário. Primeiro, quando um novo processo é criado, uma decisão precisa ser tomada a respeito de qual processo, o pai ou o filho, deve ser executado. Tendo em vista que ambos os processos estão em um estado pronto, trata-se de uma decisão de escalonamento normal e pode ser qualquer uma, isto é, o escalonador pode legitimamente escolher executar o processo pai ou o filho em seguida.

Segundo, uma decisão de escalonamento precisa ser tomada ao término de um processo. Esse processo não pode mais executar (já que ele não existe mais), então algum outro precisa ser escolhido do conjunto de processos prontos. Se nenhum está pronto, um processo ocioso gerado pelo sistema normalmente é executado.

Terceiro, quando um processo bloqueia para E/S, em um semáforo, ou por alguma outra razão, outro processo precisa ser selecionado para executar. Às vezes, a razão para bloquear pode ter um papel na escolha. Por exemplo, se *A* é um processo importante e ele está esperando por *B* para sair de sua região crítica, deixar que *B* execute em seguida permitirá que ele saia de sua região crítica e desse modo deixe que *A* continue. O problema, no entanto, é que o escalonador geralmente não tem a informação necessária para levar essa dependência em consideração.

Quarto, quando ocorre uma interrupção de E/S, uma decisão de escalonamento pode ser feita. Se a interrupção veio de um dispositivo de E/S que agora completou seu trabalho, algum processo que foi bloqueado esperando pela E/S pode agora estar pronto para executar. Cabe ao escalonador decidir se deve executar o processo que ficou pronto há pouco, o processo que estava

sendo executado no momento da interrupção, ou algum terceiro processo.

Se um hardware de relógio fornece interrupções periódicas a 50 ou 60 Hz ou alguma outra frequência, uma decisão de escalonamento pode ser feita a cada interrupção ou a cada *k*-ésima interrupção de relógio. Algoritmos de escalonamento podem ser divididos em duas categorias em relação a como lidar com interrupções de relógio. Um algoritmo de escalonamento **não preemptivo** escolhe um processo para ser executado e então o deixa ser executado até que ele seja bloqueado (seja em E/S ou esperando por outro processo), ou libera voluntariamente a CPU. Mesmo que ele execute por muitas horas, não será suspenso forçosamente. Na realidade, nenhuma decisão de escalonamento é feita durante interrupções de relógio. Após o processamento da interrupção de relógio ter sido concluído, o processo que estava executando antes da interrupção é retomado, a não ser que um processo mais prioritário esteja esperando por um tempo de espera agora satisfeito.

Por outro lado, um algoritmo de escalonamento **preemptivo** escolhe um processo e o deixa executar por no máximo um certo tempo fixado. Se ele ainda estiver executando ao fim do intervalo de tempo, ele é suspenso e o escalonador escolhe outro processo para executar (se algum estiver disponível). Realizar o escalonamento preemptivo exige que uma interrupção de relógio ocorra ao fim do intervalo para devolver o controle da CPU de volta para o escalonador. Se nenhum relógio estiver disponível, o escalonamento não preemptivo é a única solução.

Categorias de algoritmos de escalonamento

De maneira pouco surpreendente, em diferentes ambientes, distintos algoritmos de escalonamento são necessários. Essa situação surge porque diferentes áreas de aplicação (e de tipos de sistemas operacionais) têm metas diversas. Em outras palavras, o que deve ser otimizado pelo escalonador não é o mesmo em todos os sistemas. Três ambientes valem ser destacados aqui:

1. Lote.
2. Interativo.
3. Tempo real.

Sistemas em lote ainda são amplamente usados no mundo de negócios para folhas de pagamento, estoques, contas a receber, contas a pagar, cálculos de juros (em bancos), processamento de pedidos de indenização (em companhias de seguro) e outras tarefas periódicas. Em sistemas em lote, não há usuários esperando impacientemente em seus terminais para uma resposta rápida a uma solicitação menor. Em consequência, algoritmos não preemptivos, ou algoritmos preemptivos com longos períodos para cada processo são muitas vezes aceitáveis. Essa abordagem reduz os chaveamentos de processos e melhora o desempenho. Na realidade, os algoritmos em lote são bastante comuns e muitas vezes aplicáveis a outras situações também, o que torna seu estudo interessante, mesmo para pessoas não envolvidas na computação corporativa de grande porte.

Em um ambiente com usuários interativos, a preempção é essencial para evitar que um processo tome conta da CPU e negue serviço para os outros. Mesmo que nenhum processo execute de modo intencional para sempre, um erro em um programa pode levar um processo a impedir indefinidamente que todos os outros executem. A preempção é necessária para evitar esse comportamento. Os servidores também caem nessa categoria, visto que eles normalmente servem a múltiplos usuários (remotos), todos os quais estão muito apressados, assim como usuários de computadores.

Em sistemas com restrições de tempo real, a preempção às vezes, por incrível que pareça, não é necessária, porque os processos sabem que eles não podem executar por longos períodos e em geral realizam o seu trabalho e bloqueiam rapidamente. A diferença com os sistemas interativos é que os de tempo real executam apenas programas que visam ao progresso da aplicação à mão. Sistemas interativos são sistemas para fins gerais e podem executar programas arbitrários que não são cooperativos e talvez até mesmo maliciosos.

Objetivos do algoritmo de escalonamento

A fim de projetar um algoritmo de escalonamento, é necessário ter alguma ideia do que um bom algoritmo deve fazer. Certas metas dependem do ambiente (em lote, interativo ou de tempo real), mas algumas são desejáveis em todos os casos. Algumas metas estão listadas na Figura 2.40. Discutiremos essas metas a seguir.

Em qualquer circunstância, a justiça é importante. Processos comparáveis devem receber serviços comparáveis. Conceder a um processo muito mais tempo de CPU do que para um processo equivalente não é justo. É claro que categorias diferentes de processos podem ser tratadas diferentemente. Pense sobre controle de segurança e elaboração da folha de pagamento em um centro de computadores de um reator nuclear.

De certa maneira relacionado com justiça está o cumprimento das políticas do sistema. Se a política local é que os processos de controle de segurança são executados

FIGURA 2.40 Algumas metas do algoritmo de escalonamento sob diferentes circunstâncias.

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

sempre que quiserem, mesmo que isso signifique atraso de 30 segundos da folha de pagamento, o escalonador precisa certificar-se de que essa política seja cumprida.

Outra meta geral é manter todas as partes do sistema ocupadas quando possível. Se a CPU e todos os outros dispositivos de E/S podem ser mantidos executando o tempo inteiro, mais trabalho é realizado por segundo do que se alguns dos componentes estivessem ociosos. No sistema em lote, por exemplo, o escalonador tem controle sobre quais tarefas são trazidas à memória para serem executadas. Ter alguns processos limitados pela CPU e alguns limitados pela E/S juntos na memória é uma ideia melhor do que primeiro carregar e executar todas as tarefas limitadas pela CPU e, quando forem concluídas, carregar e executar todas as tarefas limitadas pela E/S. Se a segunda estratégia for usada, quando os processos limitados pela CPU estiverem sendo executados, eles disputarão a CPU e o disco ficará ocioso. Depois, quando as tarefas limitadas pela E/S entrarem, elas disputarão o disco e a CPU ficará ociosa. Assim, é melhor manter o sistema inteiro executando ao mesmo tempo mediante uma mistura cuidadosa de processos.

Os gerentes de grandes centros de computadores que executam muitas tarefas em lote costumam observar três métricas para ver como seus sistemas estão desempenhando: vazão, tempo de retorno e utilização da CPU. A **vazão** é o número de tarefas por hora que o sistema completa. Considerados todos os fatores, terminar 50 tarefas por hora é melhor do que terminar 40 tarefas por hora. O **tempo de retorno** é estatisticamente o tempo médio do momento em que a tarefa em lote é submetida até o momento em que ela é concluída. Ele mede quanto tempo o usuário médio tem de esperar pela saída. Aqui a regra é: menos é mais.

Um algoritmo de escalonamento que tenta maximizar a vazão talvez não minimize necessariamente o tempo de retorno. Por exemplo, dada uma combinação de tarefas curtas e tarefas longas, um escalonador que sempre executou tarefas curtas e nunca as longas talvez consiga uma excelente vazão (muitas tarefas curtas por hora), mas à custa de um tempo de retorno terrível para as tarefas longas. Se as tarefas curtas seguissem chegando a uma taxa aproximadamente uniforme, as tarefas longas talvez nunca fossem executadas, tornando o tempo de retorno médio infinito, conquanto alcançando uma alta vazão.

A utilização da CPU é muitas vezes usada como uma métrica nos sistemas em lote. No entanto, ela não é uma boa métrica. O que de fato importa é quantas tarefas por hora saem do sistema (vazão) e quanto tempo leva para receber uma tarefa de volta (tempo de retorno). Usar a utilização de CPU como uma métrica é como classificar carros com base em seu giro de motor. Entretanto, saber quando a utilização da CPU está próxima de 100% é útil para saber quando chegou o momento de obter mais poder computacional.

Para sistemas interativos, aplicam-se metas diferentes. A mais importante é minimizar o **tempo de resposta**, isto é, o tempo entre emitir um comando e receber o resultado. Em um computador pessoal, em que um processo de segundo plano está sendo executado (por exemplo, lendo e armazenando e-mail da rede), uma solicitação de usuário para começar um programa ou abrir um arquivo deve ter precedência sobre o trabalho de segundo plano. Atender primeiro todas as solicitações interativas será percebido como um bom serviço.

Uma questão de certa maneira relacionada é o que poderia ser chamada de **proporcionalidade**. Usuários têm uma ideia inerente (porém muitas vezes incorreta) de quanto tempo as coisas devem levar. Quando uma solicitação que o usuário percebe como complexa leva muito tempo, os usuários aceitam isso, mas quando uma solicitação percebida como simples leva muito tempo, eles ficam irritados. Por exemplo, se clicar em um ícone que envia um vídeo de 500 MB para um servidor na nuvem demorar 60 segundos, o usuário provavelmente aceitará isso como um fato da vida por não esperar que a transferência leve 5 s. Ele sabe que levará um tempo.

Por outro lado, quando um usuário clica em um ícone que desconecta a conexão com o servidor na nuvem após o vídeo ter sido enviado, ele tem expectativas diferentes. Se a desconexão não estiver completa após 30 s, o usuário provavelmente estará soltando algum palavrão e após 60 s ele estará espumando de raiva. Esse comportamento decorre da percepção comum dos usuários de que enviar um monte de dados *supostamente* leva muito mais tempo que apenas desconectar uma conexão. Em alguns casos (como esse), o escalonador não pode fazer nada a respeito do tempo de resposta, mas em outros casos ele pode, especialmente quando o atraso é causado por uma escolha ruim da ordem dos processos.

Sistemas de tempo real têm propriedades diferentes de sistemas interativos e, desse modo, metas de escalonamento diferentes. Eles são caracterizados por ter prazos que devem — ou pelo menos deveriam —, ser cumpridos. Por exemplo, se um computador está controlando um dispositivo que produz dados a uma taxa regular, deixar de executar o processo de coleta de dados em tempo pode resultar em dados perdidos. Assim, a principal exigência de um sistema de tempo real é cumprir com todos (ou a maioria) dos prazos.

Em alguns sistemas de tempo real, especialmente aqueles envolvendo multimídia, a previsibilidade é importante. Descumprir um prazo ocasional não é fatal, mas se o processo de áudio executar de maneira errática demais, a qualidade do som deteriorará rapidamente. O vídeo também é uma questão, mas o ouvido é muito mais sensível a atrasos que o olho. Para evitar esse problema, o escalonamento de processos deve ser altamente previsível e regular. Neste capítulo, estudaremos algoritmos de escalonamento interativo e em lote. O escalonamento de tempo real não é abordado no livro, mas no material extra sobre sistemas operacionais de multimídia na Sala Virtual do livro.

2.4.2 Escalonamento em sistemas em lote

Chegou o momento agora de passar das questões de escalonamento gerais para algoritmos de escalonamento específicos. Nesta seção, examinaremos os algoritmos usados em sistemas em lote. Nas seções seguintes, examinaremos sistemas interativos e de tempo real. Vale a pena destacar que alguns algoritmos são usados tanto nos sistemas interativos como nos em lote. Estudaremos esses mais tarde.

Primeiro a chegar, primeiro a ser servido

É provável que o mais simples de todos os algoritmos de escalonamento já projetados seja o **primeiro a chegar, primeiro a ser servido** (**first-come, first-served**) não preemptivo. Com esse algoritmo, a CPU é atribuída aos processos na ordem em que a requisitam. Basicamente, há uma fila única de processos prontos. Quando a primeira tarefa entrar no sistema de manhã, ela é iniciada imediatamente e deixada executar por quanto tempo ela quiser. Ela não é interrompida por ter sido executada por tempo demais. À medida que as outras tarefas chegam, elas são colocadas no fim da fila. Quando o processo que está sendo executado é bloqueado, o primeiro processo na fila é executado em seguida. Quando um processo bloqueado fica pronto — assim como uma tarefa que chegou há pouco —, ele é colocado no fim da fila, atrás dos processos em espera.

A grande força desse algoritmo é que ele é fácil de compreender e igualmente fácil de programar. Ele também é tão justo quanto alocar ingressos escassos de um concerto ou iPhones novos para pessoas que estão dispostas a esperar na fila desde às duas da manhã. Com esse algoritmo, uma única lista encadeada controla todos os processos. Escolher um processo para executar exige apenas remover um da frente da fila. Acrescentar uma nova tarefa ou desbloquear um processo exige apenas colocá-lo no fim da fila. O que poderia ser mais simples de compreender e implementar?

Infelizmente, o primeiro a chegar, primeiro a ser servido também tem uma desvantagem poderosa. Suponha que há um processo limitado pela computação que é executado por 1 s de cada vez e muitos processos limitados pela E/S que usam pouco tempo da CPU, mas cada um tem de realizar 1.000 leituras do disco para ser concluído. O processo limitado pela computação é executado por 1 s, então ele lê um bloco de disco. Todos os processos de E/S são executados agora e começam leituras de disco. Quando o processo limitado pela computação obtém seu bloco de disco, ele é executado por mais 1 s, seguido por todos os processos limitados pela E/S em rápida sucessão.

O resultado líquido é que cada processo limitado pela E/S lê 1 bloco por segundo e levará 1.000 s para terminar. Com o algoritmo de escalonamento que causasse a preempção do processo limitado pela computação a cada 10 ms, os processos limitados pela E/S terminariam em 10 s em vez de 1.000 s, e sem retardar muito o processo limitado pela computação.

Tarefa mais curta primeiro

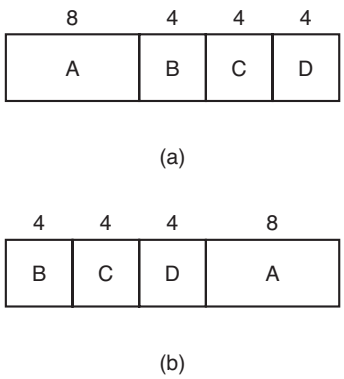
Agora vamos examinar outro algoritmo em lote não preemptivo que presume que os tempos de execução são conhecidos antecipadamente. Em uma companhia de seguros, por exemplo, as pessoas podem prever com

bastante precisão quanto tempo levará para executar um lote de 1.000 solicitações, tendo em vista que um trabalho similar é realizado todos os dias. Quando há vários trabalhos igualmente importantes esperando na fila de entrada para serem iniciados, o escalonador escolhe a **tarefa mais curta primeiro** (**shortest job first**). Observe a Figura 2.41. Nela vemos quatro tarefas *A, B, C* e *D* com tempos de execução de 8, 4, 4 e 4 minutos, respectivamente. Ao executá-las nessa ordem, o tempo de retorno para *A* é 8 minutos, para *B* é 12 minutos, para *C* é 16 minutos e para *D* é 20 minutos, resultando em uma média de 14 minutos.

Agora vamos considerar executar essas quatro tarefas usando o algoritmo *tarefa mais curta primeiro*, como mostrado na Figura 2.41(b). Os tempos de retorno são agora 4, 8, 12 e 20 minutos, resultando em uma média de 11 minutos. A tarefa mais curta primeiro é provavelmente uma ótima escolha. Considere o caso de quatro tarefas, com tempos de execução de *a, b, c* e *d*, respectivamente. A primeira tarefa termina no tempo *a*, a segunda no tempo $a + b$, e assim por diante. O tempo de retorno médio é $(4a + 3b + 2c + d)/4$. Fica claro que *a* contribui mais para a média do que os outros tempos, logo deve ser a tarefa mais curta, com *b* em seguida, então *c* e finalmente *d* como o mais longo, visto que ele afeta apenas seu próprio tempo de retorno. O mesmo argumento aplica-se igualmente bem a qualquer número de tarefas.

Vale a pena destacar que a *tarefa mais curta primeiro* é ótima apenas quando todas as tarefas estão disponíveis simultaneamente. Como um contraexemplo, considere cinco tarefas, *A* a *E*, com tempos de execução de 2, 4, 1, 1 e 1, respectivamente. Seus tempos de chegada são 0, 0, 3, 3 e 3. No início, apenas *A* ou *B* podem ser escolhidos, dado que as outras três tarefas não chegaram ainda. Usando a *tarefa mais curta primeiro*, executaremos as tarefas na ordem *A, B, C, D, E* para um tempo de espera médio de 4,6. No entanto, executá-las na ordem *B, C, D, E, A* tem um tempo de espera médio de 4,4.

FIGURA 2.41 Um exemplo do escalonamento tarefa mais curta primeiro. (a) Executando quatro tarefas na ordem original. (b) Executando-as na ordem tarefa mais curta primeiro.



Tempo restante mais curto em seguida

Uma versão preemptiva da ***tarefa mais curta primeiro*** é o **tempo restante mais curto em seguida** (*shortest remaining time next)*. Com esse algoritmo, o escalonador escolhe o processo cujo tempo de execução restante é o mais curto. De novo, o tempo de execução precisa ser conhecido antecipadamente. Quando uma nova tarefa chega, seu tempo total é comparado com o tempo restante do processo atual. Se a nova tarefa precisa de menos tempo para terminar do que o processo atual, este é suspenso e a nova tarefa iniciada. Esse esquema permite que tarefas curtas novas tenham um bom desempenho.

2.4.3 Escalonamento em sistemas interativos

Examinaremos agora alguns algoritmos que podem ser usados em sistemas interativos. Eles são comuns em computadores pessoais, servidores e outros tipos de sistemas também.

Escalonamento por chaveamento circular

Um dos algoritmos mais antigos, simples, justos e amplamente usados é o **circular** (*round-robin*). A cada processo é designado um intervalo, chamado de seu **quantum**, durante o qual ele é deixado executar. Se o processo ainda está executando ao fim do quantum, a CPU sofrerá uma preempção e receberá outro processo. Se o processo foi bloqueado ou terminado antes de o quantum ter decorrido, o chaveamento de CPU será feito quando o processo bloquear, é claro. O escalonamento circular é fácil de implementar. Tudo o que o escalonador precisa fazer é manter uma lista de processos executáveis, como mostrado na Figura 2.42(a). Quando o processo usa todo o seu quantum, ele é colocado no fim da lista, como mostrado na Figura 2.42(b).

A única questão realmente interessante em relação ao escalonamento circular é o comprimento do quantum. Chavear de um processo para o outro exige certo

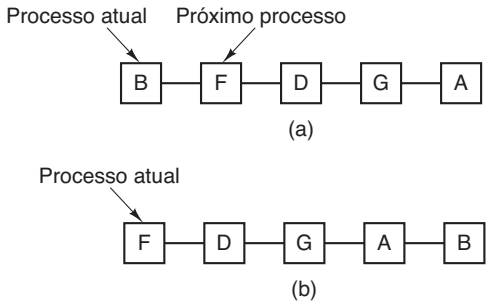


FIGURA 2.42 Escalonamento circular. (a) A lista de processos executáveis. (b) A lista de processos executáveis após *B* usar todo seu quantum.

montante de tempo para fazer toda a administração — salvando e carregando registradores e mapas de memória, atualizando várias tabelas e listas, carregando e descarregando memória cache, e assim por diante. Suponha que esse **chaveamento de processo** ou **chaveamento de contexto**, como é chamado às vezes, leva 1 ms, incluindo o chaveamento dos mapas de memória, carregar e descarregar o cache etc. Também suponha que o quantum é estabelecido em 4 ms. Com esses parâmetros, após realizar 4 ms de trabalho útil, a CPU terá de gastar (isto é, desperdiçar) 1 ms no chaveamento de processo. Desse modo, 20% do tempo da CPU será jogado fora em overhead administrativo. Claramente, isso é demais.

Para melhorar a eficiência da CPU, poderíamos configurar o quantum para, digamos, 100 ms. Agora o tempo desperdiçado é de apenas 1%. Mas considere o que acontece em um sistema de servidores se 50 solicitações entram em um intervalo muito curto e com exigências de CPU com grande variação. Cinquenta processos serão colocados na lista de processos executáveis. Se a CPU estiver ociosa, o primeiro começará imediatamente, o segundo não poderá começar até 100 ms mais tarde e assim por diante. O último azarado talvez tenha de esperar 5 s antes de ter uma chance, presumindo que todos os outros usem todo o seu quantum. A maioria dos usuários achará demorada uma resposta de 5 s para um comando curto. Essa situação seria especialmente ruim se algumas das solicitações próximas do fim da fila exigissem apenas alguns milissegundos de tempo da CPU. Com um quantum curto, eles teriam recebido um serviço melhor.

Outro fator é que se o quantum for configurado por um tempo mais longo que o surto de CPU médio, a preempção não acontecerá com muita frequência. Em vez disso, a maioria dos processos desempenhará uma operação de bloqueio antes de o quantum acabar, provocando um chaveamento de processo. Eliminar a preempção melhora o desempenho, porque os chaveamentos de processo então acontecem apenas quando são logicamente necessários, isto é, quando um processo é bloqueado e não pode continuar.

A seguinte conclusão pode ser formulada: estabelecer o quantum curto demais provoca muitos chaveamentos de processos e reduz a eficiência da CPU, mas estabelecê-lo longo demais pode provocar uma resposta ruim a solicitações interativas curtas. Um quantum em torno de 20-50 ms é muitas vezes bastante razoável.

Escalonamento por prioridades

O escalonamento circular pressupõe implicitamente que todos os processos são de igual importância. Muitas vezes, as pessoas que são proprietárias e operam computadores multiusuário têm ideias bem diferentes sobre o assunto. Em uma universidade, por exemplo, uma ordem hierárquica começaria pelo reitor, os chefes de departamento em seguida, então os professores, secretários, zeladores e, por fim, os estudantes. A necessidade de levar em consideração fatores externos leva ao **escalonamento por prioridades**. A ideia básica é direta: a cada processo é designada uma prioridade, e o processo executável com a prioridade mais alta é autorizado a executar.

Mesmo em um PC com um único proprietário, pode haver múltiplos processos, alguns dos quais são mais importantes do que os outros. Por exemplo, a um processo daemon enviando mensagens de correio eletrônico no segundo plano deve ser atribuída uma prioridade mais baixa do que a um processo exibindo um filme de vídeo na tela em tempo real.

Para evitar que processos de prioridade mais alta executem indefinidamente, o escalonador talvez diminua a prioridade do processo que está sendo executado em cada tique do relógio (isto é, em cada interrupção do relógio). Se essa ação faz que a prioridade caia abaixo daquela do próximo processo com a prioridade mais alta, ocorre um chaveamento de processo. Como alternativa, pode ser designado a cada processo um quantum de tempo máximo no qual ele é autorizado a executar. Quando esse quantum for esgotado, o processo seguinte na escala de prioridade recebe uma chance de ser executado.

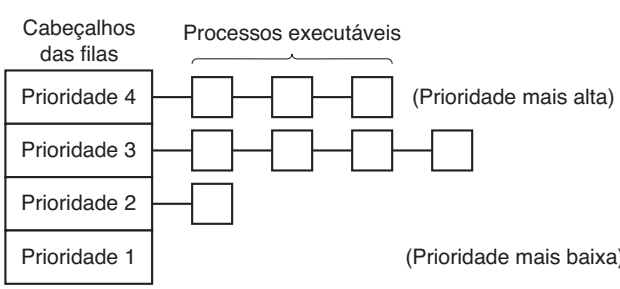
Prioridades podem ser designadas a processos estaticamente ou dinamicamente. Em um computador militar, processos iniciados por generais podem começar com uma prioridade 100, processos iniciados por coronéis a 90, majores a 80, capitães a 70, tenentes a 60 e assim por diante. Como alternativa, em uma central de computação comercial, tarefas de alta prioridade podem custar US$

100 uma hora, prioridade média US$ 75 e baixa prioridade US$ 50. O sistema UNIX tem um comando, *nice*, que permite que um usuário reduza voluntariamente a prioridade do seu processo, a fim de ser legal com os outros usuários, mas ninguém nunca o utiliza.

Prioridades também podem ser designadas dinamicamente pelo sistema para alcançar determinadas metas. Por exemplo, alguns processos são altamente limitados pela E/S e passam a maior parte do tempo esperando para a E/S ser concluída. Sempre que um processo assim quer a CPU, ele deve recebê-la imediatamente, para deixá-lo iniciar sua próxima solicitação de E/S, que pode então proceder em paralelo com outro processo que estiver de fato computando. Fazer que o processo limitado pela E/S espere muito tempo pela CPU significará apenas tê-lo ocupando a memória por um tempo desnecessariamente longo. Um algoritmo simples para proporcionar um bom serviço para processos limitados pela E/S é configurar a prioridade para $1/f$, onde f é a fração do último quantum que o processo usou. Um processo que usou apenas 1 ms do seu quantum de 50 ms receberia a prioridade 50, enquanto um que usasse 25 ms antes de bloquear receberia a prioridade 2, e um que usasse o quantum inteiro receberia a prioridade 1.

Muitas vezes é conveniente agrupar processos em classes de prioridade e usar o escalonamento de prioridades entre as classes, mas escalonamento circular dentro de cada classe. A Figura 2.43 mostra um sistema com quatro classes de prioridade. O algoritmo de escalonamento funciona do seguinte modo: desde que existam processos executáveis na classe de prioridade 4, apenas execute cada um por um quantum, estilo circular, e jamais se importe com classes de prioridade mais baixa. Se a classe de prioridade 4 estiver vazia, então execute os processos de classe 3 de maneira circular. Se ambas as classes — 4 e 3 — estiverem vazias, então execute a classe 2 de maneira circular e assim por diante. Se as prioridades não forem ajustadas ocasionalmente, classes de prioridade mais baixa podem todas morrer famintas.

FIGURA 2.43 Um algoritmo de escalonamento com quatro classes de prioridade.



Múltiplas filas

Um dos primeiros escalonadores de prioridade foi em CTSS, o sistema compatível de tempo compartilhado do MIT que operava no IBM 7094 (CORBATÓ et al., 1962). O CTSS tinha o problema que o chaveamento de processo era lento, pois o 7094 conseguia armazenar apenas um processo na memória. Cada chaveamento significava trocar o processo atual para o disco e ler em um novo a partir do disco. Os projetistas do CTSS logo perceberam que era mais eficiente dar aos processos limitados pela CPU um grande quantum de vez em quando, em vez de dar a eles pequenos quanta frequentemente (para reduzir as operações de troca). Por outro lado, dar a todos os processos um grande quantum significaria um tempo de resposta ruim, como já vimos. A solução foi estabelecer classes de prioridade. Processos na classe mais alta seriam executados por dois quanta. Processos na classe seguinte seriam executados por quatro quanta etc. Sempre que um processo consumia todos os quanta alocados para ele, era movido para uma classe inferior.

Como exemplo, considere um processo que precisasse computar continuamente por 100 quanta. De início ele receberia um quantum, então seria trocado. Da vez seguinte, ele receberia dois quanta antes de ser trocado. Em sucessivas execuções ele receberia 4, 8, 16, 32 e 64 quanta, embora ele tivesse usado apenas 37 dos 64 quanta finais para completar o trabalho. Apenas 7 trocas seriam necessárias (incluindo a carga inicial) em vez de 100 com um algoritmo circular puro. Além disso, à medida que o processo se aprofundasse nas filas de prioridade, ele seria usado de maneira cada vez menos frequente, poupando a CPU para processos interativos curtos.

A política a seguir foi adotada a fim de evitar punir para sempre um processo que precisasse ser executado por um longo tempo quando fosse iniciado pela primeira vez, mas se tornasse interativo mais tarde. Sempre que a tecla *Enter* era digitada em um terminal, o processo pertencente àquele terminal era movido para a classe de prioridade mais alta, pressupondo que ele estava prestes a tornar-se interativo. Um belo dia, algum usuário com um processo pesadamente limitado pela CPU descobriu que apenas sentar em um terminal e digitar a tecla *Enter* ao acaso de tempos em tempos ajudava e muito seu tempo de resposta. Moral da história: acertar na prática é muito mais difícil que acertar na regra.

Processo mais curto em seguida

Como a *tarefa mais curta primeiro* sempre produz o tempo de resposta médio mínimo para sistemas em

lote, seria bom se ela pudesse ser usada para processos interativos também. Até certo ponto, ela pode ser. Processos interativos geralmente seguem o padrão de esperar pelo comando, executar o comando, esperar pelo comando, executar o comando etc. Se considerarmos a execução de cada comando uma "tarefa" em separado, então podemos minimizar o tempo de resposta geral executando a tarefa mais curta primeiro. O problema é descobrir qual dos processos atualmente executáveis é o mais curto.

Uma abordagem é fazer estimativas baseadas no comportamento passado e executar o processo com o tempo de execução estimado mais curto. Suponha que o tempo estimado por comando para alguns processos é T_0. Agora suponha que a execução seguinte é mensurada como sendo T_1. Poderíamos atualizar nossa estimativa tomando a soma ponderada desses dois números, isto é, $aT_0 + (1 - a)T_1$. Pela escolha de a podemos decidir que o processo de estimativa esqueça as execuções anteriores rapidamente, ou as lembre por um longo tempo. Com $a = 1/2$, temos estimativas sucessivas de

$$T_0,\ T_0/2 + T_1/2,\ T_0/4 + T_1/4 + T_2/2,\ T_0/8 + T_1/8 + T_2/4 + T_3/2$$

Após três novas execuções, o peso de T_0 na nova estimativa caiu para 1/8.

A técnica de estimar o valor seguinte em uma série tomando a média ponderada do valor mensurado atual e a estimativa anterior é às vezes chamada de **envelhecimento** (**aging**). Ela é aplicável a muitas situações onde uma previsão precisa ser feita baseada nos valores anteriores. O envelhecimento é especialmente fácil de implementar quando $a = 1/2$. Tudo o que é preciso fazer é adicionar o novo valor à estimativa atual e dividir a soma por 2 (deslocando-a 1 bit para a direita).

Escalonamento garantido

Uma abordagem completamente diferente para o escalonamento é fazer promessas reais para os usuários a respeito do desempenho e então cumpri-las. Uma promessa realista de se fazer e fácil de cumprir é a seguinte: se n usuários estão conectados enquanto você está trabalhando, você receberá em torno de $1/n$ da potência da CPU. De modo similar, em um sistema de usuário único com n processos sendo executados, todos os fatores permanecendo os mesmos, cada um deve receber $1/n$ dos ciclos da CPU. Isso parece bastante justo.

Para cumprir essa promessa, o sistema deve controlar quanta CPU cada processo teve desde sua criação. Ele então calcula o montante de CPU a que cada um tem direito, especificamente, o tempo desde a criação dividido por n. Tendo em vista que o montante de tempo da CPU que cada processo realmente teve também é conhecido, calcular o índice de tempo de CPU real consumido com o tempo de CPU ao qual ele tem direito é algo bastante direto. Um índice de 0,5 significa que o processo teve apenas metade do que deveria, e um índice de 2,0 significa que teve duas vezes o montante de tempo ao qual ele tinha direito. O algoritmo então executará o processo com o índice mais baixo até que seu índice aumente e se aproxime do de seu competidor. Então este é escolhido para executar em seguida.

Escalonamento por loteria

Embora realizar promessas para os usuários e cumpri-las seja uma bela ideia, ela é difícil de implementar. No entanto, outro algoritmo pode ser usado para gerar resultados similarmente previsíveis com uma implementação muito mais simples. Ele é chamado de **escalonamento por loteria** (WALDSPURGER e WEIHL, 1994).

A ideia básica é dar bilhetes de loteria aos processos para vários recursos do sistema, como o tempo da CPU. Sempre que uma decisão de escalonamento tiver de ser feita, um bilhete de loteria será escolhido ao acaso, e o processo com o bilhete fica com o recurso. Quando aplicado ao escalonamento de CPU, o sistema pode realizar um sorteio 50 vezes por segundo, com cada vencedor recebendo 20 ms de tempo da CPU como prêmio.

Parafraseando George Orwell: "Todos os processos são iguais, mas alguns processos são mais iguais". Processos mais importantes podem receber bilhetes extras, para aumentar a chance de vencer. Se há 100 bilhetes emitidos e um processo tem 20 deles, ele terá uma chance de 20% de vencer cada sorteio. A longo prazo, ele terá acesso a cerca de 20% da CPU. Em comparação com o escalonador de prioridade, em que é muito difícil de afirmar o que realmente significa ter uma prioridade de 40, aqui a regra é clara: um processo que tenha uma fração f dos bilhetes terá aproximadamente uma fração f do recurso em questão.

O escalonamento de loteria tem várias propriedades interessantes. Por exemplo, se um novo processo aparece e ele ganha alguns bilhetes, no sorteio seguinte ele teria uma chance de vencer na proporção do número de bilhetes que tem em mãos. Em outras palavras, o escalonamento de loteria é altamente responsivo.

Processos cooperativos podem trocar bilhetes se assim quiserem. Por exemplo, quando um processo

cliente envia uma mensagem para um processo servidor e então bloqueia, ele pode dar todos os seus bilhetes para o servidor a fim de aumentar a chance de que o servidor seja executado em seguida. Quando o servidor tiver concluído, ele devolve os bilhetes de maneira que o cliente possa executar novamente. Na realidade, na ausência de clientes, os servidores não precisam de bilhete algum.

O escalonamento de loteria pode ser usado para solucionar problemas difíceis de lidar com outros métodos. Um exemplo é um servidor de vídeo no qual vários processos estão alimentando fluxos de vídeo para seus clientes, mas em diferentes taxas de apresentação dos quadros. Suponha que os processos precisem de quadros a 10, 20 e 25 quadros/s. Ao alocar para esses processos 10, 20 e 25 bilhetes, nessa ordem, eles automaticamente dividirão a CPU em mais ou menos a proporção correta, isto é, 10 : 20 : 25.

Escalonamento por fração justa

Até agora presumimos que cada processo é escalonado por si próprio, sem levar em consideração quem é o 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 receberá 90% da CPU e o usuário 2 apenas 10% dela.

Para evitar essa situação, alguns sistemas levam em conta qual usuário é dono de um processo antes de escaloná-lo. Nesse modelo, a cada usuário é alocada alguma fração da CPU e o escalonador escolhe processos de uma maneira que garanta essa fração. Desse modo, se dois usuários têm cada um 50% da CPU prometidos, cada um receberá isso, não importa quantos processos eles tenham em existência.

Como exemplo, considere um sistema com dois usuários, cada um tendo a promessa de 50% da CPU. O usuário 1 tem quatro processos, *A, B, C* e *D*, e o usuário 2 tem apenas um processo, *E*. Se o escalonamento circular for usado, uma sequência de escalonamento possível que atende a todas as restrições é a seguinte:

A E B E C E D E A E B E C E D E ...

Por outro lado, se o usuário 1 tem direito a duas vezes o tempo de CPU que o usuário 2, talvez tenhamos

A B E C D E A B E C D E ...

Existem numerosas outras possibilidades, é claro, e elas podem ser exploradas, dependendo de qual seja a noção de justiça.

2.4.4 Escalonamento em sistemas de tempo real

Um sistema de **tempo real** é aquele em que o tempo tem um papel essencial. Tipicamente, um ou mais dispositivos físicos externos ao computador geram estímulos, e o computador tem de reagir em conformidade dentro de um montante de tempo fixo. Por exemplo, o computador em um CD player recebe os bits à medida que eles saem do drive e deve convertê-los em música dentro de um intervalo muito estrito. Se o cálculo levar tempo demais, a música soará estranha. Outros sistemas de tempo real estão monitorando pacientes em uma UTI, o piloto automático em um avião e o controle de robôs em uma fábrica automatizada. Em todos esses casos, ter a resposta certa, mas tê-la tarde demais é muitas vezes tão ruim quanto não tê-la.

Sistemas em tempo real são geralmente categorizados como **tempo real crítico**, significando que há prazos absolutos que devem ser cumpridos — para valer! — e **tempo real não crítico**, significando que descumprir um prazo ocasional é indesejável, mas mesmo assim tolerável. Em ambos os casos, o comportamento em tempo real é conseguido dividindo o programa em uma série de processos, cada um dos quais é previsível e conhecido antecipadamente. Esses processos geralmente têm vida curta e podem ser concluídos em bem menos de um segundo. Quando um evento externo é detectado, cabe ao escalonador programar os processos de uma maneira que todos os prazos sejam atendidos.

Os eventos a que um sistema de tempo real talvez tenha de responder podem ser categorizados ainda como **periódicos** (significando que eles ocorrem em intervalos regulares) ou **aperiódicos** (significando que eles ocorrem de maneira imprevisível). Um sistema pode ter de responder a múltiplos fluxos de eventos periódicos. Dependendo de quanto tempo cada evento exige para o processamento, tratar de todos talvez não seja nem possível. Por exemplo, se há m eventos periódicos e o evento i ocorre com o período P_i e exige C_i segundos de tempo da CPU para lidar com cada evento, então a carga só pode ser tratada se

$$\sum_{i=1}^{m} \frac{C_i}{P_i} \leq 1$$

Diz-se de um sistema de tempo real que atende a esse critério que ele é **escalonável**. Isso significa que ele realmente pode ser implementado. Um processo que fracassa em atender esse teste não pode ser escalonado, pois o montante total de tempo de CPU que os processos querem coletivamente é maior do que a CPU pode proporcionar.

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 exigem 50, 30 e 100 ms de tempo da CPU, respectivamente, o sistema é escalonável, pois 0,5 + 0,15 + 0,2 < 1. Se um quarto evento com um período de 1 segundo é acrescentado, o sistema permanecerá escalonável desde que esse evento não precise de mais de 150 ms de tempo da CPU por evento. Implícito nesse cálculo está o pressuposto de que o overhead de chaveamento de contexto é tão pequeno que pode ser ignorado.

Algoritmos de escalonamento de tempo real podem ser estáticos ou dinâmicos. Os primeiros tomam suas decisões de escalonamento antes de o sistema começar a ser executado. Os últimos tomam suas decisões no tempo de execução, após ela ter começado. O escalonamento estático funciona apenas quando há uma informação perfeita disponível antecipadamente sobre o trabalho a ser feito, e os prazos que precisam ser cumpridos. Algoritmos de escalonamento dinâmico não têm essas restrições.

2.4.5 Política *versus* mecanismo

Até o momento, presumimos tacitamente que todos os processos no sistema pertencem a usuários diferentes e estão, portanto, competindo pela CPU. Embora isso seja muitas vezes verdadeiro, às vezes acontece de um processo ter muitos filhos executando sob o seu controle. Por exemplo, um processo de sistema de gerenciamento de banco de dados pode ter muitos filhos. Cada filho pode estar funcionando em uma solicitação diferente, ou cada um pode ter alguma função específica para realizar (análise sintática de consultas, acesso ao disco etc.). É inteiramente possível que o principal processo tenha uma ideia excelente de qual dos filhos é o mais importante (ou tenha tempo crítico) e qual é o menos importante. Infelizmente, nenhum dos escalonadores discutidos aceita qualquer entrada dos processos do usuário sobre decisões de escalonamento. Como resultado, o escalonador raramente faz a melhor escolha.

A solução desse problema é separar o **mecanismo de escalonamento** da **política de escalonamento**, um princípio há muito estabelecido (LEVIN et al., 1975). O que isso significa é que o algoritmo de escalonamento é parametrizado de alguma maneira, mas os parâmetros podem estar preenchidos pelos processos dos usuários. Vamos considerar o exemplo do banco de dados novamente. Suponha que o núcleo utilize um algoritmo de escalonamento de prioridades, mas fornece uma chamada de sistemas pela qual um processo pode estabelecer (e mudar) as prioridades dos seus filhos. Dessa maneira, o pai pode controlar como seus filhos são escalonados, mesmo que ele mesmo não realize o escalonamento. Aqui o mecanismo está no núcleo, mas a política é estabelecida por um processo do usuário. A separação do mecanismo de política é uma ideia fundamental.

2.4.6 Escalonamento de threads

Quando vários processos têm cada um múltiplos threads, temos dois níveis de paralelismo presentes: processos e threads. Escalonar nesses sistemas difere substancialmente, dependendo se os threads de usuário ou os threads de núcleo (ou ambos) recebem suporte.

Vamos considerar primeiro os threads de usuário. Tendo em vista que o núcleo não tem ciência da existência dos threads, ele opera como sempre fez, escolhendo um processo, digamos, *A*, e dando a *A* controle de seu quantum. O escalonador de thread dentro de *A* decide qual thread executar, digamos, *A1*. Dado que não há interrupções de relógio para multiprogramar threads, esse thread pode continuar a ser executado por quanto tempo quiser. Se ele utilizar todo o quantum do processo, o núcleo selecionará outro processo para executar.

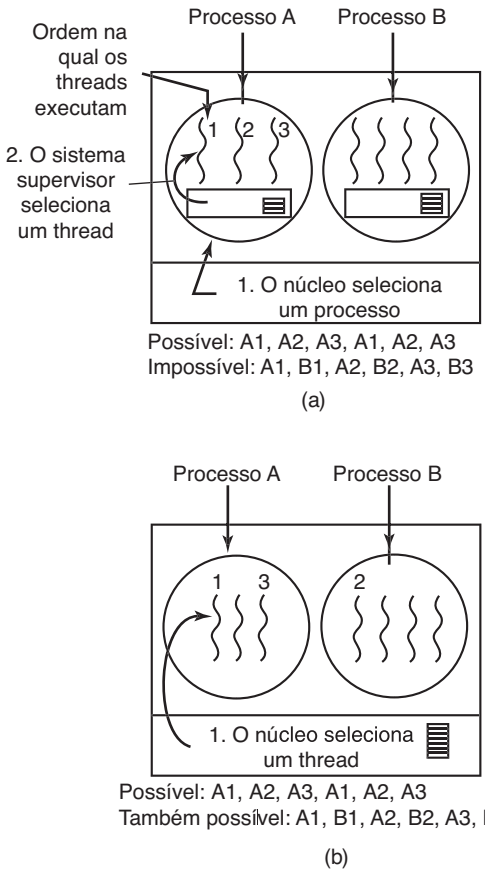
Quando o processo *A,* por fim, executar novamente, o thread *A1* retomará a execução. Ele continuará a consumir todo o tempo de *A* até que termine. No entanto, seu comportamento antissocial não afetará outros processos. Eles receberão o que quer que o escalonador considere sua fração apropriada, não importa o que estiver acontecendo dentro do processo *A*.

Agora considere o caso em que os threads de *A* tenham relativamente pouco trabalho para fazer por surto de CPU, por exemplo, 5 ms de trabalho dentro de um quantum de 50 ms. Em consequência, cada um executa por um tempo, então cede a CPU de volta para o escalonador de threads. Isso pode levar à sequência *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, antes que o núcleo chaveie para o processo *B*. Essa situação está mostrada na Figura 2.44(a).

O algoritmo de escalonamento usado pelo sistema de tempo de execução pode ser qualquer um dos descritos anteriormente. Na prática, o escalonamento circular e o de prioridade são os mais comuns. A única restrição é a ausência de um relógio para interromper um thread que esteja sendo executado há tempo demais. Visto que os threads cooperam, isso normalmente não é um problema.

Agora considere a situação com threads de núcleo. Aqui o núcleo escolhe um thread em particular para executar. Ele não precisa levar em conta a qual processo o thread pertence, porém ele pode, se assim o desejar. O thread recebe um quantum e é suspenso compulsoriamente se o exceder. Com um quantum de 50 ms, mas threads que são

FIGURA 2.44 (a) Escalonamento possível de threads de usuário com 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 que (a).



bloqueados após 5 ms, a ordem do thread por algum período de 30 ms pode ser *A1*, *B1*, *A2*, *B2*, *A3*, *B3,* algo que não é possível com esses parâmetros e threads de usuário. Essa situação está parcialmente descrita na Figura 2.44(b).

Uma diferença importante entre threads de usuário e de núcleo é o desempenho. Realizar um chaveamento de thread com threads de usuário exige um punhado de instruções de máquina. Com threads de núcleo é necessário um chaveamento de contexto completo, mudar o mapa de memória e invalidar o cache, o que representa uma demora de magnitude várias ordens maior. Por outro lado, com threads de núcleo, ter um bloqueio de thread na E/S não suspende todo o processo como ocorre com threads de usuário.

Visto que o núcleo sabe que chavear de um thread no processo *A* para um thread no processo *B* é mais caro do que executar um segundo thread no processo *A* (por ter de mudar o mapa de memória e invalidar a memória de cache), ele pode levar essa informação em conta quando toma uma decisão. Por exemplo, dados dois threads que de outra forma são igualmente importantes, com um deles pertencendo ao mesmo processo que um thread que foi bloqueado há pouco e outro pertencendo a um processo diferente, a preferência poderia ser dada ao primeiro.

Outro fator importante é que os threads de usuário podem empregar um escalonador de thread específico de uma aplicação. Considere, por exemplo, o servidor na web da Figura 2.8. Suponha que um thread operário foi bloqueado há pouco e o thread despachante e dois threads operários estão prontos. Quem deve ser executado em seguida? O sistema de tempo de execução, sabendo o que todos os threads fazem, pode facilmente escolher o despachante para ser executado em seguida, de maneira que ele possa colocar outro operário para executar. Essa estratégia maximiza o montante de paralelismo em um ambiente onde operários frequentemente são bloqueados pela E/S de disco. Com threads de núcleo, o núcleo jamais saberia o que cada thread fez (embora a eles pudessem ser atribuídas prioridades diferentes). No geral, entretanto, escalonadores de threads específicos de aplicações são capazes de ajustar uma aplicação melhor do que o núcleo.

2.5 Problemas clássicos de IPC

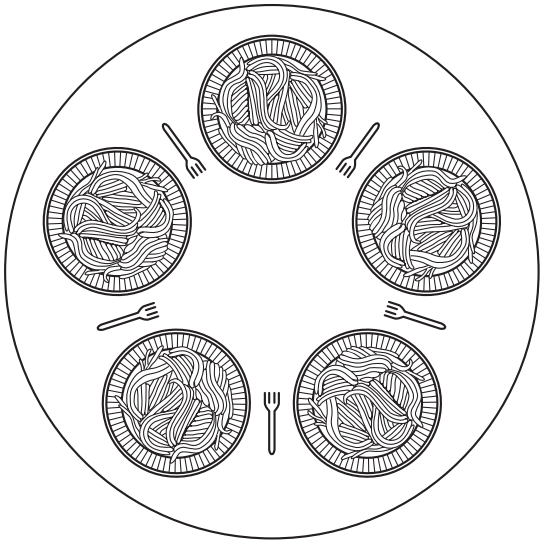
A literatura de sistemas operacionais está cheia de problemas interessantes que foram amplamente discutidos e analisados usando variados métodos de sincronização. Nas seções a seguir, examinaremos três dos problemas mais conhecidos.

2.5.1 O problema do jantar dos filósofos

Em 1965, Dijkstra formulou e então solucionou um problema de sincronização que ele chamou de **problema do jantar dos filósofos**. Desde então, todos os que inventaram mais uma primitiva de sincronização sentiram-se obrigados a demonstrar quão maravilhosa é a nova primitiva exibindo quão elegantemente ela soluciona o problema do jantar dos filósofos. O problema pode ser colocado de maneira bastante simples, como a seguir: cinco filósofos estão sentados em torno de uma mesa circular. Cada filósofo tem um prato de espaguete. O espaguete é tão escorregadio que um filósofo precisa de dois garfos para comê-lo. Entre cada par de pratos há um garfo. O desenho da mesa está ilustrado na Figura 2.45.

A vida de um filósofo consiste em alternar períodos de alimentação e pensamento. (Trata-se de um tipo de abstração, mesmo para filósofos, mas as outras atividades são irrelevantes aqui.) Quando um filósofo fica suficientemente faminto, ele tenta pegar seus garfos à esquerda

FIGURA 2.45 Hora do almoço no departamento de filosofia.



e à direita, um de cada vez, não importa a ordem. Se for bem-sucedido em pegar dois garfos, ele come por um tempo, então larga os garfos e continua a pensar. A questão fundamental é: você consegue escrever um programa para cada filósofo que faça o que deve fazer e jamais fique travado? (Já foi apontado que a necessidade de dois garfos é de certa maneira artificial; talvez devamos trocar de um prato italiano para um chinês, substituindo o espaguete por arroz e os garfos por pauzinhos.)

A Figura 2.46 mostra a solução óbvia. O procedimento *take_fork* espera até o garfo específico estar disponível e então o pega. Infelizmente, a solução óbvia está errada. Suponha que todos os cinco filósofos peguem seus garfos esquerdos simultaneamente. Nenhum será capaz de pegar seus garfos direitos, e haverá um impasse.

Poderíamos facilmente modificar o programa de maneira que após pegar o garfo esquerdo, o programa confere para ver se o garfo direito está disponível. Se não estiver, o filósofo coloca de volta o esquerdo sobre a mesa, espera por um tempo, e repete todo o processo. Essa proposta também fracassa, embora por uma razão diferente. Com um pouco de azar, todos os filósofos poderiam começar o algoritmo simultaneamente, pegando seus garfos esquerdos, vendo que seus garfos direitos não estavam disponíveis, colocando seus garfos esquerdos de volta sobre a mesa, esperando, pegando seus garfos esquerdos de novo ao mesmo tempo, assim por diante, para sempre. Uma situação como essa, na qual todos os programas continuam a executar indefinidamente, mas fracassam em realizar qualquer progresso, é chamada de **inanição** (*starvation*). (Ela é chamada de inanição mesmo quando o problema não ocorre em um restaurante italiano ou chinês.)

Agora você pode pensar que se os filósofos simplesmente esperassem um tempo aleatório em vez de ao mesmo tempo fracassarem em conseguir o garfo direito, a chance de tudo continuar em um impasse mesmo por uma hora é muito pequena. Essa observação é verdadeira, e em quase todas as aplicações tentar mais tarde não é um problema. Por exemplo, na popular rede de área local Ethernet, se dois computadores enviam um pacote ao mesmo tempo, cada um espera um tempo aleatório e tenta de novo; na prática essa solução funciona bem. No entanto, em algumas aplicações você preferiria uma solução que sempre funcionasse e não pudesse fracassar por uma série improvável de números aleatórios. Pense no controle de segurança em uma usina de energia nuclear.

Uma melhoria para a Figura 2.46 que não apresenta impasse nem inanição é proteger os cinco comandos seguindo a chamada *think* com um semáforo binário. Antes de começar a pegar garfos, um filósofo realizaria

FIGURA 2.46 Uma não solução para o problema do jantar dos filósofos.

```
#define N 5                                     /* numero de filosofos */

void philosopher(int i)                         /* i: numero do filosofo, de 0 a 4 */
{
    while (TRUE) {
        think( );                               /* o filosofo esta pensando */
        take_fork(i);                           /* pega o garfo esquerdo */
        take_fork((i+1) % N);                   /* pega o garfo direito; % e o operador modulo */
        eat( );                                 /* hummm, espaguete */
        put_fork(i);                            /* devolve o garfo esquerdo a mesa */
        put_fork((i+1) % N);                    /* devolve o garfo direito a mesa */
    }
}
```

um down em *mutex*. Após substituir os garfos, ele realizaria um up em *mutex*. Do ponto de vista teórico, essa solução é adequada. Do ponto de vista prático, ela tem um erro de desempenho: apenas um filósofo pode estar comendo a qualquer dado instante. Com cinco garfos disponíveis, deveríamos ser capazes de ter dois filósofos comendo ao mesmo tempo.

A solução apresentada na Figura 2.47 é livre de impasse e permite o máximo paralelismo para um número arbitrário de filósofos. Ela usa um arranjo, *estado*, para controlar se um filósofo está comendo, pensando, ou com fome (tentando conseguir garfos). Um filósofo pode passar para o estado comendo apenas se nenhum de seus vizinhos estiver comendo. Os vizinhos do

FIGURA 2-47 Uma solução para o problema do jantar dos filósofos.

```
#define N               5               /* numero de filosofos */
#define LEFT            (i+N–1)%N       /* numero do vizinho a esquerda de i */
#define RIGHT           (i+1)%N         /* numero do vizinho a direita de i */
#define THINKING        0               /* o filosofo esta pensando */
#define HUNGRY          1               /* o filosofo esta tentando pegar garfos */
#define EATING          2               /* o filosofo esta comendo */

typedef int semaphore;                  /* semaforos sao um tipo especial de int */
int state[N];                           /* arranjo para controlar o estado de cada um */
semaphore mutex = 1;                    /* exclusao mutua para as regioes criticas */
semaphore s[N];                         /* um semaforo por filosofo */

void philosopher(int i)                 /* i: o numero do filosofo, de 0 a N–1 */
{
    while (TRUE) {                      /* repete para sempre */
        think( );                       /* o filosofo esta pensando */
        take_forks(i);                  /* pega dois garfos ou bloqueia */
        eat( );                         /* hummm, espaguete! */
        put_forks(i);                   /* devolve os dois garfos a mesa */
    }
}

void take_forks(int i)                  /* i: o numero do filosofo, de 0 a N–1 */
{
    down(&mutex);                       /* entra na regiao critica */
    state[i] = HUNGRY;                  /* registra que o filosofo esta faminto */
    test(i);                            /* tenta pegar dois garfos */
    up(&mutex);                         /* sai da regiao critica */
    down(&s[i]);                        /* bloqueia se os garfos nao foram pegos */
}

void put_forks(i)                       /* i: o numero do filosofo, de 0 a N–1 */
{
    down(&mutex);                       /* entra na regiao critica */
    state[i] = THINKING;                /* o filosofo acabou de comer */
    test(LEFT);                        /* ve se o vizinho da esquerda pode comer agora */
    test(RIGHT);                        /* ve se o vizinho da direita pode comer agora */
    up(&mutex);                         /* sai da regiao critica */
}

void test(i)/* i: o numero do filosofo, de 0 a N–1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

filósofo *i* são definidos pelas macros *LEFT* e *RIGHT*. Em outras palavras, se *i* é 2, *LEFT* é 1 e *RIGHT* é 3.

O programa usa um conjunto de semáforos, um por filósofo, portanto os filósofos com fome 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 estão competindo pelo acesso exclusivo a um número limitado de recursos, como em dispositivos de E/S. Outro problema famoso é o problema dos leitores e escritores (COURTOIS et al., 1971), que modela o acesso a um banco de dados. Imagine, por exemplo, um sistema de reservas de uma companhia aérea, com muitos processos competindo entre si desejando ler e escrever. É aceitável ter múltiplos processos lendo o banco de dados ao mesmo tempo, mas se um processo está atualizando (escrevendo) o banco de dados, nenhum outro pode ter acesso, nem mesmo os leitores. A questão é: como programar leitores e escritores? Uma solução é mostrada na Figura 2.48.

Nessa solução, para conseguir acesso ao banco de dados, o primeiro leitor realiza um down no semáforo *db*. Leitores subsequentes apenas incrementam um contador, *rc*. À medida que os leitores saem, eles decrementam o contador, e o último a deixar realiza um up no semáforo, permitindo que um escritor bloqueado, se houver, entre.

A solução apresentada aqui contém implicitamente uma decisão sutil que vale observar. Suponha que enquanto um leitor está usando o banco de dados, aparece outro leitor. Visto que ter dois leitores ao mesmo tempo não é um problema, o segundo leitor é admitido. Leitores adicionais também podem ser admitidos se aparecerem.

FIGURA 2.48 Uma solução para o problema dos leitores e escritores.

```
typedef int semaphore;                  /* use sua imaginacao */
semaphore mutex = 1;                    /* controla o acesso a 'rc' */
semaphore db = 1;                       /* controla o acesso a base de dados */
int rc = 0;                             /* numero de processos lendo ou querendo ler */

void reader(void)
{
     while (TRUE) {                     /* repete para sempre */
          down(&mutex);                 /* obtem acesso exclusivo a 'rc' */
          rc = rc + 1;                  /* um leitor a mais agora */
          if (rc == 1) down(&db);       /* se este for o primeiro leitor ... */
          up(&mutex);                   /* libera o acesso exclusivo a 'rc' */
          read_data_base( );            /* acesso aos dados */
          down(&mutex);                 /* obtem acesso exclusivo a 'rc' */
          rc = rc – 1;                  /* um leitor a menos agora */
          if (rc == 0) up(&db);         /* se este for o ultimo leitor ... */
          up(&mutex);                   /* libera o acesso exclusivo a 'rc' */
          use_data_read( );             /* regiao nao critica */
     }
}


void writer(void)
{
     while (TRUE) {                     /* repete para sempre */
          think_up_data( );             /* regiao nao critica */
          down(&db);                    /* obtem acesso exclusivo */
          write_data_base( );           /* atualiza os dados */
          up(&db);                      /* libera o acesso exclusivo */
     }
}
```

Agora suponha que um escritor apareça. O escritor pode não ser admitido ao banco de dados, já que escritores precisam ter acesso exclusivo, então ele é suspenso. Depois, leitores adicionais aparecem. Enquanto pelo menos um leitor ainda estiver ativo, leitores subsequentes serão admitidos. Como consequência dessa estratégia, enquanto houver uma oferta uniforme de leitores, todos eles entrarão assim que chegarem. O escritor será mantido suspenso até que nenhum leitor esteja presente. Se um novo leitor aparecer, digamos, a cada 2 s, e cada leitor levar 5 s para realizar o seu trabalho, o escritor jamais entrará.

Para evitar essa situação, o programa poderia ser escrito de maneira ligeiramente diferente: quando um leitor chega e um escritor está esperando, o leitor é suspenso atrás do escritor em vez de ser admitido imediatamente. Dessa maneira, um escritor precisa esperar por leitores que estavam ativos quando ele chegou, mas não precisa esperar por leitores que chegaram depois dele. A desvantagem dessa solução é que ela alcança uma concorrência menor e assim tem um desempenho mais baixo. Courtois et al. (1971) apresentam uma solução que dá prioridade aos escritores. Para detalhes, consulte o artigo.

2.6 Pesquisas sobre processos e threads

No Capítulo 1, estudamos algumas das pesquisas atuais sobre a estrutura dos sistemas operacionais. Neste capítulo e nos subsequentes, estudaremos pesquisas mais específicas, começando com processos. Como ficará claro com o tempo, alguns assuntos são muito menos controversos do que outros. A maioria das pesquisas tende a ser sobre os tópicos novos, em vez daqueles que estão por aí há décadas.

O conceito de um processo é um exemplo de algo que já está muito bem estabelecido. Quase todo sistema tem alguma noção de um processo como um contêiner para agrupar recursos relacionados como um espaço de endereçamento, threads, arquivos abertos, permissões de proteção e assim por diante. Sistemas diferentes realizam o agrupamento de maneira ligeiramente diferente, mas trata-se apenas de diferenças de engenharia. A ideia básica não é mais muito controversa, e há pouca pesquisa nova sobre processos.

O conceito de threads é mais recente do que o de processos, mas ele, também, já foi bastante estudado. Ainda assim, o estudo ocasional sobre threads aparece de tempos em tempos, como o estudo a respeito de aglomeração de threads em multiprocessadores (TAM et al., 2007), ou sobre quão bem os sistemas operacionais modernos como o Linux lidam com muitos threads e muitos núcleos (BOYD-WICKIZER, 2010).

Uma área de pesquisa particularmente ativa lida com a gravação e a reprodução da execução de um processo (VIENNOT et al., 2013). A reprodução ajuda os desenvolvedores a procurar erros difíceis de serem encontrados e especialistas em segurança a investigar incidentes.

De modo similar, muita pesquisa na comunidade de sistemas operacionais concentra-se hoje em questões de segurança. Muitos incidentes demonstraram que os usuários precisam de uma proteção melhor contra agressores (e, ocasionalmente, contra si mesmos). Uma abordagem é controlar e restringir com cuidado os fluxos de informação em um sistema operacional (GIFFIN et al., 2012).

O escalonamento (tanto de uniprocessadores quanto de multiprocessadores) ainda é um tópico que mora no coração de alguns pesquisadores. Alguns tópicos sendo pesquisados incluem o escalonamento de dispositivos móveis em busca da eficiência energética (YUAN e NAHRSTEDT, 2006), escalonamento com tecnologia hyperthreading (BULPIN e PRATT, 2005) e escalonamento *bias-aware* (KOUFATY, 2010). Com cada vez mais computação em smartphones com restrições de energia, alguns pesquisadores propõem migrar o processo para um servidor mais potente na nuvem, quando isso for útil (GORDON et al, 2012). No entanto, poucos projetistas de sistemas andam preocupados com a falta de um algoritmo de escalonamento de threads decente, então esse tipo de pesquisa parece ser mais um interesse de pesquisadores do que uma demanda de projetistas. Como um todo, processos, threads e escalonamento, não são mais os tópicos quentes de pesquisa que já foram um dia. A pesquisa seguiu para tópicos como gerenciamento de energia, virtualização, nuvens e segurança.

2.7 Resumo

A fim de esconder os efeitos de interrupções, os sistemas operacionais fornecem um modelo conceitual que consiste de processos sequenciais executando em paralelo. 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 de um espaço de endereçamento comum.

Threads podem ser implementados no espaço do usuário ou no núcleo.

Processos podem comunicar-se uns com os outros usando primitivas de comunicação entre processos, por exemplo, semáforos, monitores ou mensagens. Essas primitivas são usadas para assegurar que jamais dois processos estejam em suas regiões críticas ao mesmo tempo, uma situação que leva ao caos. Um processo pode estar sendo executado, ser executável, ou bloqueado, e pode mudar de estado quando ele ou outro executar uma das primitivas de comunicação entre processos. A comunicação entre threads é similar.

Primitivas de comunicação entre processos podem ser usadas para solucionar problemas como o produtor-consumidor, o jantar dos filósofos e leitor-escritor. Mesmo com essas primitivas, é preciso cuidado para evitar erros e impasses.

Um número considerável de algoritmos de escalonamento foi estudado. Alguns deles são usados fundamentalmente por sistemas em lote, como o escalonamento da tarefa mais curta primeiro. Outros são comuns tanto nos sistema em lote quanto nos sistemas interativos. Esses algoritmos incluem escalonamento circular, por prioridade, de múltiplas filas, garantido, de loteria e por fração justa. Alguns sistemas fazem uma separação clara entre o mecanismo de escalonamento e a política de escalonamento, o que permite que os usuários tenham controle do algoritmo de escalonamento.

PROBLEMAS

1. Na Figura 2.2, são mostrados três estados de processos. Na teoria, com três estados, poderia haver seis transições, duas para cada. No entanto, apenas quatro transições são mostradas. Existe alguma circunstância na qual uma delas ou ambas as transições perdidas possam ocorrer?
2. Suponha que você fosse projetar uma arquitetura de computador avançada que realizasse chaveamento de processos em hardware, em vez de interrupções. De qual informação a CPU precisaria? Descreva como o processo de chaveamento por hardware poderia funcionar.
3. Em todos os computadores atuais, pelo menos parte dos tratadores de interrupções é escrita em linguagem de montagem. Por quê?
4. Quando uma interrupção ou uma chamada de sistema transfere controle para o sistema operacional, geralmente uma área da pilha do núcleo separada da pilha do processo interrompido é usada. Por quê?
5. Um sistema computacional tem espaço suficiente para conter cinco programas em sua memória principal. Esses programas estão ociosos esperando por E/S metade do tempo. Qual fração do tempo da CPU é desperdiçada?
6. Um computador tem 4 GB de RAM da qual o sistema operacional ocupa 512 MB. Os processos ocupam 256 MB cada (para simplificar) e têm as mesmas características. Se a meta é a utilização de 99% da CPU, qual é a espera de E/S máxima que pode ser tolerada?
7. Múltiplas tarefas podem ser executadas em paralelo e terminar mais rápido do que se forem executadas de modo sequencial. Suponha que duas tarefas, cada uma precisando de 20 minutos de tempo da CPU, iniciassem simultaneamente. Quanto tempo a última levará para completar se forem executadas sequencialmente? Quanto tempo se forem executadas em paralelo? Presuma uma espera de E/S de 50%.
8. Considere um sistema multiprogramado com grau de 6 (isto é, seis programas na memória ao mesmo tempo). Presuma que cada processo passe 40% do seu tempo esperando pelo dispositivo de E/S. Qual será a utilização da CPU?
9. Presuma que você esteja tentando baixar um arquivo grande de 2 GB da internet. O arquivo está disponível a partir de um conjunto de servidores espelho, cada um deles capaz de fornecer um subconjunto dos bytes do arquivo; presuma que uma determinada solicitação especifique os bytes de início e fim do arquivo. Explique como você poderia usar os threads para melhorar o tempo de download.
10. No texto foi afirmado que o modelo da Figura 2.11(a) não era adequado a um servidor de arquivos usando um cache na memória. Por que não? Será que cada processo poderia ter seu próprio cache?
11. Se um processo multithread bifurca, um problema ocorre se o filho recebe cópias de todos os threads do pai. Suponha que um dos threads originais estivesse esperando por entradas do teclado. Agora dois threads estão esperando por entradas do teclado, um em cada processo. Esse problema ocorre alguma vez em processos de thread único?
12. Um servidor web multithread é mostrado na Figura 2.8. Se a única maneira de ler de um arquivo é a chamada de sistema `read` com bloqueio normal, você acredita que threads de usuário ou threads de núcleo estão sendo usados para o servidor web? Por quê?
13. No texto, descrevemos um servidor web multithread, mostrando por que ele é melhor do que um servidor de thread único e um servidor de máquina de estado finito. Existe alguma circunstância na qual um servidor de thread único possa ser melhor? Dê um exemplo.

14. Na Figura 2.12, o conjunto de registradores é listado como um item por thread em vez de por processo. Por quê? Afinal de contas, a máquina tem apenas um conjunto de registradores.

15. Por que um thread em algum momento abriria mão voluntariamente da CPU chamando *thread_yield*? Afinal, visto que não há uma interrupção periódica de relógio, ele talvez jamais receba a CPU de volta.

16. É possível que um thread seja antecipado por uma interrupção de relógio? Se a resposta for afirmativa, em quais circunstâncias?

17. Neste problema, você deve comparar a leitura de um arquivo usando um servidor de arquivos de um thread único e um servidor com múltiplos threads. São necessários 12 ms para obter uma requisição de trabalho, despachá-la e realizar o resto do processamento, presumindo que os dados necessários estejam na cache de blocos. Se uma operação de disco for necessária, como é o caso em um terço das vezes, 75 ms adicionais são necessários, tempo em que o thread repousa. Quantas requisições/segundo o servidor consegue lidar se ele tiver um único thread? E se ele for multithread?

18. Qual é a maior vantagem de se implementar threads no espaço de usuário? Qual é a maior desvantagem?

19. Na Figura 2.15 as criações dos thread e mensagens impressas pelos threads são intercaladas ao acaso. Existe alguma maneira de se forçar que a ordem seja estritamente thread 1 criado, thread 1 imprime mensagem, thread 1 sai, thread 2 criado, thread 2 imprime mensagem, thread 2 sai e assim por diante? Se a resposta for afirmativa, como? Se não, por que não?

20. Na discussão sobre variáveis globais em threads, usamos uma rotina *create_global* para alocar memória para um ponteiro para a variável, em vez de para a própria variável. Isso é essencial, ou as rotinas poderiam funcionar somente com os próprios valores?

21. Considere um sistema no qual threads são implementados inteiramente no espaço do usuário, com o sistema de tempo de execução sofrendo uma interrupção de relógio a cada segundo. Suponha que uma interrupção de relógio ocorra enquanto um thread está executando no sistema de tempo de execução. Qual problema poderia ocorrer? Você poderia sugerir uma maneira para solucioná-lo?

22. Suponha que um sistema operacional não tem nada parecido com a chamada de sistema select para saber antecipadamente se é seguro ler de um arquivo, pipe ou dispositivo, mas ele permite que relógios de alarme sejam configurados para interromper chamadas de sistema bloqueadas. É possível implementar um pacote de threads no espaço do usuário nessas condições? Discuta.

23. A solução da espera ocupada usando a variável *turn* (Figura 2.23) 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 2.24 funciona quando o escalonamento de processos é preemptivo? E quando ele é não preemptivo?

25. O problema da inversão de prioridades discutido na Seção 2.3.4 acontece com threads de usuário? Por que ou por que não?

26. Na Seção 2.3.4, uma situação com um processo de alta prioridade, *H*, e um processo de baixa prioridade, *L*, foi descrita, o que levou *H* a entrar em um laço infinito. O mesmo problema ocorre se o escalonamento circular for usado em vez do escalonamento de prioridade? Discuta.

27. Em um sistema com threads, há uma pilha por thread ou uma pilha por processo quando threads de usuário são usados? E quando threads de núcleo são usados? Explique.

28. Quando um computador está sendo desenvolvido, normalmente ele é primeiro simulado por um programa que executa uma instrução de cada vez. Mesmo multiprocessadores são simulados de maneira estritamente sequencial. É possível que uma condição de corrida ocorra quando não há eventos simultâneos como nesses casos?

29. O problema produtor-consumidor pode ser ampliado para um sistema com múltiplos produtores e consumidores que escrevem (ou leem) para (ou de) um buffer compartilhado. Presuma que cada produtor e consumidor executem seu próprio thread. A solução apresentada na Figura 2.28 usando semáforos funcionaria para esse sistema?

30. Considere a solução a seguir para o problema da exclusão mútua envolvendo dois processos *P0* e *P1*. Presuma que a variável *turn* seja inicializada para 0. O código do processo *P0* é apresentado a seguir.

```
/* Outro codigo */
while (turn != 0) { } /* Nao fazer nada e esperar */
Critical Section /* . . . */
turn = 0;

/* Outro codigo */
```

Para o processo *P1*, substitua 0 por 1 no código anterior. Determine se a solução atende a todas as condições exigidas para uma solução de exclusão mútua.

31. Como um sistema operacional capaz de desabilitar interrupções poderia implementar semáforos?

32. Mostre como semáforos contadores (isto é, semáforos que podem armazenar um valor arbitrário) podem ser implementados usando apenas semáforos binários e instruções de máquinas ordinárias.

33. Se um sistema tem apenas dois processos, faz sentido usar uma barreira para sincronizá-los? Por que ou por que não?

34. É possível que dois threads no mesmo processo sincronizem usando um semáforo do núcleo se os threads são implementados pelo núcleo? E se eles são implementados no espaço do usuário? Presuma que nenhum thread em qualquer outro processo tenha acesso ao semáforo. Discuta suas respostas.

35. A sincronização dentro de 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 tivesse um predicado booleano arbitrário como parâmetro. Desse modo, você poderia dizer, por exemplo,

waituntil $x < 0$ ou $y + z < n$

A primitiva signal não seria mais necessária. Esse esquema é claramente mais geral do que o de Hoare ou Brinch Hansen, mas não é usado. Por que não? (*Dica*: pense a respeito da implementação.)

36. Uma lanchonete tem quatro tipos de empregados: (1) atendentes, que pegam os pedidos dos clientes; (2) cozinheiros, que preparam a comida; (3) especialistas em empacotamento, que colocam a comida nas sacolas; e (4) caixas, que dão as sacolas para os clientes e recebem seu dinheiro. Cada empregado pode ser considerado um processo sequencial comunicante. Que forma de comunicação entre processos eles usam? Relacione esse modelo aos processos em UNIX.

37. Suponha que temos um sistema de transmissão de mensagens usando caixas de correio. Quando envia para uma caixa de correio cheia ou tenta receber de uma vazia, um processo não bloqueia. Em vez disso, ele recebe de volta um código de erro. O processo responde ao código de erro apenas tentando de novo, repetidas vezes, até ter sucesso. Esse esquema leva a condições de corrida?

38. Os computadores CDC 6600 poderiam lidar com até 10 processos de E/S simultaneamente usando uma forma interessante de escalonamento circular chamado de compartilhamento de processador. Um chaveamento de processo ocorreu após cada instrução, de maneira que a instrução 1 veio do processo 1, a instrução 2 do processe 2 etc. O chaveamento de processo foi realizado por um hardware especial e a sobrecarga foi zero. Se um processo necessitasse T segundos para completar na ausência da competição, de quanto tempo ele precisaria se o compartilhamento de processador fosse usado com n processos?

39. Considere o fragmento de código C seguinte:

```
void main( ) {
fork( );
fork( );
exit( );
}
```

Quantos processos filhos são criados com a execução desse programa?

40. Escalonadores circulares em geral mantêm uma lista de todos os processos executáveis, com cada processo ocorrendo exatamente uma vez na lista. O que aconteceria se um processo ocorresse duas vezes? Você consegue pensar em qualquer razão para permitir isso?

41. É possível determinar se um processo é propenso a se tornar limitado pela CPU ou limitado pela E/S analisando o código fonte? Como isso pode ser determinado no tempo de execução?

42. Explique como o valor quantum de tempo e tempo de chaveamento de contexto afetam um ao outro, em um algoritmo de escalonamento circular.

43. Medidas de um determinado sistema mostraram que o processo típico executa por um tempo T antes de bloquear na E/S. Um chaveamento de processo exige um tempo S, que é efetivamente desperdiçado (sobrecarga). Para o escalonamento circular com quantum Q, dê uma fórmula para a eficiência da CPU para cada uma das situações a seguir:

(a) $Q = \infty$.
(b) $Q > T$.
(c) $S < Q < T$.
(d) $Q = S$.
(e) Q quase 0.

44. Cinco tarefas estão esperando para serem executadas. Seus tempos de execução esperados são 9, 6, 3, 5 e X. Em qual ordem elas devem ser executadas para minimizar o tempo de resposta médio? (Sua resposta dependerá de X.)

45. Cinco tarefas em lote, A até E, chegam a um centro de computadores quase ao mesmo tempo. Elas têm tempos de execução estimados de 10, 6, 2, 4 e 8 minutos. Suas prioridades (externamente determinadas) são 3, 5, 2, 1 e 4, respectivamente, sendo 5 a mais alta. Para cada um dos algoritmos de escalonamento a seguir, determine o tempo de retorno médio do processo. Ignore a sobrecarga de chaveamento de processo.

(a) Circular.
(b) Escalonamento por prioridade.
(c) Primeiro a chegar, primeiro a ser servido (siga a ordem 10, 6, 2, 4, 8).
(d) Tarefa mais curta primeiro.

Para (a), presuma que o sistema é multiprogramado e que cada tarefa recebe sua porção justa de tempo na CPU. Para (b) até (d), presuma que apenas uma tarefa de

cada vez é executada, até terminar. Todas as tarefas são completamente limitadas pela CPU.

46. Um processo executando em CTSS precisa de 30 quanta para ser completo. Quantas vezes ele deve ser trocado para execução, incluindo a primeiríssima vez (antes de ter sido executado)?

47. Considere um sistema de tempo real com duas chamadas de voz de periodicidade de 5 ms cada com um tempo de CPU por chamada de 1 ms, e um fluxo de vídeo de periodicidade de 33 ms com tempo de CPU por chamada de 11 ms. Esse sistema é escalonável?

48. Para o problema 47, será que outro fluxo de vídeo pode ser acrescentado e ter o sistema ainda escalonável?

49. O algoritmo de envelhecimento com $a = 1/2$ está sendo usado para prever tempos de execução. As quatro execuções anteriores, da mais antiga à mais recente, são 40, 20, 40 e 15 ms. Qual é a previsão do próximo tempo?

50. Um sistema de tempo real não crítico tem quatro eventos periódicos com períodos de 50, 100, 200 e 250 ms cada. Suponha que os quatro eventos exigem 35, 20, 10 e x ms de tempo da CPU, respectivamente. Qual é o maior valor de x para o qual o sistema é escalonável?

51. No problema do jantar dos filósofos, deixe o protocolo a seguir ser usado: um filósofo de número par sempre pega o seu garfo esquerdo antes de pegar o direito; um filósofo de número ímpar sempre pega o garfo direito antes de pegar o esquerdo. Esse protocolo vai garantir uma operação sem impasse?

52. Um sistema de tempo real precisa tratar de duas chamadas de voz onde cada uma executa a cada 6 ms e consome 1 ms de tempo da CPU por surto, mais um vídeo de 25 quadros/s, com cada quadro exigindo 20 ms de tempo de CPU. Esse sistema é escalonável?

53. Considere um sistema no qual se deseja separar política e mecanismo para o escalonamento de threads de núcleo. Proponha um meio de atingir essa meta.

54. Na solução para o problema do jantar dos filósofos (Figura 2.47), por que a variável de estado está configurada para *HUNGRY* na rotina *take_forks*?

55. Considere a rotina *put_forks* na Figura 2.47. Suponha que a variável *state[i]* foi configurada para *THINKING após* as duas chamadas para teste, em vez de *antes*. Como essa mudança afetaria a solução?

56. O problema dos leitores e escritores pode ser formulado de várias maneiras em relação a qual categoria de processos pode ser iniciada e quando. Descreva cuidadosamente três variações diferentes do problema, cada uma favorecendo (ou não favorecendo) alguma categoria de processos. Para cada variação, especifique o que acontece quando um leitor ou um escritor está pronto para acessar o banco de dados, e o que acontece quando um processo foi concluído.

57. Escreva um roteiro (*script*) shell que produz um arquivo de números sequenciais lendo o último número, adicionando 1 a ele, e então anexando-o ao arquivo. Execute uma instância do roteiro no segundo plano e uma no primeiro plano, cada uma acessando o mesmo arquivo. Quanto tempo leva até que a condição de corrida se manifeste? Qual é a região crítica? Modifique o roteiro para evitar a corrida. (*Dica*: use

```
ln file file.lock
```

para travar o arquivo de dados.)

58. Presuma que você tem um sistema operacional que fornece semáforos. Implemente um sistema de mensagens. Escreva os procedimentos para enviar e receber mensagens.

59. Solucione o problema do jantar de filósofos usando monitores em vez de semáforos.

60. Suponha que uma universidade queira mostrar o quão politicamente correta ela é, aplicando a doutrina "Separado mas igual é inerentemente desigual" da Suprema Corte dos EUA para o gênero, assim como a raça, terminando sua prática de longa data de banheiros segregados por gênero no *campus*. No entanto, como uma concessão para a tradição, ela decreta que se uma mulher está em um banheiro, outras mulheres podem entrar, mas nenhum homem, e vice-versa. Um sinal com uma placa móvel na porta de cada banheiro indica em quais dos três estados possíveis ele se encontra atualmente:

- Vazio.
- Mulheres presentes.
- Homens presentes.

Em alguma linguagem de programação de que você goste, escreva as seguintes rotinas: *woman_wants_to_enter*, *man_wants_to_enter*, *woman_leaves*, *man_leaves*. Você pode usar as técnicas de sincronização e contadores que quiser.

61. Reescreva o programa da Figura 2.23 para lidar com mais do que dois processos.

62. Escreva um problema produtor-consumidor que use threads e compartilhe de um buffer comum. No entanto, não use semáforos ou quaisquer outras primitivas de sincronização para guardar as estruturas de dados compartilhados. Apenas deixe cada thread acessá-las quando quiser. Use sleep e wakeup para lidar com condições de cheio e vazio. Veja quanto tempo leva para uma condição de corrida fatal ocorrer. Por exemplo, talvez você tenha o produtor imprimindo um número de vez em quando. Não imprima mais do que um número a cada minuto, pois a E/S poderia afetar as condições de corrida.

63. Um processo pode ser colocado em uma fila circular mais de uma vez para dar a ele uma prioridade mais alta. Executar instâncias múltiplas de um programa, cada uma trabalhando em uma parte diferente de um pool de dados

pode ter o mesmo efeito. Primeiro escreva um programa que teste uma lista de números para primalidade. Então crie um método para permitir que múltiplas instâncias do programa executem ao mesmo tempo de tal maneira que duas instâncias do programa não trabalharão sobre o mesmo número. Você consegue de fato repassar mais rápido a lista executando múltiplas cópias do programa? Observe que seus resultados dependerão do que mais seu computador estiver fazendo; em um computador pessoal executando apenas instâncias desse programa você não esperaria uma melhora, mas em um sistema com outros processos, deve conseguir ficar com uma porção maior da CPU dessa maneira.

64. O objetivo desse exercício é implementar uma solução com múltiplos threads para descobrir se um determinado número é um número perfeito. *N* é um número perfeito se a soma de todos os seus fatores, excluindo ele mesmo, for *N*; exemplos são 6 e 28. A entrada é um inteiro, *N*. A saída é verdadeira se o número for um número perfeito e falsa de outra maneira. O programa principal lerá os números *N* e *P* da linha de comando. O processo principal gerará um conjunto de threads *P*. Os números de 1 a *N* serão divididos entre esses threads de maneira que dois threads não trabalhem sobre o mesmo número. Para cada número nesse conjunto, o thread determinará se o número é um fator de *N*; se for, ele acrescenta o número a um buffer compartilhado que armazena fatores de *N*. O processo pai espera até que todos os threads terminem. Use a primitiva de sincronização apropriada aqui. O pai determinará então se o número de entrada é perfeito, isto é, se *N* é uma soma de todos os seus fatores, então reportará em conformidade. (**Nota**: você pode fazer a computação mais rápido restringindo os números buscados de 1 até a raiz quadrada de *N*).

65. Implemente um programa para contar a frequência de palavras em um arquivo de texto. O arquivo de texto é dividido em *N* segmentos. Cada segmento é processado por um thread em separado que produz a contagem de frequência intermediária para esse segmento. O processo principal espera até que todos os threads terminem; então ele calcula os dados de frequência de palavras consolidados baseados na produção dos threads individuais.