CAPÍTULO 6

IMPASSES

Os sistemas computacionais estão cheios de recursos que podem ser usados somente por um processo de cada vez. Exemplos comuns incluem impressoras, unidades de fita para backup de dados da empresa e entradas nas tabelas internas do sistema. Ter dois processos escrevendo simultaneamente para a impressora gera uma saída ininteligível. Ter dois processos usando a mesma entrada da tabela do sistema de arquivos invariavelmente levará a um sistema de arquivos corrompido. Em consequência, todos os sistemas operacionais têm a capacidade de conceder (temporariamente) acesso exclusivo a um processo a determinados recursos.

Para muitas aplicações, um processo precisa de acesso exclusivo a não somente um recurso, mas a vários. Suponha, por exemplo, que dois processos queiram cada um gravar um documento escaneado em um disco Blu-ray. O processo *A* solicita permissão para usar o scanner e ela lhe é concedida. O processo *B* é programado diferentemente e solicita o gravador Blu-ray primeiro e ele também lhe é concedido. Agora *A* pede pelo gravador Blu-ray, mas a solicitação é suspensa até que *B* o libere. Infelizmente, em vez de liberar o gravador Blu-ray, *B* pede pelo scanner. A essa altura ambos os processos estão bloqueados e assim permanecerão para sempre. Essa situação é chamada de **impasse** (**deadlock**).

Impasses também podem ocorrer entre máquinas. Por exemplo, muitos escritórios têm uma rede de área local com muitos computadores conectados a ela. Muitas vezes dispositivos como scanners, gravadores Blu-ray/DVDs, impressoras e unidades de fitas estão conectados à rede como recursos compartilhados, disponíveis para qualquer usuário em qualquer máquina. Se esses dispositivos puderem ser reservados remotamente (isto é, da máquina da casa do usuário), impasses do mesmo tipo podem ocorrer como descrito. Situações mais complicadas podem provocar impasses envolvendo três, quatro ou mais dispositivos e usuários.

Impasses também podem ocorrer em uma série de outras situações. Em um sistema de banco de dados, por exemplo, um programa pode ter de bloquear vários registros que ele está usando a fim de evitar condições de corrida. Se o processo *A* bloqueia o registro *R1* e o processo *B* bloqueia o registro *R2*, e então cada processo tenta bloquear o registro do outro, também teremos um impasse. Portanto, impasses podem ocorrer em recursos de hardware ou em recursos de software.

Neste capítulo, examinaremos vários tipos de impasses, ver como eles surgem, e estudar algumas maneiras de preveni-los ou evitá-los. Embora esses impasses surjam no contexto de sistemas operacionais, eles também ocorrem em sistemas de bancos de dados e em muitos outros contextos na ciência da computação; então, este material é aplicável, na realidade, a uma ampla gama de sistemas concorrentes.

Muito já foi escrito sobre impasses. Duas bibliografias sobre o assunto apareceram na *Operating Systems Review* e devem ser consultadas para referências (NEWTON, 1979; e ZOBEL, 1983). Embora essas bibliografias sejam muito antigas, a maior parte dos trabalhos sobre impasses foi feita bem antes de 1980, de maneira que eles ainda são úteis.

6.1 Recursos

Uma classe importante de impasses envolve recursos para os quais algum processo teve acesso exclusivo concedido. Esses recursos incluem dispositivos, registros de

dados, arquivos e assim por diante. Para tornar a discussão dos impasses mais geral possível, vamos nos referir aos objetos concedidos como **recursos**. Um recurso pode ser um dispositivo de hardware (por exemplo, uma unidade de Blu-ray) ou um fragmento de informação (por exemplo, um registro em um banco de dados). Um computador normalmente terá muitos recursos diferentes que um processo pode adquirir. Para alguns recursos, várias instâncias idênticas podem estar disponíveis, como três unidades de Blu-rays. Quando várias cópias de um recurso encontram-se disponíveis, qualquer uma delas pode ser usada para satisfazer qualquer pedido pelo recurso. Resumindo, um recurso é qualquer coisa que precisa ser adquirida, usada e liberada com o passar do tempo.

6.1.1 Recursos preemptíveis e não preemptíveis

Há dois tipos de recursos: preemptíveis e não preemptíveis. Um **recurso preemptível** é aquele que pode ser retirado do processo proprietário sem causar-lhe prejuízo algum. A memória é um exemplo de um recurso preemptível. Considere, por exemplo, um sistema com uma memória de usuário de 1 GB, uma impressora e dois processos de 1 GB cada que querem imprimir algo. O processo *A* solicita e ganha a impressora, então começa a computar os valores para imprimir. Antes que ele termine a computação, ele excede a sua parcela de tempo e é mandado para o disco.

O processo *B* executa agora e tenta, sem sucesso no fim das contas, ficar com a impressora. Potencialmente, temos agora uma situação de impasse, pois *A* tem a impressora e *B* tem a memória, e nenhum dos dois pode proceder sem o recurso contido pelo outro. Felizmente, é possível obter por preempção (tomar a memória) de *B* enviando-o para o disco e trazendo *A* de volta. Agora *A* pode executar, fazer sua impressão e então liberar a impressora. Nenhum impasse ocorre.

Um **recurso não preemptível**, por sua vez, é um recurso que não pode ser tomado do seu proprietário atual sem potencialmente causar uma falha. Se um processo começou a ser executado em um Blu-ray, tirar o gravador Blu-ray dele de repente e dá-lo a outro processo resultará em um Blu-ray bagunçado. Gravadores Blu-ray não são preemptíveis em um momento arbitrário.

A questão se um recurso é preemptível depende do contexto. Em um PC padrão, a memória é preemptível porque as páginas sempre podem ser enviadas para o disco para depois recuperá-las. No entanto, em um smartphone que não suporta trocas (swapping) ou paginação, impasses não podem ser evitados simplesmente trocando uma porção da memória.

Em geral, impasses envolvem recursos não preemptíveis. Impasses potenciais que envolvem recursos preemptíveis normalmente podem ser solucionados realocando recursos de um processo para outro. Desse modo, nosso estudo enfocará os recursos não preemptíveis.

A sequência abstrata de eventos necessários para usar um recurso é dada a seguir.

1. Solicitar o recurso.
2. Usar o recurso.
3. Liberar o recurso.

Se o recurso não está disponível quando ele é solicitado, o processo que o está solicitando é forçado a esperar. Em alguns sistemas operacionais, o processo é automaticamente bloqueado quando uma solicitação de recurso falha, e despertado quando ela torna-se disponível. Em outros sistemas, a solicitação falha com um código de erro, e cabe ao processo que está fazendo a chamada esperar um pouco e tentar de novo.

Um processo cuja solicitação de recurso foi negada há pouco, normalmente esperará em um laço estreito solicitando o recurso, dormindo ou tentando novamente. Embora esse processo não esteja bloqueado, para todos os efeitos e propósitos é como se estivesse, pois ele não pode realizar nenhum trabalho útil. Mais adiante, presumiremos que, quando um processo tem uma solicitação de recurso negada, ele é colocado para dormir.

A exata natureza da solicitação de um recurso é altamente dependente do sistema. Em alguns sistemas, uma chamada de sistema request é fornecida para permitir que os processos peçam explicitamente por recursos. Em outros, os únicos recursos que o sistema operacional conhece são arquivos especiais que somente um processo pode ter aberto de cada vez. Esses são abertos pela chamada open usual. Se o arquivo já está sendo usado, o processo chamador é bloqueado até o arquivo ser fechado pelo seu proprietário atual.

6.1.2 Aquisição de recursos

Para alguns tipos de recursos, como registros em um sistema de banco de dados, cabe aos processos do usuário, em vez do sistema, gerenciar eles mesmos o uso de recursos. Uma maneira de permitir isso é associar um semáforo a cada recurso. Esses semáforos são todos inicializados com 1. Também podem ser usadas variáveis do tipo mutex. Os três passos listados são então implementados como um down no semáforo para aquisição e utilização do recurso e, por fim, um up no semáforo para liberação do recurso. Esses passos são mostrados na Figura 6.1(a).

FIGURA 6.1 O uso de um semáforo para proteger recursos. (a) Um recurso. (b) Dois recursos.

```
typedef int semaphore;
semaphore resource_1;

void process_A(void) {
    down(&resource_1);
    use_resource_1( );
    up(&resource_1);
}
```

(a)

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}
```

(b)

Às vezes, processos precisam de dois ou mais recursos. Eles podem ser adquiridos em sequência, como mostrado na Figura 6.1(b). Se mais de dois recursos são necessários, eles são simplesmente adquiridos um depois do outro.

Até aqui, nenhum problema. Enquanto apenas um processo estiver envolvido, tudo funciona bem. É claro, com apenas um processo, não há a necessidade de adquirir formalmente recursos, já que não há competição por eles.

Agora vamos considerar uma situação com dois processos, *A* e *B*, e dois recursos. Dois cenários são descritos na Figura 6.2. Na Figura 6.2(a), ambos os processos solicitam pelos recursos na mesma ordem. Na Figura 6.2(b), eles os solicitam em uma ordem diferente. Essa diferença pode parecer menor, mas não é.

Na Figura 6.2(a), um dos processos adquirirá o primeiro recurso antes do outro. Esse processo então será bem-sucedido na aquisição do segundo recurso e realizará o seu trabalho. Se o outro processo tentar adquirir o recurso 1 antes que ele seja liberado, o outro processo simplesmente será bloqueado até que o recurso esteja disponível.

Na Figura 6.2(b), a situação é diferente. Pode ocorrer que um dos processos adquira ambos os recursos e efetivamente bloqueie o outro processo até concluir seu trabalho. No entanto, pode também acontecer de o processo *A* adquirir o recurso 1 e o processo *B* adquirir o recurso 2. Cada um bloqueará agora quando tentar adquirir o outro recurso. Nenhum processo executará novamente. Má notícia: essa situação é um impasse.

Vemos aqui o que parece ser uma diferença menor em estilo de codificação — qual recurso adquirir primeiro — no fim das contas, faz a diferença entre o programa funcionar ou falhar de uma maneira difícil de ser detectada. Como impasses podem ocorrer tão facilmente, muita pesquisa foi feita para descobrir maneiras de lidar com eles. Este capítulo discute impasses em detalhe e o que pode ser feito a seu respeito.

6.2 Introdução aos impasses

Um impasse pode ser definido formalmente como a seguir:

Um conjunto de processos estará em situação de impasse se cada processo no conjunto estiver esperando por um evento que apenas outro processo no conjunto pode causar.

FIGURA 6.2 (a) Código livre de impasses. (b) Código com impasse potencial.

```
typedef int semaphore;
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}
```

(a)

```
semaphore resource_1;
semaphore resource_2;

void process_A(void) {
    down(&resource_1);
    down(&resource_2);
    use_both_resources( );
    up(&resource_2);
    up(&resource_1);
}

void process_B(void) {
    down(&resource_2);
    down(&resource_1);
    use_both_resources( );
    up(&resource_1);
    up(&resource_2);
}
```

(b)

Como todos os processos estão esperando, nenhum deles jamais causará qualquer evento que possa despertar um dos outros membros do conjunto, e todos os processos continuam a esperar para sempre. Para esse modelo, presumimos que os processos têm um único thread e que nenhuma interrupção é possível para despertar um processo bloqueado. A condição de não haver interrupções é necessária para evitar que um processo em situação de impasse seja acordado por, digamos, um alarme, e então cause eventos que liberem outros processos no conjunto.

Na maioria dos casos, o evento que cada processo está esperando é a liberação de algum recurso atualmente possuído por outro membro do conjunto. Em outras palavras, cada membro do conjunto de processos em situação de impasse está esperando por um recurso que é de propriedade do processo em situação de impasse. Nenhum dos processos pode executar, nenhum deles pode liberar quaisquer recursos e nenhum pode ser desperto. O número de processos e o número e tipo de recursos possuídos e solicitados não têm importância. Esse resultado é válido para qualquer tipo de recurso, incluindo hardwares e softwares. Esse tipo de impasse é chamado de **impasse de recurso**, e é provavelmente o tipo mais comum, mas não o único. Estudaremos primeiro os impasses de recursos em detalhe e, então, no fim do capítulo, retornaremos brevemente para os outros tipos de impasses.

6.2.1 Condições para ocorrência de impasses

Coffman et al. (1971) demonstraram que quatro condições têm de ser válidas para haver um impasse (de recurso):

1. Condição de exclusão mútua. Cada recurso está atualmente associado a exatamente um processo ou está disponível.
2. Condição de posse e espera. Processos atualmente de posse de recursos que foram concedidos antes podem solicitar novos recursos.
3. Condição de não preempção. Recursos concedidos antes não podem ser tomados à força de um processo. Eles precisam ser explicitamente liberados pelo processo que os têm.
4. Condição de espera circular. Deve haver uma lista circular de dois ou mais processos, cada um deles esperando por um processo de posse do membro seguinte da cadeia.

Todas essas quatro condições devem estar presentes para que um impasse de recurso ocorra. Se uma delas estiver ausente, nenhum impasse de recurso será possível.

Vale a pena observar que cada condição relaciona-se com uma política que um sistema pode ter ou não. Pode um dado recurso ser designado a mais um processo ao mesmo tempo? Pode um processo ter a posse de um recurso e pedir por outro? Os recursos podem passar por preempção? Esperas circulares podem existir? Mais tarde veremos como os impasses podem ser combatidos tentando negar-lhes algumas dessas condições.

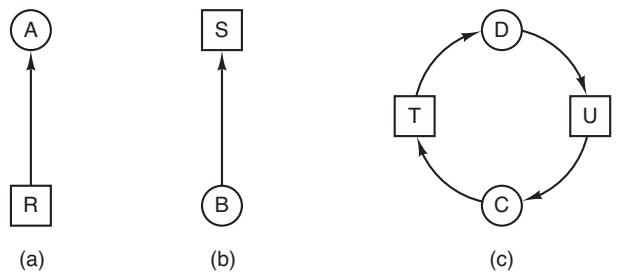
6.2.2 Modelagem de impasses

Holt (1972) demonstrou como essas quatro condições podem ser modeladas com grafos dirigidos. Os grafos têm dois tipos de nós: processos, mostrados como círculos, e recursos, mostrados como quadrados. Um arco direcionado de um nó de recurso (quadrado) para um nó de processo (círculo) significa que o recurso foi previamente solicitado, concedido e está atualmente com aquele processo. Na Figura 6.3(a), o recurso *R* está atualmente alocado ao processo *A*.

Um arco direcionado de um processo para um recurso significa que o processo está atualmente bloqueado esperando por aquele recurso. Na Figura 6.3(b), o processo *B* está esperando pelo recurso *S*. Na Figura 6.3(c) vemos um impasse: o processo *C* está esperando pelo recurso *T*, que atualmente está sendo usado pelo processo *D*. O processo *D* não está prestes a liberar o recurso *T* porque ele está esperando pelo recurso *U*, sendo usado por *C*. Ambos os processos esperarão para sempre. Um ciclo no grafo significa que há um impasse envolvendo os processos e recursos no ciclo (presumindo que há um recurso de cada tipo). Nesse exemplo, o ciclo é *C* – *T* – *D* – *U* – *C*.

Agora vamos examinar um exemplo de como grafos de recursos podem ser usados. Imagine que temos três

FIGURA 6.3 Grafos de alocação de recursos. (a) Processo de posse de um recurso. (b) Solicitação de um recurso. (c) Impasse.

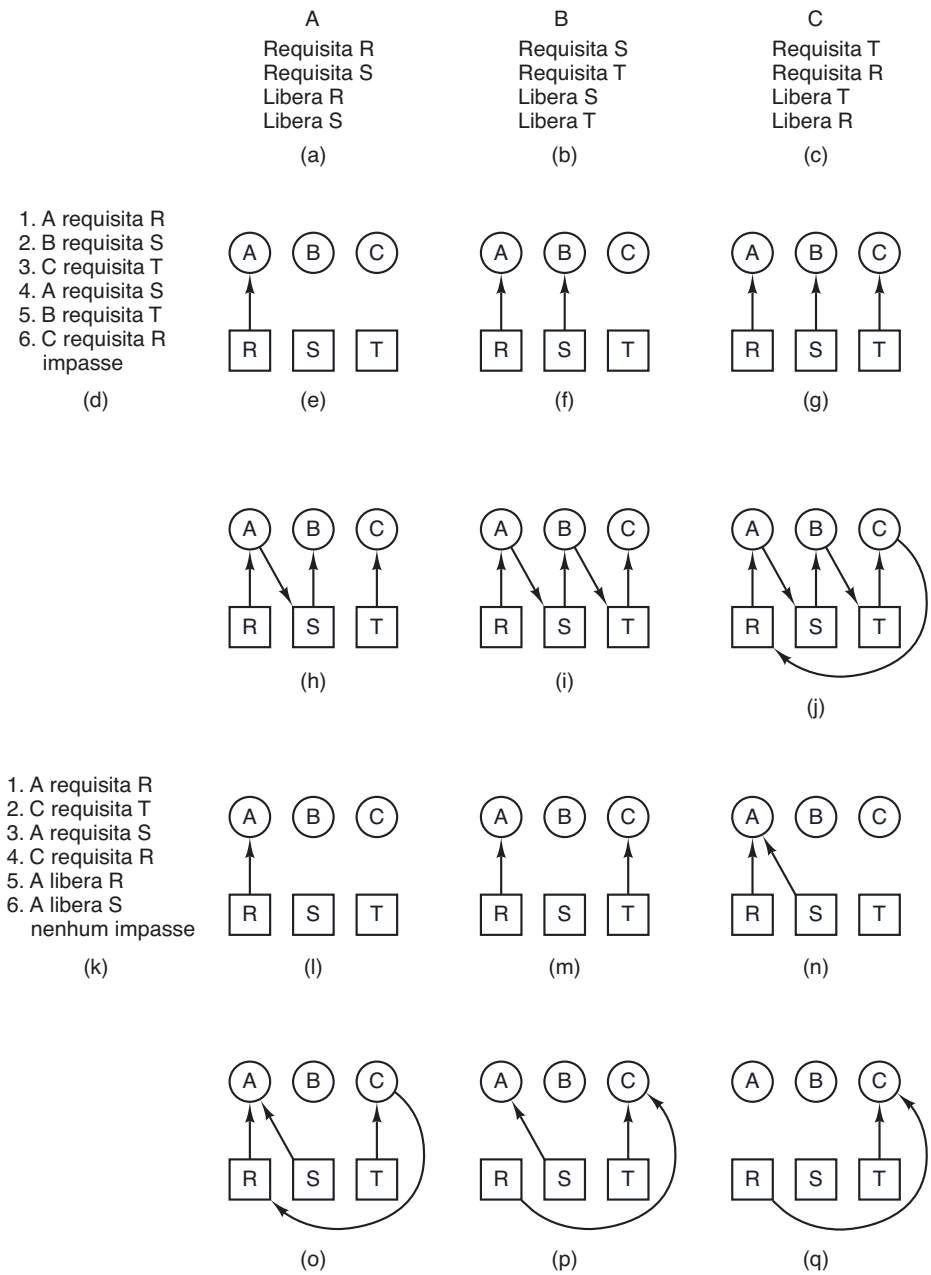


processos, *A, B* e *C*, e três recursos, *R, S* e *T*. As solicitações e as liberações dos três processos são dadas na Figura 6.4(a) a (c). O sistema operacional está liberado para executar qualquer processo desbloqueado a qualquer instante, então ele poderia decidir executar *A* até *A* ter concluído o seu trabalho, então executar *B* até sua conclusão e por fim executar *C*.

Essa ordem de execução não leva a impasse algum (pois não há competição por recursos), mas também não há paralelismo algum. Além de solicitar e liberar recursos, processos calculam e realizam E/S. Quando os processos são executados sequencialmente, não existe a possibilidade de enquanto um processo espera pela E/S, outro poder usar a CPU. Desse modo, executar os processos de maneira estritamente sequencial pode não ser uma opção ótima. Por outro lado, se nenhum dos processos realizar qualquer E/S, o algoritmo trabalho mais curto primeiro é melhor do que a alternância circular, de maneira que em determinadas circunstâncias executar todos os processos sequencialmente pode ser a melhor maneira.

Vamos supor agora que os processos realizam E/S e computação, então a alternância circular é um algoritmo de escalonamento razoável. As solicitações de recursos

FIGURA 6.4 Um exemplo de como um impasse ocorre e como ele pode ser evitado.



podem ocorrer na ordem da Figura 6.4(d). Se as seis solicitações forem executadas nessa ordem, os seis grafos de recursos resultantes serão como mostrado na Figura 6.4 (e) a (j). Após a solicitação 4 ter sido feita, *A* bloqueia esperando por *S*, como mostrado na Figura 6.4(h). Nos próximos dois passos, *B* e *C* também bloqueiam, em última análise levando a um ciclo e ao impasse da Figura 6.4(j).

No entanto, como já mencionamos, não é exigido do sistema operacional que execute os processos em qualquer ordem especial. Em particular, se conceder uma solicitação específica pode levar a um impasse, o sistema operacional pode simplesmente suspender o processo sem conceder a solicitação (isto é, apenas não escalonar o processo) até que seja seguro. Na Figura 6.4, se o sistema operacional soubesse a respeito do impasse iminente, ele poderia suspender *B* em vez de conceder-lhe *S*. Ao executar apenas *A* e *C*, teríamos as solicitações e liberações da Figura 6.4(k) em vez da Figura 6.4(d). Essa sequência conduz aos grafos de recursos da Figura 6.4(l) a (q), que não levam a um impasse.

Após o passo (q), S pode ser concedido ao processo *B* porque *A* foi concluído e *C* tem tudo de que ele precisa. Mesmo que *B* bloqueie quando solicita *T*, nenhum impasse pode ocorrer. *B* apenas esperará até que *C* tenha terminado.

Mais tarde neste capítulo estudaremos um algoritmo detalhado para tomar decisões de alocação que não levam a um impasse. Por ora, basta compreender que grafos de recursos são uma ferramenta que nos deixa ver se uma determinada sequência de solicitação/liberação pode levar a um impasse. Apenas atendemos às solicitações e liberações passo a passo e após cada passo conferimos o grafo para ver se ele contém algum ciclo. Se afirmativo, temos um impasse; se não, não há impasse. Embora nosso tratamento de grafos de recursos tenha sido para o caso de um único recurso de cada tipo, grafos de recursos também podem ser generalizados para lidar com múltiplos recursos do mesmo tipo (HOLT, 1972).

Em geral, quatro estratégias são usadas para lidar com impasses.

1. Simplesmente ignorar o problema. Se você o ignorar, quem sabe ele ignore você.
2. Detecção e recuperação. Deixe-os ocorrer, detecte-os e tome as medidas cabíveis.
3. Evitar dinamicamente pela alocação cuidadosa de recursos.
4. Prevenção, ao negar estruturalmente uma das quatro condições.

Nas próximas quatro seções, examinaremos cada um desses métodos.

6.3 Algoritmo do avestruz

A abordagem mais simples é o algoritmo do avestruz: enfie a cabeça na areia e finja que não há um problema.[1] As pessoas reagem a essa estratégia de maneiras diferentes. Matemáticos a consideram inaceitável e dizem que os impasses devem ser evitados a todo custo. Engenheiros perguntam qual a frequência que se espera que o problema ocorra, qual a frequência com que ocorrem quedas no sistema por outras razões e quão sério um impasse é realmente. Se os impasses ocorrem na média uma vez a cada cinco anos, mas quedas do sistema decorrentes de falhas no hardware e defeitos no sistema operacional ocorrem uma vez por semana, a maioria dos engenheiros não estaria disposta a pagar um alto preço em termos de desempenho ou conveniência para eliminar os impasses.

Para tornar esse contraste mais específico, considere um sistema operacional que bloqueia o chamador quando uma chamada de sistema `open` para um dispositivo físico como um driver de Blu-ray ou uma impressora não pode ser executada porque o dispositivo está ocupado. Tipicamente cabe ao driver do dispositivo decidir qual ação tomar sob essas circunstâncias. Bloquear ou retornar um código de erro são duas possibilidades óbvias. Se um processo tiver sucesso em abrir a unidade de Blu-ray e outro tiver sucesso em abrir a impressora e então os dois tentarem abrir o recurso um do outro e forem bloqueados tentando, temos um impasse. Poucos sistemas atuais detectarão isso.

6.4 Detecção e recuperação de impasses

Uma segunda técnica é a detecção e recuperação. Quando essa técnica é usada, o sistema não tenta evitar a ocorrência dos impasses. Em vez disso, ele os deixa ocorrer, tenta detectá-los quando acontecem e então toma alguma medida para recuperar-se após o fato. Nesta seção examinaremos algumas maneiras como os impasses podem ser detectados e tratados.

1 Na realidade, essa parte do folclore é uma bobagem. Avestruzes conseguem correr a 60 km/h e seu coice é poderoso o suficiente para matar qualquer leão com visões de um grande prato de frango, e os leões sabem disso. (N. A.)

6.4.1 Detecção de impasses com um recurso de cada tipo

Vamos começar com o caso mais simples: existe apenas um recurso de cada tipo. Um sistema assim poderia ter um scanner, um gravador Blu-ray, uma plotter e uma unidade de fita, mas não mais do que um de cada classe de recurso. Em outras palavras, estamos excluindo sistemas com duas impressoras por ora. Trataremos deles mais tarde, usando um método diferente.

Para esse sistema, podemos construir um grafo de recursos do tipo ilustrado na Figura 6.3. Se esse grafo contém um ou mais ciclos, há um impasse. Qualquer processo que faça parte de um ciclo está em situação de impasse. Se não existem ciclos, o sistema não está em impasse.

Como um exemplo de um sistema mais complexo do que aqueles que examinamos até o momento, considere um sistema com sete processos, *A* até *G* e seis recursos, *R* até *W*. O estado de quais recursos estão sendo atualmente usados e quais estão sendo solicitados é o seguinte:

1. O processo *A* possui *R* e solicita *S*.
2. O processo *B* não possui nada, mas solicita *T*.
3. O processo *C* não possui nada, mas solicita *S*.
4. O processo *D* possui *U* e solicita *S* e *T*.
5. O processo *E* possui *T* e solicita *V*.
6. O processo *F* possui *W* e solicita *S*.
7. O processo *G* possui *V* e solicita *U*.

A questão é: "Esse sistema está em impasse? E se estiver, quais processos estão envolvidos?".

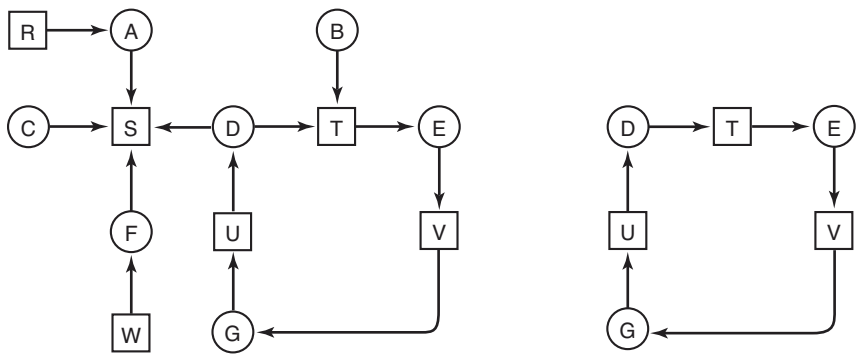
Para responder, podemos construir o grafo de recursos da Figura 6.5(a). Esse grafo contém um ciclo, que pode ser visto por inspeção visual. O ciclo é mostrado na Figura 6.5(b). Desse ciclo, podemos ver que os processos *D, E* e *G* estão todos em situação de impasse. Os processos *A, C* e *F* não estão em situação de impasse porque *S* pode ser alocado para qualquer um deles, permitindo sua conclusão e retorno do recurso. Então os outros dois podem obtê-lo por sua vez e também ser finalizados. (Observe que a fim de tornar esse exemplo mais interessante permitimos que processos, a saber *D*, solicitasse por dois recursos ao mesmo tempo.)

Embora seja relativamente simples escolher os processos em situação de impasse mediante inspeção visual de um grafo simples, para usar em sistemas reais precisamos de um algoritmo formal para detectar impasses. Muitos algoritmos para detectar ciclos em grafos direcionados são conhecidos. A seguir exibiremos um algoritmo simples que inspeciona um grafo e termina quando ele encontrou um ciclo ou quando demonstrou que nenhum existe. Ele usa uma estrutura de dados dinâmica, *L*, uma lista de nós, assim como uma lista de arcos. Durante o algoritmo, a fim de evitar inspeções repetidas, arcos serão marcados para indicar que já foram inspecionados.

O algoritmo opera executando os passos a seguir como especificado:

1. Para cada nó, *N*, no grafo, execute os cinco passos a seguir com *N* como o nó de partida.
2. Inicialize *L* como uma lista vazia e designe todos os arcos como desmarcados.
3. Adicione o nó atual ao final de *L* e confira para ver se o nó aparece agora em *L* duas vezes. Se ele aparecer, o grafo contém um ciclo (listado em *L*) e o algoritmo termina.
4. A partir do referido nó, verifique se há algum arco de saída desmarcado. Se afirmativo, vá para o passo 5; se não, vá para o passo 6.
5. Escolha aleatoriamente um arco de saída desmarcado e marque-o. Então siga-o para gerar o novo nó atual e vá para o passo 3.
6. Se esse nó é o inicial, o grafo não contém ciclo algum e o algoritmo termina. De outra maneira,

FIGURA 6.5 (a) Um grafo de recursos. (b) Um ciclo extraído de (a).



chegamos agora a um beco sem saída. Remova-o e volte ao nó anterior, isto é, aquele que era atual imediatamente antes desse, faça dele o nó atual e vá para o passo 3.

O que esse algoritmo faz é tomar cada nó, um de cada vez, como a raiz do que ele espera ser uma árvore e então realiza uma busca do tipo busca em profundidade nele. Se acontecer de ele voltar a um nó que já havia encontrado, então o algoritmo encontrou um ciclo. Se ele exaurir todos os arcos de um dado nó, ele retorna ao nó anterior. Se ele retornar até a raiz e não conseguir seguir adiante, o subgrafo alcançável a partir do nó atual não contém ciclo algum. Se essa propriedade for válida para todos os nós, o grafo inteiro está livre de ciclos, então o sistema não está em impasse.

Para ver como o algoritmo funciona na prática, vamos usá-lo no grafo da Figura 6.5(a). A ordem de processamento dos nós é arbitrária; portanto, vamos apenas inspecioná-los da esquerda para a direita, de cima para baixo, executando primeiro o algoritmo começando em *R*, então sucessivamente *A, B, C, S, D, T, E, F* e assim por diante. Se depararmos com um ciclo, o algoritmo para.

Começamos em *R* e inicializamos *L* como lista vazia. Então adicionamos *R* à lista e vamos para a única possibilidade, *A*, e a adicionamos a *L*, resultando em *L* = [*R, A*]. De *A* vamos para *S*, resultando em *L* = [*R, A, S*]. *S* não tem arcos de saída, então trata-se de um beco sem saída, forçando-nos a recuar para *A*. Já que *A* não tem arcos de saída desmarcados, recuamos para *R*, completando nossa inspeção de *R*.

Agora reinicializamos o algoritmo começando em *A*, reinicializando *L* para a lista vazia. Essa busca também para rapidamente, então começamos novamente em *B*. De *B* continuamos para seguir os arcos de saída até chegarmos a *D*, momento em que *L* = [*B, T, E, V, G, U, D*]. Agora precisamos fazer uma escolha (ao acaso). Se escolhermos *S*, chegaremos a um beco sem saída e recuaremos para *D*. Na segunda tentativa, escolhemos *T* e atualizamos *L* para ser [*B, T, E, V, G, U, D, T*], ponto em que descobrimos o ciclo e paramos o algoritmo.

Esse algoritmo está longe de ser ótimo. Para um algoritmo melhor, ver Even (1979). Mesmo assim, ele demonstra que existe um algoritmo para detecção de impasses.

6.4.2 Detecção de impasses com múltiplos recursos de cada tipo

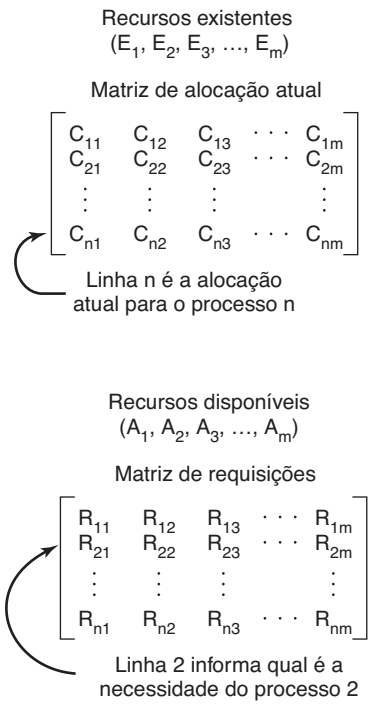
Quando existem múltiplas cópias de alguns dos recursos, é necessária uma abordagem diferente para detectar impasses. Apresentaremos agora um algoritmo baseado em matrizes para detectar impasses entre *n* processos, P_1 a P_n. Seja *m* o número de classes de recursos, com E_1 recursos de classe 1, E_2 recursos de classe 2, e geralmente, E_i recursos de classe *i* ($1 \le i \le m$). *E* é o **vetor de recursos existentes**. Ele fornece o número total de instâncias de cada recurso existente. Por exemplo, se a classe 1 for de unidades de fita, então $E_1 = 2$ significa que o sistema tem duas unidades de fita.

A qualquer instante, alguns dos recursos são alocados e não se encontram disponíveis. Seja *A* o **vetor de recursos disponíveis**, com A_i dando o número de instâncias de recurso *i* atualmente disponíveis (isto é, não alocadas). Se ambas as nossas unidades de fita estiverem alocadas, A_1 será 0.

Agora precisamos de dois arranjos, *C*, a **matriz de alocação atual** e *R*, a **matriz de requisição**. A *i*-ésima linha de *C* informa quantas instâncias de cada classe de recurso P_i atualmente possui. Desse modo, C_{ij} é o número de instâncias do recurso *j* que são possuídas pelo processo *i*. Similarmente, R_{ij} é o número de instâncias do recurso *j* que P_i quer. Essas quatro estruturas de dados são mostradas na Figura 6.6.

Uma importante condição invariante se aplica a essas quatro estruturas de dados. Em particular, cada

FIGURA 6.6 As quatro estruturas de dados necessárias ao algoritmo de detecção de impasses.



recurso é alocado ou está disponível. Essa observação significa que

$$\sum_{i=1}^{n} C_{ij} + A_j = E_j$$

Em outras palavras, se somarmos todas as instâncias do recurso j que foram alocadas e a isso adicionarmos todas as instâncias que estão disponíveis, o resultado será o número de instâncias existentes daquela classe de recursos.

O algoritmo de detecção de impasses é baseado em vetores de comparação. Vamos definir a relação $A \leq B$, entre dois vetores A e B, para indicar que cada elemento de A é menor do que ou igual ao elemento correspondente de B. Matematicamente, $A \leq B$ se mantém se e somente se $A_i \leq B_i$ para $1 \leq i \leq m$.

Diz-se que cada processo, inicialmente, está desmarcado. À medida que o algoritmo progride, os processos serão marcados, indicando que eles são capazes de completar e portanto não estão em uma situação de impasse. Quando o algoritmo termina, sabe-se que quaisquer processos desmarcados estão em situação de impasse. Esse algoritmo presume o pior cenário possível: todos os processos mantêm todos os recursos adquiridos até que terminem.

O algoritmo de detecção de impasses pode ser dado agora como a seguir.

1. Procure por um processo desmarcado, P_i, para o qual a i-ésima linha de R seja menor ou igual a A.
2. Se um processo assim for encontrado, adicione a i-ésima linha de C a A, marque o processo e volte ao passo 1.
3. Se esse processo não existir, o algoritmo termina.

Quando o algoritmo terminar, todos os processos desmarcados, se houver algum, estarão em situação de impasse.

O que o algoritmo está fazendo no passo 1 é procurar por um processo que possa ser executado até o fim. Esse processo é caracterizado por ter demandas de recursos que podem ser atendidas pelos recursos atualmente disponíveis. O processo escolhido é então executado até ser concluído, momento em que ele retorna os recursos a ele alocados para o pool de recursos disponíveis. Ele então é marcado como concluído. Se todos os processos são, em última análise, capazes de serem executados até a sua conclusão, nenhum deles está em situação de impasse. Se alguns deles jamais puderem ser concluídos, eles estão em situação de impasse. Embora o algoritmo seja não determinístico (pois ele pode executar os processos em qualquer ordem possível), o resultado é sempre o mesmo.

Como um exemplo de como o algoritmo de detecção de impasses funciona, observe a Figura 6.7. Aqui temos três processos e quatro classes de recursos, os quais rotulamos arbitrariamente como unidades de fita, plotters, scanners e unidades de Blu-ray. O processo 1 tem um scanner. O processo 2 tem duas unidades de fitas e uma unidade de Blu-ray. O processo 3 tem uma plotter e dois scanners. Cada processo precisa de recursos adicionais, como mostrado na matriz R.

Para executar o algoritmo de detecção de impasses, procuramos por um processo cuja solicitação de recursos possa ser satisfeita. O primeiro não pode ser satisfeito, pois não há uma unidade de Blu-ray disponível. O segundo também não pode ser satisfeito, pois não há um scanner liberado. Felizmente, o terceiro pode ser satisfeito, de maneira que o processo 3 executa e eventualmente retorna todos os seus recursos, resultando em

A = (2 2 2 0)

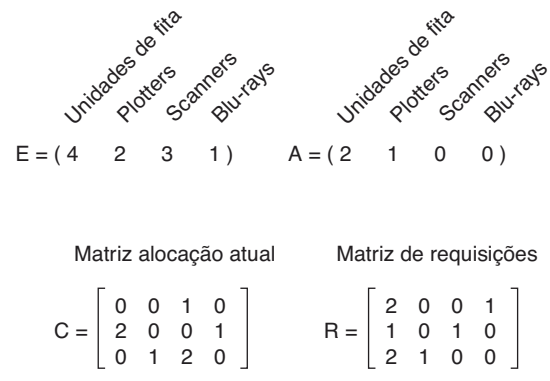
Nesse ponto o processo 2 pode executar e retornar os seus recursos, resultando em

A = (4 2 2 1)

Agora o restante do processo pode executar. Não há um impasse no sistema.

Agora considere uma variação menor da situação da Figura 6.7. Suponha que o processo 3 precisa de uma unidade de Blu-ray, assim como as duas unidades de fitas e a plotter. Nenhuma das solicitações pode ser satisfeita, então o sistema inteiro eventualmente estará em uma situação de impasse. Mesmo se dermos ao processo 3 suas duas unidades de fitas e uma plotter, o sistema entrará em uma situação de impasse quando ele solicitar a unidade de Blu-ray.

FIGURA 6.7 Um exemplo para o algoritmo de detecção de impasses.



Agora que sabemos como detectar impasses (pelo menos com solicitações de recursos estáticos conhecidas antecipadamente), surge a questão de quando procurar por eles. Uma possibilidade é verificar todas as vezes que uma solicitação de recursos for feita. Isso é certo que irá detectá-las o mais cedo possível, mas é uma alternativa potencialmente cara em termos de tempo da CPU. Uma estratégia possível é fazer a verificação a cada *k* minutos, ou talvez somente quando a utilização da CPU cair abaixo de algum limiar. A razão para considerar a utilização da CPU é que se um número suficiente de processos estiver em situação de impasse, haverá menos processos executáveis, e a CPU muitas vezes estará ociosa.

6.4.3 Recuperação de um impasse

Suponha que nosso algoritmo de detecção de impasses teve sucesso e detectou um impasse. O que fazer então? Alguma maneira é necessária para recuperar o sistema e colocá-lo em funcionamento novamente. Nesta seção discutiremos várias maneiras de conseguir isso. Nenhuma delas é especialmente atraente, no entanto.

Recuperação mediante preempção

Em alguns casos pode ser viável tomar temporariamente um recurso do seu proprietário atual e dá-lo a outro processo. Em muitos casos, pode ser necessária a intervenção manual, especialmente em sistemas operacionais de processamento em lote executando em computadores de grande porte.

Por exemplo, para tomar uma impressora a laser de seu processo-proprietário, o operador pode juntar todas as folhas já impressas e colocá-las em uma pilha. Então o processo pode ser suspenso (marcado como não executável). Nesse ponto, a impressora pode ser alocada para outro processo. Quando esse processo terminar, a pilha de folhas impressas pode ser colocada de volta na bandeja de saída da impressora, e o processo original reinicializado.

A capacidade de tirar um recurso de um processo, entregá-lo a outro para usá-lo e então devolvê-lo sem que o processo note isso é algo altamente dependente da natureza do recurso. A recuperação dessa maneira é com frequência difícil ou impossível. Escolher o processo a ser suspenso depende em grande parte de quais processos têm recursos que podem ser facilmente devolvidos.

Recuperação mediante retrocesso

Se os projetistas de sistemas e operadores de máquinas souberem que a probabilidade da ocorrência de impasses é grande, eles podem arranjar para que os processos gerem **pontos de salvaguarda** (checkpoints) periodicamente. Gerar este ponto de salvaguarda de um processo significa que o seu estado é escrito para um arquivo, para que assim ele possa ser reinicializado mais tarde. O ponto de salvaguarda contém não apenas a imagem da memória, mas também o estado dos recursos, em outras palavras, quais recursos estão atualmente alocados para o processo. Para serem mais eficientes, novos pontos de salvaguarda não devem sobrescrever sobre os antigos, mas serem escritos para os arquivos novos, de maneira que à medida que o processo executa, toda uma sequência se acumula.

Quando um impasse é detectado, é fácil ver quais recursos são necessários. Para realizar a recuperação, um processo que tem um recurso necessário é retrocedido até um ponto no tempo anterior ao momento em que ele adquiriu aquele recurso, reiniciando em um de seus pontos de salvaguarda anteriores. Todo o trabalho realizado desde o ponto de salvaguarda é perdido (por exemplo, a produção impressa desde o ponto de salvaguarda deve ser descartada, tendo em vista que ela será impressa novamente). Na realidade, o processo é reiniciado para um momento anterior quando ele não tinha o recurso, que agora é alocado para um dos processos em situação de impasse. Se o processo reiniciado tentar adquirir o recurso novamente, terá de esperar até que ele se torne disponível.

Recuperação mediante a eliminação de processos

A maneira mais bruta de eliminar um impasse, mas também a mais simples, é matar um ou mais processos. Uma possibilidade é matar um processo no ciclo. Com um pouco de sorte, os outros processos serão capazes de continuar. Se isso não ajudar, essa ação pode ser repetida até que o ciclo seja rompido.

Como alternativa, um processo que não está no ciclo pode ser escolhido como vítima a fim liberar os seus recursos. Nessa abordagem, o processo a ser morto é cuidadosamente escolhido porque ele tem em mãos recursos que algum processo no ciclo precisa. Por exemplo, um processo pode ter uma impressora e querer uma plotter, com outro processo tendo uma plotter e querendo uma impressora. Ambos estão em situação de impasse. Um terceiro processo pode ter outra impressora idêntica e outra plotter idêntica e estar executando feliz da vida. Matar o terceiro processo liberará esses recursos e acabará com o impasse envolvendo os dois primeiros.

Sempre que possível, é melhor matar um processo que pode ser reexecutado desde o início sem efeitos danosos. Por exemplo, uma compilação sempre pode ser

reexecutada, pois tudo o que ela faz é ler um arquivo-fonte e produzir um arquivo-objeto. Se ela for morta durante uma execução, a primeira execução não terá influência alguma sobre a segunda.

Por outro lado, um processo que atualiza um banco de dados nem sempre pode executar uma segunda vez com segurança. Se ele adicionar 1 a algum campo de uma tabela no banco de dados, executá-lo uma vez, matá-lo e então executá-lo novamente adicionará 2 ao campo, o que é incorreto.

6.5 Evitando impasses

Na discussão da detecção de impasses, presumimos tacitamente que, quando um processo pede por recursos, ele pede por todos eles ao mesmo tempo (a matriz *R* da Figura 6.6). Na maioria dos sistemas, no entanto, os recursos são solicitados um de cada vez. O sistema precisa ser capaz de decidir se conceder um recurso é seguro ou não e fazer a alocação somente quando for. Desse modo, surge a questão: existe um algoritmo que possa sempre evitar o impasse fazendo a escolha certa o tempo inteiro? A resposta é um sim qualificado — podemos evitar impasses, mas somente se determinadas informações estiverem disponíveis. Nesta seção examinaremos maneiras de evitar um impasse por meio da alocação cuidadosa de recursos.

6.5.1 Trajetórias de recursos

Os principais algoritmos para evitar impasses são baseados no conceito de estados seguros. Antes de descrevê-los, faremos uma ligeira digressão para examinar o conceito de segurança em uma maneira gráfica e fácil de compreender. Embora a abordagem gráfica não se traduza diretamente em um algoritmo utilizável, ela proporciona um bom sentimento intuitivo para a natureza do problema.

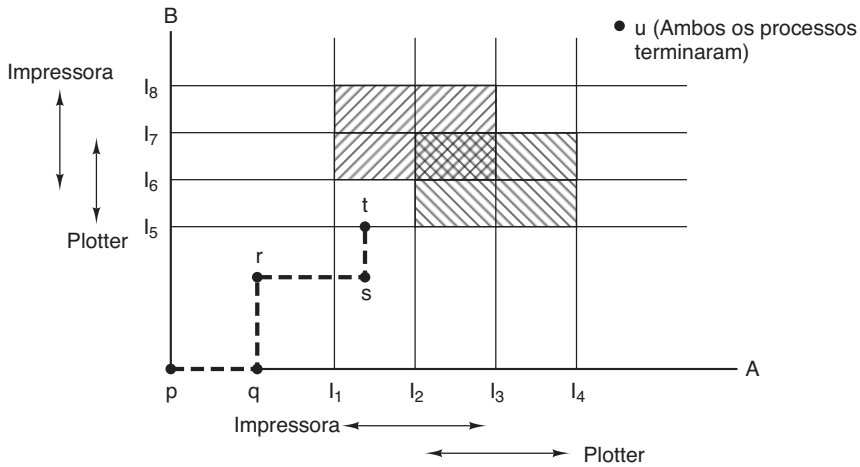
Na Figura 6.8 vemos um modelo para lidar com dois processos e dois recursos, por exemplo, uma impressora e uma plotter. O eixo horizontal representa o número de instruções executadas pelo processo *A*. O eixo vertical representa o número de instruções executadas pelo processo *B*. Em I_1 o processo *A* solicita uma impressora; em I_2 ele precisa de uma plotter. A impressora e a plotter são liberadas em I_3 e I_4, respectivamente. O processo *B* precisa da plotter de I_5 a I_7 e a impressora, de I_6 a I_8.

Todo ponto no diagrama representa um estado de junção dos dois processos. No início, o estado está em *p*, com nenhum processo tendo executado quaisquer instruções. Se o escalonador escolher executar *A* primeiro, chegamos ao ponto *q*, no qual *A* executou uma série de instruções, mas *B* não executou nenhuma. No ponto *q* a trajetória torna-se vertical, indicando que o escalonador escolheu executar *B*. Com um único processador, todos os caminhos devem ser horizontais ou verticais, jamais diagonais. Além disso, o movimento é sempre para o norte ou leste, jamais para o sul ou oeste (pois processos não podem voltar atrás em suas execuções, é claro).

Quando *A* cruza a linha I_1 no caminho de *r* para *s*, ele solicita a impressora e esta lhe é concedida. Quando *B* atinge o pont o *t*, ele solicita a plotter.

As regiões sombreadas são especialmente interessantes. A região com linhas inclinadas à direita representa ambos os processos tendo a impressora. A regra

FIGURA 6.8 A trajetória de recursos de dois processos.



da exclusão mútua torna impossível entrar essa região. Similarmente, a região sombreada no outro sentido representa ambos processos tendo a plotter e é igualmente impossível.

Sempre que o sistema entrar no quadrado delimitado por I_1 e I_2 nas laterais e I_5 e I_6 na parte de cima e de baixo, ele eventualmente entrará em uma situação de impasse quando chegar à interseção de I_2 e I_6. Nesse ponto, *A* está solicitando a plotter e *B*, a impressora, e ambas já foram alocadas. O quadrado inteiro é inseguro e não deve ser adentrado. No ponto *t*, a única coisa segura a ser feita é executar o processo *A* até ele chegar a I_4. Além disso, qualquer trajetória para *u* será segura.

A questão importante a atentarmos aqui é que no ponto *t*, *B* está solicitando um recurso. O sistema precisa decidir se deseja concedê-lo ou não. Se a concessão for feita, o sistema entrará em uma região insegura e eventualmente em uma situação de impasse. Para evitar o impasse, *B* deve ser suspenso até que *A* tenha solicitado e liberado a plotter.

6.5.2 Estados seguros e inseguros

Os algoritmos que evitam impasses que estudaremos usam as informações da Figura 6.6. A qualquer instante no tempo, há um estado atual consistindo de *E, A, C* e *R*. Diz-se de um estado que ele é **seguro** se existir alguma ordem de escalonamento na qual todos os processos puderem ser executados até sua conclusão mesmo que todos eles subitamente solicitem seu número máximo de recursos imediatamente. É mais fácil ilustrar esse conceito por meio de um exemplo usando um recurso. Na Figura 6.9(a) temos um estado no qual *A* tem três instâncias do recurso, mas talvez precise de até nove instâncias. *B* atualmente tem duas e talvez precise de até quatro no total, mais tarde. De modo similar, *C* também tem duas, mas talvez precise de cinco adicionais. Existe um total de 10 instâncias de recursos, então com sete recursos já alocados, três ainda estão livres.

O estado da Figura 6.9(a) é seguro porque existe uma sequência de alocações que permite que todos os processos sejam concluídos. A saber, o escalonador pode apenas executar *B* exclusivamente, até ele pedir e receber mais duas instâncias do recurso, levando ao estado da Figura 6.9(b). Quando *B* termina, temos o estado da Figura 6.9(c). Então o escalonador pode executar *C*, levando em consequência à Figura 6.9(d). Quando *C* termina, temos a Figura 6.9(e). Agora *A* pode ter as seis instâncias do recurso que ele precisa e também terminar. Assim, o estado da Figura 6.9(a) é seguro porque o sistema, pelo escalonamento cuidadoso, pode evitar o impasse.

Agora suponha que tenhamos o estado inicial mostrado na Figura 6.10(a), mas dessa vez *A* solicita e recebe outro recurso, gerando a Figura 6.10(b). É possível encontrarmos uma sequência que seja garantida que funcione? Vamos tentar. O escalonador poderia executar *B* até que ele pedisse por todos os seus recursos, como mostrado na Figura 6.10(c).

Por fim, *B* termina e temos o estado da Figura 6.10(d). Nesse ponto estamos presos. Temos apenas quatro instâncias do recurso disponíveis, e cada um dos processos ativos precisa de cinco. Não há uma sequência que garanta a conclusão. Assim, a decisão de alocação que moveu o sistema da Figura 6.10(a) para a Figura 6.10(b) foi de um estado seguro para um inseguro. Executar *A* ou *C* em seguida começando na Figura 6.10(b) também não funciona. Em retrospectiva, a solicitação de *A* não deveria ter sido concedida.

Vale a pena observar que um estado inseguro não é um estado em situação de impasse. Começando na Figura 6.10(b), o sistema pode executar por um tempo. Na realidade, um processo pode até terminar. Além disso, é possível que *A* consiga liberar um recurso antes de pedir por mais, permitindo a *C* que termine e evitando inteiramente um impasse. Assim, a diferença entre um estado seguro e um inseguro é que a partir de um seguro o sistema pode *garantir* que todos os processos terminarão; a partir de um estado inseguro, nenhuma garantia nesse sentido pode ser dada.

FIGURA 6.9 Demonstração de que o estado em (a) é seguro.

	Possui	máximo
A	3	9
B	2	4
C	2	7

Disponível: 3
(a)

	Possui	máximo
A	3	9
B	4	4
C	2	7

Disponível: 1
(b)

	Possui	máximo
A	3	9
B	0	–
C	2	7

Disponível: 5
(c)

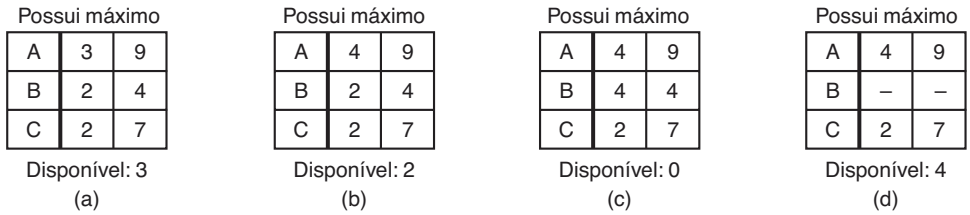
	Possui	máximo
A	3	9
B	0	–
C	7	7

Disponível: 0
(d)

	Possui	máximo
A	3	9
B	0	–
C	0	–

Disponível: 7
(e)

FIGURA 6.10 Demonstração de que o estado em (b) é inseguro.



6.5.3 O algoritmo do banqueiro para um único recurso

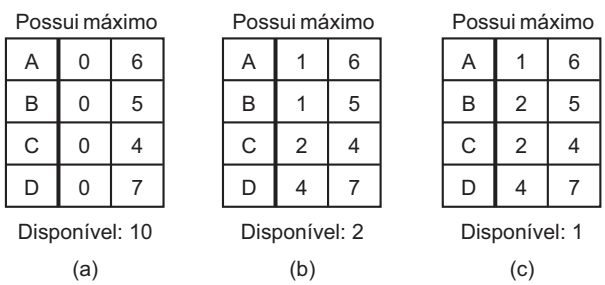
Um algoritmo de escalonamento que pode evitar impasses foi desenvolvido por Dijkstra (1965); ele é conhecido como o **algoritmo do banqueiro** e é uma extensão do algoritmo de detecção de impasses dado na Seção 3.4.1. Ele é modelado da maneira pela qual um banqueiro de uma cidade pequena poderia lidar com um grupo de clientes para os quais ele concedeu linhas de crédito. (Anos atrás, os bancos não emprestavam dinheiro a não ser que tivessem certeza de que poderiam ser ressarcidos.) O que o algoritmo faz é conferir para ver se conceder a solicitação leva a um estado inseguro. Se afirmativo, a solicitação é negada. Se conceder a solicitação conduz a um estado seguro, ela é levada adiante. Na Figura 6.11(a) vemos quatro clientes, *A, B, C* e *D*, cada um tendo recebido um determinado número de unidades de crédito (por exemplo, 1 unidade é 1K dólares). O banqueiro sabe que nem todos os clientes precisarão de seu crédito máximo imediatamente, então ele reservou apenas 10 unidades em vez de 22 para servi-los. (Nessa analogia, os clientes são processos, as unidades são, digamos, unidades de fita, e o banqueiro é o sistema operacional.)

Os clientes cuidam de seus respectivos negócios, fazendo solicitações de empréstimos de tempos em tempos (isto é, pedindo por recursos). Em um determinado momento, a situação é como a mostrada na Figura 6.11(b). Esse estado é seguro porque restando duas unidades, o banqueiro pode postergar quaisquer solicitações exceto as de *C*, deixando então *C* terminar e liberar todos os seus quatro recursos. Com quatro unidades nas mãos, o banqueiro pode deixar *D* ou *B* terem as unidades necessárias, e assim por diante.

Considere o que aconteceria se uma solicitação de *B* por mais uma unidade fosse concedida na Figura 6.11(b). Teríamos a situação da Figura 6.11(c), que é insegura. Se todos os clientes subitamente pedissem por seus empréstimos máximos, o banqueiro não poderia satisfazer nenhum deles, e teríamos um impasse. Um estado inseguro não *precisa* levar a um impasse, tendo em vista que um cliente talvez não precise de toda a linha de crédito disponível, mas o banqueiro não pode contar com esse comportamento.

O algoritmo do banqueiro considera cada solicitação à medida que ela ocorre, vendo se concedê-la leva a um estado seguro. Se afirmativo, a solicitação é concedida; de outra maneira, ela é adiada. Para ver se um estado é seguro, o banqueiro confere para ver se ele tem recursos suficientes para satisfazer algum cliente. Se afirmativo, presume-se que os empréstimos a esse cliente serão ressarcidos, e o cliente agora mais próximo do limite é conferido, e assim por diante. Se todos os empréstimos puderem ser ressarcidos por fim, o estado é seguro e a solicitação inicial pode ser concedida.

FIGURA 6.11 Três estados de alocação de recursos: (a) Seguro. (b) Seguro. (c) Inseguro.

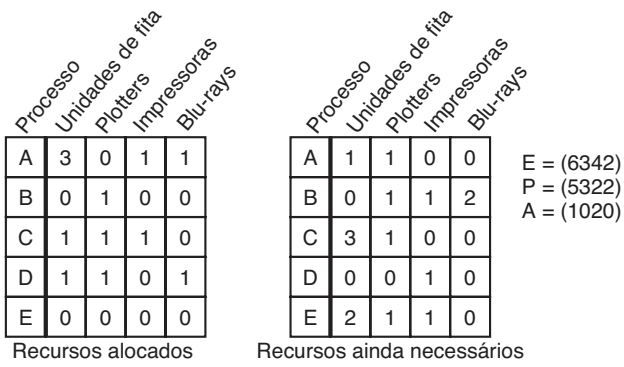


6.5.4 O algoritmo do banqueiro com múltiplos recursos

O algoritmo do banqueiro pode ser generalizado para lidar com múltiplos recursos. A Figura 6.12 mostra como isso funciona.

Na Figura 6.12 vemos duas matrizes. A primeira, à esquerda, mostra quanto de cada recurso está atualmente alocado para cada um dos cinco processos. A matriz à direita mostra de quantos recursos cada processo ainda precisa a fim de terminar. Essas matrizes são simplesmente *C* e *R* da Figura 6.6. Como no caso do recurso único, os processos precisam declarar

FIGURA 6.12 O algoritmo do banqueiro com múltiplos recursos.



suas necessidades de recursos totais antes de executar, de maneira que o sistema possa calcular a matriz à direita a cada instante.

Os três vetores à direita da figura mostram os recursos existentes, *E*, os recursos possuídos, *P*, e os recursos disponíveis, *A*, respectivamente. A partir de *E* vemos que o sistema tem seis unidades de fita, três plotters, quatro impressoras e duas unidades de Blu-ray. Dessas, cinco unidades de fita, três plotters, duas impressoras e duas unidades de Blu-ray estão atualmente alocadas. Esse fato pode ser visto somando-se as entradas nas quatro colunas de recursos na matriz à esquerda. O vetor de recursos disponíveis é apenas a diferença entre o que o sistema tem e o que está atualmente sendo usado.

O algoritmo para verificar se um estado é seguro pode ser descrito agora.

1. Procure por uma linha, *R*, cujas necessidades de recursos não atendidas sejam todas menores ou iguais a *A*. Se essa linha não existir, o sistema irá em algum momento chegar a um impasse, dado que nenhum processo pode executar até o fim (presumindo que os processos mantenham todos os recursos até sua saída).
2. Presuma que o processo da linha escolhida solicita todos os recursos que ele precisa (o que é garantido que seja possível) e termina. Marque esse processo como concluído e adicione todos os seus recursos ao vetor *A*.
3. Repita os passos 1 e 2 até que todos os processos estejam marcados como terminados (caso em que o estado inicial era seguro) ou nenhum processo cujas necessidades de recursos possam ser atendidas seja deixado (caso em que o sistema não era seguro).

Se vários processos são elegíveis para serem escolhidos no passo 1, não importa qual seja escolhido: o pool de recursos disponíveis ou fica maior, ou na pior das hipóteses, fica o mesmo.

Agora vamos voltar para o exemplo da Figura 6.12. O estado atual é seguro. Suponha que o processo *B* faça agora uma solicitação para a impressora. Essa solicitação pode ser concedida porque o estado resultante ainda é seguro (o processo *D* pode terminar, e então os processos *A* ou *E*, seguidos pelo resto).

Agora imagine que após dar a *B* uma das duas impressoras restantes, *E* quer a última impressora. Conceder essa solicitação reduziria o vetor de recursos disponíveis para (1 0 0 0), o que leva a um impasse, portanto a solicitação de *E* deve ser negada por um tempo.

O algoritmo do banqueiro foi publicado pela primeira vez por Dijkstra em 1965. Desde então, quase todo livro sobre sistemas operacionais o descreveu detalhadamente. Inúmeros estudos foram escritos a respeito de vários de seus aspectos. Infelizmente, poucos autores tiveram a audácia de apontar que embora na teoria o algoritmo seja maravilhoso, na prática ele é essencialmente inútil, pois é raro que os processos saibam por antecipação quais serão suas necessidades máximas de recursos. Além disso, o número de processos não é fixo, mas dinamicamente variável, à medida que novos usuários se conectam e desconectam. Ademais, recursos que se acreditava estarem disponíveis podem subitamente desaparecer (unidades de fita podem quebrar). Desse modo, na prática, poucos — se algum — sistemas existentes usam o algoritmo do banqueiro para evitar impasses. Alguns sistemas, no entanto, usam heurísticas similares àquelas do algoritmo do banqueiro para evitar um impasse. Por exemplo, redes podem regular o tráfego quando a utilização do buffer atinge um nível mais alto do que, digamos, 70% — estimando que os 30% restantes serão suficientes para os usuários atuais completarem o seu serviço e retornarem seus recursos.

6.6 Prevenção de impasses

Tendo visto que evitar impasses é algo essencialmente impossível, pois isso exige informações a respeito de solicitações futuras, que não são conhecidas, como os sistemas reais os evitam? A resposta é voltar para as quatro condições colocadas por Coffman et al. (1971) para ver se elas podem fornecer uma pista. Se pudermos assegurar que pelo menos uma dessas condições jamais seja satisfeita, então os impasses serão estruturalmente impossíveis (HAVENDER, 1968).

6.6.1 Atacando a condição de exclusão mútua

Primeiro vamos atacar a condição da exclusão mútua. Se nunca acontecer de um recurso ser alocado exclusivamente para um único processo, jamais teremos impasses. Para dados, o método mais simples é tornar os dados somente para leitura, de maneira que os processos podem usá-los simultaneamente. No entanto, está igualmente claro que permitir que dois processos escrevam na impressora ao mesmo tempo levará ao caos. Utilizar a técnica de spooling na impressora, vários processos podem gerar saídas ao mesmo tempo. Nesse modelo, o único processo que realmente solicita a impressora física é o daemon de impressão. Dado que o daemon jamais solicita quaisquer outros recursos, podemos eliminar o impasse para a impressora.

Se o daemon estiver programado para começar a imprimir mesmo antes de toda a produção ter passado pelo spool, a impressora pode ficar ociosa se um processo de saída decidir esperar várias horas após o primeiro surto de saída. Por essa razão, daemons são normalmente programados para imprimir somente após o arquivo de saída completo estar disponível. No entanto, essa decisão em si poderia levar a um impasse. O que aconteceria se dois processos ainda não tivessem completado suas saídas, embora tivessem preenchido metade do espaço disponível de spool com suas saídas? Nesse caso, teríamos dois processos que haveriam terminado parte de sua saída, mas não toda, e não poderiam continuar. Nenhum processo será concluído, então teríamos um impasse no disco.

Mesmo assim, há um princípio de uma ideia aqui que é muitas vezes aplicável. Evitar alocar um recurso a não ser que seja absolutamente necessário, e tentar certificar-se de que o menor número possível de processos possa, realmente, requisitar o recurso.

6.6.2 Atacando a condição de posse e espera

A segunda das condições estabelecida por Coffman et al. parece ligeiramente mais promissora. Se pudermos evitar que processos que já possuem recursos esperem por mais recursos, poderemos eliminar os impasses. Uma maneira de atingir essa meta é exigir que todos os processos solicitem todos os seus recursos antes de iniciar a execução. Se tudo estiver disponível, o processo terá alocado para si o que ele precisar e pode então executar até o fim. Se um ou mais recursos estiverem ocupados, nada será alocado e o processo simplesmente esperará.

Um problema imediato com essa abordagem é que muitos processos não sabem de quantos recursos eles precisarão até começarem a executar. Na realidade, se soubessem, o algoritmo do banqueiro poderia ser usado. Outro problema é que os recursos não serão usados de maneira otimizada com essa abordagem. Tome como exemplo um processo que lê dados de uma fita de entrada, os analisa por uma hora e então escreve uma fita de saída, assim como imprime os resultados em uma plotter. Se todos os resultados precisam ser solicitados antecipadamente, o processo emperrará a unidade de fita de saída e a plotter por uma hora.

Mesmo assim, alguns sistemas em lote de computadores de grande porte exigem que o usuário liste todos os recursos na primeira linha de cada tarefa. O sistema então prealoca todos os recursos imediatamente e não os libera até que eles não sejam mais necessários pela tarefa (ou no caso mais simples, até a tarefa ser concluída). Embora esse método coloque um fardo sobre o programador e desperdice recursos, ele evita impasses.

Uma maneira ligeiramente diferente de romper com a condição de posse e espera é exigir que um processo que solicita um recurso primeiro libere temporariamente todos os recursos atualmente em suas mãos. Então ele tenta, de uma só vez, conseguir todos os recursos de que precisa.

6.6.3 Atacando a condição de não preempção

Atacar a terceira condição (não preempção) também é uma possibilidade. Se a impressora foi alocada a um processo e ele está no meio da impressão de sua saída, tomar à força a impressora porque uma plotter de que esse processo também necessita não está disponível é algo complicado na melhor das hipóteses e impossível no pior cenário. No entanto, alguns recursos podem ser virtualizados para essa situação. Promover o spooling da saída da impressora para o disco e permitir que apenas o daemon da impressora acesse a impressora real elimina impasses envolvendo a impressora, embora crie o potencial para um impasse sobre o espaço em disco. Com discos grandes, no entanto, ficar sem espaço em disco é algo improvável de acontecer.

Entretanto, nem todos os recursos podem ser virtualizados dessa forma. Por exemplo, registros em bancos de dados ou tabelas dentro do sistema operacional devem ser travados para serem usados e aí encontra-se o potencial para um impasse.

6.6.4 Atacando a condição da espera circular

Resta-nos apenas uma condição. A espera circular pode ser eliminada de várias maneiras. Uma delas é

simplesmente ter uma regra dizendo que um processo tem o direito a apenas um único recurso de cada vez. Se ele precisar de um segundo recurso, precisa liberar o primeiro. Para um processo que precisa copiar um arquivo enorme de uma fita para uma impressora, essa restrição é inaceitável.

Outra maneira de se evitar uma espera circular é fornecer uma numeração global de todos os recursos, como mostrado na Figura 6.13(a). Agora a regra é esta: processos podem solicitar recursos sempre que eles quiserem, mas todas as solicitações precisam ser feitas em ordem numérica. Um processo pode solicitar primeiro uma impressora e então uma unidade de fita, mas ele não pode solicitar primeiro uma plotter e então uma impressora.

Com essa regra, o grafo de alocação de recursos jamais pode ter ciclos. Vamos examinar por que isso é verdade para o caso de dois processos, na Figura 6.13(b). Só pode haver um impasse se A solicitar o recurso j, e B solicitar o recurso i. Presumindo que i e j são recursos distintos, eles terão números diferentes. Se $i > j$, então A não tem permissão para solicitar j porque este tem ordem menor do que a do recurso já obtido por A. Se $i < j$, então B não tem permissão para solicitar i porque este tem ordem menor do que a do recurso já obtido por B. De qualquer maneira, o impasse é impossível.

Com mais do que dois processos a mesma lógica se mantém. A cada instante, um dos recursos alocados será o mais alto. O processo possuindo aquele recurso jamais pedirá por um recurso já alocado. Ele finalizará, ou, na pior das hipóteses, requisitará até mesmo recursos de ordens maiores, todos os quais disponíveis. Em consequência, ele finalizará e libertará seus recursos. Nesse ponto, algum outro processo possuirá o recurso de mais alta ordem e também poderá finalizar. Resumindo, existe um cenário no qual todos os processos finalizam, de maneira que não há impasse algum.

Uma variação menor desse algoritmo é abandonar a exigência de que os recursos sejam adquiridos em uma sequência estritamente crescente e meramente insistir que nenhum processo solicite um recurso de ordem mais baixa do que o recurso que ele já possui. Se um processo solicita inicialmente 9 e 10, e então libera os dois, ele está efetivamente começando tudo de novo, assim não há razão para proibi-lo de agora solicitar o recurso 1.

Embora ordenar numericamente os recursos elimine o problema dos impasses, pode ser impossível encontrar uma ordem que satisfaça a todos. Quando os recursos incluem entradas da tabela de processos, espaço em disco para spooling, registros travados de bancos de dados e outros recursos abstratos, o número de recursos potenciais e usos diferentes pode ser tão grande que nenhuma ordem teria chance de funcionar.

Várias abordagens para a prevenção de impasses estão resumidas na Figura 6.14.

FIGURA 6.13 (a) Recursos ordenados numericamente. (b) Um grafo de recursos.

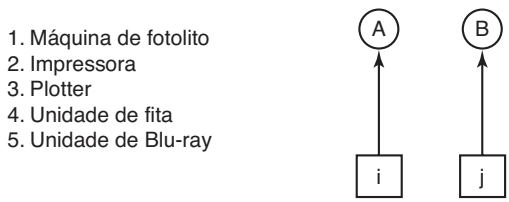


FIGURA 6.14 Resumo das abordagens para prevenir impasses.

Condição	Abordagem contra impasses
Exclusão mútua	Usar spool em tudo
Posse e espera	Requisitar todos os recursos necessários no início
Não preempção	Retomar os recursos alocados
Espera circular	Ordenar numericamente os recursos

6.7 Outras questões

Nesta seção discutiremos algumas questões diversas relacionadas com impasses. Elas incluem o travamento em duas fases, impasses que não envolvem recursos e inanições.

6.7.1 Travamento em duas fases

Embora os meios para evitar e prevenir impasses não sejam muito promissores em geral, para aplicações específicas, são conhecidos muitos algoritmos excelentes para fins especiais. Como um exemplo, em muitos sistemas de bancos de dados, uma operação que ocorre frequentemente é solicitar travas em vários registros e então atualizar todos os registros travados. Quando múltiplos processos estão executando ao mesmo tempo, há um perigo real de impasse.

A abordagem muitas vezes usada é chamada de **travamento em duas fases** (*two-phase locking*). Na primeira fase, o processo tenta travar todos os registros de que precisa, um de cada vez. Se for bem-sucedido, ele começa a segunda fase, desempenhando as atualizações e liberando as travas. Nenhum trabalho de verdade é feito na primeira fase.

Se, durante a primeira fase, algum registro for necessário que já esteja travado, o processo simplesmente libera todas as travas e começa a primeira fase desde o início. De certa maneira, essa abordagem é similar a solicitar todos os recursos necessários antecipadamente, ou pelo menos antes que qualquer ato irreversível seja feito. Em algumas versões do travamento em duas fases não há liberação e reinício se um registro travado for encontrado durante a primeira fase. Nessas versões, pode ocorrer um impasse.

No entanto, essa estratégia em geral não é aplicável. Em sistemas de tempo real e sistemas de controle de processos, por exemplo, não é aceitável apenas terminar um processo no meio do caminho porque um recurso não está disponível e começar tudo de novo. Tampouco é aceitável reiniciar se o processo já leu ou escreveu mensagens para a rede, arquivos atualizados ou qualquer coisa que não possa ser repetida seguramente. O algoritmo funciona apenas naquelas situações em que o programador arranjou as coisas muito cuidadosamente de modo que o programa pode ser parado em qualquer ponto durante a primeira fase e reiniciado. Muitas aplicações não podem ser estruturadas dessa maneira.

6.7.2 Impasses de comunicação

Todo o nosso trabalho até o momento concentrou-se nos impasses de recursos. Um processo quer algo que outro processo tem e deve esperar até que o primeiro abra mão dele. Às vezes os recursos são objetos de hardware ou software, como unidades de Blu-ray ou registros de bancos de dados, mas às vezes eles são mais abstratos. O impasse de recursos é um problema de **sincronização de competição**. Processos independentes completariam seus serviços se a sua execução não sofresse a competição de outros processos. Um processo trava recursos a fim de evitar estados de recursos inconsistentes causados pelo acesso intercalado a recursos. O acesso intervalado a recursos bloqueados, no entanto, proporciona o impasse de recursos. Na Figura 6.2 vimos um impasse de recursos em que eles eram semáforos. Um semáforo é um pouco mais abstrato do que uma unidade de Blu-ray, mas nesse exemplo cada processo adquiriu de maneira bem-sucedida um recurso (um dos semáforos) e entraram em situação em impasse ao tentar adquirir outro (o outro semáforo). Essa situação é um clássico impasse de recursos.

No entanto, como mencionamos no início do capítulo, embora impasses de recursos sejam o tipo mais comum, eles não são o único. Outro tipo de impasse pode ocorrer em sistemas de comunicação (por exemplo, redes), em que dois ou mais processos comunicam-se enviando mensagens. Um arranjo comum é o processo *A* enviar uma mensagem de solicitação ao processo *B*, e então bloquear até *B* enviar de volta uma mensagem de resposta. Suponha que a mensagem de solicitação se perca. *A* está bloqueado esperando pela resposta. *B* está bloqueado esperando por uma solicitação pedindo a ele para fazer algo. Temos um impasse.

Este, no entanto, não é o impasse de recursos clássico. *A* não possui nenhum recurso que *B* quer, e vice-versa. Na realidade, não há recurso algum à vista. Mas trata-se de um impasse de acordo com nossa definição formal, pois temos um conjunto de (dois) processos, cada um bloqueado esperando por um evento que somente o outro pode causar. Essa situação é chamada de **impasse de comunicação** para contrastá-la com o impasse de recursos mais comum. O impasse de comunicação é uma anomalia de *sincronização de cooperação*. Os processos nesse tipo de impasse não poderiam completar o serviço se executados independentemente.

Impasses de comunicação não podem ser prevenidos ordenando os recursos (dado que não há recursos) ou evitados mediante um escalonamento cuidadoso (já que não há momentos em que uma solicitação poderia ser adiada). Felizmente, existe outra técnica que pode ser empregada para acabar com os impasses de comunicação: controles de limite de tempo (timeouts). Na maioria dos sistemas de comunicação, sempre que uma mensagem é enviada para a qual uma resposta é esperada, um temporizador é inicializado. Se o limite de tempo for ultrapassado antes de a resposta chegar, o emissor da mensagem presume que ela foi perdida e a envia de novo (e quantas vezes for necessário). Dessa maneira, o impasse é rompido. Colocando a questão de outra maneira, o limite de tempo serve como uma heurística para detectar impasses e possibilitar a recuperação, e é aplicável também a impasses de recurso. Da mesma maneira, usuários com drivers de dispositivos temperamentais ou defeituosos que geram impasses ou "congelamentos" contam com elas para resolver essas questões.

É claro, se a mensagem original não foi perdida, mas a resposta apenas foi atrasada, o destinatário pode receber a mensagem duas vezes ou mais, possivelmente com consequências indesejáveis. Pense em um sistema bancário eletrônico no qual a mensagem contém instruções para realizar um pagamento. É claro, isso não deve ser repetido (e executado) múltiplas vezes só porque a rede é lenta ou o timeout curto demais. Projetar as regras de comunicação, chamadas de **protocolo**, para

deixar tudo certo é um assunto complexo, mas fora do escopo deste livro. Leitores interessados em protocolos de rede poderiam interessar-se por outro livro de um dos autores, *Redes de computadores* (TANENBAUM e WETHERALL, 2010).

Nem todos os impasses que ocorrem em sistemas de comunicação ou redes são impasses de comunicação. Impasses de recursos também acontecem. Considere, por exemplo, a rede da Figura 6.15. Trata-se de uma visão simplificada da internet. Muito simplificada. A internet consiste em dois tipos de computadores: hospedeiros e roteadores. Um **hospedeiro** (**host**) é um computador de usuário, seja o tablet ou o PC na casa de alguém, um PC em uma empresa ou um servidor corporativo. Hospedeiros trabalham para pessoas. Um **roteador** é um computador de comunicações especializado que move pacotes de dados da fonte para o destino. Cada hospedeiro é conectado a um ou mais roteadores, seja por linha DSL, conexão de TV a cabo, LAN, conexão dial-up, rede sem fio, fibra ótica ou algo mais.

Quando um pacote chega a um roteador vindo de um dos seus hospedeiros, ele é colocado em um buffer para transmissão subsequente para outro roteador e então outro até que chegue ao destino. Esses buffers são recursos e há um número finito deles. Na Figura 6.16 cada roteador tem apenas oito buffers (na prática eles têm milhões, mas isso não muda a natureza do impasse potencial, apenas sua frequência). Suponha que todos os pacotes no roteador *A* precisam ir para *B*, todos os pacotes em *B* precisam ir para *C*, todos os pacotes em *C* precisam ir para *D* e todos os pacotes em *D* precisam ir para *A*. Nenhum pacote pode se movimentar porque não há um buffer na outra extremidade e temos um clássico impasse de recursos, embora no meio de um sistema de comunicação.

FIGURA 6.15 Um impasse de recursos em uma rede.

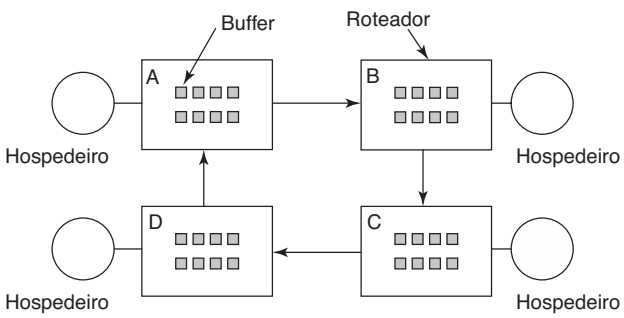


FIGURA 6.16 Processos educados que podem causar um livelock.

```
void process_A(void) {
      acquire_lock(&resource_1);
      while (try_lock(&resource_2) == FAIL) {
            release_lock(&resource_1);
            wait_fixed_time();
            acquire_lock(&resource_1);
      }
      use_both_resources( );
      release_lock(&resource_2);
      release_lock(&resource_1);
}

void process_A(void) {
      acquire_lock(&resource_2);
      while (try_lock(&resource_1) == FAIL) {
            release_lock(&resource_2);
            wait_fixed_time();
            acquire_lock(&resource_2);
      }
      use_both_resources( );
      release_lock(&resource_1);
      release_lock(&resource_2);
}
```

6.7.3 Livelock

Em algumas situações, um processo tenta ser educado abrindo mão dos bloqueios que ele já adquiriu sempre que nota que não pode obter o bloqueio seguinte de que precisa. Então ele espera um milissegundo, digamos, e tenta de novo. Em princípio, isso é bom e deve ajudar a detectar e a evitar impasses. No entanto, se o outro processo faz a mesma coisa exatamente no mesmo momento, eles estarão na situação de duas pessoas tentando passar uma pela outra quando ambas educadamente dão um passo para o lado e, no entanto, nenhum progresso é possível, pois elas seguem dando um passo ao lado na mesma direção ao mesmo tempo.

Considere um primitivo atômico *try_lock* no qual o processo chamador testa um mutex e o pega ou retorna uma falha. Em outras palavras, ele jamais bloqueia. Programadores podem usá-lo junto com *acquire_lock* que também tenta pegar a trava (lock), mas bloqueia se ela não estiver disponível. Agora imagine um par de processos executando em paralelo (talvez em núcleos diferentes) que usam dois recursos, como mostrado na Figura 6.16. Cada um precisa de dois recursos e usa o primitivo *try_lock* para tentar adquirir as travas necessárias. Se a tentativa falha, o processo abre mão da trava que ele possui e tenta novamente. Na Figura 6.16, o processo *A* executa e adquire o recurso 1, enquanto o

processo 2 executa e adquire o recurso 2. Em seguida, eles tentam adquirir a outra trava e falham. Para serem educados, eles abrem mão da trava que possuem atualmente e tentam de novo. Essa rotina se repete até que um usuário entediado (ou alguma outra entidade) acaba com o sofrimento de um desses processos. É claro, nenhum processo é bloqueado e poderíamos até dizer que as coisas estão acontecendo, então isso não é um impasse. Ainda assim, nenhum progresso é possível, então temos algo equivalente: um **livelock**.[2]

Livelocks e impasses podem ocorrer de maneiras surpreendentes. Em alguns sistemas, o número total de processos permitidos é determinado pelo número de entradas na tabela de processos. Desse modo, vagas na tabela de processo são recursos finitos. Se um fork falha porque a tabela está cheia, uma abordagem razoável para o programa realizando o fork é esperar um tempo qualquer e tentar novamente.

Agora suponha que um sistema UNIX tem cem entradas para processos. Dez programas estão executando, cada um deles precisa criar 12 filhos. Após cada um ter criado 9 processos, os 10 processos originais e os 90 novos exauriram a tabela. Cada um dos 10 processos originais encontra-se agora em um laço sem fim realizando *fork* e falhando — um livelock. A probabilidade de isso acontecer é minúscula, mas *poderia* acontecer. Deveríamos abandonar os processos e a chamada fork para eliminar o problema?

O número máximo de arquivos abertos é similarmente restrito pelo tamanho da tabela de i-nós, assim um problema similar ocorre quando ela enche. O espaço de swap no disco é outro recurso limitado. Na realidade, quase todas as tabelas no sistema operacional representam um recurso finito. Deveríamos abolir todas elas porque poderia acontecer de uma coleção de n processos reivindicar $1/n$ do total, e então cada um tentar reivindicar outro? Provavelmente não é uma boa ideia.

A maioria dos sistemas operacionais, incluindo UNIX e Windows, na essência apenas ignora o problema presumindo que a maioria dos usuários preferiria um livelock ocasional (ou mesmo um impasse) a uma regra restringindo todos os usuários a um processo, um arquivo aberto e um de tudo. Se esses problemas pudessem ser eliminados gratuitamente, não haveria muita discussão. A questão é que o preço é alto, principalmente por causa da aplicação de restrições inconvenientes sobre os processos. Desse modo, estamos diante de uma escolha desagradável entre conveniência e correção, e muita discussão sobre o que é mais importante, e para quem.

6.7.4 Inanição

Um problema relacionado de muito perto com o impasse e o livelock é a **inanição** (***starvation***). Em um sistema dinâmico, solicitações para recursos acontecem o tempo todo. Alguma política é necessária para tomar uma decisão sobre quem recebe qual recurso e quando. Essa política, embora aparentemente razoável, pode levar a alguns processos nunca serem servidos, embora não estejam em situação de impasse.

Como exemplo, considere a alocação da impressora. Imagine que o sistema utilize algum algoritmo para assegurar que a alocação da impressora não leve a um impasse. Agora suponha que vários processos a queiram ao mesmo tempo. Quem deve ficar com ela?

Um algoritmo de alocação possível é dá-la ao processo com o menor arquivo para imprimir (presumindo que essa informação esteja disponível). Essa abordagem maximiza o número de clientes felizes e parece justa. Agora considere o que acontece em um sistema ocupado quando um processo tem um arquivo enorme para imprimir. Toda vez que a impressora estiver livre, o sistema procurará à sua volta e escolherá o processo com o arquivo mais curto. Se houver um fluxo constante de processos com arquivos curtos, o processo com o arquivo enorme jamais terá a impressora alocada para si. Ele simplesmente morrerá de inanição (será postergado indefinidamente, embora não esteja bloqueado).

A inanição pode ser evitada com uma política de alocação de recursos primeiro a chegar, primeiro a ser servido. Com essa abordagem, o processo que estiver esperando há mais tempo é servido em seguida. No devido momento, qualquer dado processo será consequentemente o mais antigo e, desse modo, receberá o recurso de que necessita.

Vale a pena mencionar que algumas pessoas não fazem distinção entre a inanição e o impasse, porque em ambos os casos não há um progresso. Outros creem tratar-se de conceitos fundamentalmente diferentes, pois um processo poderia com facilidade ser programado a tentar fazer algo n vezes e, se todas elas falhassem, tentar algo mais. Um processo bloqueado não tem essa escolha.

6.8 Pesquisas sobre impasses

Se há um assunto que foi pesquisado extensamente no princípio dos sistemas operacionais, foi o impasse. A razão é que a detecção de impasses é um problema

2 Lembrando que um "impasse" é um "deadlock". (N. T).

de teoria de grafos interessante sobre o qual um estudante universitário inclinado à matemática poderia se debruçar por quatro anos. Muitos algoritmos foram desenvolvidos, cada um mais exótico e menos prático do que o anterior. A maior parte desses trabalhos caiu no esquecimento, mas mesmo assim, alguns estudos ainda estão sendo publicados sobre impasses.

Trabalhos recentes sobre impasses incluem a pesquisa sobre a imunidade a impasses (JULA et al., 2011). A principal ideia dessa abordagem é que as aplicações detectam impasses quando eles ocorrem e então salvam suas "assinaturas", de maneira a evitar o mesmo impasse em execuções futuras. Marino et al. (2013), por outro lado, usam o controle de concorrência para certificar-se de que os impasses não possam ocorrer em primeiro lugar.

Outra direção de pesquisa é tentar e detectar impasses. Trabalhos recentes sobre a detecção de impasses foram apresentados por Pyla e Varadarajan (2012). O trabalho de Cai e Chan (2012) apresenta um novo esquema de detecção de impasses dinâmico que iterativamente apara dependências entre travas que não têm arestas de entrada ou saída.

O problema do impasse aparece por toda parte. Wu et al. (2013) descrevem um sistema de controle de impasses para sistemas de manufatura automatizados. Ele modela esses sistemas usando redes de Petri para procurar por condições necessárias e suficientes a fim de permitir um controle de impasses permissivo.

Há também muita pesquisa sobre a detecção distribuída de impasses, especialmente em computação de alto desempenho. Por exemplo, há um conjunto de trabalhos significativo sobre a detecção de impasses baseada no escalonamento. Wang e Lu (2013) apresentam um algoritmo de escalonamento para cálculos de fluxo de trabalho na presença de restrições de armazenamento. Hilbrich et al. (2013) descrevem a detecção de impasses em tempo de execução para MPI. Por fim, há uma quantidade enorme de trabalhos teóricos sobre a detecção de impasses distribuídos. No entanto, não a consideraremos aqui, pois (1) está fora do escopo deste livro e (2) nada disso chega a ser remotamente prático em sistemas reais. A sua principal função parece ser manter fora das ruas teóricos de grafos que de outra maneira estariam desempregados.

6.9 Resumo

O impasse é um problema potencial em qualquer sistema operacional. Ele ocorre quando todos os membros de um conjunto de processos são bloqueados esperando por um evento que apenas outros membros do mesmo conjunto podem causar. Essa situação faz que todos os processos esperem para sempre. Comumente, o evento pelo qual os processos estão esperando é a liberação de algum recurso nas mãos de outro membro do conjunto. Outra situação na qual o impasse é possível ocorre quando todos os processos de um conjunto de processos de comunicação estão esperando por uma mensagem e o canal de comunicação está vazio e não há timeouts pendentes.

O impasse de recursos pode ser evitado controlando quais estados são seguros e quais são inseguros. Um estado seguro é aquele no qual existe uma sequência de eventos garantindo que todos os processos possam ser concluídos. Um estado inseguro não tem essa garantia. O algoritmo do banqueiro evita o impasse ao não conceder uma solicitação se ela colocar o sistema em um estado inseguro.

O impasse de recursos pode ser evitado estruturalmente projetando o sistema de tal maneira que ele jamais possa ocorrer. Por exemplo, ao permitir que um processo possua somente um recurso a qualquer instante, a condição da espera circular necessária para um impasse é derrubada. O impasse de recursos também pode ser evitado numerando todos os recursos e obrigando os processos a requisitá-los somente na ordem crescente.

O impasse de recursos não é o único tipo existente. O impasse de comunicação também é um problema em potencial em alguns sistemas, embora ele possa muitas vezes ser resolvido via estabelecimento de timeouts apropriados.

O livelock é similar ao impasse no sentido de que ele pode parar todo o progresso, mas ele é tecnicamente diferente, pois envolve processos que não estão realmente bloqueados. A inanição pode ser evitada mediante uma política de alocação "primeiro a chegar, primeiro a ser servido".

PROBLEMAS

1. Dê um exemplo de um impasse tirado da política.
2. Estudantes trabalhando em PCs individuais em um laboratório de computadores enviam seus arquivos para serem impressos por um servidor que envia os arquivos para o seu disco rígido através de spooling. Em quais condições pode ocorrer um impasse se o espaço em disco para o spool de impressão é limitado? Como o impasse pode ser evitado?
3. Na questão anterior, quais recursos podem ser obtidos por preempção e quais não podem ser obtidos dessa maneira?
4. Na Figura 6.1 os recursos são retornados na ordem inversa da sua aquisição. Devolvê-los na outra ordem seria igualmente correto?
5. As quatro condições (exclusão mútua, posse e espera, não preempção e espera circular) são necessárias para que o impasse de um recurso ocorra. Dê um exemplo mostrando que essas condições não são suficientes para que ocorra um impasse de um recurso. Quando tais condições são suficientes para que ocorra esse impasse?
6. As ruas da cidade são vulneráveis a uma condição de bloqueio circular chamada engarrafamento, na qual os cruzamentos são bloqueados pelos carros que então bloqueiam os carros atrás deles que então bloqueiam os carros que estão tentando entrar no cruzamento anterior etc. Todos os cruzamentos em torno de um quarteirão da cidade estão cheios de veículos que bloqueiam o tráfego que está chegando de uma maneira circular. O engarrafamento é um impasse de recursos e um problema de sincronização da competição. O algoritmo de prevenção da cidade de Nova York, chamado "não bloqueie o espaço", proíbe os carros de entrar em um cruzamento a não ser que o espaço após o cruzamento esteja também disponível. Qual algoritmo de prevenção é esse? Você teria em mente algum outro algoritmo de prevenção para engarrafamentos?
7. Suponha que quatro carros se aproximem de um cruzamento vindos de quatro direções diferentes simultaneamente. Cada esquina da interseção tem um sinal de "pare". Presuma que as normas do trânsito exijam que, quando dois carros se aproximam adjacentes a sinais de "pare" ao mesmo tempo, o carro à esquerda deve ceder para o carro à direita. Desse modo, quando quatro carros avançam até seus sinais de "pare" individuais, cada um espera (indefinidamente) pelo carro da esquerda seguir. Essa anomalia é um impasse de comunicação? É um impasse de recursos?
8. É possível que um impasse de recurso envolva múltiplas unidades de um tipo e uma única unidade de outro? Se afirmativo, dê um exemplo.
9. A Figura 6.3 mostra o conceito de um grafo de recursos. Existem grafos ilegais, isto é, grafos que violam estruturalmente o modelo que usamos para a utilização de recursos? Se afirmativo, dê um exemplo de um.
10. Considere a Figura 6.4. Suponha que no passo (o) *C* solicitou *S* em vez de *R*. Isso levaria a um impasse? Suponha que ele tenha solicitado tanto *S* como *R*.
11. Suponha que há um impasse de recursos em um sistema. Dê um exemplo para mostrar que o conjunto de processos em situação de impasse pode incluir processos que não estão na cadeia circular no grafo de alocação de recursos correspondente.
12. A fim de controlar o tráfego, um roteador de rede, *A*, envia periodicamente uma mensagem para seu vizinho, *B*, dizendo-lhe para aumentar ou reduzir o número de pacotes com que ele pode lidar. Em determinado ponto no tempo, o Roteador *A* é inundado com tráfego e envia a *B* uma mensagem dizendo-lhe para cessar de enviar tráfego. Ele faz isso especificando que o número de bytes que *B* pode enviar (tamanho da janela de *A*) é 0. À medida que os surtos de tráfego diminuem, *A* envia uma nova mensagem, dizendo a *B* para reiniciar a transmissão. Ele faz isso aumentando o tamanho da janela de 0 para um número positivo. Essa mensagem é perdida. Como descrito, nenhum lado jamais transmitirá. Que tipo de impasse é esse?
13. A discussão do algoritmo do avestruz menciona a possibilidade de entradas da tabela de processos ou outras tabelas do sistema encherem. Você poderia sugerir uma maneira de capacitar um administrador de sistemas a recuperar de uma situação dessas?
14. Considere o estado a seguir de um sistema com quatro processos, *P1*, *P2*, *P3* e *P4*, e cinco tipos de recursos, *RS1*, *RS2*, *RS3*, *RS4* e *RS5*.

C =

0	1	1	1	2
0	1	0	1	0
0	0	0	0	1
2	1	0	0	0

R =

1	1	0	2	1
0	1	0	2	1
0	2	0	3	1
0	2	1	1	0

E = (24144)

A = (01021)

Usando o algoritmo de detecção de impasses descrito na Seção 6.4.2, mostre que há um impasse no sistema. Identifique os processos que estão em situação de impasse.

15. Explique como o sistema pode se recuperar do impasse no problema anterior usando
    (a) recuperação mediante preempção.
    (b) recuperação mediante retrocesso.
    (c) recuperação mediante eliminação de processos.

16. Suponha que na Figura 6.6 $C_{ij} + R_{ij} > E_j$ para algum i. Quais implicações isso tem para o sistema?

17. Todas as trajetórias na Figura 6.8 são horizontais ou verticais. Você consegue imaginar alguma circunstância na qual trajetórias diagonais também sejam possíveis?

18. O esquema de trajetória de recursos da Figura 6.8 também poderia ser usado para ilustrar o problema de impasses com três processos e três recursos? Se afirmativo, como isso pode ser feito? Se não for possível, por que não?

19. Na teoria, grafos da trajetória de recursos poderiam ser usados para evitar impasses. Com um escalonamento inteligente, o sistema operacional poderia evitar regiões inseguras. Existe uma maneira prática de realmente se fazer isso?

20. É possível que um sistema esteja em um estado que não seja de impasse nem tampouco seguro? Se afirmativo, dê um exemplo. Se não, prove que todos os estados são seguros ou se encontram em situação de impasse.

21. Examine cuidadosamente a Figura 6.11(b). Se D pedir por mais uma unidade, isso leva a um estado seguro ou inseguro? E se a solicitação vier de C em vez de D?

22. Um sistema tem dois processos e três recursos idênticos. Cada processo precisa de um máximo de dois recursos. Um impasse é possível? Explique a sua resposta.

23. Considere o problema anterior novamente, mas agora com p processos cada um necessitando de um máximo de m recursos e um total de r recursos disponíveis. Qual condição deve se manter para tornar o sistema livre de impasses?

24. Suponha que o processo A na Figura 6.12 exige a última unidade de fita. Essa ação leva a um impasse?

25. O algoritmo do banqueiro está sendo executado em um sistema com m classes de recursos e n processos. No limite de m e n grandes, o número de operações que precisam ser realizadas para verificar a segurança de um estado é proporcional a $m^a n^b$. Quais são os valores de a e b?

26. Um sistema tem quatro processos e cinco recursos alocáveis. A alocação atual e as necessidades máximas são as seguintes:

	Alocado	*Máximo*	*Disponível*
Processo A	10211	11213	00x11
Processo B	20110	22210	
Processo C	11010	21310	
Processo D	11110	11221	

Qual é o menor valor de x para o qual esse é um estado seguro?

27. Uma maneira de se eliminar a espera circular é ter uma regra dizendo que um processo tem direito a somente um único recurso a qualquer dado momento. Dê um exemplo para mostrar que essa restrição é inaceitável em muitos casos.

28. Dois processos, A e B, têm cada um três registros, 1, 2 e 3, em um banco de dados. Se A pede por eles na ordem 1, 2, 3 e B pede por eles na mesma ordem, o impasse não é possível. No entanto, se B pedir por eles na ordem 3, 2, 1, então o impasse é possível. Com três recursos, há 3! ou seis combinações possíveis nas quais cada processo pode solicitá-los. Qual fração de todas as combinações é garantida que seja livre de impasses?

29. Um sistema distribuído usando caixas de correio tem duas primitivas de IPC, send e receive. A segunda primitiva especifica um processo do qual deve receber e bloqueia se nenhuma mensagem desse processo estiver disponível, embora possa haver mensagens esperando de outros processos. Não há recursos compartilhados, mas os processos precisam comunicar-se frequentemente a respeito de outras questões. É possível ocorrer um impasse? Discuta.

30. Em um sistema de transferência de fundos eletrônico, há centenas de processos idênticos que funcionam como a seguir. Cada processo lê uma linha de entrada especificando uma quantidade de dinheiro, a conta a ser creditada e a conta a ser debitada. Então ele bloqueia ambas as contas e transfere o dinheiro, liberando as travas quando concluída a transferência. Com muitos processos executando em paralelo, há um perigo muito real de que um processo tendo bloqueado a conta x será incapaz de desbloquear y porque y foi bloqueada por um processo agora esperando por x. Projete um esquema que evite os impasses. Não libere um registro de conta até você ter completado as transações. (Em outras palavras, soluções que bloqueiam uma conta e então a liberam imediatamente se a outra estiver bloqueada não são permitidas.)

31. Uma maneira de evitar os impasses é eliminar a condição de posse e espera. No texto, foi proposto que antes de pedir por um novo recurso, um processo deve primeiro liberar quaisquer recursos que ele já possui (presumindo que isso seja possível). No entanto, fazê-lo introduz o perigo de que ele possa receber o novo recurso, mas perder alguns dos recursos existentes para processos que estão competindo com ele. Proponha uma melhoria para esse esquema.

32. Um estudante de ciência da computação designado para trabalhar com impasses pensa na seguinte maneira brilhante de eliminar os impasses. Quando um processo solicita um recurso, ele especifica um limite de tempo. Se o processo bloqueia porque o recurso não está disponível, um temporizador é inicializado. Se o limite de tempo for excedido, o processo é liberado e pode executar novamente. Se você fosse o professor, qual nota daria a essa proposta e por quê?

33. Unidades de memória principal passam por preempção em sistema de memória virtual e swapping. O processador passa por preempção em ambientes de tempo compartilhado. Você acredita que esses métodos de preempção foram desenvolvidos para lidar com o impasse de recursos ou para outros fins? Quão alta é a sua sobrecarga?

34. Explique a diferença entre impasse, livelock e inanição.

35. Presuma que dois processos estejam emitindo um comando de busca para reposicionar o mecanismo de acesso ao disco e possibilitar um comando de leitura. Cada processo é interrompido antes de executar a sua leitura, e descobre que o outro moveu o braço do disco. Cada um então reemite o comando de busca, mas é de novo interrompido pelo outro. Essa sequência se repete continuamente. Isso é um impasse ou um livelock de recursos? Quais métodos você recomendaria para lidar com a anomalia?

36. Redes de área local utilizam um método de acesso à mídia chamado CSMA/CD, no qual as estações compartilhando um barramento podem conferir o meio e detectar transmissões, assim como colisões. No protocolo Ethernet, as estações solicitando o canal compartilhado não transmitem quadros se perceberem que o meio está ocupado. Quando uma transmissão é concluída, as estações esperando transmitem seus quadros. Dois quadros que forem transmitidos ao mesmo tempo colidirão. Se as estações imediata e repetidamente retransmitem após a detecção da colisão, elas continuarão a colidir indefinidamente.

(a) Estamos falando de um impasse ou um livelock de recursos?

(b) Você poderia sugerir uma solução para essa anomalia?

(c) A inanição poderia ocorrer nesse cenário?

37. Um programa contém um erro na ordem de mecanismos de cooperação e competição, resultando em um processo consumidor bloqueando um mutex (semáforo de exclusão mútua) antes que ele bloqueie um buffer vazio. O processo produtor bloqueia no mutex antes que ele possa colocar um valor no buffer vazio e despertar o consumidor. Desse modo, ambos os processos estão bloqueados para sempre, o produtor esperando que o mutex seja desbloqueado e o consumidor esperando por um sinal do produtor. Estamos falando de um impasse de recursos ou um impasse de comunicação? Sugira métodos para o seu controle.

38. Cinderela e o Príncipe estão se divorciando. Para dividir sua propriedade, eles concordaram com o algoritmo a seguir. Cada manhã, um deles pode enviar uma carta para o advogado do outro exigindo um item da propriedade. Como leva um dia para as cartas serem entregues, eles concordaram que se ambos descobrirem que eles solicitaram o mesmo item no mesmo dia, no dia seguinte eles mandarão uma carta cancelando o pedido. Entre seus bens há o seu cão, Woofer. A casa de cachorro do Woofer, seu canário, Tweeter, e a gaiola dele. Os animais adoram suas casas, então foi acordado que qualquer divisão de propriedade separando um animal da sua casa é inválida, exigindo que toda a divisão começasse do início. Tanto Cinderela quanto Príncipe querem desesperadamente ficar com Woofer. Para que pudessem sair de férias (separados), cada um programou um computador pessoal para lidar com a negociação. Quando voltaram das férias, os computadores ainda estavam negociando. Por quê? Um impasse é possível? Inanição? Discuta sua resposta.

39. Um estudante especializando-se em antropologia e interessado em ciência de computação embarcou em um projeto de pesquisa para ver se os babuínos africanos podem ser ensinados a respeito de impasses. Ele localiza um cânion profundo e amarra uma corda de um lado ao outro, de maneira que os babuínos podem atravessá-lo agarrando-se com as mãos. Vários babuínos podem atravessar ao mesmo tempo, desde que eles todos sigam na mesma direção. Se babuínos seguindo na direção leste e outros seguindo na direção oeste se encontrarem na corda ao mesmo tempo, ocorrerá um impasse (eles ficarão presos no meio do caminho), pois é impossível que um passe sobre o outro. Se um babuíno quiser atravessar o cânion, ele tem de conferir para ver se não há nenhum outro babuíno cruzando o cânion no momento na direção oposta. Escreva um programa usando semáforos que evite o impasse. Não se preocupe com uma série de babuínos movendo-se na direção leste impedindo indefinidamente a passagem de babuínos movendo-se na direção oeste.

40. Repita o problema anterior, mas agora evite a inanição. Quando um babuíno que quiser atravessar para leste chegar à corda e encontrar babuínos cruzando na direção contrária, ele espera até que a corda esteja vazia, mas nenhum outro babuíno deslocando-se para oeste tem permissão de começar a travessia até que pelo menos um babuíno tenha atravessado na outra direção.

41. Programe uma simulação do algoritmo do banqueiro. O programa deve passar por cada um dos clientes do banco fazendo uma solicitação e avaliando se ela é segura ou insegura. Envie um histórico de solicitações e decisões para um arquivo.

42. Escreva um programa para implementar o algoritmo de detecção de impasses com múltiplos recursos de cada tipo. O seu programa deve ler de um arquivo as seguintes entradas: o número de processos, o número de tipos de recursos, o número de recursos de cada tipo em existência (vetor E), a matriz de alocação atual C (primeira fila,

seguida pela segunda fila e assim por diante), a matriz de solicitações *R* (primeira fila, seguida pela segunda fila, e assim por diante). A saída do seu programa deve indicar se há um impasse no sistema. Caso exista, o programa deve imprimir as identidades de todos os processos que estão em situação de impasse.

43. Escreva um programa que detecte se há um impasse no sistema usando um grafo de alocação de recursos. O seu programa deve ler de um arquivo as seguintes entradas: o número de processos e o número de recursos. Para cada processo ele deve ler quatro números: o número de recursos que tem em mãos no momento, as identidades dos recursos que tem em mãos, o número de recursos que ele está solicitando atualmente e as identidades dos recursos que está solicitando. A saída do programa deve indicar se há um impasse no sistema. Caso exista, o programa deve imprimir as identidades de todos os processos em situação de impasse.

44. Em determinados países, quando duas pessoas se encontram, elas inclinam-se para a frente como forma de cumprimento. O protocolo manda que uma delas se incline para a frente primeiro e permaneça inclinada até que a outra a cumprimente da mesma forma. Se elas se cumprimentarem ao mesmo tempo, permanecerão inclinadas para a frente para sempre. Escreva um programa que evite um impasse.

CAPÍTULO 7

VIRTUALIZAÇÃO E A NUVEM

Em algumas situações, uma organização tem um multicomputador, mas na realidade não gostaria de tê-lo. Um exemplo comum ocorre quando uma empresa tem um servidor de e-mail, um de internet, um FTP, alguns de e-commerce, e outros. Todos são executados em computadores diferentes no mesmo rack de equipamentos, todos conectados por uma rede de alta velocidade, em outras palavras, um multicomputador. Uma razão para todos esses servidores serem executados em máquinas separadas pode ser que uma máquina não consiga lidar com a carga, mas outra é a confiabilidade: a administração simplesmente não confia que o sistema operacional vá funcionar 24 horas, 365 ou 366 dias sem falhas. Ao colocar cada serviço em um computador diferente, se um dos servidores falhar, pelo menos os outros não serão afetados. Isso é bom para a segurança também. Mesmo que algum invasor maligno comprometa o servidor da internet, ele não terá acesso imediatamente a e-mails importantes também — uma propriedade referida às vezes como **caixa de areia** (**sandboxing**). Embora o isolamento e a tolerância a falhas sejam conseguidos dessa maneira, essa solução é cara e difícil de gerenciar, por haver tantas máquinas envolvidas.

É importante salientar que essas são apenas duas dentre muitas razões para se manterem máquinas separadas. Por exemplo, as organizações muitas vezes dependem de mais do que um sistema operacional para suas operações diárias: um servidor de internet em Linux, um servidor de e-mail em Windows, um servidor de e-commerce para clientes executando em OS X e alguns outros serviços executando em diversas variações do UNIX. Mais uma vez, essa solução funciona, mas barata ela definitivamente não é.

O que fazer? Uma solução possível (e popular) é usar a tecnologia de máquinas virtuais, o que soa muito inovador e moderno, mas a ideia é antiga, surgida nos anos 1960. Mesmo assim, a maneira como as usamos hoje em dia é definitivamente nova. A ideia principal é que um **Monitor de Máquina Virtual** (**VMM — Virtual Machine Monitor**) cria a ilusão de múltiplas máquinas (virtuais) no mesmo hardware físico. Um VMM também é conhecido como **hipervisor**. Como discutimos na Seção 1.7.5, distinguimos entre os hipervisores tipo 1 que são executados diretamente sobre o hardware (bare metal), e os hipervisores tipo 2 que podem fazer uso de todos os serviços e abstrações maravilhosos oferecidos pelo sistema operacional subjacente. De qualquer maneira, a **virtualização** permite que um único computador seja o hospedeiro de múltiplas máquinas virtuais, cada uma executando potencialmente um sistema operacional completamente diferente.

A vantagem dessa abordagem é que uma falha em uma máquina virtual não derruba nenhuma outra. Em um sistema virtualizado, diferentes servidores podem executar em diferentes máquinas virtuais, mantendo desse modo um modelo de falha parcial que um multicomputador tem, mas a um custo mais baixo e com uma manutenção mais fácil. Além disso, podemos agora executar múltiplos sistemas operacionais diferentes no mesmo hardware, beneficiar-nos do isolamento da máquina virtual diante de ataques e aproveitar outras coisas boas.

É claro, consolidar servidores dessa maneira é como colocar todos os ovos em uma cesta. Se o servidor executando todas as máquinas virtuais falhar, o resultado é ainda mais catastrófico do que a falha de um único