Capítulo 2
Processos e threads

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

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

2.1 Processos

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

Agora considere um usuário de PC. Quando o sistema é inicializado, muitos processos muitas vezes desconhecidos ao usuário começam secretamente. Por exemplo, um processo pode ser iniciado para espera de e-mails que chegam. Outro processo pode ser executado pelo programa de antivírus para verificar periodicamente se há novas definições de antivírus disponíveis. Além disso, processos de usuários explícitos podem estar sendo executados, imprimindo arquivos e gravando um CD-ROM, tudo enquanto o usuário está navegando na Web. Toda essa atividade tem de ser administrada, e um sistema multiprogramado que sustente múltiplos processos é bastante útil nesse caso.

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

2.1.1 O modelo de processo

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

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

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

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

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

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

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

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

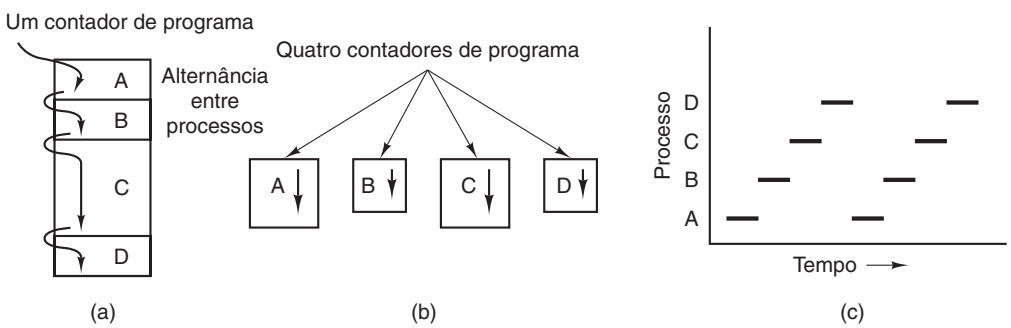


Figura 2.1 (a) Multiprogramação de quatro programas. (b) Modelo conceitual de quatro processos sequenciais independentes. (c) Somente um programa está ativo a cada momento.

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

2.1.2 | Criação de processos

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

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

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

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

Além dos processos criados durante a carga do sistema operacional, novos processos podem ser criados depois disso. Muitas vezes, um processo em execução fará chamadas de sistema (*system calls*) para criar um ou mais novos processos para ajudá-lo em seu trabalho. Criar novos processos é particularmente interessante quando a tarefa a ser executada puder ser facilmente dividida em vários processos relacionados, mas interagindo de maneira independente. Por exemplo, se uma grande quantidade de dados estiver sendo trazida via rede para que seja subsequentemente processada, poderá ser conveniente criar um processo para trazer esses dados e armazená-los em um local compartilhado da memória, enquanto um segundo processo remove os dados e os processa. Em um sistema multiprocessador, permitir que cada processo execute em uma CPU diferente também torna o trabalho mais rápido.

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

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

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

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

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

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

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

2.1.3 | Término de processos

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

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

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

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

```
cc foo.c
```

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

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

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

2.1.4 | Hierarquias de processos

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

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

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

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

2.1.5 | Estados de processos

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

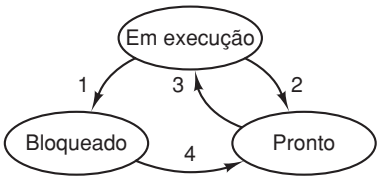
```
cat chapter1 chapter2 chapter3 | grep tree
```

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

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

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

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



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

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

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

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

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

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

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

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

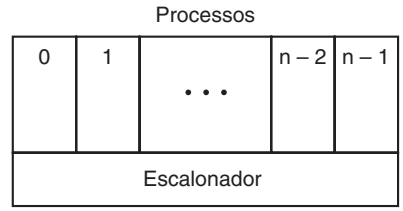


Figura 2.3 O nível mais baixo de um sistema operacional estruturado em processos controla interrupções e escalonamento. Acima desse nível estão processos sequenciais.

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 processo. (Alguns autores chamam essas entradas de **process control blocks** — blocos de controle de processo.) Essa entrada contém informações sobre o estado do processo, seu contador de programa, o ponteiro da pilha, a alocação de memória, os estados de seus arquivos abertos, sua informação sobre contabilidade e escalonamento e tudo o mais sobre o processo que deva ser salvo quando o processo passar do estado *em execução* para o estado *pronto* ou *bloqueado*, para que ele possa ser reiniciado depois, como se nunca tivesse sido bloqueado.

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

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

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

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

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

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

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

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

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

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

2.1.7 Modelando a multiprogramação

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

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

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

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

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

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

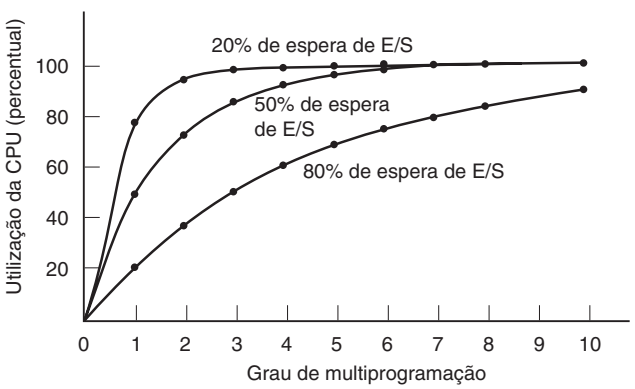


Figura 2.4 Utilização da CPU como função do número de processos na memória.

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

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

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

2.2 Threads

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

2.2.1 O uso de thread

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

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

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

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

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

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

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

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

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

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

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

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

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

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

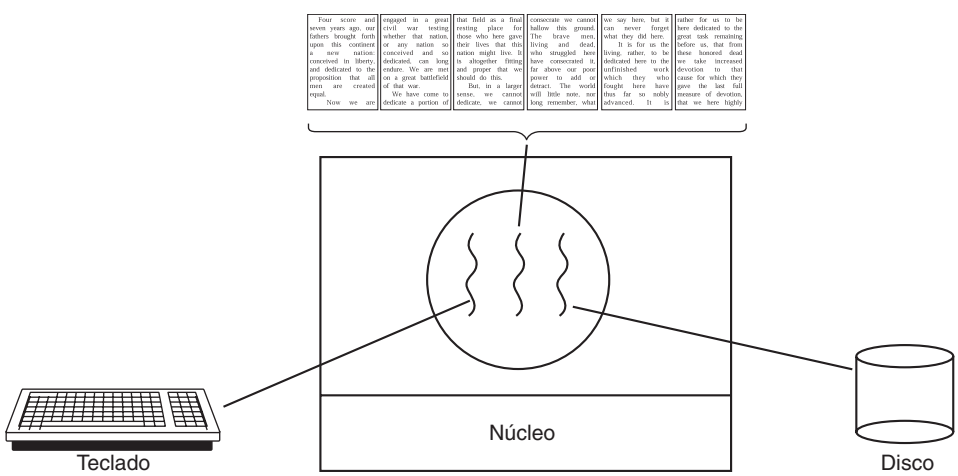


Figura 2.5 Um processador de textos com três threads.

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

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

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

Esse modelo permite que o servidor seja escrito como uma coleção de threads sequenciais. O programa do despachante consiste em um laço infinito para obter requisições de trabalho e entregá-las a um operário. Cada código de operário consiste em um laço infinito, que acata uma requisição de um despachante e verifica se a página está presente na cache de páginas da Web. Se estiver, entrega-a ao cliente e o operário bloqueia esperando uma nova requisição. Do contrário, ele busca a página no disco, entrega-a ao cliente e bloqueia esperando uma nova requisição.

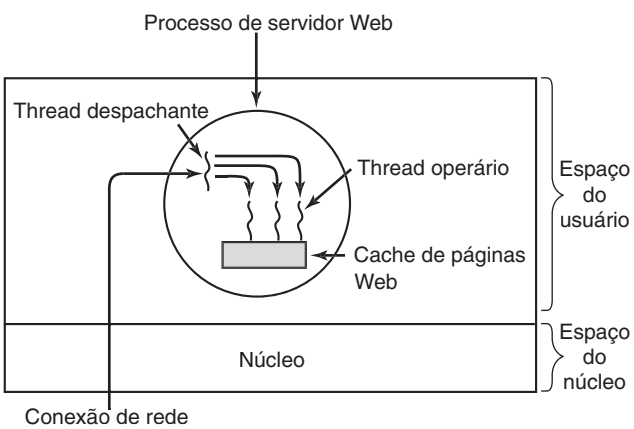


Figura 2.6 Um servidor Web multithread.

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

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

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

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

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

(a)

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

(b)

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

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

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

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

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

Os threads oferecem uma solução. O processo poderia ser estruturado com um thread de entrada, um thread de processamento e um thread de saída. O thread de entrada lê os dados em um buffer de entrada. O thread de processamento tira os dados do buffer de entrada, processa-os e põe os resultados em um buffer de saída. O thread de saída escreve esses resultados de volta no disco. Desse modo, entrada, saída e processamento podem funcionar todos ao mesmo tempo. É claro que esse modelo funciona somente se uma chamada de sistema bloqueia apenas o thread que está chamando, e não todo o processo.

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

Tabela 2.3 Três modos de construir um servidor.

2.2.2 O modelo de thread clássico

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

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

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

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

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

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

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

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

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

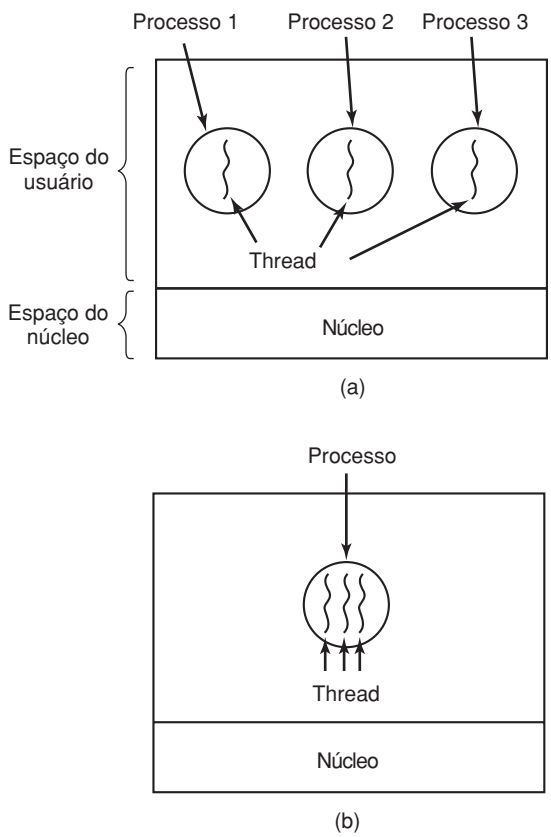


Figura 2.8 (a) Três processos, cada um com um thread. (b) Um processo com três threads.

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

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

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

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

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

Quando termina seu trabalho, um thread pode terminar sua execução chamando uma rotina de biblioteca — digamos, *thread_exit*. Ele então desaparece e não é mais escalonável. Em alguns sistemas de thread, um thread pode esperar pela saída de um thread (específico) chamando um procedimento *thread_join*, por exemplo. Essa rotina bloqueia o thread que executou a chamada até que um thread (específico) tenha terminado. Sendo assim, a criação e o término do thread são muito parecidos com a criação e o término de processos, inclusive com quase as mesmas opções.

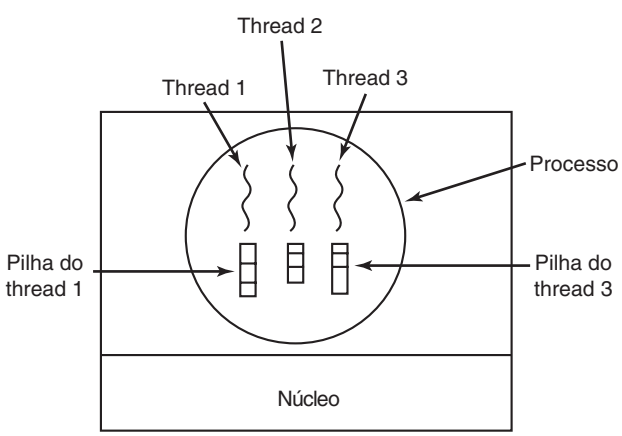


Figura 2.9 Cada thread tem sua própria pilha.

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

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

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

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

2.2.3 Threads POSIX

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

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

Tabela 2.5 Algumas das chamadas de função de Pthreads.

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

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

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

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

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

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

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

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

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

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

2.2.4 Implementação de threads no espaço do usuário

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

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

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

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

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

#define NUMBER_OF_THREADS     10

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

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

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

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

Figura 2.10 Um exemplo de programa usando threads.

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

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

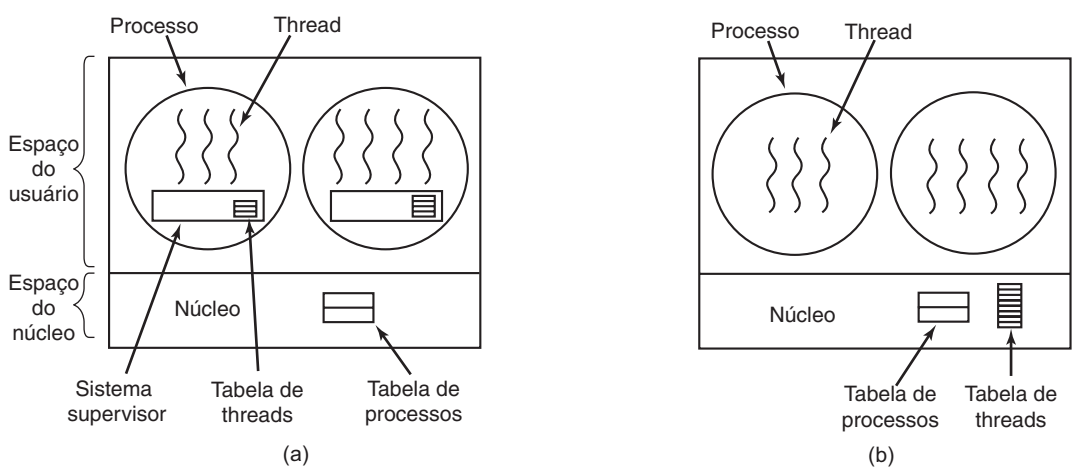


Figura 2.11 (a) Um pacote de threads de usuário. (b) Um pacote de threads administrado pelo núcleo.

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

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

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

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

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

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

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

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

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

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

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

2.2.5 Implementação de threads no núcleo

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

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

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

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

Os threads de núcleo não precisam de nenhuma chamada de sistema não-bloqueante. Além disso, se um thread em um processo causa uma falta de página, o núcleo pode facilmente verificar se o processo tem threads para execução e, em caso afirmativo, pode executá-los enquanto aguarda a página requisitada ser trazida do disco. A principal desvantagem é que o custo de uma chamada de sistema é alto e, portanto, a ocorrência frequente de operações de thread (criação, término etc.) causará uma sobrecarga muito maior.

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

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

2.2.6 Implementações híbridas

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

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

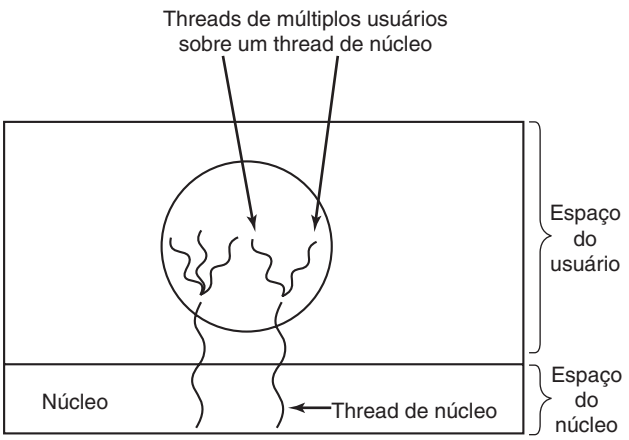


Figura 2.12 Multiplexando threads de usuários sobre threads de núcleo.

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

2.2.7 | Ativações do escalonador

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

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

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

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

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

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

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

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

2.2.8 | Threads pop-up

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

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

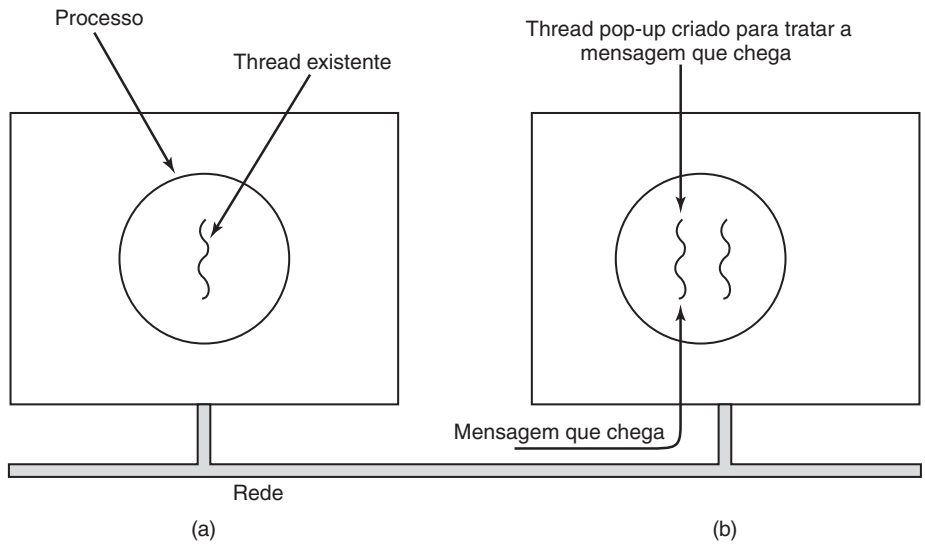


Figura 2.13 Criação de um novo thread quando chega uma mensagem. (a) Antes da chegada da mensagem. (b) Após a chegada da mensagem.

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

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

2.2.9 Convertendo o código monothread em código multithread

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

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

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

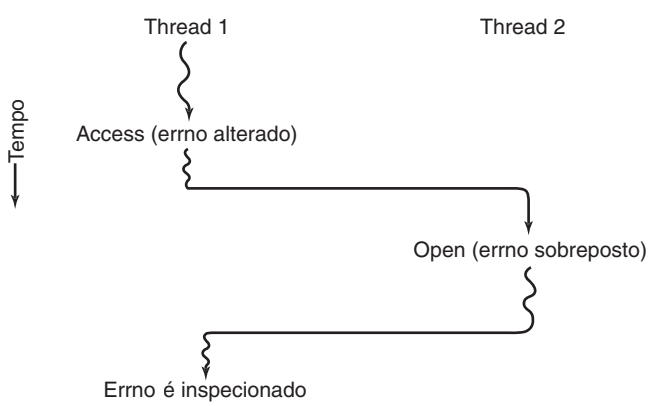


Figura 2.14 Conflitos entre threads sobre o uso de uma variável global.

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

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

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

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

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

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

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

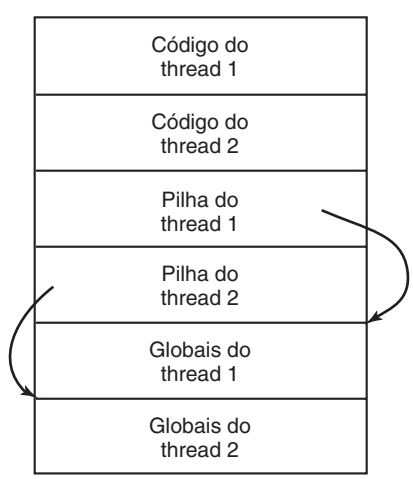


Figura 2.15 Threads podem ter variáveis globais individuais.

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

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

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

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

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

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

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

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

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

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

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

2.3 Comunicação entre processos

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

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

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

2.3.1 Condições de corrida

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

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

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

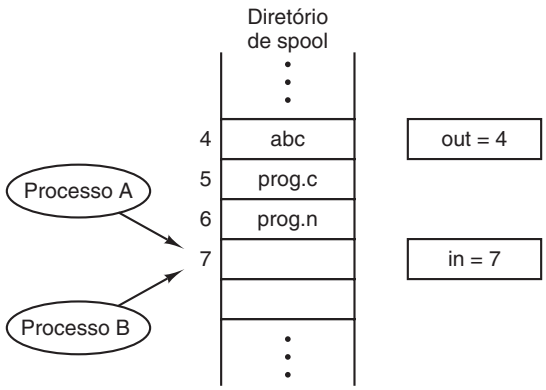


Figura 2.16 Dois processos querem acessar a memória compartilhada ao mesmo tempo.

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

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

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

2.3.2 | Regiões críticas

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

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

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

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

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

2.3.3 | Exclusão mútua com espera ociosa

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

Desabilitando interrupções

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

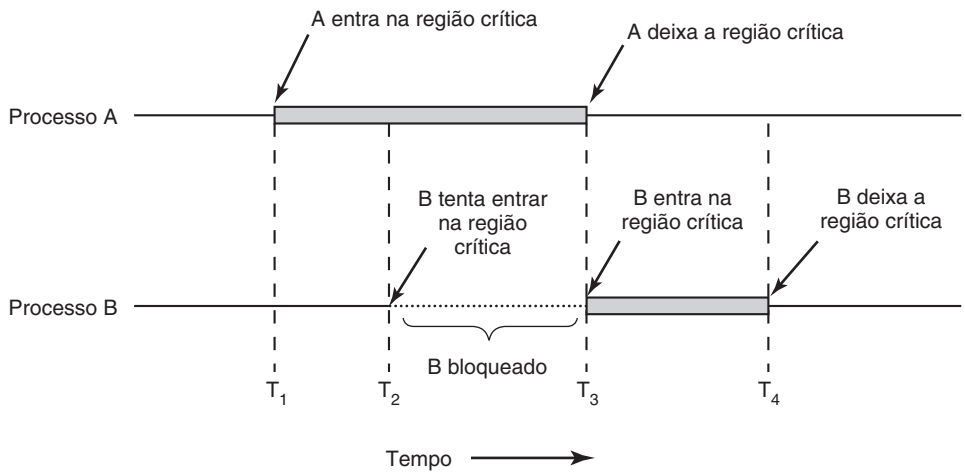


Figura 2.17 Exclusão mútua usando regiões críticas.

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

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

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

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

Variáveis do tipo trava (*lock*)

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

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

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

Chaveamento obrigatório

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

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

(a)

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

(b)

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

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

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

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

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

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

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

Solução de Peterson

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

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

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

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

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

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

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

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

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

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

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

A instrução TSL

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

```
TSL RX,LOCK
```

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

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

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

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

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

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

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

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

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

2.3.4 | Dormir e acordar

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

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

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

O problema do produtor–consumidor

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

```
enter_region:
    MOVE REGISTER,#1      | insira 1 no registrador
    XCHG REGISTER,LOCK    | substitua os conteúdos do registrador e a variação de lock
    CMP REGISTER,#0       | lock valia zero?
    JNE enter_region      | se fosse diferente de zero, lock estaria ligado, portanto continue no laço de repetição
    RET                   | retorna a quem chamou; entrou na região crítica

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

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

1. Do inglês *exchange* = troca (N.R.T).

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

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

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

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

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

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

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

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

void producer(void)
{
     int item;

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

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

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

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

2.3.5 | Semáforos

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

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

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

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

Resolvendo o problema produtor–consumidor usando semáforos

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

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

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

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

```
#define N 100                                   /* número de lugares no buffer */
typedef int semaphore;                          /* semáforos são um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso à região crítica */
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 é a constante 1 */
          item = produce_item( );               /* gera algo para pôr no buffer */
          down(&empty);                         /* decresce o contador empty */
          down(&mutex);                         /* entra na região crítica */
          insert_item(item);                    /* põe novo item no buffer */
          up(&mutex);                           /* sai da região crítica */
          up(&full);                            /* incrementa o contador de lugares preenchidos */
     }
}


void consumer(void)
{
     int item;

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

Figura 2.23 O problema produtor–consumidor usando semáforos.

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

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

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

2.3.6 Mutexes

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

Um **mutex** é uma variável que pode estar em um dos dois estados seguintes: desimpedido ou impedido. Consequentemente, somente 1 bit é necessário para representá-lo, mas, na prática, muitas vezes se usa um inteiro, com 0 para

desimpedido e qualquer outro valor para impedido. Duas rotinas são usadas com mutexes. Quando um thread (ou processo) precisa ter acesso a uma região crítica, ele chama *mutex_lock*. Se o mutex estiver desimpedido (indicando que a região crítica está disponível), a chamada prosseguirá e o thread que chamou *mutex_lock* ficará livre para entrar na região crítica.

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

Por serem muito simples, os mutexes podem ser implementados facilmente no espaço de usuário, se houver uma instrução TSL ou XCHG disponível. Os códigos de *mutex_lock* e *mutex_unlock*, para que sejam usados com um pacote de threads de usuário, são mostrados na Figura 2.24. A solução com XCHG é essencialmente a mesma.

O código do *mutex_lock* é similar ao código do *enter_region* da Figura 2.20, mas com uma diferença fundamental. Quando falha ao entrar na região crítica, o *enter_region* continua testando repetidamente a variável de trava (espera ociosa). Ao final, o tempo de CPU se esgota e algum outro processo é escalonado para executar. Cedo ou tarde o processo que detém a trava é executado e o libera.

Com threads (de usuário), a situação é diferente porque não há relógio que pare os threads que estiverem executando há muito tempo. Consequentemente, um thread que tentar obter a variável de trava pela espera ociosa ficará em um laço infinito e nunca conseguirá obter essa variável, pois ele nunca permitirá que qualquer outro thread execute e libere a variável de trava.

Eis a diferença entre *enter_region* e *mutex_lock*: quando falha em verificar a variável de trava, *mutex_lock* chama *thread_yield* para que abra mão da CPU em favor de outro thread. Consequentemente, não há espera ocupada. Quando executar na próxima vez, o thread verificará a variável de trava novamente.

O *thread_yield* é muito rápido, pois é apenas uma chamada do escalonador de threads no espaço do usuário. Como consequência, nem *mutex_lock* nem *mutex_unlock* requerem qualquer chamada ao núcleo. Usando-as, threads de usuário podem sincronizar totalmente dentro do espaço de usuário, com rotinas que exigem somente algumas instruções.

O sistema mutex que descrevemos anteriormente é um conjunto mínimo de chamadas. Para todo software há sempre uma exigência por aperfeiçoamentos, e o caso das primitivas de sincronização não é exceção. Por exemplo, algumas vezes um pacote de thread oferece uma chamada *mutex_trylock* que obtém a variável de trava ou retorna um código de falha sem bloquear. Essa chamada dá ao thread a flexibilidade de decidir o que fazer se houver alternativas a apenas esperar.

Até agora não tratamos de um determinado tópico, mas é melhor explicitá-lo. Com o pacote de threads no espaço de usuário, não há problema de existirem múltiplos threads com acesso ao mesmo mutex, já que todos os threads operam em um espaço de endereçamento comum. Contudo, como a maioria das soluções anteriores — como o algoritmo de Peterson e os semáforos —, há uma hipótese, não comentada, de que múltiplos processos tenham acesso a pelo menos alguma memória compartilhada, talvez somente a uma única palavra na memória, mas pelo menos alguma. Se os processos possuírem espaços de endereçamento disjuntos, conforme temos dito insistentemente, como eles poderiam compartilhar a variável *turn* no algoritmo de Peterson, ou os semáforos, ou um buffer comum?

Há duas respostas. Primeiro, algumas das estruturas de dados compartilhadas, como os semáforos, podem ser armazenadas no núcleo e somente ter seu acesso disponível via chamadas de sistema. Esse método elimina o problema. Em segundo, a maioria dos sistemas operacionais modernos (incluindo UNIX e Windows) oferece meios para que processos compartilhem alguma parte de seus espaços de endereçamento com outros processos. Desse modo, buffers e outras estruturas de dados podem ser compartilhados. Em um caso extremo, em que nada mais é possível, pode-se usar um arquivo compartilhado.

Se dois ou mais processos compartilharem a maior parte ou a totalidade do espaço de endereçamento, a distinção entre processos e threads torna-se algo difusa, mas ainda presente. Dois processos que compartilham um espaço de

```
mutex_lock:
    TSL REGISTER,MUTEX          | copia mutex para o registrador e atribui  a ele o valor 1
    CMP REGISTER,#0             | o mutex era zero?
    JZE ok                      | se era zero, o mutex estava desimpedido, portanto retorne
    CALL thread_yield           | o mutex está ocupado; escalone um outro thread
    JMP mutex_lock              | tente novamente mais tarde
ok:  RET                        | retorna a quem chamou; entrou na região crítica

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

Figura 2.24 Implementação do *mutex_lock* e do *mutex_unlock*.

endereçamento comum ainda têm arquivos abertos diferentes, temporizadores de alarme e outras propriedades por processo, enquanto os threads dentro de um único processo compartilham todas as propriedades. É sempre válida a afirmação de que múltiplos processos que compartilham um espaço de endereçamento comum nunca têm a eficiência de threads de usuário, pois o núcleo estará profundamente envolvido no gerenciamento desses processos.

Mutexes em pthreads

Os Pthreads fornecem várias funções que podem ser usadas para sincronizar threads. O mecanismo básico usa uma variável mutex, que pode ser travada ou destravada, para proteger cada região crítica. Um thread que queira entrar em uma região crítica primeiro tenta travar o mutex associado. Se o mutex estiver destravado, o thread pode entrar imediatamente e uma trava é estabelecida atomicamente, evitando que outros threads entrem. Se o mutex já estiver travado, o thread que chama é bloqueado até que ele seja destravado. Se múltiplos threads estão esperando pelo mesmo mutex, quando ele for destravado, apenas um deles é autorizado a continuar e travá-lo novamente. Essas travas não são obrigatórias. Cabe ao programador assegurar que os threads os utilizem corretamente.

As principais chamadas relacionadas a mutexes são mostradas na Tabela 2.6. Como era de se esperar, eles podem ser criados e destruídos. As chamadas para executar essas operações são *pthread_mutex_init* e *pthread_mutex_destroy,* respectivamente. Eles também podem ser travados por *pthread_mutex_lock* — que tenta conquistar a trava e é bloqueado, se o mutex já estiver impedido. Também há a opção de tentar travar um mutex e falhar com um código de erro em vez de bloqueá-lo, se ele já estiver bloqueado. Essa chamada é *pthread_mutex_trylock*. Essa chamada permite que um thread entre em espera caso seja necessário. Por fim, *pthread_mutex_unlock* ocupada um mutex e libera exatamente um thread se um ou mais estiverem esperando por ele. Os mutexes também podem ter atributos, mas esses são usados apenas para objetivos especializados.

Além dos mutexes, os Pthreads oferecem um segundo mecanismo de sincronização: **variáveis de condição.** Os mutexes são úteis para permitir ou bloquear o acesso a uma região crítica. As variáveis de condição permitem que os threads bloqueiem em virtude de alguma condição não satisfeita. Quase sempre os dois métodos são usados juntos. Examinemos agora em maiores detalhes a interação de threads, mutexes e variáveis de condição.

Considere novamente o exemplo simples da situação produtor–consumidor: um thread coloca coisas em um buffer e outro as retira. Se o produtor descobrir que não há mais espaço disponível no buffer, ele deve bloquear até que algum se torne disponível. Os mutexes permitem fazer a verificação automaticamente sem interferência de outros threads, mas, diante da descoberta de que o buffer está cheio, o produtor precisa de uma maneira para bloquear e despertar mais tarde. É isso que as variáveis de condição permitem.

Algumas das chamadas relacionadas às variáveis de condição são mostradas na Tabela 2.7. Como você provavelmente já esperava, há chamadas para criar e destruir variáveis de condição. Elas podem ter atributos e há várias chamadas para administrá-las (não mostradas). As operações principais de variáveis de condição são *pthread_cond_wait* e *pthread_cond_signal*. A primeira bloqueia o thread que chama até que algum outro thread sinalize (usando a última chamada). É claro que as razões para bloquear e esperar não são parte do protocolo de espera e sinalização. O thread bloqueado muitas vezes está esperando que o thread que sinaliza faça algum trabalho, libere algum recurso ou execute alguma outra atividade. Somente depois disso o thread que bloqueia pode prosseguir. As variáveis de condição permitem que essa espera e bloqueio sejam feitos atomicamente. A chamada *pthread_cond_broadcast* é usada quando potencialmente há múltiplos threads bloqueados e esperando pelo mesmo sinal.

Variáveis de condição e mutexes são sempre usados em conjunto. O padrão é um thread travar um mutex e então esperar por uma variável condicional quando não puder obter o que precisa. Eventualmente, outro thread sinalizará e ele pode continuar. O *pthread_cond_wait* chama atomicamente e destrava atomicamente o mutex que está controlando. Por essa razão, o mutex é um dos parâmetros.

Chamada de thread	Descrição
pthread_mutex_init	Cria um mutex
pthread_mutex_destroy	Destrói um mutex existente
pthread_mutex_lock	Conquista uma trava ou bloqueio
pthread_mutex_trylock	Conquista uma trava ou falha
pthread_mutex_unlock	Libera uma trava

Tabela 2.6 Algumas chamadas de Pthreads relacionadas a mutexes.

Chamada de thread	Descrição
pthread_cond_init	Cria uma variável de condição
pthread_cond_destroy	Destrói uma variável de condição
pthread_cond_wait	Bloqueio esperando por um sinal
pthread_cond_signal	Sinaliza para outro thread e o desperta
pthread_cond_broadcast	Sinaliza para múltiplos threads e desperta todos eles

Tabela 2.7 Algumas chamadas de Pthreads relacionadas a variáveis de condição.

É importante observar também que as variáveis de condição (à diferença dos semáforos) não têm memória. Se um sinal é enviado para uma variável de condição pela qual nenhum thread está esperando, o sinal é perdido. Os programadores devem ser cuidadosos para não perder sinais.

Como exemplo do modo como mutexes e variáveis de condição são usados, a Figura 2.25 mostra um problema de consumidor–produtor muito simples com um único buffer. Quando o produtor tiver enchido o buffer, ele deve esperar até que o consumidor o esvazie antes de gerar o próximo item. De modo semelhante, quando o consumidor tiver removido um item, ele deve esperar até que o produtor tenha gerado outro. Embora seja muito simples, esse exemplo ilustra os mecanismos básicos. O comando que coloca um thread para dormir sempre deveria verificar a condição para assegurar que ela seja satisfeita antes de prosseguir, visto que o thread poderia ter sido despertado em virtude de um sinal do UNIX ou por alguma outra razão.

2.3.7 Monitores

Com o uso de semáforos e mutexes, a comunicação entre processos parece fácil, não é? Você ainda não viu nada! Observe com bastante atenção a ordem dos downs antes de

```
#include <stdio.h>
#include <pthread.h>
#define MAX 1000000000                          /* quantos números produzir */
pthread_mutex_t the_mutex;
pthread_cond_t condc, condp;                    /* usado para sinalização */
int buffer = 0;                                 /* buffer usado entre produtor e consumidor */

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

      for (i= 1; i <= MAX; i++) {
            pthread_mutex_lock(&the_mutex);     /* obtém acesso exclusivo ao buffer */
            while (buffer != 0) pthread_cond_wait(&condp, &the_mutex);
            buffer = i;                         /* põe item no buffer */
            pthread_cond_signal(&condc);        /* acorda consumidor */
            pthread_mutex_unlock(&the_mutex);/* libera acesso ao buffer */
      }
      pthread_exit(0);
}

void *consumer(void *ptr)                       /* dados do consumidor */
{     int i;

      for (i = 1; i <= MAX; i++) {
            pthread_mutex_lock(&the_mutex);     /* obtém acesso exclusivo ao buffer */
            while (buffer ==0 ) pthread_cond_wait(&condc, &the_mutex);
            buffer = 0;                         /* retire o item do buffer */
            pthread_cond_signal(&condp);        /* acorda o produtor */
            pthread_mutex_unlock(&the_mutex);/* libera acesso ao buffer */
      }
      pthread_exit(0);
}

int main(int argc, char **argv)
{
      pthread_t pro, con;
      pthread_mutex_init(&the_mutex, 0);
      pthread_cond_init(&condc, 0);
      pthread_cond_init(&condp, 0);
      pthread_create(&con, 0, consumer, 0);
      pthread_create(&pro, 0, producer, 0);
      pthread_join(pro, 0);
      pthread_join(con, 0);
      pthread_cond_destroy(&condc);
      pthread_cond_destroy(&condp);
      pthread_mutex_destroy(&the_mutex);
}
```

Figura 2.25 Usando threads para resolver o problema produtor–consumidor.

inserir ou remover os itens do buffer na Figura 2.23. Suponha que os dois downs, no código do produtor, estivessem invertidos, de modo que o *mutex* seria decrescido antes de *vazio* em vez de depois dele. Se o buffer estivesse completamente cheio, o produtor seria bloqueado com *mutex* em 0. Consequentemente, na vez seguinte em que o consumidor tentasse ter acesso ao buffer, faria um down no *mutex*, agora em 0, e seria bloqueado também. Ambos os processos permaneceriam eternamente bloqueados e nunca mais funcionariam. Essa situação infeliz é chamada de impasse (*deadlock*). Estudaremos os impasses em detalhes no Capítulo 6.

Esse problema foi levantado para mostrar o cuidado que se deve ter no uso de semáforos. Um erro sutil pode pôr tudo a perder. É como programar em linguagem assembly — ou até pior, pois os erros são condições de corrida, impasses e outros modos de comportamento imprevisível e irreprodutível.

Para facilitar a escrita correta de programas, Hoare (1974) e Brinch Hansen (1973) propuseram uma unidade básica de sincronização de alto nível chamada **monitor**. As propostas deles eram um pouco diferentes, conforme veremos a seguir. Um monitor é uma coleção de rotinas, variáveis e estruturas de dados, tudo isso agrupado em um tipo especial de módulo ou pacote. Os processos podem chamar as rotinas em um monitor quando quiserem, mas não podem ter acesso direto às estruturas internas de dados ao monitor a partir de rotinas declaradas fora dele. A Figura 2.26 ilustra um monitor escrito em uma linguagem imaginária, a Pascal Pidgin. C não pode ser usada aqui porque os monitores são um conceito de *linguagem* e C não os tem.

Os monitores apresentam uma propriedade importante que os torna úteis para realizar a exclusão mútua: somente um processo pode estar ativo em um monitor em um dado momento. O monitor é uma construção da linguagem de programação e, portanto, os compiladores sabem que eles são especiais e, por isso, tratam as chamadas a rotinas do monitor de modo diferente de outras chamadas de procedimento. Em geral, quando um processo chama uma rotina do monitor, algumas das primeiras instruções da rotina verificarão se qualquer outro processo está atualmente ativo dentro do monitor. Se estiver, o processo que chamou será suspenso até que o outro processo deixe o monitor. Se nenhum outro processo estiver usando o monitor, o processo que chamou poderá entrar.

```
monitor example
    integer i;
    condition c;

    procedure producer ( );
    .
    .
    .
    end;

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

Figura 2.26 Um monitor.

Cabe ao compilador implementar a exclusão mútua nas entradas do monitor, mas um modo comum é usar um mutex ou um semáforo binário. Como é o compilador e não o programador que providencia a exclusão mútua, é muito menos provável que algo dê errado. De qualquer maneira, quem codifica o monitor não precisa saber como o compilador implementou a exclusão mútua. Basta saber que, convertendo todas as regiões críticas em rotinas do monitor, dois processos nunca executarão suas regiões críticas ao mesmo tempo.

Embora monitores ofereçam um modo fácil de fazer a exclusão mútua (como vimos anteriormente), isso não é o bastante. É preciso também um modo de bloquear processos quando não puderem continuar. No problema produtor–consumidor, é muito fácil colocar todos os testes de buffer cheio e buffer vazio nas rotinas do monitor, mas como o produtor seria bloqueado quando ele encontrasse o buffer cheio?

A solução está na introdução de **variáveis condicionais**, com duas operações sobre elas: wait e signal. Quando uma rotina do monitor descobre que não pode prosseguir (por exemplo, o produtor percebe que o buffer está cheio), executa um wait sobre alguma variável condicional — por exemplo, *cheio*. Essa ação resulta no bloqueio do processo que está chamando. Ela também permite que outro processo anteriormente proibido de entrar no monitor agora entre. Vimos variáveis de condição e essas operações no contexto de Pthreads anteriormente.

Esse outro processo — por exemplo, o consumidor — pode acordar seu parceiro adormecido a partir da emissão de um signal para a variável condicional que seu parceiro está esperando. Para evitar que dois processos permaneçam no monitor ao mesmo tempo, precisamos de uma regra que determine o que acontece depois de um signal. Hoare propôs deixar o processo recém-acordado executar, suspendendo o outro. Brinch Hansen sugeriu uma maneira astuta de resolver o problema: exigir que um processo que emitir um signal saia do monitor imediatamente. Em outras palavras, um comando signal só poderá aparecer como o último comando de uma rotina do monitor. Usaremos a proposta de Brinch Hansen porque ela é conceitualmente mais simples e também mais fácil de implementar. Se um signal é emitido sobre uma variável condicional pela qual vários processos estejam esperando, somente um deles, determinado pelo escalonador do sistema, é despertado.

É preciso mencionar que há uma terceira solução que não foi proposta por Hoare nem por Brinch Hansen. Essa solução deixa o emissor do sinal prosseguir sua execução e permite ao processo em espera começar a executar somente depois que o emissor do sinal tenha saído do monitor.

Variáveis condicionais não são contadores. Elas não acumulam sinais para usá-los depois, como fazem os semáforos. Assim, se uma variável condicional for sinalizada sem ninguém estar esperando pelo sinal, este ficará perdido para sempre. Em outras palavras, wait deve vir antes do signal. Essa regra torna a implementação muito mais simples. Na prática não é um problema, pois é fácil manter o controle do estado de cada processo com variáveis, se for necessário. Um processo capaz de emitir um signal pode perceber, a partir da verificação das variáveis, que essa operação não é necessária.

Um esqueleto do problema produtor–consumidor com monitores é mostrado na Figura 2.27 em uma linguagem imaginária, a Pascal Pidgin. A vantagem de usar Pascal Pidgin é que ela é pura, simples e segue exatamente o modelo de Hoare/Brinch Hansen.

Você pode estar pensando que as operações wait e signal são parecidas com as operações sleep e wakeup, que vimos anteriormente ao tratar das condições de corrida fatais. Elas *são* muito similares, mas apresentam uma diferença fundamental: sleep e wakeup falharam porque, enquanto um processo estava tentando ir dormir, o outro tentava acordá-lo. Com monitores, isso não acontece. A exclusão mútua automática das rotinas do monitor garante, por exemplo, que, se o produtor dentro de uma rotina do monitor descobrir que o buffer está cheio, esse produtor será capaz de terminar a operação wait sem se preocupar com a possibilidade de o escalonador chavear para o consumidor um pouco antes de wait terminar. O consumidor nem mesmo será permitido dentro do monitor até que wait tenha terminado e o produtor tenha sido marcado como não mais executável.

```
monitor ProducerConsumer
    condition full, empty;
    integer count,

    procedure insert(item:integer);
    begin
        if count= N then wait (full);
        insert_item(item);
        count:= count + 1;
        if count= 1 then signal (empty)
    end ;

    function remove:integer;
    begin
        if count= 0 then wait (empty);
        remove =remove_item;
        count:= count– 1;
        if count= N – 1 then signal (full)
    end;

    count:= 0;
end monitor;

procedure producer;
begin
    while true do
    begin
        item = produce_item;
        ProducerConsumer.insert(item)
    end
end ;

procedure consumer;
begin
    while true do
    begin
        item = ProducerConsumer.remove;
        consume_item(item)
    end
end ;
```

Figura 2.27 Um esqueleto do problema produtor–consumidor com monitores. Somente uma rotina está ativa por vez no monitor. O buffer tem *N* lugares.

Embora a Pascal Pidgin seja uma linguagem imaginária, algumas linguagens de programação reais também dão suporte a monitores, embora nem sempre conforme o projetado por Hoare e Brinch Hansen. Uma dessas linguagens é Java. Esta é uma linguagem orientada a objetos que dão suporte a threads de usuário e também permite que métodos (rotinas) sejam agrupados em classes. Adicionando-se a palavra-chave synchronized à declaração de um método, Java garante que, uma vez iniciado qualquer thread executando aquele método, a nenhum outro thread será permitido executar qualquer outro método synchronized naquela classe.

Uma solução para o problema produtor–consumidor com base no uso de monitores em Java é mostrada na Figura 2.28. A solução é constituída de quatro classes. A classe mais externa, *ProducerConsumer*, cria e inicia dois threads, *p* e *c*. A segunda e a terceira classe, *producer* e *consumer*, respectivamente, contêm o código para o produtor e o consumidor. Por fim, a classe *our_monitor* é o monitor; ela contém dois threads sincronizados que são usados, na verdade, para inserir elementos no buffer compartilhado e tirá-los de lá. Diferentemente dos exemplos anteriores, agora mostramos o código completo para *insert* e *remove*.

Os threads produtor e consumidor são funcionalmente idênticos a seus correspondentes em todos os nossos exemplos anteriores. O produtor possui um laço infinito que gera dados e os põe no buffer comum. O consumidor tem igualmente um laço infinito, que tira dados do buffer comum e faz algo útil com ele.

A parte interessante desse programa é a classe *our_monitor*, que contém o buffer, as variáveis de administração e dois métodos sincronizados. Quando está ativo dentro do *insert*, o produtor sabe com certeza que o consumidor não pode estar ativo dentro do *remove*, tornando seguras as atualizações das variáveis e do buffer sem o temor das condições de corrida. A variável *count* controla o número de itens que estão no buffer. Ela pode assumir qualquer valor entre 0 e $N - 1$ inclusive. A variável *lo* aponta para um lugar do buffer que contém o próximo item a ser buscado. Da mesma maneira, *hi* aponta para um lugar do buffer onde o próximo item será colocado. É permitido que *lo* = *hi*, o que

```
public class ProducerConsumer {
    static final int N = 100        // constante com o tamanho do buffer
    static producer p = new producer( );   // instância de um novo thread produtor
    static consumer c = new consumer( );// instância de um novo thread consumidor
    static our_monitor mon = new our_monitor( );  // instância de um novo monitor

    public static void main(String args[ ]) {
       p.start( );       // inicia o thread produtor
       c.start( );       // inicia o thread consumidor
    }
    static class producer extends Thread {
       public void run( )  {// o método run contém o código do thread
          int item;
          while (true) {    // laço do produtor
             item = produce_item( );
             mon.insert(item);
          }
       }
       private int produce_item( ) { ... }     // realmente produz
    }
    static class consumer extends Thread {
       public void run( )  {método run contém o código do thread
          int item;
          while (true) {     // laço do consumidor
             item = mon.remove( );
             consume_item (item);
          }
       }
       private void consume_item(int item) { ... }// realmente consome
    }
    static class our_monitor {// este é o monitor
       private int buffer[ ] = new int[N];
       private int count = 0, lo = 0, hi = 0; // contadores e índices

       public synchronized void insert(int val) {
          if (count == N) go_to_sleep( );   // se o buffer estiver cheio, vá dormir
          buffer [hi] = val; // insere um item no buffer
          hi = (hi + 1) % N;        // lugar para colocar o próximo item
          count = count + 1;      // mais um item no buffer agora
          if (count == 1) notify( );      // se o consumidor estava dormindo, acorde-o
       }
       public synchronized int remove( ) {
          int val;
          if (count == 0) go_to_sleep( );    // se o buffer estiver vazio, vá dormir
          val = buffer [lo];  // busca um item no buffer
          lo = (lo + 1) % N;       // lugar de onde buscar o próximo item
          count = count – 1;     // um item a menos no buffer
          if (count == N – 1) notify( );     // se o produtor estava dormindo, acorde-o
          return val;
       }
       private void go_to_sleep( ) { try{wait( );} catch(InterruptedException exc) {};}
    }
}
```

Figura 2.28 Uma solução para o problema produtor–consumidor em Java.

significa que 0 item ou *N* itens estão no buffer. O valor de *count* indica qual desses casos ocorre.

Métodos sincronizados em Java são essencialmente diferentes dos monitores clássicos: Java não tem variáveis condicionais. Em vez disso, ela oferece dois métodos, *wait* e *notify*, equivalentes a *sleep* e *wakeup*, exceto que, quando usados dentro de métodos sincronizados, não estão sujeitos às condições de corrida. Teoricamente, o método *wait* pode ser interrompido — que é o papel do código que o envolve. Java requer que o tratamento de exceções seja explícito. Para nosso propósito, imagine apenas que *go_to_sleep* seja o caminho para ir dormir.

Tornando automática a exclusão mútua das regiões críticas, os monitores deixam a programação paralela muito menos sujeita a erros que com semáforos. Ainda assim eles também têm alguns problemas. Não é à toa que nossos dois exemplos de monitores estavam escritos em Pascal Pidgin e em Java — em vez de C, como os outros exemplos

deste livro. Conforme foi mencionado anteriormente, os monitores são um conceito de linguagem de programação. O compilador deve reconhecê-los e organizá-los para a exclusão mútua de alguma maneira. C, Pascal e a maioria das outras linguagens não possuem monitores; portanto, não é razoável esperar que seus compiladores imponham alguma regra de exclusão mútua. Na verdade, como poderia o compilador saber até mesmo quais rotinas estavam nos monitores e quais não estavam?

Essas mesmas linguagens não apresentam semáforos, mas incluí-los é fácil: tudo o que você precisa fazer é adicionar à biblioteca duas pequenas rotinas, em código de linguagem assembly, a fim de emitir as chamadas de sistema up e down. Os compiladores nem sequer precisam saber que elas existem. Obviamente, os sistemas operacionais precisam ser informados sobre os semáforos, mas, caso se tenha um sistema operacional baseado em semáforos, é possível ainda escrever os programas de usuário para ele em C ou C++ (ou até mesmo em linguagem assembly, se você for masoquista o suficiente). Para os monitores, você precisa de uma linguagem que os tenha construído.

Outro problema com monitores e também com semáforos é que eles foram projetados para resolver o problema da exclusão mútua em uma ou mais CPUs, todas com acesso a uma memória comum. Pondo os semáforos na memória compartilhada e protegendo-os com as instruções TSL, ou XCHG, podemos evitar disputas. Quando vamos para um sistema distribuído formado por múltiplas CPUs, cada qual com sua própria memória privada e conectada por uma rede local, essas primitivas tornam-se inaplicáveis. A conclusão é que os semáforos são de nível muito baixo e os monitores não são úteis, exceto para algumas linguagens de programação. Além disso, nenhuma dessas primitivas permite troca de informações entre as máquinas. Algo diferente se faz necessário.

2.3.8 Troca de mensagens

Esse algo diferente é a **troca de mensagens** (*message passing*). Esse método de comunicação entre processos usa duas primitivas, send e receive, que, assim como os semáforos mas diferentemente dos monitores, são chamadas de sistema e não construções de linguagem. Dessa maneira, elas podem ser facilmente colocadas em rotinas de biblioteca, como

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

e

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

A primeira chamada envia uma mensagem para um dado destino; a segunda recebe uma mensagem de uma dada origem (ou de uma origem qualquer, se o receptor não se importar). Se nenhuma mensagem estiver disponível, o receptor poderá ficar bloqueado até que alguma mensagem chegue. Como alternativa, ele pode retornar imediatamente acompanhado de um código de erro.

Projeto de sistemas de troca de mensagens

Sistemas de troca de mensagens apresentam muitos problemas complexos e dificuldades de projeto que não ocorrem com semáforos ou monitores — especialmente se os processos comunicantes estiverem em máquinas diferentes conectadas por uma rede. Por exemplo, as mensagens podem ser perdidas pela rede. Para se prevenir contra mensagens perdidas, o emissor e o receptor podem combinar que, assim que uma mensagem tenha sido recebida, o receptor enviará de volta uma mensagem especial de **confirmação de recebimento** (*acknowledgement*). Se o emissor não tiver recebido a confirmação de recebimento dentro de um certo intervalo de tempo, ele retransmitirá a mensagem.

Agora, pense no que acontece se a própria mensagem for recebida corretamente, mas a confirmação de recebimento tiver sido perdida. O emissor retransmitirá a mensagem e, portanto, o receptor a receberá duas vezes. É fundamental que o receptor seja capaz de distinguir entre uma mensagem nova e a retransmissão de uma mensagem antiga. Normalmente, esse problema é resolvido mediante a colocação de números em uma sequência consecutiva em cada mensagem original. Se o receptor obtém uma mensagem que carregue o mesmo número sequencial de uma mensagem anterior, ele sabe que a mensagem é uma duplicata que pode ser ignorada. A comunicação bem-sucedida diante de trocas de mensagens não confiáveis é uma importante parte do estudo sobre redes de computadores. Para mais informações, veja Tanenbaum (1996).

Os sistemas de mensagens também precisam lidar com a questão dos nomes dos processos, para que o processo especificado em uma chamada send ou receive não seja ambíguo. A **autenticação** também é um tópico de sistemas de mensagens: como o cliente pode saber que está se comunicando com o servidor de arquivos real e não com um impostor?

Na outra ponta do espectro, há ainda tópicos de projeto que são importantes quando o emissor e o receptor estão na mesma máquina. Um desses tópicos é o desempenho. Copiar mensagens de um processo para outro é sempre mais lento que realizar uma operação de semáforo ou entrar em um monitor. Muito se tem feito para tornar a troca de mensagens eficiente. Cheriton (1984), por exemplo, sugeriu um tamanho de mensagem limitado, que caiba nos registradores das máquinas, para então serem realizadas as trocas de mensagens com o uso dos registradores.

O problema produtor–consumidor com troca de mensagens

Agora vejamos como o problema produtor–consumidor pode ser resolvido com a troca de mensagens e sem qualquer memória compartilhada. Uma solução possível é mostrada na Figura 2.29. Partimos do pressuposto de que todas as mensagens são do mesmo tamanho e que as

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

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

     while (TRUE) {
          item = produce_item( );               /* 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 */
     }
}
```

Figura 2.29 O problema produtor–consumidor com *N* mensagens.

mensagens enviadas, mas ainda não recebidas, são armazenadas automaticamente pelo sistema operacional. Nessa solução, é usado um total de *N* mensagens, analogamente aos *N* lugares no buffer em uma memória compartilhada. O consumidor começa enviando *N* mensagens vazias para o produtor. Se tiver algum item para fornecer ao consumidor, o produtor pegará uma mensagem vazia e enviará de volta uma mensagem cheia. Desse modo, o número total de mensagens no sistema permanece constante com o decorrer do tempo, e assim elas podem ser armazenadas em uma quantidade de memória previamente conhecida.

Se o produtor trabalhar mais rápido que o consumidor, todas as mensagens serão preenchidas, à espera do consumidor; o produtor será bloqueado, aguardando que uma mensagem vazia volte. Se o consumidor trabalhar mais rápido, então acontecerá o inverso: todas as mensagens estarão vazias esperando que o produtor as preencha; o consumidor será bloqueado, esperando por uma mensagem cheia.

Há muitas variações possíveis do mecanismo de troca de mensagens. Para começar, observemos como as mensagens são endereçadas. Um meio para isso é atribuir a cada processo um endereço único e fazer as mensagens serem endereçadas aos processos. Um outro modo é inventar uma nova estrutura de dados, chamada **caixa postal**. Uma caixa postal é um local para armazenar temporariamente um certo número de mensagens, normalmente especificado quando ela é criada. Quando as caixas postais são usadas, os parâmetros de endereço nas chamadas send e receive são as caixas postais, não os processos. Ao tentar enviar para uma caixa postal que esteja cheia, um processo é suspenso até que uma mensagem seja removida daquela caixa postal e dê lugar a uma nova.

Para o problema produtor–consumidor, tanto o produtor quanto o consumidor criariam caixas postais suficientemente grandes para conter *N* mensagens. O produtor enviaria mensagens contendo dados à caixa postal do consumidor e este mandaria mensagens vazias para a caixa postal do produtor. O mecanismo de buffer das caixas postais é bastante simples: a caixa postal de destino contém mensagens enviadas ao processo de destino, mas ainda não aceitas.

O outro extremo das caixas postais é eliminar todo o armazenamento temporário. Quando se opta por esse caminho, se o send é emitido antes do receive, o processo emissor permanece bloqueado até que ocorra o receive, momento no qual a mensagem pode ser copiada diretamente do emissor para o receptor, sem armazenamento intermediário. Da mesma maneira, se o receive é emitido antes, o receptor é bloqueado até que ocorra um send. Essa estratégia é mais conhecida como ***rendezvous***[2]. Ela é mais fácil de implementar que um esquema de armazenamento de mensagens, mas é menos flexível, pois o emissor e o receptor são forçados a executar de maneira interdependente.

A troca de mensagens é bastante usada em sistemas de programação paralela. Um sistema de troca de men-

2. Expressão em francês para 'encontro marcado' (N.T.).

sagens bem conhecido, por exemplo, é o **MPI** (*message-passing interface* — **interface de troca de mensagem**), amplamente usado em computação científica. Para mais informações sobre ele, veja, por exemplo, Gropp et al. (1994) e Snir et al. (1996).

2.3.9 Barreiras

Nosso último mecanismo de sincronização é dirigido aos grupos de processos em vez de situações que envolvem dois processos do tipo produtor–consumidor. Algumas aplicações são divididas em fases e têm como regra que nenhum processo pode avançar para a próxima fase até que todos os processos estejam prontos a fazê-lo. Isso pode ser conseguido por meio da colocação de uma **barreira** no final de cada fase. Quando alcança a barreira, um processo permanece bloqueado até que todos os processos alcancem a barreira. A operação de uma barreira é ilustrada na Figura 2.30.

Na Figura 2.30(a), vemos quatro processos chegando a uma barreira, o que significa que eles estão apenas computando e ainda não atingiram o final da fase atual. Depois de um tempo, o primeiro processo termina toda a computação atribuída a ele para a primeira fase. Ele então executa a primitiva barrier, em geral por intermédio da chamada a uma rotina de biblioteca. O processo é, então, suspenso. Mais tarde, a primeira fase é terminada por um segundo e depois por um terceiro processo, que também executam a primitiva barrier. Essa situação é ilustrada na Figura 2.30(b). Por fim, quando o último processo, *C*, atinge a barreira, todos os processos são liberados, conforme ilustrado na Figura 2.30(c).

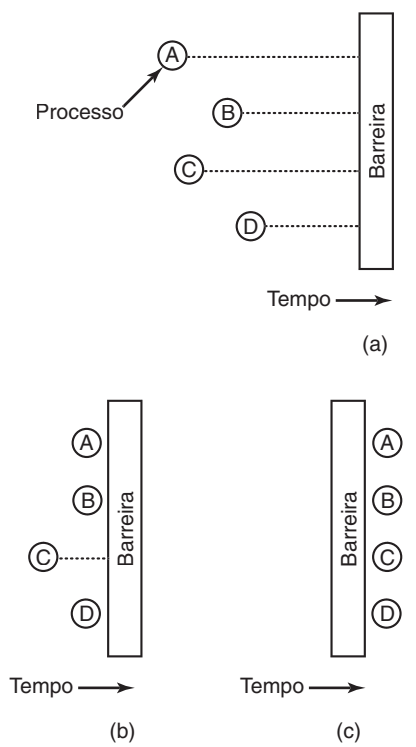


Figura 2.30 Uso de uma barreira. (a) Processos se aproximando de uma barreira. (b) Todos os processos, exceto um, estão bloqueados pela barreira. (c) Quando o último processo chega à barreira, todos passam por ela.

Como exemplo de uma situação que requer barreiras, considere um problema típico de relaxação, da física ou da engenharia. Há em geral uma matriz que contém alguns valores iniciais. Os valores podem representar temperaturas em vários pontos de uma placa de metal. O objetivo pode ser calcular quanto tempo leva para que o efeito de uma chama localizada em um canto se propague por toda a placa.

Começando com os valores atuais, uma transformação é aplicada à matriz para obter uma segunda versão da matriz — por exemplo, aplicando-se as leis da termodinâmica para verificar todas as temperaturas em um instante ΔT mais tarde. O processo é, então, repetido várias vezes e fornece as temperaturas nos pontos de amostragem como uma função do tempo, à medida que a placa é aquecida. O algoritmo produz, portanto, uma série de matrizes ao longo do tempo.

Agora, imagine que a matriz seja muito grande (digamos, um milhão por um milhão), exigindo o uso de processamento paralelo (possivelmente em um sistema multiprocessador) para aumentar a velocidade do cálculo. Processos diferentes trabalham com diferentes partes da matriz, calculando os elementos da nova matriz a partir dos valores anteriores e de acordo com as leis da física. Contudo, um processo só pode começar uma iteração $n + 1$ quando a iteração n terminar, isto é, quando todos os processos terminarem seus trabalhos atuais. O meio de chegar a esse objetivo é programar cada processo de maneira que ele execute uma operação barrier depois que terminar sua parte da iteração. Quando todos tiverem feito sua parte, a nova matriz (a entrada para a próxima iteração) estará pronta e todos os processos serão simultaneamente liberados para inicializar a próxima iteração.

2.4 Escalonamento

Quando um computador é multiprogramado, ele muitas vezes tem múltiplos processos ou threads que competem pela CPU ao mesmo tempo. Essa situação ocorre sempre que dois ou mais processos estão simultaneamente no estado pronto. Se somente uma CPU se encontrar disponível, deverá ser feita uma escolha de qual processo executará em seguida. A parte do sistema operacional que faz a escolha é chamada de **escalonador**, e o algoritmo que ele usa é o **algoritmo de escalonamento**. Esses tópicos formam o assunto das próximas seções.

Muitos dos problemas que se aplicam ao escalonamento de processos também são válidos para o escalonamento de threads, embora haja diferenças. Quando o núcleo gerencia threads, o escalonamento normalmente é feito por thread, dando pouca ou nenhuma atenção ao processo ao qual o thread pertence. Inicialmente nos concentraremos em questões de escalonamento que se aplicam tanto a processos como a threads. Em seguida estudaremos especifica-

mente o escalonamento de threads e alguns dos problemas exclusivos que suscita. Lidaremos com chips multinúcleo no Capítulo 8.

2.4.1 Introdução ao escalonamento

De volta aos velhos tempos dos sistemas em lote, com a entrada na forma de imagens de cartões em uma fita magnética, o algoritmo de escalonamento era simples: apenas execute a próxima tarefa que está na fita. Com os sistemas multiprogramados, o algoritmo de escalonamento tornou-se mais complexo porque, em geral, havia vários usuários esperando por um serviço. Alguns computadores de grande porte ainda combinam serviços em lote e de tempo compartilhado, exigindo assim que o escalonador decida se uma tarefa em lote ou um usuário interativo em um terminal deve ser atendido. (Atente para o seguinte: uma tarefa em lote pode ser uma requisição para executar uma sucessão de vários programas, mas, para esta seção, vamos supor que seja uma requisição para executar um único programa.) Como o tempo de CPU é um recurso escasso nessas máquinas, um bom escalonador pode fazer uma grande diferença no desempenho observado e na satisfação do usuário. Consequentemente, muito se fez tendo em vista desenvolver algoritmos de escalonamento inteligentes e eficientes.

Com o advento dos computadores pessoais, a situação mudou de duas maneiras. Primeiro, na maior parte do tempo existe apenas um processo ativo. É improvável que um usuário esteja, simultaneamente, entrando com um documento em um processador de textos e compilando um programa em segundo plano. Quando o usuário digita um comando para o processador de textos, o escalonador não precisa trabalhar muito para perceber qual processo executar — o processador de textos é o único candidato.

Em segundo lugar, os computadores, com o passar dos anos, ficaram tão mais rápidos que a CPU raramente chegará a ser um recurso escasso. A maioria dos programas para computadores pessoais é limitada pela velocidade com que o usuário pode entrar dados (digitando ou clicando), e não pela taxa na qual a CPU é capaz de processá-los. Até os compiladores — grandes consumidores de ciclos de CPU no passado — atualmente levam, no máximo, alguns segundos. Mesmo quando dois programas estiverem executando de modo simultâneo — como um processador de textos e uma planilha —, dificilmente importará qual começa primeiro, já que o usuário estará esperando, provavelmente, que ambos terminem. Como consequência, o escalonamento não é tão importante em PCs simples. Claro, há aplicações que praticamente esgotam a CPU: exibir uma hora de vídeo de alta resolução pode exigir processamento de imagens de altíssima capacidade para lidar com cada um dos 108 mil quadros em NTSC (90 mil no PAL), mas essas aplicações são exceções, não a regra.

Quando nos concentramos em servidores e estações de trabalho de alto desempenho em rede, a situação muda. Nesse caso, é comum haver múltiplos processos competindo pela CPU, e, portanto, o escalonamento torna-se importante novamente. Por exemplo, quando a CPU precisar decidir entre executar um processo que reúne estatísticas diárias e um que atende às solicitações dos usuários, estes ficarão muito mais satisfeitos se o último tiver precedência na CPU.

Além de escolher o processo certo para executar, o escalonador também deve se preocupar em fazer um uso eficiente da CPU, pois chavear processos é muito custoso. De início, deve ocorrer um chaveamento do modo de usuário para o modo núcleo. Depois, o estado atual do processo deve ser salvo, armazenando-se inclusive seus registradores na tabela de processos, para que possam ser recarregados posteriormente. Em muitos sistemas, o mapa de memória (por exemplo, os bits de referência à memória na tabela de páginas) também deve ser salvo. Em seguida, um novo processo precisa ser selecionado pela execução do algoritmo de escalonamento. Depois disso, a MMU (*memory management unit* — unidade de gerenciamento de memória) tem de ser recarregada com o mapa de memória do novo processo. Por fim, o novo processo precisa ser iniciado. Além disso tudo, o chaveamento do processo normalmente invalida toda a memória cache, forçando-a a ser dinamicamente recarregada da memória principal por duas vezes (ao entrar no núcleo e ao sair dele). De modo geral, realizar muitos chaveamentos de processos por segundo pode comprometer uma grande quantidade do tempo de CPU; portanto, todo cuidado é pouco.

Comportamento do processo

Quase todos os processos alternam surtos de computação com requisições de E/S (de disco), conforme mostra a Figura 2.31. Em geral, a CPU executa indefinidamente e então é feita uma chamada de sistema para ler de um arquivo ou escrever nele. Quando a chamada de sistema termina, a CPU computa novamente até que ela requisite ou tenha de escrever mais dados, e assim continua. Perceba que algumas atividades de E/S contam como computação. Por exemplo, quando a CPU copia bits para uma RAM de vídeo a fim de atualizar a tela, ela está computando, não fazendo E/S, pois a CPU se encontra em uso. E/S, nesse sentido, é o que ocorre quando um processo entra no estado bloqueado esperando que um dispositivo externo termine o que está fazendo.

O que é importante observar na Figura 2.31 é que alguns processos, como os da Figura 2.31(a), gastam a maior parte do tempo computando, enquanto outros, como os da Figura 2.31(b), passam a maior parte de seu tempo esperando E/S. Os primeiros são chamados **limitados pela CPU** (*compute-bound* ou *CPU-bound*); os últimos são os **limitados pela E/S** (*I/O-bound*). Os processos limitados pela CPU apresentam, em geral, longos surtos de uso da CPU e esporádicas esperas por E/S; já os processos limitados por E/S têm pequenos surtos de uso da CPU e esperas frequentes

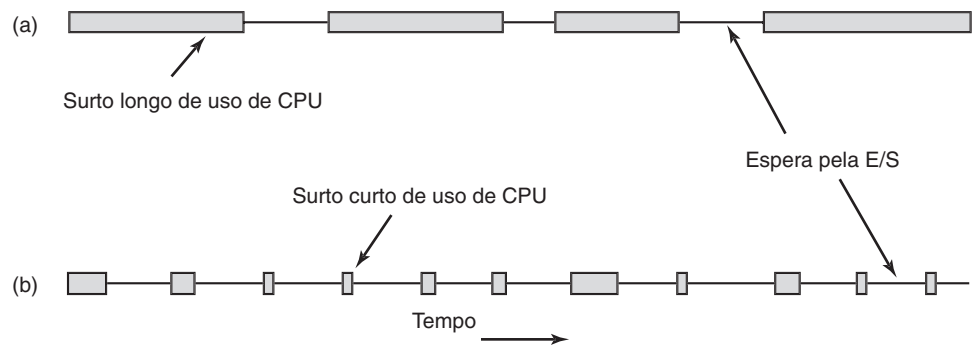


Figura 2.31 Usos de surtos de CPU se alternam com períodos de espera por E/S. (a) Um processo orientado à CPU. (b) Um processo orientado à E/S.

por E/S. Note que o fator principal é o tamanho do surto de CPU, não o tamanho do surto de E/S. Os processos orientados à E/S são assim chamados porque, entre uma requisição e outra por E/S, eles não realizam muita computação, não porque tenham requisições por E/S especialmente demoradas. O tempo para a leitura de um bloco de disco é sempre o mesmo, independentemente do quanto demore processar os dados que chegam depois.

Convém observar que, à medida que as CPUs se tornam mais rápidas, os processos tendem a ficar mais limitados por E/S. Esse efeito ocorre porque as CPUs estão ficando muito mais rápidas que os discos. Como consequência, o escalonamento de processos orientados à E/S deverá ser um assunto mais importante no futuro. A ideia básica é que, se um processo orientado à E/S quiser executar, essa oportunidade deve ser rapidamente dada a ele, pois assim ele executará suas requisições de disco, mantendo o disco ocupado. Como vimos na Figura 2.4, quando os processos são orientados à E/S, são necessários alguns deles para manter a CPU totalmente ocupada.

Quando escalonar

Um tópico fundamental, relacionado ao escalonamento, é o momento certo de tomar as decisões de escalonar. É claro que há uma variedade de situações nas quais o escalonamento é necessário. Primeiro, quando se cria um novo processo, é necessário tomar uma decisão entre executar o processo pai ou o processo filho. Como ambos os processos estão no estado pronto, essa é uma decisão normal de escalonamento e pode levar à escolha de um ou de outro — isto é, o escalonador pode escolher legitimamente executar o pai ou o filho.

Em segundo lugar, uma decisão de escalonamento deve ser tomada ao término de um processo. Como o processo não pode executar mais (já que ele não existe mais), algum outro processo deve ser escolhido entre os processos prontos. Se nenhum processo estiver pronto, é executado um processo ocioso gerado pelo sistema.

Em terceiro lugar, quando um processo bloqueia para E/S, sobre um semáforo ou por alguma outra razão, outro processo precisa ser selecionado para executar. O motivo do bloqueio pode, algumas vezes, influenciar na escolha. Por exemplo, se *A* for um processo importante e estiver esperando *B* sair de sua região crítica, deixar *B* executar em seguida permitirá que ele saia de sua região crítica e, portanto, permite que *A* continue. O problema, contudo, é que geralmente o escalonador não possui a informação necessária para considerar essa dependência.

Em quarto lugar, quando ocorre uma interrupção de E/S, pode-se tomar uma decisão de escalonamento. Se a interrupção vem de um dispositivo de E/S que acabou de fazer seu trabalho, o processo que estava bloqueado, esperando pela E/S, pode agora ficar pronto para execução. É o escalonador quem decide se executa o processo que acabou de ficar pronto, o processo que estava executando no momento da interrupção ou algum terceiro processo.

Se um hardware de relógio fornece interrupções periódicas a 50 Hz, 60 Hz ou alguma outra frequência, uma decisão de escalonamento pode ser tomada a cada interrupção de relógio ou a cada *k*-ésima interrupção de relógio. Os algoritmos de escalonamento podem ser divididos em duas categorias quanto ao modo como tratam essas interrupções. Um algoritmo de escalonamento **não preemptivo** escolhe um processo para executar e, então, o deixa executar até que seja bloqueado (à espera de E/S ou de um outro processo) ou até que ele voluntariamente libere a CPU. Mesmo que ele execute por horas, não será compulsoriamente suspenso. Na verdade, nenhuma decisão de escalonamento é tomada durante as interrupções de relógio. Depois que o processamento da interrupção de relógio termina, o processo que estava executando antes da interrupção prossegue até acabar, a menos que um processo de prioridade mais alta esteja esperando por um tempo de espera agora satisfeito.

Por outro lado, um algoritmo de escalonamento **preemptivo** escolhe um processo e o deixa em execução por um tempo máximo fixado. Se ainda estiver executando ao final desse intervalo de tempo, o processo será suspenso e o escalonador escolherá outro processo para executar (se houver algum disponível). O escalonamento preemptivo

requer a existência de uma interrupção de relógio ao fim do intervalo de tempo para que o controle sobre a CPU seja devolvido ao escalonador. Se não houver relógio disponível, o escalonamento não preemptivo será a única opção.

Categorias de algoritmos de escalonamento

É claro que, para ambientes diferentes, são necessários diferentes algoritmos de escalonamento. Essa situação ocorre porque diferentes áreas de aplicação (e diferentes tipos de sistemas operacionais) têm objetivos diferentes. Em outras palavras, o que deve ser otimizado pelo escalonador não é o mesmo para todos os sistemas. Três ambientes merecem distinção:

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

Os sistemas em lote ainda são amplamente utilizados pelas empresas para folhas de pagamento, estoque, contas a receber, contas a pagar, cálculo de juros (em bancos), processamento de pedidos de indenização (em companhias de seguros) e outras tarefas periódicas. Nos sistemas em lote não há, em seus terminais, usuários esperando impacientes por uma resposta rápida. Consequentemente, os algoritmos não preemptivos ou preemptivos com longo intervalo de tempo para cada processo são, em geral, aceitáveis. Essa tática reduz os chaveamentos entre processos e assim melhora o desempenho. Na verdade, os algoritmos de lote são bastante comuns e muitas vezes aplicáveis a outras situações também, o que torna importante estudá-los, até para pessoas que não estejam envolvidas em computação central corporativa.

Em um ambiente com usuários interativos, a preempção é essencial para evitar que um processo se aposse da CPU e, com isso, negue serviço aos outros. Mesmo que nenhum processo execute intencionalmente para sempre, uma falha em um programa pode levar um processo a impedir indefinidamente que todos os outros executem. A preempção é necessária para impedir esse comportamento. Os servidores também caem nessa categoria, visto que normalmente servem a usuários (remotos) múltiplos, todos muito apressados.

Em sistemas com restrições de tempo real, a preempção é, estranhamente, algumas vezes desnecessária, pois os processos sabem que não podem executar por longos períodos e, em geral, fazem seus trabalhos e bloqueiam rapidamente. A diferença com relação aos sistemas interativos é que os sistemas de tempo real executam apenas programas que visam ao progresso da aplicação. Já os sistemas interativos são de propósito geral e podem executar programas arbitrários não cooperativos ou até mal-intencionados.

Objetivos do algoritmo de escalonamento

Para projetar um algoritmo de escalonamento, é necessário ter alguma ideia do que um bom algoritmo deve fazer. Alguns objetivos dependem do ambiente (lote, interativo ou tempo real), mas há também aqueles que são desejáveis para todos os casos. Alguns objetivos são relacionados na Tabela 2.8. Discutiremos isso logo a seguir.

Em qualquer circunstância, justiça é algo importante. Processos semelhantes devem ter serviços semelhantes. Não é justo dar mais tempo de CPU a um processo do que a outro equivalente. É claro que categorias diferentes de processos podem ser tratadas de modo muito diverso. Pense no controle de segurança e na folha de pagamento de um centro de computação de uma usina nuclear.

De alguma maneira relacionada à justiça está o cumprimento das políticas do sistema. Se a política local estabelece que os processos do controle de segurança executam quando quiserem, mesmo que isso signifique que a folha de pagamento atrase 30 segundos, o escalonador deve assegurar que essa política seja cumprida.

Todos os sistemas
Justiça — dar a cada processo uma porção justa da CPU
Aplicação da política — verificar se a política estabelecida é cumprida
Equilíbrio — manter ocupadas todas as partes do sistema
Sistemas em lote
Vazão (*throughput*) — maximizar o número de tarefas por hora
Tempo de retorno — minimizar o tempo entre a submissão e o término
Utilização de CPU — manter a CPU ocupada o tempo todo
Sistemas interativos
Tempo de resposta — responder rapidamente às requisições
Proporcionalidade — satisfazer às expectativas dos usuários
Sistemas de tempo real
Cumprimento dos prazos — evitar a perda de dados
Previsibilidade — evitar a degradação da qualidade em sistemas multimídia

Tabela 2.8 Alguns objetivos do algoritmo de escalonamento sob diferentes circunstâncias.

Outro objetivo geral é manter, quando possível, todas as partes do sistema ocupadas. Se a CPU e os demais dispositivos de E/S puderem ser mantidos em execução o tempo todo, mais trabalho por segundo será feito do que se algum dos componentes estiver ocioso. Em um sistema em lote, por exemplo, o escalonador tem o controle de quais tarefas são trazidos para a memória para executar. É melhor ter juntos na memória alguns processos limitados pela CPU e outros limitados por E/S do que carregar todas as tarefas limitadas pela CPU primeiro e quando terminarem, carregar e executar todas as tarefas limitadas por E/S. Se a última estratégia for usada, quando os processos orientados à CPU estiverem executando, disputarão a CPU e, assim, o disco ficará ocioso. Em seguida, quando as tarefas limitadas por E/S executarem, disputarão o disco e a CPU se encontrará ociosa. É melhor manter o sistema todo executando de uma vez formulando-se cuidadosamente essa mistura de processos.

Os gerentes de grandes centros de computação — que executam muitas tarefas em lote — observam, em geral, três métricas para verificar se os sistemas deles estão executando bem ou não: vazão, tempo de retorno e utilização da CPU. **Vazão** é o número de tarefas por hora que o sistema termina. Considerando tudo o que foi discutido, terminar 50 tarefas por hora é melhor do que terminar 40 no mesmo período. O **tempo de retorno** é estatisticamente o tempo médio do momento em que uma tarefa em lote é submetido até o momento em que ele é terminado. Ele indica quanto tempo, em média, o usuário tem de esperar pelo fim de um trabalho. Aqui a regra é: quanto menor, melhor.

Um algoritmo de escalonamento que maximize a vazão pode não necessariamente minimizar o tempo de retorno. Por exemplo, dada uma mistura de tarefas curtas e longas, um escalonador que execute sempre tarefas curtas e nunca tarefas longas pode conseguir uma excelente vazão (muitas tarefas curtas por hora), mas à custa de um enorme tempo de retorno para as tarefas longas. Se as tarefas curtas mantiverem uma taxa de chegada constante, as tarefas longas poderão nunca executar, tornando o tempo médio de retorno infinito, embora atingindo alta vazão.

A utilização da CPU também é muitas vezes usada como parâmetro em sistemas em lote. Na verdade, esse não é um bom parâmetro. O que realmente interessa é quantas tarefas por hora saem do sistema (vazão) e quanto tempo leva para receber o resultado do trabalho (tempo de retorno). Tomar a utilização da CPU como medida é o mesmo que avaliar um carro pelo número de giros que seu motor dá a cada hora. Por outro lado, saber quando a utilização da CPU está se aproximando de 100 por cento é útil para identificar o momento de obter mais potência para o computador.

Para sistemas interativos, aplicam-se objetivos diferentes. O que importa é minimizar o **tempo de resposta**, isto é, o tempo entre a emissão de um comando e a obtenção do resultado. Em um computador pessoal, no qual está sendo executado um processo em segundo plano (por exemplo, lendo e armazenando mensagens de correio eletrônico da rede), uma requisição de usuário, para inicializar um programa ou abrir um arquivo, deveria ter precedência sobre o trabalho em segundo plano. Atender antes a todas as requisições interativas será considerado um bom serviço.

Uma questão relacionada a esse tópico é a chamada **proporcionalidade**. Os usuários têm uma intuição (mas muitas vezes errada) de quanto tempo as coisas devem durar. Quando uma requisição tida como complexa demora, os usuários aceitam isso, mas, quando a demora ocorre com uma requisição considerada simples, os usuários ficam irritados. Por exemplo, se, ao clicar em um ícone que inicia o envio de um fax, são necessários 60 segundos para concluí-lo, o usuário provavelmente encarará isso como inevitável porque não espera que um fax seja enviado em cinco segundos.

Por outro lado, quando o usuário clicar em um ícone para interromper a conexão telefônica após o envio do fax, ele tem expectativas diferentes. Se não tiver sido concluído após 30 segundos, o usuário provavelmente começará a reclamar e, após 60 segundos, estará espumando de raiva. Esse comportamento é causado pela inevitável comparação que o usuário faz que realizar uma chamada telefônica e passar um fax deveria demorar muito mais do que desligar o telefone. Em alguns casos (como esse), o escalonador não pode fazer nada com relação ao tempo de resposta, mas em outros casos sim, especialmente naqueles em que o atraso é decorrente de uma má escolha da ordem dos processos.

Sistemas de tempo real têm propriedades diferentes dos sistemas interativos e, portanto, objetivos diferentes. Eles são caracterizados por prazos que devem — ou pelo menos deveriam — ser cumpridos. Por exemplo, em um computador encarregado de controlar um dispositivo que produz dados a uma taxa constante, uma falha ao executar o processo de coleta de dados em tempo hábil pode resultar na perda de dados. Assim, a principal exigência de um sistema de tempo real é cumprir todos os prazos (ou a maior parte deles).

Em alguns sistemas de tempo real, especialmente naqueles que envolvem multimídia, a previsibilidade é importante. Deixar de cumprir um prazo ocasional não é fatal, mas, se o processo de áudio, por exemplo, executar erraticamente, a qualidade do som vai deteriorar rápido. O vídeo também é um problema, mas o ouvido é muito mais sensível a atrasos que a visão. Para evitar esse problema, o escalonamento de processos deve ser altamente previsível e regular. Ainda neste capítulo, estudaremos os algoritmos de escalonamento em lote e interativos, mas adiaremos a maior parte de nosso estudo sobre o escalonamento em tempo real para o Capítulo 7, que trata de sistemas operacionais multimídia.

2.4.2 Escalonamento em sistemas em lote

É chegada a hora de passar dos tópicos gerais de escalonamento para os algoritmos específicos desse processo. Nesta seção, estudaremos os algoritmos usados em sistemas em lote. Em seguida, veremos sistemas interativos e de tempo real. Convém ressaltar que alguns algoritmos são usados tanto em sistemas em lote quanto em sistemas interativos. Esses serão estudados depois. Agora, o foco será mantido sobre algoritmos adequados somente a sistemas em lote.

Primeiro a chegar, primeiro a ser servido

Provavelmente o mais simples algoritmo de escalonamento seja o não preemptivo **primeiro a chegar, primeiro a ser servido** (*first come, first served* — **FCFS**). Com esse algoritmo, a CPU é atribuída aos processos na ordem em que eles a requisitam. Basicamente, há uma fila única de processos prontos. Quando a primeira tarefa entra no sistema, logo quando chega de manhã, é iniciado imediatamente e autorizado a executar por quanto tempo queira. Ele não é interrompido porque está sendo executado há muito tempo. À medida que chegam as outras tarefas, eles são encaminhados para o fim da fila. Quando o processo em execução é bloqueado, o primeiro processo na fila é o próximo a executar. Quando um processo bloqueado fica pronto — assim como uma tarefa que acabou de chegar —, ele é posto no fim da fila.

A grande vantagem desse algoritmo é que ele é fácil de entender e igualmente fácil de programar. É também um algoritmo justo, assim como é justo destinar escassos ingressos para eventos esportivos ou musicais para as pessoas que estejam dispostas a ficar na fila desde as 2 da madrugada. Com esse algoritmo, uma única lista encadeada controla todos os processos prontos. Adicionar uma nova tarefa ou um processo desbloqueado requer apenas a inserção dele no final da fila. O que poderia ser mais simples de entender e implementar?

Infelizmente, o algoritmo *primeiro a chegar, primeiro a ser servido* apresenta uma grande desvantagem. Imagine um processo orientado à computação que execute durante um segundo por vez e muitos outros processos limitados por E/S que usem pouco tempo de CPU, mas que precisem realizar, cada um, mil leituras de disco antes de terminar. O processo orientado à computação executa por um segundo e então lê um bloco de disco (bloqueia). Com esse processo bloqueado à espera de E/S, todos os outros processos limitados por E/S executam e iniciam as leituras de disco. Quando o processo orientado à computação obtém seu bloco de dados (desbloqueia), ele executa por mais um segundo, seguido novamente por todos os processos limitados por E/S, em uma rápida sucessão.

O resultado líquido é que cada um dos processos limitados por E/S lê um bloco por segundo e, portanto, demorará mil segundos para terminar. Com um algoritmo de escalonamento que causasse a preempção do processo orientado à computação a cada dez milissegundos (em vez de a cada um segundo), os processos de E/S terminariam em dez segundos, e não em mil segundos, sem atrasar tanto o processo orientado à computação.

Tarefa mais curta primeiro

Vejamos um outro algoritmo em lote não preemptivo que supõe como previamente conhecidos todos os tempos de execução. Em uma companhia de seguros, por exemplo, as pessoas podem prever, com bastante precisão, quanto tempo será necessário para executar um lote de mil solicitações, já que um trabalho similar é feito todos os dias. Quando várias tarefas igualmente importantes estiverem postados na fila de entrada à espera de serem iniciados, o escalonador escolhe a **tarefa mais curto primeiro** (*shortest job first*). Veja a Figura 2.32. Nela encontramos quatro tarefas — *A*, *B*, *C* e *D* — com seus respectivos tempos de execução — 8, 4, 4 e 4 minutos, respectivamente. Ao executá-los nessa ordem, o tempo de retorno para *A* é de oito minutos, para *B* é de 12 minutos, para *C* é de 16 minutos e para *D* é de 20 minutos, o que resulta em uma média de 14 minutos.

Consideremos agora a execução desses quatro tarefas a partir do algoritmo *tarefa mais curta primeiro*, conforme ilustrado na Figura 2.32(b). Os tempos de retorno são agora 4, 8, 12 e 20 minutos, com uma média de 11 minutos. A *tarefa mais curta primeiro* parece ótimo, não? Considere o caso de quatro tarefas, com tempos de execução *a*, *b*, *c* e *d*, respectivamente. A primeira tarefa termina no tempo *a*, o segundo termina no tempo $a + b$ e assim por diante. O tempo médio de retorno é $(4a + 3b + 2c + d)/4$. É claro que *a* contribui mais para a média que os outros tempos; portanto, ele deveria ser a tarefa mais curto, com o *b* depois, então o *c* e, por fim, o *d* — sendo este o mais demorado e que afeta somente seu próprio tempo de retorno. O mesmo argumento se aplica igualmente bem a qualquer número de tarefas.

Convém observar que a *tarefa mais curta primeiro* é adequado somente para situações em que todas as tarefas estejam disponíveis simultaneamente. Como um contraexemplo,

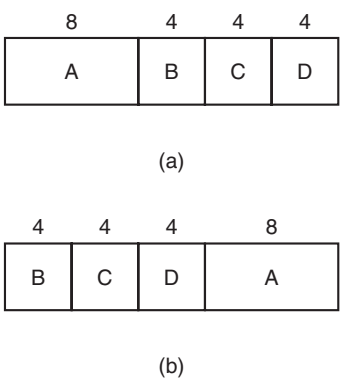


Figura 2.32 Um exemplo do escalonamento *tarefa mais curta primeiro*. (a) Execução de quatro tarefas na ordem original. (b) Execução na ordem *tarefa mais curta primeiro*.

considere cinco tarefas, de *A* a *E*, com tempos de execução 2, 4, 1, 1 e 1, respectivamente. Seus tempos de chegada são 0, 0, 3, 3 e 3. De início, somente *A* ou *B* podem ser escolhidas, já que os outras três tarefas ainda não chegaram. Usando a *tarefa mais curto primeiro*, executaremos as tarefas na ordem *A*, *B*, *C*, *D*, *E* para um tempo médio de espera de 4,6. Contudo, executá-los na ordem *B*, *C*, *D*, *E*, *A* implica um tempo médio de espera de 4,4.

Próximo de menor tempo restante

Uma versão preemptiva da *tarefa mais curta primeiro* é o **próximo de menor tempo restante** (*shortest remaining time next*). Com esse algoritmo, o escalonador sempre escolhe o processo cujo tempo de execução restante seja o menor. Novamente, o tempo de execução deve ser previamente conhecido. Quando chega uma nova tarefa, seu tempo total é comparado ao tempo restante do processo em curso. Se, para terminar, a nova tarefa precisar de menos tempo que o processo corrente, então esse será suspenso e a nova tarefa será iniciado. Esse esquema permite que novas tarefas curtos obtenham um bom desempenho.

2.4.3 Escalonamento em sistemas interativos

Vejamos então alguns algoritmos aplicados a sistemas interativos. Eles são comuns em computadores pessoais, servidores e outros tipos de sistemas também.

Escalonamento por chaveamento circular (*round-robin*)

Um dos algoritmos mais antigos, simples, justos e amplamente usados é o **circular**. A cada processo é atribuído um intervalo de tempo, o seu **quantum**, no qual ele é permitido executar. Se, ao final do quantum, o processo ainda estiver executando, a CPU sofrerá preempção e será dada a outro processo. Se o processo foi bloqueado ou terminou antes que o quantum tenha decorrido, a CPU é chaveada para outro processo. O escalonamento circular é fácil de implementar. O escalonador só precisa manter uma lista de processos executáveis, conforme mostra a Figura 2.33(a). Quando o processo usa todo o seu quantum, ele é colocado no final da lista, como mostra a Figura 2.33(b).

O que interessa para o escalonamento circular é o tamanho do quantum. O chaveamento de um processo para outro requer uma certa quantidade de tempo para sua administração — salvar e carregar registradores e mapas de memória, atualizar várias listas e tabelas, carregar e descarregar a memória cache etc. Suponha que esse **chaveamento de processo** — ou **chaveamento de contexto**, como é algumas vezes chamado — dure 1 ms, incluindo o chaveamento dos mapas de memória, descarga e recarga da cache etc. Suponha também que o quantum seja de 4 ms. Com esses parâmetros, depois de fazer 4 ms de trabalho útil, a CPU

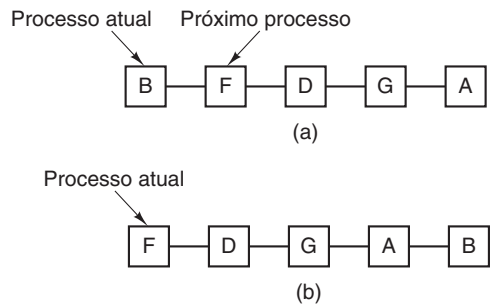


Figura 2.33 Escalonamento circular (*round robin*). (a) Lista de processos executáveis. (b) Lista de processos executáveis depois que *B* usou todo o seu quantum.

terá de gastar (ou melhor, desperdiçar) 1 ms para chavear o processo. Nesse exemplo, 20 por cento do tempo de CPU será gasto em administração, o que sem dúvida é demais.

Para melhorar a eficiência da CPU, poderíamos estabelecer o valor do quantum em, digamos, 100 ms. Agora, o tempo gasto é de apenas 1 por cento. No entanto, considere o que pode acontecer em um sistema de tempo compartilhado se 50 solicitações forem feitas dentro de um curto intervalo de tempo e com grande variação nas necessidades de CPU. Cinquenta processos serão colocados na lista de processos executáveis. Se a CPU estiver ociosa, o primeiro dos processos inicializará imediatamente, o segundo não poderá inicializar enquanto não se passarem 100 ms e assim por diante. O último azarado pode ter de esperar cinco segundos antes de ter uma oportunidade — supondo que todos os outros usem inteiramente seus quanta[3]. A maioria dos usuários verá como problema uma resposta se um pequeno comando demorar cinco segundos. Essa situação é especialmente ruim se alguma das solicitações próximas ao fim da fila exigir apenas alguns milissegundos de tempo da CPU. Com um quantum curto, os usuários teriam obtido o melhor serviço.

Outro fator é o seguinte: se o quantum for maior que o surto médio de CPU, a preempção raramente ocorrerá. Na verdade, a maior parte dos processos bloqueará antes que o quantum acabe, causando um chaveamento de processo. Eliminar a preempção melhora o desempenho porque o chaveamento de processo somente ocorre quando é logicamente necessário, isto é, quando um processo bloqueia e não é mais capaz de continuar.

A conclusão pode ser formulada assim: adotar um quantum muito curto causa muitos chaveamentos de processo e reduz a eficiência da CPU, mas um quantum muito longo pode gerar uma resposta pobre às requisições interativas curtas. Um quantum em torno de 20 ms a 50 ms é bastante razoável.

Escalonamento por prioridades

O escalonamento circular pressupõe que todos os processos sejam igualmente importantes. É frequente as pes-

3. Quanta = plural de quantum (N. R. T.).

soas que possuem e operam computadores multiusuário pensarem de modo diferente sobre o assunto. Em uma universidade, por exemplo, uma ordem hierárquica seria encabeçada pelo reitor, e então viriam os professores, os secretários, os porteiros e finalmente os estudantes. Da necessidade de se considerarem fatores externos resulta o **escalonamento por prioridades**. A ideia básica é simples: a cada processo é atribuída uma prioridade, e ao processo executável com a prioridade mais alta é permitido executar.

Mesmo em um PC com um único proprietário, pode haver múltiplos processos, alguns mais importantes que outros. Por exemplo, a um processo daemon, que envia mensagens de correio eletrônico em segundo plano, deve ser atribuída uma prioridade mais baixa que a um processo que exibe um vídeo na tela em tempo real.

Para evitar que processos de alta prioridade executem indefinidamente, o escalonador pode reduzir a prioridade do processo em execução a cada tique de relógio (isto é, a cada interrupção de relógio). Se isso fizer com que sua prioridade caia abaixo da prioridade do próximo processo com prioridade mais alta, então ocorrerá um chaveamento de processo. Outra possibilidade é atribuir a cada processo um quantum máximo no qual ele pode executar. Quando esse quantum estiver esgotado, será dada a oportunidade para que o próximo processo com prioridade mais alta execute.

Prioridades podem ser atribuídas aos processos estática ou dinamicamente. Em um computador militar, os processos iniciados por generais podem partir com prioridade em 100; os processos iniciados por coronéis, em 90; os de majores, em 80; os de capitães, em 70; os de tenentes, em 60, e assim por diante. De outra forma, em um centro de computação comercial, trabalhos de alta prioridade podem custar cem dólares por uma hora; um de prioridade média, 75 dólares por hora; e os de prioridade baixa, 50 dólares pelo mesmo período. O sistema UNIX tem um comando, *nice*, que permite que um usuário reduza voluntariamente a prioridade de seu processo e, assim, seja gentil com os outros usuários. Usuários nunca o utilizam.

O sistema também pode atribuir dinamicamente as prioridades para atingir certos objetivos. Por exemplo, alguns processos são altamente orientados à E/S e gastam a maior parte de seu tempo esperando que uma E/S termine. Se um processo como esse quisesse a CPU, deveria recebê-la imediatamente, para deixá-lo inicializar sua próxima requisição de E/S, a qual poderia então continuar em paralelo com outro processo que estivesse realmente computando. Fazer o processo orientado à E/S esperar um longo tempo pela CPU significa tê-lo ocupando a memória por tempo demais desnecessariamente. Um algoritmo simples e que funciona bem para processos orientados à E/S é atribuir $1/f$ à prioridade, sendo f a fração do último quantum que o processo usou. Um processo que tivesse utilizado somente 1 ms de seu quantum de 50 ms obteria prioridade 50, enquanto um processo que executa 25 ms antes de bloquear teria prioridade 2, e um processo que houvesse consumido todo o quantum teria prioridade 1.

Muitas vezes é conveniente agrupar processos em classes de prioridade e usar o escalonamento por prioridades entre as classes — contudo, dentro de cada classe, usar o escalonamento circular. A Figura 2.34 mostra um sistema com quatro classes de prioridade. O algoritmo de escalonamento é o seguinte: enquanto houver processos executáveis na classe de prioridade 4, execute apenas um por quantum usando escalonamento circular e nunca perca tempo com as classes de baixa prioridade. Se a classe de prioridade 4 estiver vazia (sem processos para executar), então execute os processos da classe 3 em chaveamento circular. Se as classes 4 e 3 estiverem ambas vazias, então execute a classe 2 em chaveamento circular, e assim por diante. Se as prioridades não forem ocasionalmente ajustadas, as classes de prioridade mais baixas poderão todas morrer de fome.

Filas múltiplas

Um dos primeiros escalonadores por prioridades foi implementado no CTSS, o sistema compatível de tempo compartilhado do MIT que operava no IBM 7094 (Corbató et al., 1962). O CTSS tinha um problema: o chaveamento de processo era muito lento porque o 7094 só podia manter na memória um processo por vez. Cada chaveamento significava trocar todo o processo, ou seja, enviá-lo para o disco e ler outro do disco. Os projetistas do CTSS logo perceberam que era mais eficiente dar, de vez em quando, um quantum grande para os processos limitados pela CPU do que fornecer frequentemente um quantum pequeno (para reduzir as operações de troca entre o disco e a memória). Por outro lado, dar a todos os processos um quantum grande significaria ter um tempo de resposta inadequado, conforme vimos. A solução, então, foi definir classes de prioridade. Os processos na classe de prioridade mais alta eram executados por um quantum. Os processos na classe seguinte de prioridade mais alta executavam por dois quanta. Os processos na próxima classe executavam por quatro quanta e assim por diante. Se um processo utilizasse todos os seus quanta, seria movido para uma classe inferior.

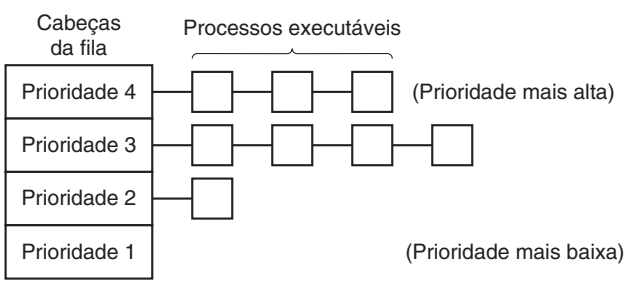


Figura 2.34 Um algoritmo de escalonamento com quatro classes de prioridade.

Como exemplo, imagine um processo que precisasse computar continuamente por 100 quanta. Inicialmente, a ele seria dado um quantum, e ele então seria levado da memória para o disco (troca para o disco). Na vez seguinte ele teria dois quanta antes de ocorrer a troca para o disco. As próximas execuções obteriam 4, 8, 16, 32 e 64 quanta, embora ele tivesse usado apenas 37 dos últimos 64 quanta para realizar seu trabalho. Seriam necessárias somente sete trocas entre a memória e o disco (incluindo a carga inicial), em vez de cem para um algoritmo puramente circular. Além disso, à medida que o processo se aprofundasse mais nas filas de prioridade, ele seria cada vez menos frequentemente executado, liberando a CPU para processos interativos e rápidos.

Foi então adotada a seguinte política para impedir uma longa punição a um processo que, quando iniciado pela primeira vez, precisasse executar por um longo intervalo de tempo, mas que depois se tornasse interativo. Se fosse digitado um <Enter> em um terminal, o processo pertencente àquele terminal era movido para a classe de prioridade mais alta, na suposição de que ele estivesse prestes a se tornar interativo. Certo dia, algum usuário com um processo pesadamente limitado pela CPU descobriu que, sentando ao terminal e digitando <Entra> de maneira aleatória e a intervalos de poucos segundos, poderia fazer maravilhas por seu tempo de resposta. Ele contou isso para todos os seus amigos. Moral da história: conseguir acertar na prática é muito mais difícil que acertar na teoria.

Muitos outros algoritmos foram usados para atribuir processos a classes de prioridade. Por exemplo, o influente sistema XDS 940 (Lampson, 1968), construído em Berkeley, possuía quatro classes de prioridade: terminal, E/S, quantum curto e quantum longo. Quando um processo que estivesse esperando pela entrada de um terminal finalmente acordasse, ele iria para classe de prioridade mais alta (terminal). Quando um processo bloqueado pelo disco ficasse pronto, ele iria para a segunda classe. Se, enquanto um processo ainda estivesse executando, seu quantum acabasse, seria inicialmente alocado na terceira classe. Contudo, se um processo terminasse seu quantum várias vezes sem ser bloqueado pelo terminal ou por outra E/S, iria para a última fila. Muitos outros sistemas usam algo semelhante para favorecer os usuários interativos mais do que os processos em segundo plano.

Próximo processo mais curto (*shortest process next*)

Como a *tarefa mais curta primeiro* sempre resulta no mínimo tempo médio de resposta para sistemas em lote, seria bom se ele também pudesse ser usado para processos interativos. Até certo ponto, isso é possível. Processos interativos geralmente seguem o padrão de esperar por comando, executar comando, esperar por comando, executar comando e assim por diante. Se víssemos a execução de cada comando como uma 'tarefa' isolado, então poderíamos minimizar o tempo de resposta geral executando a *tarefa mais curta primeiro*. O único problema é saber qual dos processos atualmente executáveis é o mais curto.

Uma saída é realizar uma estimativa com base no comportamento passado e, então, executar o processo cujo tempo de execução estimado seja o menor. Suponha que o tempo estimado por comando para algum terminal seja T_0 e que sua próxima execução seja medida como T_1. Poderíamos atualizar nossa estimativa tomando uma soma ponderada desses dois números, isto é, $aT_0 + (1 - a)T_1$. Pela escolha de a, podemos decidir se o processo de estimativa esquecerá rapidamente as execuções anteriores ou se lembrará delas por um longo tempo. Com $a = 1/2$, obtemos estimativas sucessivas de

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

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

A técnica de estimar o valor seguinte da série, tomando a média ponderada do valor sendo medido e a estimativa anterior, é algumas vezes chamada de **aging** (envelhecimento). Essa técnica é aplicável a muitas situações nas quais é preciso uma previsão baseada nos valores anteriores. Aging é especialmente fácil de implementar quando $a = 1/2$. Basta apenas adicionar o novo valor à estimativa atual e dividir a soma por 2 (deslocando 1 bit à direita).

Escalonamento garantido

Um método completamente diferente de lidar com o escalonamento é fazer promessas reais sobre o desempenho aos usuários e, então, satisfazê-los. Uma promessa realista e fácil de cumprir é esta: se houver n usuários conectados enquanto você estiver trabalhando, você receberá cerca de $1/n$ de CPU. De modo semelhante, em um sistema monousuário com n processos em execução, todos iguais, cada um deve receber $1/n$ ciclos de CPU. Isso parece suficientemente justo.

Para fazer valer essa promessa, o sistema deve manter o controle da quantidade de CPU que cada processo recebe desde sua criação. Ele então calcula a quantidade de CPU destinada a cada um ou simplesmente o tempo desde a criação dividido por n. Como a quantidade de tempo de CPU que cada processo realmente teve é também conhecida, torna-se fácil calcular a taxa entre o tempo de CPU de fato consumido e o tempo de CPU destinado a cada processo. Uma taxa de 0,5 significa que um processo teve somente a metade do que ele deveria ter, e uma taxa de 2,0 significa que um processo teve duas vezes mais do que lhe foi destinado. O algoritmo então executará o processo com a taxa mais baixa, até que sua taxa cresça e se aproxime da de seu competidor.

Escalonamento por loteria

Fazer promessas aos usuários e satisfazê-los é uma boa ideia, porém difícil de implementar. Contudo, um outro algoritmo pode ser usado com resultados similarmente previsí-

veis, mas de implementação muito mais simples. É o chamado **escalonamento por loteria** (Waldspurger e Weihl, 1994).

A ideia básica é dar bilhetes de loteria aos processos, cujos prêmios são vários recursos do sistema, como tempo de CPU. Se houver uma decisão de escalonamento, um bilhete de loteria será escolhido aleatoriamente e o processo que tem o bilhete conseguirá o recurso. Quando aplicado ao escalonamento de CPU, o sistema pode fazer um sorteio 50 vezes por segundo e, portanto, cada vencedor terá 20 ms de tempo de CPU como prêmio.

Parafraseando George Orwell: "Todos os processos são iguais, mas alguns são mais iguais que os outros". Aos processos mais importantes podem ser atribuídos bilhetes extras para aumentar suas probabilidades de vitória. Se houver cem bilhetes extras e um processo detiver 20 deles, esse processo terá uma chance de 20 por cento de vencer cada loteria. Ao longo da execução, ele obterá 20 por cento da CPU. Diferentemente de um escalonador por prioridades, no qual é muito difícil estabelecer o que de fato significa uma prioridade 40, aqui há uma regra clara: um processo que detenha uma fração *f* dos bilhetes obterá em torno de uma fração *f* do recurso em questão.

O escalonamento por loteria tem várias propriedades interessantes. Por exemplo, se aparece um novo processo e a ele são atribuídos alguns bilhetes, já no próximo sorteio da loteria sua probabilidade de vencer será proporcional ao número de bilhetes que ele tiver. Em outras palavras, o escalonamento por loteria é altamente responsivo.

Os processos cooperativos podem trocar bilhetes entre si, se assim desejarem. Por exemplo, quando um processo cliente envia uma mensagem para um processo servidor e, então, é bloqueado, ele pode dar todos os seus bilhetes ao servidor, para que aumentem as probabilidades de o servidor executar logo. Quando o servidor termina, retorna os bilhetes para o cliente executar novamente. Na verdade, na ausência de clientes, os servidores nem precisam de bilhetes.

O escalonamento por loteria pode ser usado para resolver problemas difíceis de solucionar a partir de outros métodos. Um exemplo é o de um servidor de vídeo, no qual vários processos alimentam o fluxo de vídeo de seus clientes, mas em diferentes taxas de apresentação dos quadros. Suponha que os processos precisem de taxas em 10, 20 e 25 quadros/s. Alocando a esses processos dez, 20 e 25 bilhetes, respectivamente, eles vão automaticamente dividir a CPU aproximadamente na proporção correta, que é 10:20:25.

Escalonamento por fração justa (fair-share)

Até agora temos partido do pressuposto de que cada processo é escalonado por si próprio, sem nos preocuparmos com quem é seu dono. Como resultado, se o usuário 1 inicia nove processos e o usuário 2 inicia um processo, com chaveamento circular ou com prioridades iguais, o usuário 1 obterá 90 por cento da CPU e o usuário 2 terá somente 10 por cento dela.

Para evitar isso, alguns sistemas consideram a propriedade do processo antes de escaloná-lo. Nesse modelo, a cada usuário é alocada uma fração da CPU, e o escalonador escolhe os processos de modo que garanta essa fração. Assim, se dois usuários tiverem 50 por cento da CPU prometida a cada um deles, cada um obterá os 50 por cento, não importando quantos processos eles tenham gerado.

Como exemplo, imagine um sistema com dois usuários, cada qual com 50 por cento da CPU prometida a ele. O usuário 1 tem quatro processos, *A*, *B*, *C* e *D*, e o usuário 2 tem somente um processo, *E*. Se for usado o escalonamento circular, uma sequência possível de escalonamento que cumpra todas as exigências será a seguinte:

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

Por outro lado, se ao usuário 1 se destinar duas vezes mais tempo de CPU que para o usuário 2, poderemos obter:

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

É claro que existem inúmeras outras possibilidades, igualmente passíveis de serem exploradas, dependendo da noção de justiça.

2.4.4 Escalonamento em sistemas de tempo real

Um sistema de **tempo real** é aquele no qual o tempo tem uma função essencial. Em geral, um ou mais dispositivos físicos externos ao computador geram estímulos, e o computador deve reagir apropriadamente a eles dentro de um dado intervalo de tempo. Por exemplo, o computador em um CD player obtém os bits que chegam do drive e precisa convertê-los em música em um intervalo de tempo muito curto. Se o cálculo que ele fizer for muito demorado, a música soará diferente. Outros exemplos de sistemas de tempo real incluem: monitoração de pacientes em unidades de terapia intensiva de hospitais, piloto automático de aeronaves e robôs de controle em fábricas automatizadas. Em todos esses casos, ter a resposta certa, mas tardia, é tão ruim quanto não ter nada.

Sistemas de tempo real são em geral categorizados como **tempo real crítico**, isto é, há prazos absolutos que devem ser cumpridos ou, então, como **tempo real não crítico**, no qual o descumprimento ocasional de um prazo é indesejável, contudo tolerável. Em ambos os casos, o comportamento de tempo real é implementado dividindo-se o programa em vários processos cujo comportamento é previamente conhecido. De modo geral, esses processos têm vida curta e podem executar em bem menos de um segundo. Quando é detectado um evento externo, o trabalho do escalonador é escalonar os processos de tal maneira que todos os prazos sejam cumpridos.

Os eventos aos quais um sistema de tempo real pode precisar responder podem ser categorizados ainda como **periódicos** (ocorrem em intervalos regulares) ou **aperió-**

dicos (acontecem de modo imprevisível). Um sistema pode ter de responder a múltiplos fluxos de eventos periódicos. Dependendo de quanto tempo cada evento requeira para processar, talvez nem seja possível tratar de todos. Por exemplo, se houver m eventos periódicos e o evento i ocorrer com período P_i e requerer C_i segundos de CPU para tratar cada evento, então a carga poderá ser tratada somente se

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

Um sistema de tempo real que satisfaça esse critério é chamado de **escalonável**.

Como exemplo, considere um sistema de tempo real não crítico com três eventos periódicos, com períodos de 100, 200 e 500 ms, respectivamente. Se esses eventos requererem 50, 30 e 100 ms de tempo de CPU por evento, nessa ordem, o sistema é escalonável porque 0,5 + 0,15 + 0,2 < 1. Se um quarto evento, com período de 1 s, for adicionado, o sistema permanecerá escalonável desde que esse evento não precise de mais de 150 ms do tempo de CPU por evento. Está implícita nesse cálculo a hipótese de que o custo extra do chaveamento de contexto é tão pequeno que pode ser desprezado.

Os algoritmos de escalonamento de tempo real podem ser estáticos ou dinâmicos. Os primeiros tomam suas decisões de escalonamento antes de o sistema começar a executar. Os últimos o fazem em tempo de execução. O escalonamento estático só funciona quando há prévia informação perfeita disponível sobre o trabalho necessário a ser feito e os prazos que devem ser cumpridos. Os algoritmos de escalonamento dinâmico não apresentam essas restrições. Adiaremos nosso estudo sobre algoritmos específicos para quando tratarmos de sistemas multimídia de tempo real no Capítulo 7.

2.4.5 Política *versus* mecanismo

Até agora, temos presumido tacitamente que todos os processos no sistema pertencem a usuários diferentes e estão, portanto, competindo pela CPU. Embora isso muitas vezes seja verdade, um processo pode ter muitos filhos executando sob seu controle — por exemplo, um processo de um sistema de gerenciamento de bancos de dados. Cada filho pode estar atendendo a uma requisição diferente ou ter uma função específica para realizar (análise sintática de consultas, acesso a disco etc.). É totalmente possível que o processo principal tenha uma ideia clara de quais de seus filhos sejam os mais importantes (ou tenham tempo crítico) e quais sejam os menos importantes. Infelizmente, nenhum dos escalonadores discutidos anteriormente aceita qualquer entrada proveniente de processos do usuário sobre decisões de escalonamento. Como resultado, o escalonador raramente faz a melhor escolha.

A solução para esse problema é separar o **mecanismo de escalonamento** da **política de escalonamento**, um princípio estabelecido há muito tempo (Levin et al., 1975). Isso significa que o algoritmo de escalonamento é de algum modo parametrizado, mas os parâmetros podem ser preenchidos pelos processos dos usuários. Consideremos novamente o exemplo do banco de dados. Suponha que o núcleo use um algoritmo de escalonamento por prioridades, mas que disponibilize uma chamada de sistema na qual um processo seja capaz de configurar (e alterar) as prioridades e seus filhos. Desse modo, o pai pode controlar em detalhes como seus filhos são escalonados, mesmo que ele próprio não faça o escalonamento. Nesse exemplo, o mecanismo de escalonamento está no núcleo, mas a política é estabelecida por um processo de usuário.

2.4.6 Escalonamento de threads

Quando cada um dentre vários processos tem múltiplos threads, ocorrem dois níveis de paralelismo: processos e threads. O escalonamento nesses sistemas pode diferir de modo substancial, dependendo de os threads serem de usuário ou de núcleo (ou ambos).

Consideremos primeiro os threads de usuário. Como o núcleo não sabe da existência de threads, ele opera como sempre faz, escolhendo um processo — por exemplo, *A* — e dando-lhe o controle de seu quantum. O escalonador do thread em *A* decide qual thread deve executar — por exemplo, *A1*. Como não há interrupções de relógio para multiprogramar threads, esse thread pode continuar executando enquanto quiser. Se ele usar todo o quantum do processo, o núcleo selecionará um outro processo para executar.

Quando o processo *A* finalmente voltar a executar, o thread *A1* permanecerá executando. Ele continuará a consumir todo o tempo de *A* até que termine. Contudo, seu comportamento antissocial não afetará outros processos: eles obterão aquilo que o escalonador considerar uma fração apropriada, não importando o que esteja acontecendo dentro do processo *A*.

Agora, imagine que os threads de *A* tenham relativamente pouco trabalho a fazer por surto de CPU — por exemplo, 5 ms de trabalho para um quantum de 50 ms. Consequentemente, cada um executa por um pouquinho de tempo e então cede a CPU de volta para o escalonador de thread. Isso pode levar à sequência *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, *A2*, *A3*, *A1*, antes de o núcleo chavear para o processo *B*. Essa situação é ilustrada na Figura 2.35(a).

O algoritmo de escalonamento usado pelo sistema de tempo de execução pode ser qualquer um dos que acabam de ser descritos. Na prática, o escalonamento circular e o escalonamento por prioridades são os mais comuns. A única limitação é a ausência de uma interrupção de relógio para interromper um thread que esteja executando há muito tempo.

Agora, considere a situação com os threads de núcleo. Nesse caso, o núcleo escolhe um thread para executar. Ele não precisa levar em conta a qual processo o thread pertence, mas, se quiser, pode considerar esse fato. Ao thread é

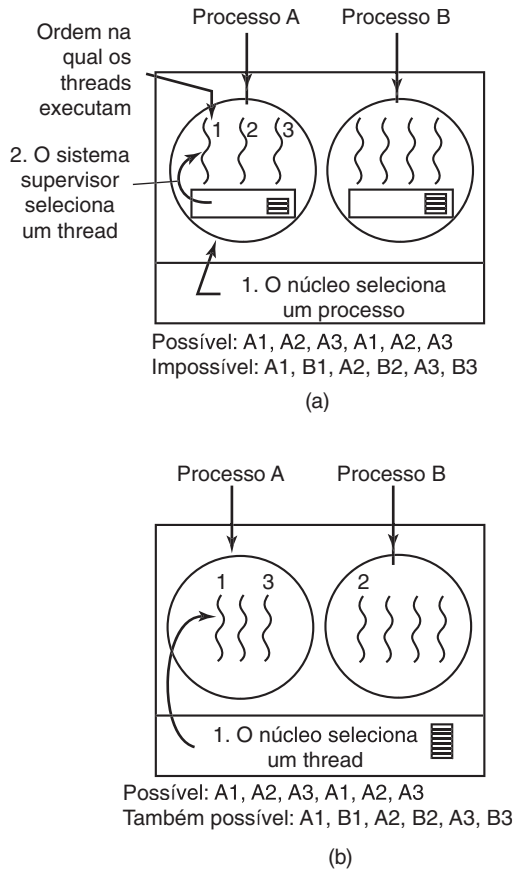


Figura 2.35 (a) Escalonamento possível de threads de usuários com um quantum de processo de 50 ms e threads que executam 5 ms por surto de CPU. (b) Escalonamento possível de threads de núcleo com as mesmas características de (a).

dado um quantum, e ele será compulsoriamente suspenso se exceder o quantum. Com um quantum de 50 ms, mas com threads que bloqueiam depois de 5 ms, a ordem dos threads por um período de 30 ms pode ser *A1*, *B1*, *A2*, *B2*, *A3*, *B3*, algo impossível de conseguir com esses parâmetros e com threads de usuário. Essa situação é parcialmente mostrada na Figura 2.35(b).

Uma diferença importante entre os threads de usuário e os threads de núcleo é o desempenho. O chaveamento de thread com threads de usuário usa poucas instruções de máquina. Para os threads de núcleo, o chaveamento requer um chaveamento completo do contexto, com alteração do mapa de memória e invalidação da cache — o que significa uma demora várias ordens de magnitude. Por outro lado, para os threads de núcleo, um thread bloqueado pela E/S não suspende o processo inteiro, como ocorre nos threads de usuário.

Como o núcleo sabe que o chaveamento de um thread no processo *A* para um thread no processo *B* custa mais do que executar um segundo thread no processo *A* (pois terá de mudar o mapa de memória e invalidar a memória cache), ele pode considerar essa informação quando for tomar uma decisão. Por exemplo, dados dois threads igualmente importantes, sendo que um deles pertence ao mesmo processo de um thread que acabou de ser bloqueado e o outro pertence a um processo diferente, a preferência poderia ser dada ao primeiro.

Outro fator importante é que os threads de usuário podem utilizar um escalonador de thread específico para uma aplicação. Considere, por exemplo, o servidor da Web da Figura 2.6. Suponha que um thread operário tenha acabado de ser bloqueado e que o thread despachante e dois threads operários estejam prontos. Qual deveria executar? O sistema de tempo de execução — que sabe o que cada thread faz — pode facilmente escolher o despachante como o próximo thread a executar, para que este coloque outro operário para executar. Essa estratégia maximiza a quantidade de paralelismo em um ambiente no qual os operários frequentemente são bloqueados pela E/S de disco. Já no caso dos threads de núcleo, este nunca saberia o que cada thread fez (embora a eles pudessem ser atribuídas diferentes prioridades). Contudo, em geral os escalonadores de threads específicos para uma aplicação são capazes de ajustar uma aplicação melhor do que o núcleo pode fazê-lo.

2.5 Problemas clássicos de IPC

A literatura sobre sistemas operacionais está repleta de problemas interessantes que têm sido amplamente discutidos e analisados a partir de vários métodos de sincronização. Nas próximas seções examinaremos três desses problemas mais comuns.

2.5.1 O problema do jantar dos filósofos

Em 1965, Dijkstra formulou e resolveu um problema de sincronização que ele chamou de **problema do jantar dos filósofos**. Desde então, cada um que inventasse mais uma primitiva de sincronização via-se obrigado a demonstrar até que ponto essa nova primitiva era maravilhosa, mostrando com que elegância ela resolvia o problema do jantar dos filósofos. O problema pode ser explicado de maneira muito simples. Cinco filósofos estão sentados em torno de uma mesa circular. Cada filósofo tem um prato de espaguete. O espaguete está tão escorregadio que um filósofo precisa de dois garfos para comê-lo. Entre cada par de pratos está um garfo. O diagrama da mesa é ilustrado na Figura 2.36.

A vida de um filósofo consiste em alternar períodos de comer e pensar. (Trata-se apenas de uma abstração, mesmo para os filósofos; as outras atividades são irrelevantes ao problema.) Quando uma filósofa fica com fome, ela tenta pegar os garfos à sua direita e à sua esquerda, um de cada vez, em qualquer ordem. Se conseguir pegar dois garfos, ela comerá durante um determinado tempo e, então, colocará os garfos na mesa novamente e continuará a pen-

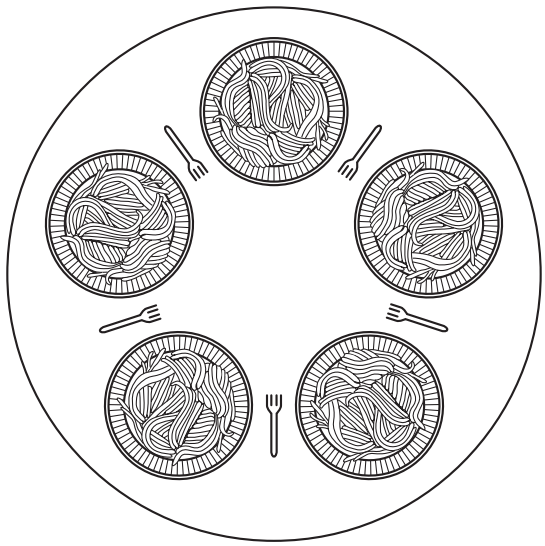


Figura 2.36 Hora do almoço no Departamento de Filosofia.

sar. A questão fundamental é: você consegue escrever um programa para cada filósofo que faça o que deve fazer e nunca trave? (Observe que a necessidade de ter dois garfos é artificial; talvez fosse melhor mudar de comida italiana para comida chinesa, substituindo o espaguete por arroz e os garfos por pauzinhos.)

A Figura 2.37 mostra a solução óbvia. A rotina *take_fork* espera até que o garfo específico esteja disponível e então o pega. Infelizmente, a solução óbvia está errada. Suponha que todos os cinco filósofos resolvam usar seus garfos simultaneamente. Nenhum deles será capaz de pegar o garfo que estiver a sua direita e, assim, ocorrerá um impasse.

Podemos fazer modificações para que o programa, depois de pegar o garfo esquerdo, verifique se o garfo direito está disponível. Se não estiver, o filósofo devolverá o garfo esquerdo à mesa, esperará por algum tempo e então repetirá todo o processo. Mas essa proposta também falha, embora por uma razão diferente. Com um pouco de azar, todos os filósofos poderiam começar o algoritmo simultaneamente; pegando seus garfos esquerdos e, vendo que seus garfos direitos não estariam disponíveis, devolveriam seus garfos esquerdos, esperariam, de novo pegariam seus garfos esquerdos simultaneamente, e assim permaneceriam para sempre. Uma situação como essa — na qual todos os programas continuam executando indefinidamente, mas falham ao tentar progredir — é chamada de **inanição** (*starvation*). (E é assim chamada mesmo quando o problema não ocorre em um restaurante italiano ou chinês.)

Mas então você poderia pensar que, se os filósofos esperassem por um tempo aleatório, em vez de esperarem por um tempo fixo depois de falharem ao pegar o garfo do lado direito, a probabilidade de tudo continuar intertravado, mesmo que por uma hora, seria muito pequena. Essa observação é verdadeira, e, em quase todas as aplicações, tentar de novo mais tarde é uma abordagem adotada. Por exemplo, na popular rede local Ethernet, se dois computadores enviam um pacote ao mesmo tempo (levando a uma colisão de pacotes), cada um espera por um tempo aleatório antes de tentar novamente; na prática, essa solução funciona bem. Contudo, para algumas aplicações seria preferível uma solução que sempre fosse válida e não falhasse por causa de uma série improvável de números aleatórios. Pense, por exemplo, no controle de segurança em uma usina de energia nuclear.

Um aperfeiçoamento da solução mostrada na Figura 2.37 que não apresenta impasse nem inanição é proteger os cinco comandos que seguem a chamada *think* com um semáforo binário. Antes de começar a pegar garfos, um filósofo faria um down no *mutex*. Depois de trocar os garfos, ele faria um up no *mutex*. Do ponto de vista teórico, essa solução é adequada. Do ponto de vista prático, ela apresenta um problema de desempenho: somente um filósofo por vez pode comer a qualquer instante. Com cinco garfos disponíveis, seria possível permitir que dois filósofos comessem ao mesmo tempo.

A solução apresentada na Figura 2.38 é livre de impasse e permite o máximo paralelismo a um número arbitrário de filósofos. Ela usa um arranjo, *estado*, para controlar se

```
#define N 5                                     /* número de filósofos */

void philosopher(int i)                         /* i: número do filósofo, de 0 a 4 */
{
    while (TRUE) {
        think( );                               /* o filósofo está pensando */
        take_fork(i);                           /* pega o garfo esquerdo */
        take_fork((i+1) % N);                   /* pega o garfo direito; % é o operador módulo */
        eat( );                                 /* hummm! Espaguete*/
        put_fork(i);                            /* devolve o garfo esquerdo à mesa */
        put_fork((i+1) % N);                    /* devolve o garfo direito à mesa */
    }
}
```

Figura 2.37 Uma solução errada para o problema do jantar dos filósofos.

```
#define N              5              /* número de filósofos */
#define LEFT           (i+N–1)%N      /* número do vizinho à esquerda de i */
#define RIGHT          (i+1)%N        /* número do vizinho à direita de i */
#define THINKING       0              /* o filósofo está pensando */
#define HUNGRY         1              /* o filósofo está tentando pegar garfos */
#define EATING         2              /* o filósofo está comendo */
typedef int semaphore;                /* semáforos são um tipo especial de int */
int state[N];                         /* arranjo para controlar o estado de cada um */
semaphore mutex = 1;                  /* exclusão mútua para as regiões críticas */
semaphore s[N];                       /* um semáforo por filósofo */

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


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

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

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

Figura 2.38 Uma solução para o problema do jantar dos filósofos.

um filósofo está comendo, pensando ou faminto (tentando pegar garfos). Um filósofo só pode mudar para o estado 'comendo' se nenhum dos vizinhos estiver comendo. Os vizinhos do filósofo *i* são definidos pelas macros *LEFT* e *RIGHT*. Em outras palavras, se *i* for 2, *LEFT* será 1 e *RIGHT* será 3.

O programa usa um arranjo de semáforos, um por filósofo; assim, filósofos famintos podem ser bloqueados se os garfos necessários estiverem ocupados. Observe que cada processo executa a rotina *philosopher* como seu código principal, mas as outras rotinas — *take_ forks*, *put_ forks* e *test* — são rotinas ordinárias, e não processos separados.

2.5.2 O problema dos leitores e escritores

O problema do jantar dos filósofos é útil para modelar processos que competem pelo acesso exclusivo a um número limitado de recursos, como dispositivos de E/S. Outro problema famoso é o caso dos leitores e escritores (Courtois et al., 1971), que modela o acesso a uma base de dados. Imagine, por exemplo, um sistema de reserva de linhas aéreas, com muitos processos em competição, querendo ler e escrever. É aceitável que múltiplos processos leiam a base de dados ao mesmo tempo, mas, se um processo estiver atualizando (escrevendo) na base de dados, nenhum outro

processo pode ter acesso ao banco de dados, nem mesmo os leitores. A questão é: como programar os leitores e os escritores? Uma solução é mostrada na Figura 2.39.

A partir dessa solução, para obter o acesso à base de dados, o primeiro leitor faz um down no semáforo *db*. Os leitores subsequentes meramente incrementam um contador, *rc*. Conforme saem, os leitores decrementam o contador de 1 e o último leitor a sair faz um up no semáforo, permitindo que um eventual escritor bloqueado entre.

A solução apresentada aqui contém implicitamente uma decisão sutil que vale a pena comentar. Suponha que, enquanto um leitor está usando a base de dados, um outro leitor chegue. Como ter dois leitores ao mesmo tempo não é um problema, o segundo leitor é admitido. Leitores adicionais também podem ser admitidos se chegarem.

Agora imagine que apareça um escritor. Este não pode ser admitido na base de dados, pois escritores devem ter acesso exclusivo. O escritor é, então, suspenso. Leitores adicionais chegam. Enquanto houver pelo menos um leitor ativo, leitores subsequentes serão admitidos. Como consequência dessa estratégia, enquanto houver um fluxo estável de leitores chegando, todos entrarão assim que chegarem. O escritor permanecerá suspenso até que nenhum leitor esteja presente. Se um novo leitor chegar — digamos, a cada dois segundos — e cada leitor levar cinco segundos para fazer seu trabalho, o escritor nunca entrará.

Para evitar essa situação, o programa poderia ser escrito de modo um pouco diferente: se um leitor chegar quando um escritor estiver esperando, o leitor será suspenso logo depois do escritor, em vez de ser admitido de imediato. Dessa maneira, um escritor, para terminar, precisa esperar por leitores que estavam ativos quando ele chegou, mas não por leitores que chegaram depois dele. A desvantagem dessa solução é que se consegue menos concorrência e, portanto, um desempenho menor. Courtois et al. apresentam uma solução que dá prioridade aos escritores. Para mais detalhes, consulte o artigo.

2.6 Pesquisas em processos e threads

No Capítulo 1, estudamos algumas das pesquisas atuais em estrutura de sistemas operacionais. Neste capítulo e nos subsequentes, examinaremos pesquisas mais específicas, iniciando com os processos. Aos poucos, torna-se claro que alguns assuntos são muito mais estáveis que outros. A maioria das pesquisas tende a ser sobre tópicos novos, em vez daqueles que nos rodeiam há décadas.

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

void reader(void)
{
     while (TRUE) {                     /* repete para sempre */
          down(&mutex);                 /* obtém 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);                 /* obtém acesso exclusivo a 'rc' */
          rc = rc – 1;                  /* um leitor a menos agora */
          if (rc == 0) up(&db);         /* se este for o último leitor ... */
          up(&mutex);                   /* libera o acesso exclusivo a 'rc' */
          use_data_read( );             /* região não crítica */
     }
}


void writer(void)
{
     while (TRUE) {                     /* repete para sempre */
          think_up_data( );             /* região não crítica */
          down(&db);                    /* obtém acesso exclusivo */
          write_data_base( );           /* atualiza os dados */
          up(&db);                      /* libera o acesso exclusivo */
     }
}
```

Figura 2.39 Uma solução para o problema dos leitores e escritores.

O conceito de um processo é um exemplo de algo muito bem estabelecido. Quase todo sistema tem alguma noção de um processo como um recipiente para agrupar recursos relacionados, como um espaço de endereçamento, threads, arquivos abertos, permissões de proteção etc. Sistemas diferentes fazem esse agrupamento de maneira um pouco diferente, mas trata-se apenas de diferenças de engenharia. A ideia básica não é tão controversa e há pouca pesquisa nova sobre o tema processos.

Os threads são uma ideia mais nova que os processos, mas tem havido bastantes reflexões sobre eles. Além disso, ocasionalmente aparecem artigos sobre threads tratando, por exemplo, de aglomerados de multiprocessadores (Tam et al., 2007) ou do dimensionamento do número de threads em um processo para cem mil (Von Behren et al., 2003).

A sincronização de processos está muito mais definida agora, mas de vez em quando ainda há artigos, como aqueles sobre processamento concorrente sem locks (Fraser e Harris, 2007) ou sincronização no modo não bloqueante em sistemas de tempo real (Hothmuth e Haertig, 2001).

O escalonamento (tanto uniprocessador quanto multiprocessador) ainda é um tópico recente e caro a alguns pesquisadores. Alguns tópicos sendo pesquisados incluem escalonamento de dispositivos móveis em termos de eficiência energética (Yuan e Nahrstedt, 2006), escalonamento com tecnologia hyperthreading (Bulpin e Pratt, 2005), modos de reduzir a ociosidade da CPU (Eggert e Touch, 2005) e escalonamento de sistemas de tempo virtual (Nieh et al., 2001). Contudo, poucos projetistas de sistemas operacionais andam desesperados pela falta de um algoritmo decente para o escalonamento de threads — portanto, parece que esse tipo de pesquisa é mais um desejo do que uma necessidade de pesquisadores. De modo geral, processos, threads e escalonamento não são mais tópicos de pesquisa tão procurados como antes. A pesquisa avançou.

2.7 Resumo

Para ocultar os efeitos das interrupções, os sistemas operacionais oferecem um modelo conceitual que consiste em processos sequenciais executando em paralelo. Os processos podem ser criados e terminados dinamicamente. Cada processo tem seu próprio espaço de endereçamento.

Para algumas aplicações, é útil ter múltiplos threads de controle dentro de um único processo. Esses threads são escalonados independentemente e cada um tem sua própria pilha, mas todos os threads em um processo compartilham um espaço de endereçamento comum. Threads podem ser implementados no espaço do usuário ou no núcleo.

Os processos podem se comunicar uns com os outros por meio de primitivas de comunicação entre processos, como semáforos, monitores ou mensagens. Essas unidades básicas são usadas para assegurar que dois processos nunca estarão em suas regiões críticas ao mesmo tempo — uma situação que levaria ao caos. Um processo pode estar executando, ser executável ou bloqueado e alterar o estado quando ele ou um outro processo executa uma das unidades básicas de comunicação entre processos. A comunicação interthread é semelhante.

As primitivas de comunicação entre processos podem ser usadas para resolver problemas como o produtor–consumidor, o jantar dos filósofos e o leitor–escritor. Mesmo com essas primitivas, devem-se tomar cuidados para evitar erros e impasses.

Muitos algoritmos de escalonamento têm sido estudados. Alguns deles são usados principalmente em sistemas em lote, como a *tarefa mais curta primeiro*. Outros são comuns aos sistemas em lote e aos sistemas interativos — como o escalonamento circular, o escalonamento por prioridades, as filas múltiplas, o escalonamento garantido, o escalonamento por loteria e o escalonamento por fração justa. Alguns sistemas fazem uma separação entre o mecanismo de escalonamento e a política de escalonamento, o que permite aos usuários um controle sobre o algoritmo de escalonamento.

Problemas

1. Na Figura 2.2, são mostrados três estados de processos. Na teoria, com três estados poderia haver seis transições, duas para cada estado. Contudo, somente quatro transições são mostradas. Há alguma circunstância na qual uma delas ou ambas as transições não ilustradas possam ocorrer?
2. Suponha que você seja o projetista de uma arquitetura de computador avançada que fez o chaveamento entre processos por hardware em vez de usar interrupções. De que informação a CPU precisaria? Descreva como o processo de chaveamento por hardware poderia funcionar.
3. Em todos os computadores atuais, pelo menos uma parte dos manipuladores de interrupção (interrupt handlers) é escrita em linguagem assembly. Por quê?
4. Quando uma interrupção ou uma chamada de sistema transfere o controle para o sistema operacional, geralmente é usada uma área da pilha do núcleo separada da pilha do processo interrompido. Por quê?
5. Tarefas múltiplas podem ser executadas paralelamente e terminar mais rápido do que se tivessem sido executados sucessivamente. Suponha que duas tarefas, cada uma precisando de dez minutos do tempo da CPU, começassem simultaneamente. De quanto tempo o último precisará para terminar se elas forem executados sucessivamente? Quanto tempo se forem executadas paralelamente? Suponha 50 por cento de espera de E/S.
6. No texto, foi estabelecido que o modelo da Figura 2.8(a) não era adequado para um servidor de arquivos que usasse uma cache na memória. Por que não? Cada processo poderia ter sua própria cache?
7. Se um processo multithread bifurcar, há problemas se o filho copia todos os threads do pai. Suponha que um dos threads originais estivesse esperando por uma entrada do

teclado. Agora dois threads estão esperando pela entrada do teclado, um em cada processo. Esse problema pode ocorrer em processos de thread único?

8. Na Figura 2.6, é mostrado um servidor da Web multithread. Se o único modo de ler a partir de um arquivo for o bloqueio normal da chamada de sistema read, você acha que threads de usuário ou de núcleo estão sendo usados para o servidor da Web? Por quê?
9. No texto, descrevemos um servidor da Web multithread, mostrando por que ele é melhor que um servidor de thread único e um servidor de máquina de estados finitos. Há alguma circunstância na qual um servidor de thread único poderia ser melhor? Dê um exemplo.
10. Na Tabela 2.4, o conjunto de registradores é relacionado como um item por thread, e não por processo. Por quê? (Afinal, a máquina tem somente um conjunto de registradores.)
11. O que faria um thread desistir voluntariamente da CPU chamando *thread_yield*? (Afinal, como não há interrupção periódica de relógio, ele pode nunca mais obter a CPU de volta.)
12. Um thread pode sofrer preempção por uma interrupção de relógio? Em caso afirmativo, sob quais circunstâncias? Do contrário, por que não?
13. Neste problema, você deve comparar a leitura de um arquivo usando um servidor de arquivos monothread e um servidor multithread. São necessários 15 ms para obter uma requisição de trabalho, despachá-la e fazer o restante do processamento necessário, presumindo que os dados essenciais estejam na cache de blocos. Se for necessária uma operação de disco — como ocorre em um terço das vezes —, será preciso um tempo adicional de 75 ms, durante o qual o thread dorme. Quantas requisições/segundo o servidor pode tratar se for monothread? E se for multithread?
14. Qual a maior vantagem de implementar threads no espaço do usuário? Qual é a maior desvantagem?
15. Na Figura 2.10, as criações de threads e as mensagens impressas pelos threads são intercaladas aleatoriamente. Há algum modo de impor que a ordem seja estritamente thread 1 criado, thread 1 imprime mensagem, thread 1 sai, thread 2 criado, thread 2 imprime a mensagem, thread 2 sai e assim por diante? Em caso de resposta afirmativa, qual é esse modo? Em caso de resposta negativa, por que não?
16. Na discussão sobre variáveis globais em threads, usamos uma rotina *create_ global* para alocar memória a um ponteiro para a variável, em vez de alocar diretamente a própria variável. Isso é essencial ou as rotinas poderiam funcionar muito bem apenas com os próprios valores?
17. Considere um sistema no qual threads são implementados inteiramente no espaço do usuário, sendo que o sistema de tempo de execução sofre uma interrupção de relógio a cada segundo. Suponha que uma interrupção de relógio ocorra enquanto algum thread estiver executando no sistema de tempo de execução. Que problema poderia ocorrer? O que você sugere para resolvê-lo?
18. Suponha que um sistema operacional não tenha uma chamada de sistema como a select para verificar previamente se é seguro ler um arquivo, um pipe ou algum dispositivo, mas ele permite que "alarm clocks" sejam setados, os quais interrompem chamadas de sistema bloqueadas. É possível implementar um pacote de threads, no espaço de usuário, sob essas condições? Comente.
19. O problema de inversão de prioridades discutido na Seção 2.3.4 pode acontecer com threads de usuário? Por quê?
20. Na Seção 2.3.4, foi descrita uma situação com um processo de alta prioridade, *H*, e um de baixa prioridade, *L*, que levava *H* a um laço infinito. O mesmo problema ocorreria se fosse usado o escalonamento circular em vez do escalonamento por prioridades? Comente.
21. Em um sistema com threads, quando são utilizados threads de usuário, há uma pilha por thread ou uma pilha por processo? E quando se usam threads de núcleo? Explique.
22. Quando um computador está sendo desenvolvido, ele é antes simulado por um programa que executa uma instrução por vez. Mesmo os multiprocessadores são simulados de modo estritamente sequencial. É possível que ocorra uma condição de corrida quando não há eventos simultâneos como nessas simulações?
23. A solução de espera ociosa usando a variável *turn* (Figura 2.18) funciona quando os dois processos estão executando em um multiprocessador de memória compartilhada, isto é, duas CPUs compartilhando uma memória comum?
24. A solução de Peterson para o problema da exclusão mútua, mostrado na Figura 2.19, funciona quando o escalonamento do processo for preemptivo? E quando o escalonamento não for preemptivo?
25. Faça um esboço de como um sistema operacional capaz de desabilitar interrupções poderia implementar semáforos.
26. Mostre como os semáforos contadores (isto é, os semáforos que podem conter um valor arbitrário) podem ser implementados usando somente semáforos binários e simples instruções de máquina.
27. Se um sistema tem somente dois processos, tem sentido usar uma barreira para sincronizá-los? Por quê?
28. Dois threads podem, no mesmo processo, sincronizar a partir do uso de um semáforo de núcleo se os threads forem implementados pelo núcleo? E se os threads fossem implementados no espaço do usuário? (Suponha que nenhum thread em qualquer outro processo tenha acesso ao semáforo.) Comente suas respostas.
29. Sincronização com monitores usa variáveis de condição e duas operações especiais, wait e signal. Uma forma mais geral de sincronização seria ter uma única primitiva, waituntil, que possuísse um predicado booleano como parâmetro. Assim, alguém poderia dizer, por exemplo,

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

    A primitiva signal não seria mais necessária. Esse esquema é claramente mais geral que o proposto por Hoare ou Brinch Hansen, mas não é utilizado. Por quê? *Dica:* pense na implementação.

30. Um restaurante de fast-food tem quatro tipos de empregados: (1) anotadores de pedido, que anotam o pedido dos clientes; (2) cozinheiros, que preparam a comida; (3) embaladores, que colocam a comida nas sacolas; e (4) caixas, que entregam as sacolas para os clientes e recebem o dinheiro deles. Cada empregado pode ser observado como um processo sequencial de comunicação. Qual forma de comunicação entre processos eles usariam? Relacione esse modelo aos processos no UNIX.

31. Imagine um sistema por troca de mensagens que use caixas postais. Quando se envia para uma caixa postal cheia ou tenta-se receber de uma caixa postal vazia, um processo não bloqueia. Na verdade, ele obtém um código de erro. O processo responde ao código de erro apenas tentando novamente, sucessivamente, até que ele consiga. Esse esquema leva a condições de corrida?

32. Os computadores CDC 6600 podiam lidar simultaneamente com até dez processos de E/S, usando uma forma interessante de escalonamento circular chamada **compartilhamento de processador**. Um chaveamento de processo ocorria depois de cada instrução; assim, a instrução 1 vinha do processo 1, a instrução 2 vinha do processo 2 e assim por diante. O chaveamento do processo era feito por um hardware especial e a sobrecarga era zero. Se um processo precisasse de T segundos para terminar sua execução, na ausência de competição, quanto tempo seria necessário se o compartilhamento do processador fosse usado com n processos?

33. Seria possível estabelecer uma medida sobre o quanto um processo é limitado pela CPU ou limitados por E/S analisando o código-fonte? Como isso poderia ser determinado em tempo de execução?

34. Na seção 'Quando escalonar', foi mencionado que, algumas vezes, o escalonamento poderia ser melhorado se um processo importante fosse passível de desempenhar um papel, ao ser bloqueado, na seleção do próximo processo a executar. Pense em uma situação na qual isso poderia ser usado e explique como.

35. As medidas de um certo sistema mostram que o processo médio executa por um tempo T antes de ser bloqueado para E/S. Um chaveamento de processos requer um tempo S efetivamente gasto (sobrecarga). Para o escalonamento circular com um quantum Q, dê uma fórmula para a eficiência da CPU em cada um dos seguintes casos:

(a) $Q = \infty$.

(b) $Q > T$.

(c) $S < Q < T$.

(d) $Q = S$.

(e) Q próximo de 0.

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

37. Cinco tarefas em lote, A a E, chegam a um centro de computação quase ao mesmo tempo. Elas têm tempos de execução estimados em 10, 6, 2, 4 e 8 minutos. Suas prioridades (externamente determinadas) são 3, 5, 2, 1 e 4, respectivamente, sendo 5 a prioridade mais alta. Para cada um dos seguintes algoritmos de escalonamento, determine o tempo médio de ida e volta. Ignore a sobrecarga de chaveamento de processos.

(a) Circular.

(b) Escalonamento por prioridades.

(c) *Primeiro a chegar, primeiro a ser servido* (execute na ordem 10, 6, 2, 4, 8).

(d) *Tarefa mais curta primeiro*.

Para (a), presuma que o sistema é multiprogramado e que cada tarefa obtenha sua fração justa da CPU. Para os itens (b) a (d), considere a execução de somente uma tarefa por vez, até que termine. Todas as tarefas são completamente limitadas pela CPU.

38. Um processo executando no CTSS precisa de 30 quanta para terminar. Quantas vezes ocorrerá uma troca para a memória, incluindo a primeira vez (antes de executar qualquer coisa)?

39. Você tem ideia de como impedir que o sistema de prioridade do CTSS seja enganado digitando-se a tecla <Entra> aleatoriamente?

40. O algoritmo do envelhecimento (*aging*) com $a = 1/2$ está sendo usado para prever tempos de execução. As quatro execuções anteriores, da primeira à mais recente, são 40, 20, 40 e 15 ms. Qual é a previsão da próxima execução?

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

42. Explique por que o escalonamento em dois níveis é bastante usado.

43. Um sistema de tempo real precisa controlar duas chamadas de voz, cada uma delas executada a cada 5 ms e consumindo 1 ms do tempo da CPU por surto, além de um vídeo de 25 quadros/s, e cada quadro requer 20 ms do tempo da CPU. Esse sistema pode ser escalonado?

44. Considere um sistema no qual se deseja separar a política e o mecanismo para o algoritmo de escalonamento dos threads de núcleo. Proponha um meio de chegar a esse objetivo.

45. Na solução para o problema do jantar dos filósofos (Figura 2.38), por que é atribuído *HUNGRY* à variável de estado na rotina *take_forks*?

46. Observe a rotina *put_forks* da Figura 2.38. Suponha que à variável *state*[*i*] fosse atribuída *THINKING depois* das duas chamadas de *test*, e não *antes*. Como isso poderia afetar a solução?

47. O problema dos leitores e escritores pode ser formulado de várias maneiras, dependendo de quando cada categoria de processos pode ser iniciada. Descreva, detalhadamente, três variações diferentes do problema, cada uma favorecendo (ou não) alguma categoria de processos. Para

cada variação, especifique o que acontece quando um leitor ou um escritor fica pronto para ter acesso ao banco de dados e o que ocorre quando um processo acaba de usar o banco de dados.

48. Escreva um script do shell que produza um arquivo de números sequenciais lendo-se o último número no arquivo, adicionando-se 1 a ele e, então, anexando-o ao arquivo. Execute uma instância do script em background (segundo plano) e outra em foreground (primeiro plano), cada uma realizando acessos ao mesmo arquivo. Quanto tempo transcorre antes de se manifestar uma condição de disputa? Qual é a região crítica? Modifique o script para impedir a disputa. (*Dica:* use

```
ln file file.lock
```

para proteger o arquivo de dados.)

49. Imagine um sistema operacional que permita semáforos. Implemente um sistema de mensagens. Escreva as rotinas para enviar e receber mensagens.

50. Resolva o problema do jantar dos filósofos usando monitores em vez de semáforos.

51. Suponha que uma universidade, para mostrar como é politicamente correta, aplique a doutrina da Suprema Corte dos Estados Unidos, "Separado mas igual é inerentemente desigual", para gênero e raça, pondo fim a sua prática de longa data de banheiros no campus segregados por gênero. Contudo, como uma concessão à tradição, ela decreta que, quando uma mulher estiver no banheiro, outra mulher poderá entrar, mas um homem não e vice-versa. Um sinal com um marcador deslizante, na porta de cada banheiro, indica em qual dos três estados o banheiro se encontra:

Vazio

Com mulher

Com homem

Escreva, em sua linguagem de programação favorita, as seguintes rotinas: *mulher_quer_entrar, homem_quer_entrar, mulher_sai, homem_sai*. Você pode usar os contadores e as técnicas de sincronização que quiser.

52. Reescreva o programa da Figura 2.18 para tratar mais de dois processos.

53. Escreva um problema produtor–consumidor que use threads e compartilhe um buffer comum. Contudo, não use semáforos ou qualquer outra primitiva de sincronização para proteger a estrutura de dados compartilhada. Apenas deixe cada thread ter acesso a eles quando quiser. Use `sleep` e `wakeup` para tratar as condições de buffer cheio e buffer vazio. Veja quanto tempo leva até ocorrer uma condição de disputa fatal. Por exemplo, você pode ter o produtor imprimindo um número a cada intervalo de tempo. Não imprima mais do que um número a cada minuto, porque a E/S poderia afetar as condições de corrida.