Capítulo 13

Impasses

O controle de concorrência entre tarefas acessando recursos compartilhados implica em suspender algumas tarefas enquanto outras acessam os recursos, de forma a garantir a consistência dos mesmos. Para isso, a cada recurso é associado um semáforo ou outro mecanismo equivalente. Assim, as tarefas solicitam e aguardam a liberação de cada semáforo para poder acessar o recurso correspondente.

Em alguns casos, o uso de semáforos ou *mutexes* pode levar a situações de **impasse** (ou *deadlock*), nas quais todas as tarefas envolvidas ficam bloqueadas aguardando a liberação de semáforos, e nada mais acontece. Este capítulo visa compreender os impasses e como tratá-los.

13.1 Exemplo de impasse

Para ilustrar uma situação de impasse, será utilizado o exemplo de acesso a uma conta bancária apresentado na Seção 10.1. O código a seguir implementa uma operação de transferência de fundos entre duas contas bancárias. A cada conta está associado um *mutex*, usado para prover acesso exclusivo aos dados da conta e assim evitar condições de disputa:

```
1  typedef struct conta_t
2  {
3     int saldo ;                    // saldo atual da conta
4     mutex m ;                      // mutex associado à conta
5     ...                            // outras informações da conta
6  } conta_t ;
7
8  void transferir (conta_t* contaDeb, conta_t* contaCred, int valor)
9  {
10    lock (contaDeb->m) ;           // obtém acesso a contaDeb
11    lock (contaCred->m) ;          // obtém acesso a contCred
12
13    if (contaDeb->saldo >= valor)
14    {
15       contaDeb->saldo -= valor ;  // debita valor de contaDeb
16       contaCred->saldo += valor ; // credita valor em contaCred
17    }
18    unlock (contaDeb->m) ;         // libera acesso a contaDeb
19    unlock (contaCred->m) ;        // libera acesso a contaCred
20 }
```

Caso dois clientes do banco (representados por duas tarefas t_1 e t_2) resolvam fazer simultaneamente operações de transferência entre suas contas (t_1 transfere um valor v_1 de c_1 para c_2 e t_2 transfere um valor v_2 de c_2 para c_1), poderá ocorrer uma situação de impasse, como mostra o diagrama de tempo da Figura 13.1.

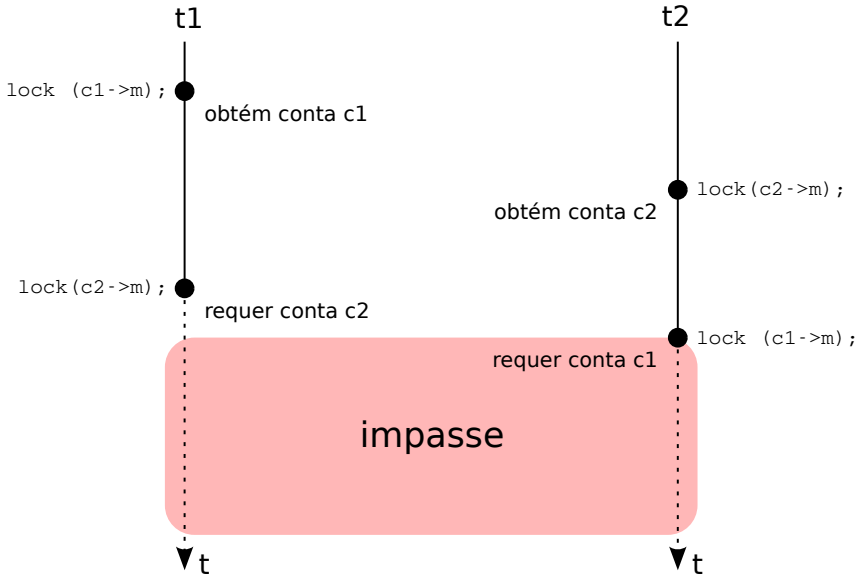


Figura 13.1: Impasse entre duas transferências.

Nessa situação, a tarefa t_1 detém o *mutex* de c_1 e solicita o *mutex* de c_2, enquanto t_2 detém o *mutex* de c_2 e solicita o *mutex* de c_1. Como nenhuma das duas tarefas poderá prosseguir sem obter o *mutex* desejado, nem poderá liberar o *mutex* de sua conta antes de obter o outro *mutex* e realizar a transferência, se estabelece um impasse.

Impasses são situações muito frequentes em programas concorrentes, mas também podem ocorrer em sistemas distribuídos e mesmo em situações fora da informática. A Figura 13.2 mostra como exemplo uma situação de impasse ocorrida em um cruzamento de São Paulo SP, no início de 2017. Antes de conhecer as técnicas

de tratamento de impasses, é importante compreender suas principais causas e saber caracterizá-los adequadamente, o que será estudado nas próximas seções.

Figura 13.2: Uma situação de impasse no trânsito.

13.2 Condições para impasses

Em um impasse, duas ou mais tarefas se encontram bloqueadas, aguardando eventos que dependem somente delas, como a liberação de semáforos. Em outras palavras, não existe influência de entidades externas em uma situação de impasse. Além disso, como as tarefas envolvidas detêm alguns recursos compartilhados (representados por semáforos), outras tarefas que vierem a requisitar esses recursos também ficarão bloqueadas, aumentando gradativamente o impasse, o que pode levar o sistema inteiro a parar de funcionar.

Formalmente, um conjunto de N tarefas se encontra em um impasse se cada uma das tarefas aguarda um evento que somente outra tarefa do conjunto poderá produzir. Quatro condições fundamentais são necessárias para que os impasses possam ocorrer [Coffman et al., 1971; Ben-Ari, 1990]:

Exclusão mútua: o acesso aos recursos deve ser feito de forma mutuamente exclusiva, controlada por semáforos ou mecanismos equivalentes. No exemplo da conta corrente, apenas uma tarefa por vez pode acessar cada conta.

Posse e espera: uma tarefa pode solicitar o acesso a outros recursos sem ter de liberar os recursos que já detém. No exemplo da conta corrente, cada tarefa detém o semáforo de uma conta e solicita o semáforo da outra conta para poder prosseguir.

Não-preempção: uma tarefa somente libera os recursos que detém quando assim o decidir, e não os perde de forma imprevista (ou seja, o sistema operacional não

retira à força os recursos alocados às tarefas). No exemplo da conta corrente, cada tarefa detém os *mutexes* obtidos até liberá-los explicitamente.

Espera circular: existe um ciclo de esperas pela liberação de recursos entre as tarefas envolvidas: a tarefa t_1 aguarda um recurso retido pela tarefa t_2 (formalmente, $t_1 \rightarrow t_2$), que aguarda um recurso retido pela tarefa t_3, e assim por diante, sendo que a tarefa t_n aguarda um recurso retido por t_1. Essa dependência circular pode ser expressa formalmente da seguinte forma: $t_1 \rightarrow t_2 \rightarrow t_3 \rightarrow \cdots \rightarrow t_n \rightarrow t_1$. No exemplo da conta corrente, pode-se observar claramente que $t_1 \rightarrow t_2 \rightarrow t_1$.

Deve-se observar que essas quatro condições são **necessárias** para a formação de impasses; se uma delas não for verificada, não existem impasses no sistema. Por outro lado, não são condições **suficientes** para a existência de impasses, ou seja, a verificação dessas quatro condições não garante a presença de um impasse no sistema. Essas condições somente são suficientes se existir apenas uma instância de cada tipo de recurso, como será discutido na próxima seção.

13.3 Grafos de alocação de recursos

É possível representar graficamente a alocação de recursos entre as tarefas de um sistema concorrente. A representação gráfica provê uma visão mais clara da distribuição dos recursos e permite detectar visualmente a presença de esperas circulares que podem caracterizar impasses. Em um *grafo de alocação de recursos* [Holt, 1972], as tarefas são representadas por círculos (○) e os recursos por retângulos (□). A posse de um recurso por uma tarefa é representada como □ → ○ (lido como "o recurso está alocado à tarefa"), enquanto a requisição de um recurso por uma tarefa é indicada por ○ ⇢ □ (lido como "a tarefa requer o recurso").

A Figura 13.3 apresenta o grafo de alocação de recursos da situação de impasse ocorrida na transferência de valores entre contas bancárias da Figura 13.1. Nessa figura percebe-se claramente a dependência cíclica entre tarefas e recursos no ciclo $t_1 \dashrightarrow c_2 \rightarrow t_2 \dashrightarrow c_1 \rightarrow t_1$, que neste caso evidencia um impasse. Como há um só recurso de cada tipo (apenas uma conta c_1 e uma conta c_2), as quatro condições necessárias se mostram também suficientes para caracterizar um impasse.

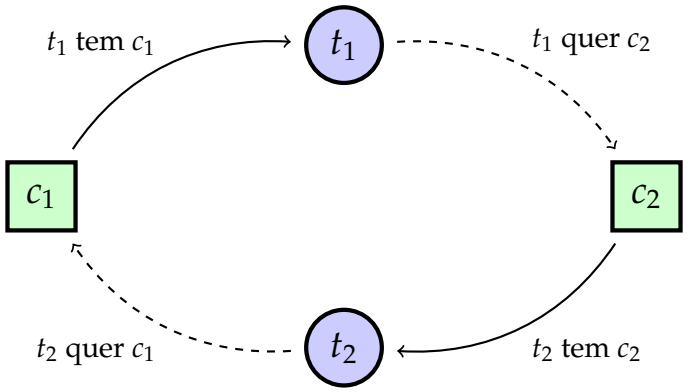


Figura 13.3: Grafo de alocação de recursos com impasse.

Alguns recursos lógicos ou físicos de um sistema computacional podem ter múltiplas instâncias: por exemplo, um sistema pode ter duas impressoras idênticas

instaladas, o que constituiria um recurso (impressora) com duas instâncias equivalentes, que podem ser alocadas de forma independente. No grafo de alocação de recursos, a existência de múltiplas instâncias de um recurso é representada através de "fichas" dentro dos retângulos. Por exemplo, as duas instâncias de impressora seriam indicadas no grafo como [••] . A Figura 13.4 apresenta um grafo de alocação de recursos considerando alguns recursos com múltiplas instâncias.

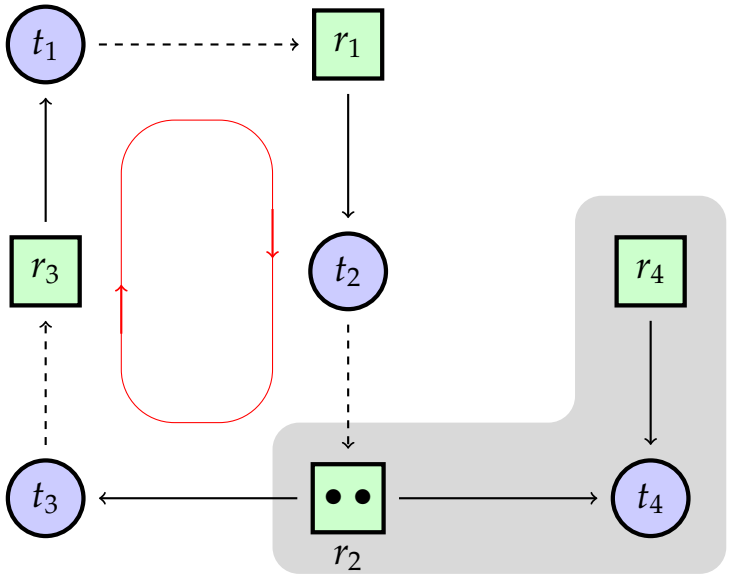


Figura 13.4: Grafo de alocação com múltiplas instâncias de recursos.

É importante observar que a ocorrência de ciclos em um grafo de alocação, envolvendo recursos com múltiplas instâncias, **pode indicar** a presença de um impasse, mas **não garante** sua existência. Por exemplo, o ciclo $t_1 \dashrightarrow r_1 \rightarrow t_2 \dashrightarrow r_2 \rightarrow t_3 \dashrightarrow r_3 \rightarrow t_1$ presente no diagrama da Figura 13.4 não representa um impasse, porque a qualquer momento a tarefa t_4 (que não está esperando recursos) pode liberar uma instância do recurso r_2, solicitado por t_2, permitindo atender a demanda de t_2 e desfazendo assim o ciclo. Um algoritmo de detecção de impasses envolvendo recursos com múltiplas instâncias é apresentado em [Tanenbaum, 2003].

13.4 Técnicas de tratamento de impasses

Como os impasses paralisam tarefas que detêm recursos, sua ocorrência pode gerar consequências graves, como a paralisação gradativa de todas as tarefas que dependam dos recursos envolvidos, o que pode levar à paralisação de todo o sistema. Devido a esse risco, diversas técnicas de tratamento de impasses foram propostas. Essas técnicas podem definir regras estruturais que previnam impasses, podem atuar de forma proativa, se antecipando aos impasses e impedindo sua ocorrência, ou podem agir de forma reativa, detectando os impasses que se formam no sistema e tomando medidas para resolvê-los.

Embora o risco de impasses seja uma questão importante, os sistemas operacionais de mercado (Windows, Linux, MacOS, etc.) adotam a solução mais simples: **ignorar o risco**, na maioria das situações. Devido ao custo computacional necessário ao tratamento de impasses e à sua forte dependência da lógica das aplicações envolvidas, os

projetistas de sistemas operacionais normalmente preferem deixar a gestão de impasses por conta dos desenvolvedores de aplicações.

As principais técnicas usadas para tratar impasses em um sistema concorrente são: **prevenir** impasses através, de regras rígidas para a programação dos sistemas, **impedir** impasses, por meio do acompanhamento contínuo da alocação dos recursos às tarefas, e **detectar e resolver** impasses. Essas técnicas serão detalhadas nas próximas seções.

13.4.1 Prevenção de impasses

As técnicas de prevenção de impasses buscam garantir que impasses nunca possam ocorrer no sistema. Para alcançar esse objetivo, a estrutura do sistema e a lógica das aplicações devem ser construídas de forma que as quatro condições fundamentais para a ocorrência de impasses, apresentadas na Seção 13.2, nunca sejam integralmente satisfeitas. Se ao menos uma das quatro condições for quebrada por essas regras estruturais, os impasses não poderão ocorrer. A seguir, cada uma das condições necessárias é analisada de acordo com essa premissa:

Exclusão mútua: se não houver exclusão mútua no acesso a recursos, não poderão ocorrer impasses. Mas, como garantir a integridade de recursos compartilhados sem usar mecanismos de exclusão mútua? Uma solução interessante é usada na gerência de impressoras: um processo *servidor de impressão* (*printer spooler*) gerencia a impressora e atende as solicitações dos demais processos. Com isso, os processos que desejam usar a impressora não precisam obter acesso exclusivo a ela. A técnica de *spooling* previne impasses envolvendo as impressoras, mas não é facilmente aplicável a certos tipos de recurso, como arquivos em disco e áreas de memória compartilhada.

Posse e espera: caso as tarefas usem apenas um recurso de cada vez, solicitando-o e liberando-o logo após o uso, impasses não poderão ocorrer. No exemplo da transferência de fundos da Figura 13.1, seria possível separar a operação de transferência em duas operações isoladas: débito em c_1 e crédito em c_2 (ou vice-versa), sem a necessidade de acesso exclusivo simultâneo às duas contas. Com isso, a condição de posse e espera seria quebrada e o impasse evitado.

Outra possibilidade seria somente permitir a execução de tarefas que detenham todos os recursos necessários antes de iniciar. Todavia, essa abordagem poderia levar as tarefas a reter os recursos por muito mais tempo que o necessário para suas operações, degradando o desempenho do sistema.

Uma terceira possibilidade seria associar um prazo (*time-out*) às solicitações de recursos: ao solicitar um recurso, a tarefa define um tempo máximo de espera por ele; caso o prazo expire, a tarefa pode tentar novamente ou desistir, liberando os demais recursos que detém.

Não-preempção: normalmente uma tarefa obtém e libera os recursos de que necessita, de acordo com sua lógica interna. Se for possível "arrancar" um recurso da tarefa, sem que esta o libere explicitamente, impasses envolvendo aquele recurso não poderão ocorrer. Essa técnica é frequentemente usada em recursos cujo estado interno pode ser salvo e restaurado de forma transparente para a tarefa, como páginas de memória e o próprio processador (nas trocas de contexto).

No entanto, é de difícil aplicação sobre recursos como arquivos ou áreas de memória compartilhada, porque a preempção viola a exclusão mútua e pode provocar inconsistências no estado interno do recurso.

Espera circular: um impasse é uma cadeia de dependências entre tarefas e recursos que forma um ciclo. Ao prevenir a formação de tais ciclos, impasses não poderão ocorrer. A estratégia mais simples para prevenir a formação de ciclos é ordenar todos os recursos do sistema de acordo com uma ordem global única, e forçar as tarefas a solicitar os recursos obedecendo a essa ordem.

No exemplo da transferência de fundos da Figura 13.1, o número de conta bancária poderia definir uma ordem global ($c_1 < c_2$, por exemplo). Assim, todas as tarefas deveriam solicitar primeiro o acesso à conta mais antiga e depois à mais recente (ou vice-versa, mas sempre na mesma ordem para todas as tarefas). Com isso, elimina-se a possibilidade de impasses.

Essa solução também pode ser aplicada ao problema do jantar dos filósofos, fazendo com que os filósofos peguem os palitos em ordem crescente: para 5 filósofos, os filósofos $f_{i=0...4}$ pegarão o palito p_i e depois o palito p_{i+1}; por sua vez, f_4 deverá pegar o palito p_0 e depois o palito p_4.

As técnicas de prevenção de impasses devem ser consideradas na construção de aplicações multitarefas complexas, pois permitem prevenir impasses sem muito esforço computacional durante a execução. Frequentemente, uma reorganização de pequenos trechos do código da aplicação é suficiente para prevenir impasses, como ocorre no exemplo da transferência bancária.

13.4.2 Impedimento de impasses

Outra forma de tratar os impasses preventivamente consiste em acompanhar a alocação dos recursos às tarefas e, de acordo com algum algoritmo, negar acessos de recursos que possam levar a impasses. Uma noção essencial nas técnicas de impedimento de impasses é o conceito de **estado seguro**. Cada estado do sistema é definido pela distribuição dos recursos entre as tarefas.

O conjunto de todos os estados possíveis do sistema durante sua execução forma um grafo de estados, no qual as arestas indicam as alocações e liberações de recursos. Um determinado estado é considerado seguro se, a partir dele, é possível concluir as tarefas pendentes. Caso o estado em questão somente leve a impasses, ele é considerado um **estado inseguro**. As técnicas de impedimento de impasses devem portanto manter o sistema sempre em um estado seguro, evitando entrar em estados inseguros.

A Figura 13.5 ilustra o grafo de estados do sistema de transferência de valores com duas tarefas analisado no início deste capítulo. Cada estado desse grafo é a combinação dos estados individuais das duas tarefas[1]. Pode-se observar no grafo que o estado e_{10} corresponde a um impasse, pois a partir dele não há mais nenhuma possibilidade de evolução do sistema a outros estados. Além disso, os estados e_4, e_7 e e_8 são considerados estados inseguros, pois levam invariavelmente na direção do impasse em e_{10}. Os demais estados são considerados seguros, pois a partir de qualquer um deles

[1]Este grafo de estados é simplificado; o grafo completo, detalhando cada solicitação, alocação e liberação de recursos, tem cerca de 40 estados possíveis.

é possível continuar a execução e retornar ao estado inicial e_0. Obviamente, operações que levem a estados inseguros devem ser impedidas, como $e_1 \rightarrow e_4$ e $e_2 \rightarrow e_4$.

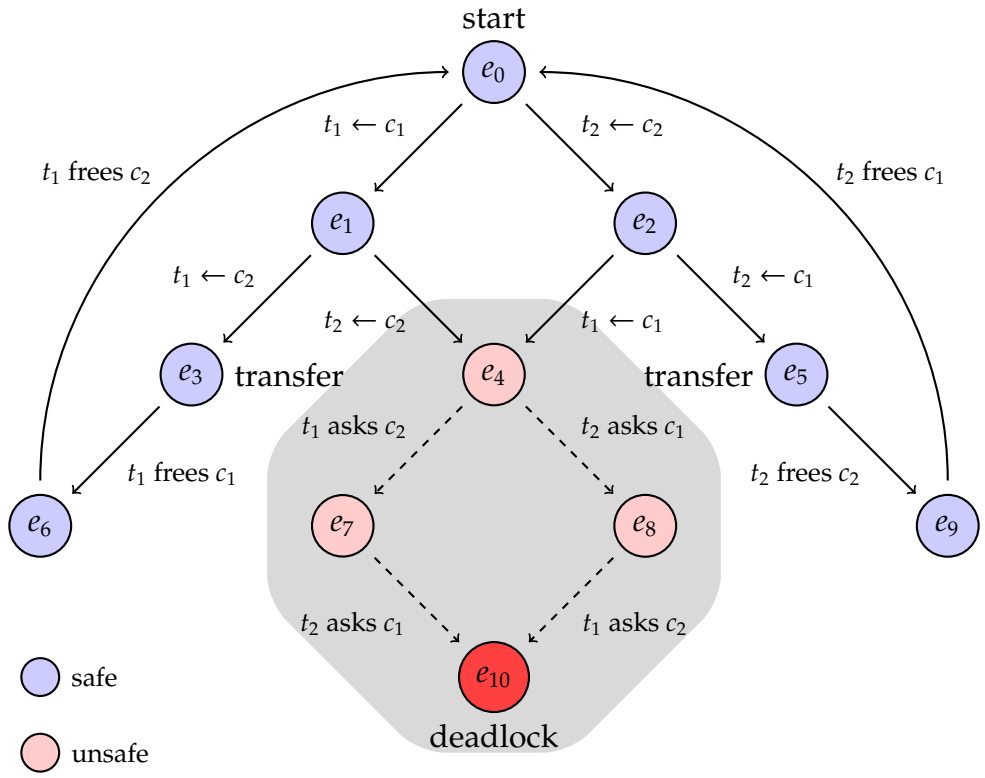


Figura 13.5: Grafo de estados do sistema de transferências com duas tarefas.

A técnica de impedimento de impasses mais conhecida é o *algoritmo do banqueiro*, criado por Dijkstra em 1965 [Tanenbaum, 2003]. Esse algoritmo faz uma analogia entre as tarefas de um sistema e os clientes de um banco, tratando os recursos como créditos emprestados às tarefas para a realização de suas atividades. O banqueiro decide que solicitações de empréstimo deve atender para conservar suas finanças em um estado seguro.

As técnicas de impedimento de impasses necessitam de algum conhecimento prévio sobre o comportamento das tarefas para poder operar. Normalmente é necessário conhecer com antecedência que recursos serão acessados por cada tarefa, quantas instâncias de cada um serão necessárias e qual a ordem de acesso aos recursos. Por essa razão, são pouco utilizadas na prática.

13.4.3 Detecção e resolução de impasses

Nesta abordagem, nenhuma medida preventiva é adotada para prevenir ou evitar impasses. As tarefas executam normalmente suas atividades, alocando e liberando recursos conforme suas necessidades. Quando ocorrer um impasse, o sistema deve detectá-lo, determinar quais as tarefas e recursos envolvidos e tomar medidas para desfazê-lo. Para aplicar essa técnica, duas questões importantes devem ser respondidas: como detectar os impasses? E como resolvê-los?

A **detecção de impasses** pode ser feita através da inspeção do grafo de alocação de recursos (Seção 13.3), que deve ser mantido pelo sistema e atualizado a cada alocação ou liberação de recurso. Um algoritmo de detecção de ciclos no grafo deve ser executado

periodicamente, para verificar a presença das dependências cíclicas que podem indicar impasses.

Alguns problemas decorrentes dessa estratégia são o custo de manutenção contínua do grafo de alocação e, sobretudo, o custo de sua análise: algoritmos de busca de ciclos em grafos têm custo computacional elevado, portanto sua ativação com muita frequência poderá prejudicar o desempenho do sistema. Por outro lado, se a detecção for ativada apenas esporadicamente, impasses podem demorar muito para ser detectados, o que também é ruim para o desempenho do sistema.

Uma vez detectado um impasse e identificadas as tarefas e recursos envolvidos, o sistema deve proceder à **resolução do impasse**, que pode ser feita de duas formas:

Eliminar tarefas: uma ou mais tarefas envolvidas no impasse são eliminadas, liberando seus recursos para que as demais tarefas possam prosseguir. A escolha das tarefas a eliminar deve levar em conta vários fatores, como o tempo de vida de cada uma, a quantidade de recursos que cada tarefa detém, o prejuízo para os usuários que dependem dessas tarefas, etc.

Retroceder tarefas: uma ou mais tarefas envolvidas no impasse têm sua execução parcialmente desfeita (uma técnica chamada *rollback*), de forma a fazer o sistema retornar a um estado seguro anterior ao impasse. Para retroceder a execução de uma tarefa, é necessário salvar periodicamente seu estado, de forma a poder recuperar um estado anterior quando necessário[2]. Além disso, operações envolvendo a rede ou interações com o usuário podem ser muito difíceis ou mesmo impossíveis de retroceder: como desfazer o envio de um pacote de rede, ou a reprodução de um arquivo de áudio na tela do usuário?

A detecção e resolução de impasses é uma abordagem interessante, mas relativamente pouco usada fora de situações muito específicas, porque o custo de detecção pode ser elevado e as alternativas de resolução sempre implicam perder tarefas ou parte das execuções já realizadas. Essa técnica é aplicada, por exemplo, no gerenciamento de transações em sistemas de bancos de dados, pois são providos mecanismos para criar *checkpoints* dos registros envolvidos antes da transação e para efetuar o *rollback* da mesma em caso de impasse.

Exercícios

1. Explique cada uma das quatro condições necessárias para a ocorrência de impasses.

2. Na prevenção de impasses:

   (a) Como pode ser feita a quebra da condição de posse e espera?

   (b) Como pode ser feita a quebra da condição de exclusão mútua?

   (c) Como pode ser feita a quebra da condição de espera circular?

   (d) Como pode ser feita a quebra da condição de não-preempção?

[2] Essa técnica é conhecida como *checkpointing* e os estados anteriores salvos são denominados *checkpoints*.

3. Como pode ser detectada a ocorrência de impasses, considerando disponível apenas um recurso de cada tipo?

4. Uma vez detectado um impasse, quais as abordagens possíveis para resolvê-lo? Explique-as e comente sua viabilidade.

5. Sobre as afirmações a seguir, relativas a impasses, indique quais são incorretas, justificando sua resposta:

   (a) Impasses ocorrem porque vários processos tentam usar o processador ao mesmo tempo.

   (b) Os sistemas operacionais atuais provêem vários recursos de baixo nível para o tratamento de impasses.

   (c) Podemos encontrar impasses em sistemas de processos que interagem unicamente por mensagens.

   (d) As condições necessárias para a ocorrência de impasses são também suficientes se houver somente um recurso de cada tipo no conjunto de processos considerado.

6. Sobre as afirmações a seguir, relativas às condições para ocorrência de impasses, indique quais são incorretas, justificando sua resposta:

   (a) As condições necessárias para a ocorrência de impasses são: exclusão mútua, posse e espera, não-preempção e espera circular.

   (b) O principal problema com a quebra da condição de posse e espera é que a taxa de uso dos recursos pode se tornar bastante baixa.

   (c) A condição de não-preempção indica que os processos envolvidos no impasse devem ser escalonados de forma não-preemptiva.

   (d) A condição de não-preempção pode ser detectada graficamente, no grafo de alocação de recursos.

   (e) A condição de exclusão mútua pode ser quebrada através do uso de processos gerenciadores de recursos ou de áreas de *spool*.

   (f) A quebra da condição de não-preempção só pode ser aplicada a recursos simples como arquivos e semáforos.

   (g) A quebra da condição de posse e espera consiste em forçar todos os processos a solicitar seus recursos em uma ordem global única e pré-fixada.

7. Sobre as afirmações a seguir, relativas à detecção e resolução de impasses, indique quais são incorretas, justificando sua resposta:

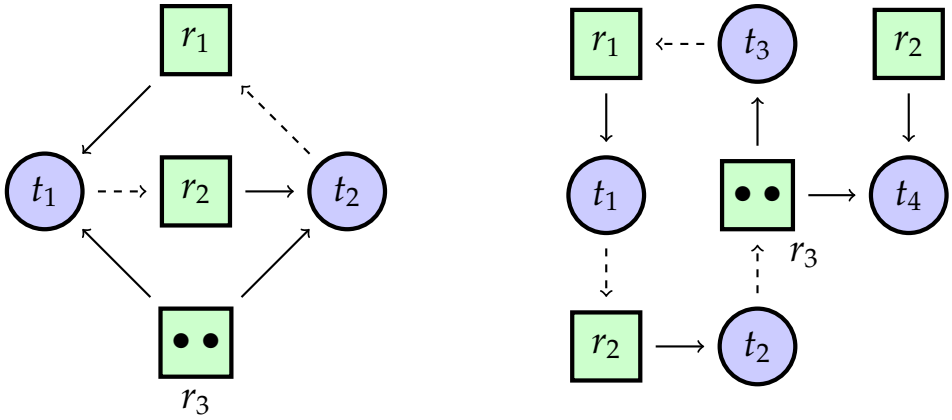
   (a) A detecção e recuperação de impasses é bastante usada, pois as técnicas de recuperação são facilmente aplicáveis.

   (b) A resolução de impasses através de *rollback* só pode ser implementada em processos que executem I/O ou interação com o usuário.

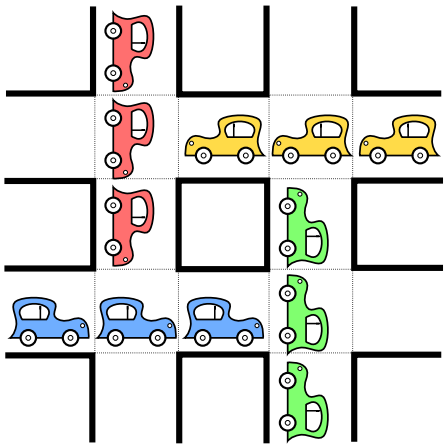
   (c) Uma vez detectado um impasse, ele pode ser facilmente resolvido através da preempção dos recursos envolvidos.

(d) O algoritmo de detecção de impasses deve ser executado com a maior frequência possível, a fim de evitar que um impasse já formado se alastre.

8. Nos grafos de alocação de recursos da figura a seguir, indique o(s) ciclo(s) onde existe um impasse:



9. A figura a seguir representa uma situação de impasse em um cruzamento de trânsito. Todas as ruas têm largura para um carro e sentido único. Mostre que as **quatro condições necessárias** para a ocorrência de impasses estão presentes nessa situação. Em seguida, defina uma regra simples a ser seguida por cada carro para **evitar** essa situação; regras envolvendo algum tipo de informação centralizada não devem ser usadas.



10. O trecho de código a seguir apresenta uma solução para o problema do jantar dos filósofos, mas ele está sujeito a impasses. Explique como o impasse pode ocorrer. A seguir, modifique o código para que ele funcione corretamente e explique sua solução.

```
#define N 5

sem_t garfo[5] ; // 5 semáforos iniciados em 1

void filosofo (int i) // 5 threads (i varia de 0 a 4)
{
   while (1)
   {
      medita ();
      sem_down (garfo [i]) ;
      sem_down (garfo [(i+1) % N]) ;
      come ();
      sem_up (garfo [i]) ;
      sem_up (garfo [(i+1) % N]) ;
   }
}
```

Atividades

1. Escreva um programa implementando o jantar dos filósofos, para um número N genérico de filósofos. Em seguida, avalie o desempenho das soluções do saleiro (Seção 12.4) e de quebra da espera circular (Seção 13.4.1), em função do número de filósofos (varie entre 5 e 1.000 filósofos). Como critério de desempenho, pode ser usado o percentual de filósofos que conseguem comer por segundo.

Referências

M. Ben-Ari. *Principles of Concurrent and Distributed Programming*. Prentice-Hall, 1990.

E. Coffman, M. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3 (2):67–78, 1971.

R. Holt. Some deadlock properties of computer systems. *ACM Computing Surveys*, 4(3): 179–196, september 1972.

A. Tanenbaum. *Sistemas Operacionais Modernos, 2ª edição*. Pearson – Prentice-Hall, 2003.