2.3 Comunicação entre processos

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

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

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

2.3.1 Condições de corrida

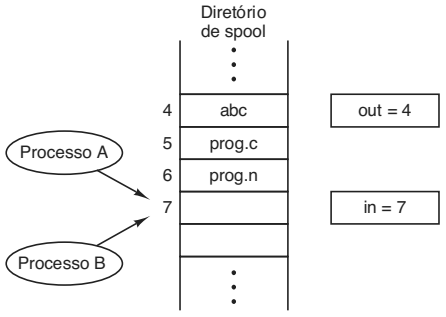
Em alguns sistemas operacionais, processos que estão trabalhando juntos podem compartilhar de alguma memória comum que cada um pode ler e escrever. A memória compartilhada pode encontrar-se na memória principal (possivelmente em uma estrutura de dados de núcleo) ou ser um arquivo compartilhado; o local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

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

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

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

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



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

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

2.3.2 Regiões críticas

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

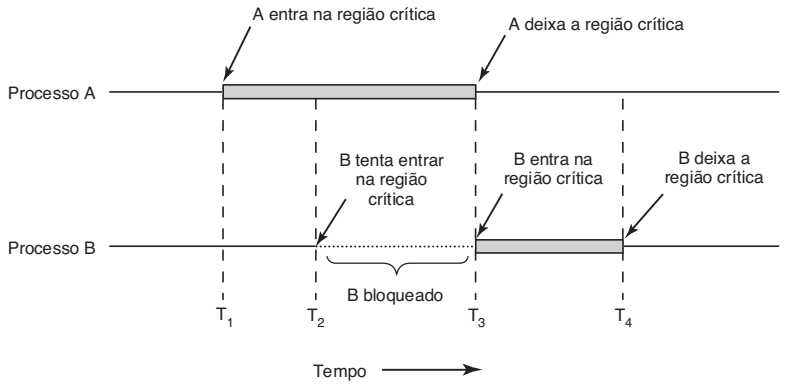
O problema de evitar condições de corrida também pode ser formulado de uma maneira abstrata. Durante parte do tempo, um processo está ocupado realizando computações internas e outras coisas que não levam a condições de corrida. No entanto, às vezes um processo tem de acessar uma memória compartilhada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

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

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

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

FIGURA 2.22 Exclusão mútua usando regiões críticas.



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

2.3.3 Exclusão mútua com espera ocupada

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

Desabilitando interrupções

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

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

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

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

Variáveis do tipo trava

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

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

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

Alternância explícita

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

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

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

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

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

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

(a)

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

(b)

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

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

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

Solução de Peterson

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

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

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

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

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

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

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

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

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

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

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

A instrução TSL

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

```
TSL RX,LOCK
```

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

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

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

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

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

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

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

2.3.4 Dormir e acordar

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

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

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

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

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

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

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

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

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

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

O problema do produtor-consumidor

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

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

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

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

Para expressar chamadas de sistema como sleep e wakeup em C, nós as mostraremos como chamadas para rotinas de biblioteca. Elas não fazem parte da biblioteca C padrão, mas presumivelmente estariam disponíveis em qualquer sistema que de fato tivesse essas chamadas de sistema. As rotinas *insert_item* e *remove_item*, que não são mostradas, lidam com o controle da inserção e retirada de itens do buffer.

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

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

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

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

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

void producer(void)
{
     int item;

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


void consumer(void)
{
     int item;

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

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

2.3.5 Semáforos

Essa era a situação em 1965, quando E. W. Dijkstra (1965) sugeriu usar uma variável inteira para contar o número de sinais de acordar salvos para uso futuro. Em sua proposta, um novo tipo de variável, que ele chamava de **semáforo**, foi introduzido. Um semáforo podia ter o valor 0, indicando que nenhum sinal de despertar fora salvo, ou algum valor positivo se um ou mais sinais de acordar estivessem pendentes.

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

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

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

Solucionando o problema produtor-consumidor usando semáforos

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

FIGURA 2.28 O problema do produtor-consumidor usando semáforos.

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

void producer(void)
{
     int item;

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


void consumer(void)
{
     int item;

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

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

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

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

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

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

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

2.3.6 Mutexes

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

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

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

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