Sistemas Operacionais: Conceitos e Mecanismos

IV - Mecanismos de Coordenação

Prof. Carlos Alberto Maziero
DInf UFPR
http://www.inf.ufpr.br/maziero

4 de agosto de 2017

Este texto está licenciado sob a Licença *Attribution-NonCommercial-ShareAlike 3.0 Unported* da *Creative Commons* (CC). Em resumo, você deve creditar a obra da forma especificada pelo autor ou licenciante (mas não de maneira que sugira que estes concedem qualquer aval a você ou ao seu uso da obra). Você não pode usar esta obra para fins comerciais. Se você alterar, transformar ou criar com base nesta obra, você poderá distribuir a obra resultante apenas sob a mesma licença, ou sob uma licença similar à presente. Para ver uma cópia desta licença, visite http://creativecommons.org/licenses/by-nc-sa/3.0/.

Este texto foi produzido usando exclusivamente software livre: Sistema Operacional *GNU/Linux* (distribuições *Fedora* e *Ubuntu*), compilador de texto LaTeX 2_ε, gerenciador de referências *BibTeX*, editor gráfico *Inkscape*, criadores de gráficos *GNUPlot* e *GraphViz* e processador PS/PDF *GhostScript*, entre outros.

Sumário

1 Objetivos **3**

2 Condições de disputa **3**

3 Seções críticas **6**

4 Inibição de interrupções **7**

5 Soluções com espera ocupada **8**
- 5.1 A solução óbvia . . . 8
- 5.2 Alternância de uso . . . 9
- 5.3 O algoritmo de Peterson . . . 10
- 5.4 Instruções *Test-and-Set* . . . 10
- 5.5 Problemas . . . 12

6 Semáforos **12**

7 Variáveis de condição **15**

8 Monitores **17**

9 Problemas clássicos de coordenação **19**
- 9.1 O problema dos produtores/consumidores . . . 19
- 9.2 O problema dos leitores/escritores . . . 21
- 9.3 O jantar dos filósofos . . . 23

10 Impasses **25**
- 10.1 Caracterização de impasses . . . 27
- 10.2 Grafos de alocação de recursos . . . 28
- 10.3 Técnicas de tratamento de impasses . . . 29
  - 10.3.1 Prevenção de impasses . . . 30
  - 10.3.2 Impedimento de impasses . . . 31
  - 10.3.3 Detecção e resolução de impasses . . . 32

Resumo

Muitas implementações de sistemas complexos são estruturadas como várias tarefas interdependentes, que cooperam entre si para atingir os objetivos da aplicação, como por exemplo em um navegador Web. Para que as várias tarefas que compõem uma aplicação possam cooperar, elas precisam comunicar informações umas às outras e coordenar suas atividades, para garantir que os resultados obtidos sejam coerentes. Este módulo apresenta os principais conceitos, problemas e soluções referentes à coordenação entre tarefas.

1 Objetivos

Em um sistema multitarefas, várias tarefas podem executar simultaneamente, acessando recursos compartilhados como áreas de memória, arquivos, conexões de rede, etc. Neste capítulo serão estudados os problemas que podem ocorrer quando duas ou mais tarefas acessam os mesmos recursos de forma concorrente; também serão apresentadas as principais técnicas usadas para coordenar de forma eficiente os acessos das tarefas aos recursos compartilhados.

2 Condições de disputa

Quando duas ou mais tarefas acessam simultaneamente um recurso compartilhado, podem ocorrer problemas de consistência dos dados ou do estado do recurso acessado. Esta seção descreve detalhadamente a origem dessas inconsistências, através de um exemplo simples, mas que permite ilustrar claramente o problema.

O código apresentado a seguir implementa de forma simplificada a operação de depósito (função `depositar`) de um valor em uma conta bancária informada como parâmetro. Para facilitar a compreensão do código de máquina apresentado na sequência, todos os valores manipulados são inteiros.

```
1  typedef struct conta_t
2  {
3     int saldo ;     // saldo atual da conta
4     ...             // outras informações da conta
5  } conta_t ;
6
7  void depositar (conta_t* conta, int valor)
8  {
9     conta->saldo += valor ;
10 }
```

Após a compilação em uma plataforma *Intel i386*, a função `depositar` assume a seguinte forma em código de máquina (nos comentários ao lado do código, reg_i é um registrador e $mem(x)$ é a posição de memória onde está armazenada a variável x):

```
00000000 <depositar>:
push %ebp              # guarda o valor do "stack frame"
mov  %esp,%ebp         # ajusta o stack frame para executar a função

mov  0x8(%ebp),%ecx    # mem(saldo) → reg1
mov  0x8(%ebp),%edx    # mem(saldo) → reg2
mov  0xc(%ebp),%eax    # mem(valor) → reg3
add  (%edx),%eax       # [reg1, reg2] + reg3 → [reg1, reg2]
mov  %eax,(%ecx)       # [reg1, reg2] → mem(saldo)

leave                  # restaura o stack frame anterior
ret                    # retorna à função anterior
```

Considere que a função `depositar` faz parte de um sistema mais amplo de controle de contas bancárias, que pode ser acessado simultaneamente por centenas ou milhares de usuários em terminais distintos. Caso dois clientes em terminais diferentes tentem depositar valores na mesma conta ao mesmo tempo, existirão duas tarefas acessando os dados (variáveis) da conta de forma concorrente. A Figura 1 ilustra esse cenário.

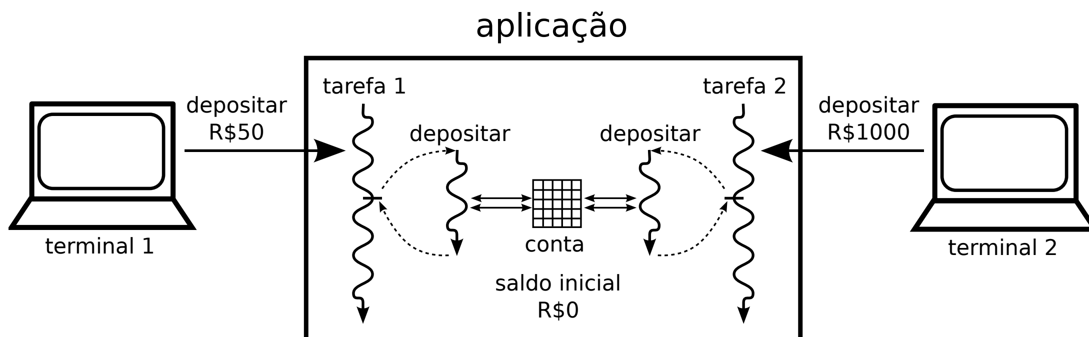

Figura 1: Acessos concorrentes a variáveis compartilhadas.

O comportamento dinâmico da aplicação pode ser modelado através de diagramas de tempo. Caso o depósito da tarefa t_1 execute integralmente **antes** ou **depois** do depósito efetuado por t_2, teremos os diagramas de tempo da Figura 2. Em ambas as execuções o saldo inicial da conta passou de R$ 0,00 para R$ 1050,00, conforme esperado.

No entanto, caso as operações de depósito de t_1 e de t_2 se entrelacem, podem ocorrer interferências entre ambas, levando a resultados incorretos. Em sistemas monoprocessados, a sobreposição pode acontecer caso ocorram trocas de contexto durante a execução do depósito. Em sistemas multiprocessados a situação é ainda mais complexa, pois cada tarefa poderá estar executando em um processador distinto.

Os diagramas de tempo apresentados na Figura 3 mostram execuções onde houve entrelaçamento das operações de depósito de t_1 e de t_2. Em ambas as execuções o saldo final **não** corresponde ao resultado esperado, pois um dos depósitos é perdido. No caso, apenas é concretizado o depósito da tarefa que realizou a operação $mem(saldo) \leftarrow reg_1$ por último[1].

[1]Não há problema em ambas as tarefas usarem os mesmos registradores reg_1 e reg_2, pois os valores de todos os registradores são salvos/restaurados a cada troca de contexto entre tarefas.

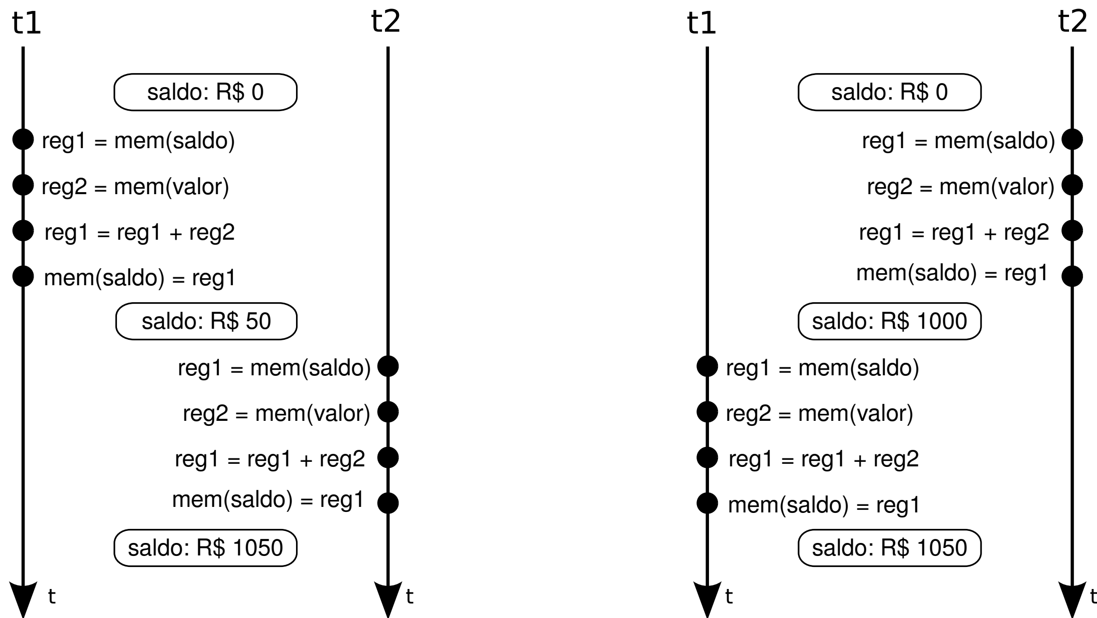

Figura 2: Operações de depósitos não-concorrentes.

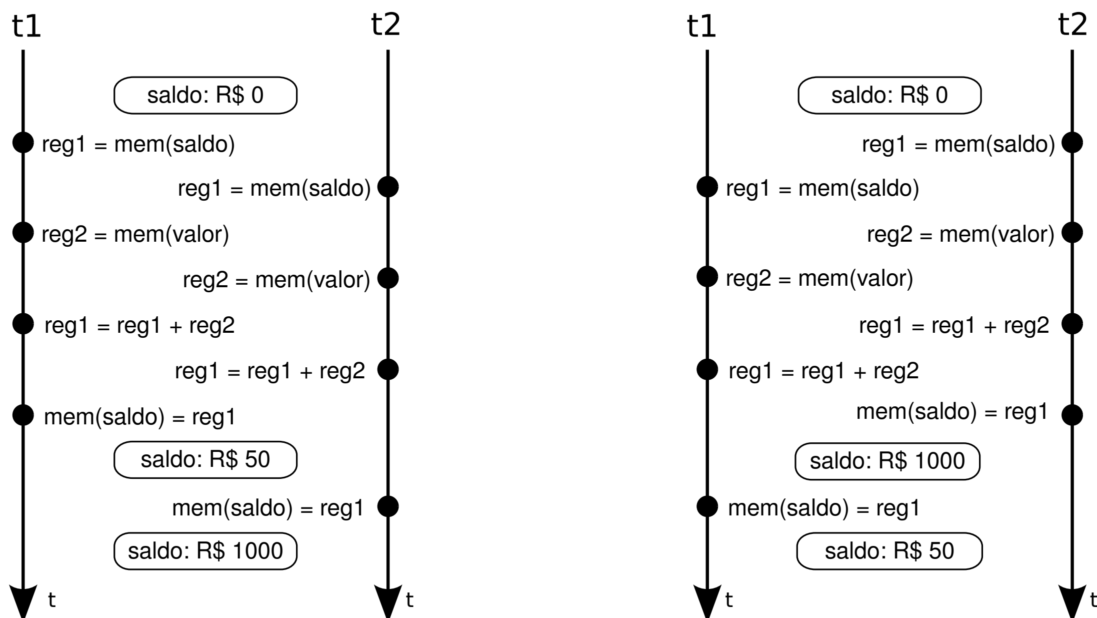

Figura 3: Operações de depósito concorrentes.

Os erros e inconsistências gerados por acessos concorrentes a dados compartilhados, como os ilustrados na Figura 3, são denominados **condições de disputa**, ou condições de corrida (do inglês *race conditions*). Condições de disputa podem ocorrer em qualquer sistema onde várias tarefas (processos ou *threads*) acessam de forma concorrente recursos compartilhados (variáveis, áreas de memória, arquivos abertos, etc.). Finalmente, condições de disputa somente existem caso ao menos uma das operações envolvidas seja de escrita; acessos de leitura concorrentes entre si não geram condições de disputa.

É importante observar que condições de disputa são erros *dinâmicos*, ou seja, que não aparecem no código fonte e que só se manifestam durante a execução, sendo dificilmente detectáveis através da análise do código fonte. Além disso, erros dessa natureza não se manifestam a cada execução, mas apenas quando certos entrelaçamentos ocorrerem. Assim, uma condição de disputa poderá permanecer latente no código durante anos, ou mesmo nunca se manifestar. A depuração de programas contendo condições de disputa pode ser muito complexa, pois o problema só se manifesta com acessos simultâneos aos mesmos dados, o que pode ocorrer raramente e ser difícil de reproduzir durante a depuração. Por isso, é importante conhecer técnicas que previnam a ocorrência de condições de disputa.

3 Seções críticas

Na seção anterior vimos que tarefas acessando dados compartilhados de forma concorrente podem ocasionar condições de disputa. Os trechos de código de cada tarefa que acessam dados compartilhados são denominados **seções críticas** (ou *regiões críticas*). No caso da Figura 1, as seções críticas das tarefas t_1 e t_2 são idênticas e resumidas à seguinte linha de código:

```
1 conta.saldo += valor ;
```

De modo geral, seções críticas são todos os trechos de código que manipulam dados compartilhados onde podem ocorrer condições de disputa. Um programa pode ter várias seções críticas, relacionadas entre si ou não (caso manipulem dados compartilhados distintos). Para assegurar a correção de uma implementação, deve-se impedir o entrelaçamento de seções críticas: apenas uma tarefa pode estar na seção crítica a cada instante. Essa propriedade é conhecida como **exclusão mútua**.

Diversos mecanismos podem ser definidos para impedir o entrelaçamento de seções críticas e assim prover a exclusão mútua. Todos eles exigem que o programador defina os limites (início e o final) de cada seção crítica. Genericamente, cada seção crítica i pode ser associada a um identificador cs_i e são definidas as primitivas $enter(t_a, cs_i)$, para que a tarefa t_a indique que deseja entrar na seção crítica cs_i, e $leave(t_a, cs_i)$, para que t_a informe que está saindo da seção crítica cs_i. A primitiva $enter(cs_i)$ é bloqueante: caso uma tarefa já esteja ocupando a seção crítica cs_i, as demais tarefas que tentarem entrar deverão aguardar até que a primeira libere a seção crítica, através da primitiva $leave(cs_i)$.

Usando as primitivas *enter* e *leave*, o código da operação de depósito visto na Seção 2 pode ser reescrito como segue:

```
typedef struct conta_t
{
   int saldo ;              // saldo atual da conta
   int numero ;             // identificação da conta (seção crítica)
   ...                      // outras informações da conta
} conta_t ;

void depositar (conta_t* conta, int valor)
{
   enter (conta->numero) ;  // tenta entrar na seção crítica
   conta->saldo += valor ;  // está na seção crítica
   leave (conta->numero) ;  // sai da seção crítica
}
```

Nas próximas seções serão estudadas várias soluções para a implementação das primitivas *enter* e *leave*, bem como abordagens alternativas. As soluções propostas devem atender a alguns critérios básicos que são enumerados a seguir:

Exclusão mútua : somente uma tarefa pode estar dentro da seção crítica em cada instante.

Espera limitada : uma tarefa que aguarda acesso a uma seção crítica deve ter esse acesso garantido em um tempo finito.

Independência de outras tarefas : a decisão sobre o uso de uma seção crítica deve depender somente das tarefas que estão tentando entrar na mesma. Outras tarefas do sistema, que no momento não estejam interessadas em entrar na região crítica, não podem ter influência sobre essa decisão.

Independência de fatores físicos : a solução deve ser puramente lógica e não depender da velocidade de execução das tarefas, de temporizações, do número de processadores no sistema ou de outros fatores físicos.

4 Inibição de interrupções

Uma solução simples para a implementação das primitivas *enter* e *leave* consiste em impedir as trocas de contexto dentro da seção crítica. Ao entrar em uma seção crítica, a tarefa desativa (mascara) as interrupções que possam provocar trocas de contexto, e as reativa ao sair da seção crítica. Apesar de simples, essa solução raramente é usada para a construção de aplicações devido a vários problemas:

- Ao desligar as interrupções, a preempção por tempo ou por recursos deixa de funcionar; caso a tarefa entre em um laço infinito dentro da seção crítica, o sistema inteiro será bloqueado. Uma tarefa mal intencionada pode forçar essa situação e travar o sistema.

- Enquanto as interrupções estão desativadas, os dispositivos de entrada/saída deixam de ser atendidos pelo núcleo, o que pode causar perdas de dados ou outros problemas. Por exemplo, uma placa de rede pode perder novos pacotes se seus buffers estiverem cheios e não forem tratados pelo núcleo em tempo hábil.

- A tarefa que está na seção crítica não pode realizar operações de entrada/saída, pois os dispositivos não irão responder.
- Esta solução só funciona em sistemas monoprocessados; em uma máquina multiprocessada ou multicore, duas tarefas concorrentes podem executar simultaneamente em processadores separados, acessando a seção crítica ao mesmo tempo.

Devido a esses problemas, a inibição de interrupções é uma operação privilegiada e somente utilizada em algumas seções críticas dentro do núcleo do sistema operacional e nunca pelas aplicações.

5 Soluções com espera ocupada

Uma primeira classe de soluções para o problema da exclusão mútua no acesso a seções críticas consiste em testar continuamente uma condição que indica se a seção desejada está livre ou ocupada. Esta seção apresenta algumas soluções clássicas usando essa abordagem.

5.1 A solução óbvia

Uma solução aparentemente trivial para o problema da seção crítica consiste em usar uma variável *busy* para indicar se a seção crítica desejada está livre ou ocupada. Usando essa abordagem, a implementação das primitivas *enter* e *leave* poderia ser escrita assim:

```
1  int busy = 0 ;        // a seção está inicialmente livre
2
3  void enter (int task)
4  {
5     while (busy) ;     // espera enquanto a seção estiver ocupada
6     busy = 1 ;         // marca a seção como ocupada
7  }
8
9  void leave (int task)
10 {
11    busy = 0 ;         // libera a seção (marca como livre)
12 }
```

Infelizmente, essa solução óbvia e simples **não funciona**! Seu grande defeito é que o teste da variável *busy* (na linha 5) e sua atribuição (na linha 6) são feitos em momentos distintos; caso ocorra uma troca de contexto entre as linhas 5 e 6 do código, poderá ocorrer uma condição de disputa envolvendo a variável *busy*, que terá como consequência a violação da exclusão mútua: duas ou mais tarefas poderão entrar simultaneamente na seção crítica (vide o diagrama de tempo da Figura 4). Em outras palavras, as linhas 5 e 6 da implementação também formam uma seção crítica, que deve ser protegida.

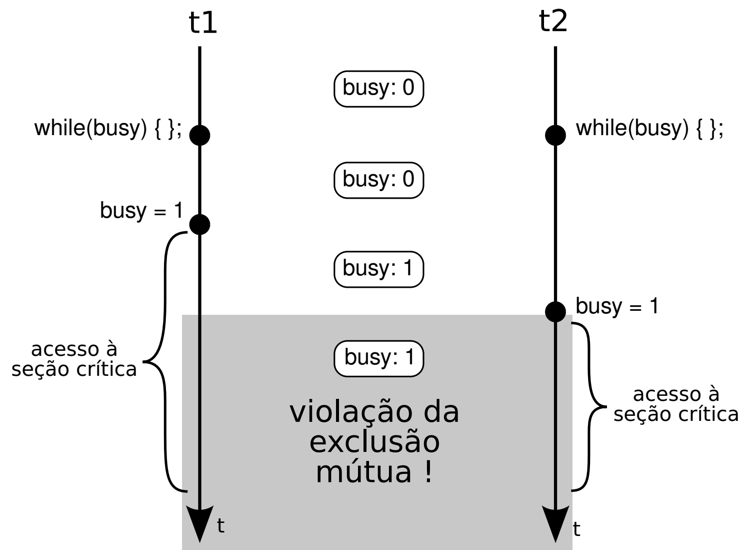

Figura 4: Condição de disputa no acesso à variável *busy*.

5.2 Alternância de uso

Outra solução simples para a implementação das primitivas *enter* e *leave* consiste em definir uma variável *turno*, que indica de quem é a vez de entrar na seção crítica. Essa variável deve ser ajustada cada vez que uma tarefa sai da seção crítica, para indicar a próxima tarefa a usá-la. A implementação das duas primitivas fica assim:

```
int turn = 0 ;
int num_tasks ;

void enter (int task)       // task vale 0, 1, ..., num_tasks-1
{
   while (turn != task) ;   // a tarefa espera seu turno
}

void leave (int task)
{
   if (turn < num_tasks-1)  // o turno é da próxima tarefa
      turn ++ ;
   else
      turn = 0 ;            // volta à primeira tarefa
}
```

Nessa solução, cada tarefa aguarda seu turno de usar a seção crítica, em uma sequência circular: $t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \cdots \rightarrow t_{n-1} \rightarrow t_0$. Essa abordagem garante a exclusão mútua entre as tarefas e independe de fatores externos, mas não atende os demais critérios: caso uma tarefa t_i não deseje usar a seção crítica, todas as tarefas t_j com $j > i$ ficarão impedidas de fazê-lo, pois a variável *turno* não irá evoluir.

5.3 O algoritmo de Peterson

Uma solução correta para a exclusão mútua no acesso a uma seção crítica por duas tarefas foi proposta inicialmente por Dekker em 1965. Em 1981, Gary Peterson propôs uma solução mais simples e elegante para o mesmo problema [Raynal, 1986]. O algoritmo de Peterson pode ser resumido no código a seguir:

```
int turn = 0 ;          // indica de quem é a vez
int wants[2] = {0, 0} ; // indica se a tarefa i quer acessar a seção crítica

void enter (int task)   // task pode valer 0 ou 1
{
   int other = 1 - task ; // indica a outra tarefa
   wants[task] = 1 ;     // task quer acessar a seção crítica
   turn = task ;
   while ((turn == task) && wants[other]) ; // espera ocupada
}

void leave (int task)
{
  wants[task] = 0 ;     // task libera a seção crítica
}
```

Os algoritmos de Dekker e de Peterson foram desenvolvidos para garantir a exclusão mútua entre duas tarefas, garantindo também o critério de espera limitada[2]. Diversas generalizações para n tarefas podem ser encontradas na literatura [Raynal, 1986], sendo a mais conhecida delas o *algoritmo da padaria*, proposto por Leslie Lamport [Lamport, 1974].

5.4 Instruções *Test-and-Set*

O uso de uma variável *busy* para controlar a entrada em uma seção crítica é uma ideia interessante, que infelizmente não funciona porque o teste da variável *busy* e seu ajuste são feitos em momentos distintos do código, permitindo condições de corrida.

Para resolver esse problema, projetistas de hardware criaram instruções em código de máquina que permitem testar e atribuir um valor a uma variável de forma *atômica* ou indivisível, sem possibilidade de troca de contexto entre essas duas operações. A execução atômica das operações de teste e atribuição (*Test & Set instructions*) impede a ocorrência de condições de disputa. Uma implementação básica dessa ideia está na instrução de máquina *Test-and-Set Lock* (TSL), cujo comportamento é descrito pelo seguinte pseudocódigo (que é executado atomicamente pelo processador):

[2]Este algoritmo **pode falhar** quando usado em algumas máquinas multinúcleo ou multiprocessadas, pois algumas arquiteturas permitem acesso fora de ordem à memória, ou seja, permitem que operações de leitura na memória se antecipem a operações de escrita executadas posteriormente, para obter mais desempenho. Este é o caso dos processadores *Pentium* e *AMD*.

$$
\begin{aligned}
TSL(x):\quad & old \leftarrow x \\
& x \leftarrow 1 \\
& return(old)
\end{aligned}
$$

A implementação das primitivas *enter* e *leave* usando a instrução TSL assume a seguinte forma:

```
int lock = 0 ;            // variável de trava

void enter (int *lock)    // passa o endereço da trava
{
   while ( TSL (*lock) ) ; // espera ocupada
}

void leave (int *lock)
{
  (*lock) = 0 ;           // libera a seção crítica
}
```

Outros processadores oferecem uma instrução que efetua a troca atômica de conteúdo (*swapping*) entre dois registradores, ou entre um registrador e uma posição de memória. No caso da família de processadores Intel i386 (incluindo o Pentium e seus sucessores), a instrução de troca se chama XCHG (do inglês *exchange*) e sua funcionalidade pode ser resumida assim:

$$XCHG\ op_1, op_2 :\ op_1 \rightleftarrows op_2$$

A implementação das primitivas *enter* e *leave* usando a instrução XCHG é um pouco mais complexa:

```
int lock ;                // variável de trava

enter (int *lock)
{
   int key = 1 ;          // variável auxiliar (local)
   while (key)            // espera ocupada
     XCHG (lock, &key) ;  // alterna valores de lock e key
}

leave (int *lock)
{
   (*lock) = 0 ;          // libera a seção crítica
}
```

Os mecanismos de exclusão mútua usando instruções atômicas no estilo TSL são amplamente usados no interior do sistema operacional, para controlar o acesso a seções críticas internas do núcleo, como descritores de tarefas, buffers de arquivos ou de conexões de rede, etc. Nesse contexto, eles são muitas vezes denominados

spinlocks. Todavia, mecanismos de espera ocupada são inadequados para a construção de aplicações de usuário, como será visto a seguir.

5.5 Problemas

Apesar das soluções para o problema de acesso à seção crítica usando espera ocupada garantirem a exclusão mútua, elas sofrem de alguns problemas que impedem seu uso em larga escala nas aplicações de usuário:

Ineficiência : as tarefas que aguardam o acesso a uma seção crítica ficam testando continuamente uma condição, consumindo tempo de processador sem necessidade. O procedimento adequado seria suspender essas tarefas até que a seção crítica solicitada seja liberada.

Injustiça : não há garantia de ordem no acesso à seção crítica; dependendo da duração de *quantum* e da política de escalonamento, uma tarefa pode entrar e sair da seção crítica várias vezes, antes que outras tarefas consigam acessá-la.

Por estas razões, as soluções com espera ocupada são pouco usadas na construção de aplicações. Seu maior uso se encontra na programação de estruturas de controle de concorrência dentro do núcleo do sistema operacional (onde se chamam *spinlocks*), e na construção de sistemas de computação dedicados, como controladores embarcados mais simples.

6 Semáforos

Em 1965, o matemático holandês E. Dijkstra propôs um mecanismo de coordenação eficiente e flexível para o controle da exclusão mútua entre n tarefas: o **semáforo** [Raynal, 1986]. Apesar de antigo, o semáforo continua sendo o mecanismo de sincronização mais utilizado na construção de aplicações concorrentes, sendo usado de forma explícita ou implícita (na construção de mecanismos de coordenação mais abstratos, como os monitores).

Um semáforo pode ser visto como uma variável s, que representa uma seção crítica e cujo conteúdo não é diretamente acessível ao programador. Internamente, cada semáforo contém um contador inteiro *s.counter* e uma fila de tarefas *s.queue*, inicialmente vazia. Sobre essa variável podem ser aplicadas duas operações atômicas, descritas a seguir:

Down(s) : usado para solicitar acesso à seção crítica associada a s. Caso a seção esteja livre, a operação retorna imediatamente e a tarefa pode continuar sua execução; caso contrário, a tarefa solicitante é suspensa e adicionada à fila do semáforo; o contador associado ao semáforo é decrementado[3]. Dijkstra denominou essa operação $P(s)$ (do holandês *probeer*, que significa *tentar*).

[3] Alguns sistemas implementam também a chamada *TryDown*(s), cuja semântica é não-bloqueante: caso o semáforo solicitado esteja ocupado, a chamada retorna imediatamente, com um código de erro.

> *Down*(s): // a executar de forma atômica
> $s.counter \leftarrow s.counter - 1$
> **if** $s.counter < 0$ **then**
>   põe a tarefa corrente no final de *s.queue*
>   suspende a tarefa corrente
> **end if**

Up(s) : invocado para liberar a seção crítica associada a s; o contador associado ao semáforo é incrementado. Caso a fila do semáforo não esteja vazia, a primeira tarefa da fila é acordada, sai da fila do semáforo e volta à fila de tarefas prontas para retomar sua execução. Essa operação foi inicialmente denominada $V(s)$ (do holandês *verhoog*, que significa *incrementar*). Deve-se observar que esta chamada não é bloqueante: a tarefa não precisa ser suspensa ao executá-la.

> *Up*(s): // a executar de forma atômica
> $s.counter \leftarrow s.counter + 1$
> **if** $s.counter \leq 0$ **then**
>   retira a primeira tarefa t de *s.queue*
>   devolve t à fila de tarefas prontas (ou seja, acorda t)
> **end if**

As operações de acesso aos semáforos são geralmente implementadas pelo núcleo do sistema operacional, na forma de chamadas de sistema. É importante observar que a execução das operações *Down*(s) e *Up*(s) deve ser **atômica**, ou seja, não devem ocorrer acessos concorrentes às variáveis internas do semáforo, para evitar condições de disputa sobre as mesmas. Para garantir a atomicidade dessas operações em um sistema monoprocessador, seria suficiente inibir as interrupções durante a execução das mesmas; no caso de sistemas com mais de um núcleo, torna-se necessário usar outros mecanismos de controle de concorrência, como operações TSL, para proteger a integridade interna do semáforo. Nestes casos, a espera ocupada não constitui um problema, pois a execução dessas operações é muito rápida.

Usando semáforos, o código de depósito em conta bancária apresentado na Seção 2 poderia ser reescrito da seguinte forma:

```
1  typedef struct conta_t
2  {
3     int saldo ;            // saldo atual da conta
4     sem_t s = 1;           // semáforo associado à conta, valor inicial 1
5     ...                    // outras informações da conta
6  } conta_t ;
7
8  void depositar (conta_t * conta, int valor)
9  {
10    down (conta->s) ;      // solicita acesso à conta
11    conta->saldo += valor ; // seção crítica
12    up (conta->s) ;        // libera o acesso à conta
13 }
```

A suspensão das tarefas que aguardam o acesso à seção crítica elimina a espera ocupada, o que torna esse mecanismo mais eficiente no uso do processador que os anteriores. A fila de tarefas associada ao semáforo contém todas as tarefas que foram suspensas ao solicitar acesso à seção crítica usando a chamada *Down*(*s*). Como a fila obedece uma política FIFO, garante-se a também a justiça no acesso à seção crítica, pois todos os processos que aguardam no semáforo serão atendidos em sequência[4]. Por sua vez, o valor inteiro associado ao semáforo funciona como um contador de recursos: caso seja positivo, indica quantas instâncias daquele recurso estão disponíveis. Caso seja negativo, indica quantas tarefas estão aguardando para usar aquele recurso. Seu valor inicial permite expressar diferentes situações de sincronização, como será visto na Seção 9.

A listagem a seguir apresenta um exemplo hipotético de uso de um semáforo para controlar o acesso a um estacionamento. O valor inicial do semáforo `vagas` representa o número de vagas inicialmente livres no estacionamento (500). Quando um carro deseja entrar no estacionamento ele solicita uma vaga; enquanto o semáforo for positivo não haverão bloqueios, pois há vagas livres. Caso não existam mais vagas livres, a chamada `carro_entra()` ficará bloqueada até que alguma vaga seja liberada, o que ocorre quando outro carro acionar a chamada `carro_sai()`. Esta solução simples pode ser aplicada a um estacionamento com várias entradas e várias saídas simultâneas.

```
1  sem_t vagas = 500 ;
2
3  void carro_entra ()
4  {
5     down (vagas) ;        // solicita uma vaga de estacionamento
6     ...                   // demais ações específicas da aplicação
7  }
8
9  void carro_sai ()
10 {
11    up (vagas) ;         // libera uma vaga de estacionamento
12    ...                  // demais ações específicas da aplicação
13 }
```

A API POSIX define várias chamadas para a criação e manipulação de semáforos. As chamadas mais frequentemente utilizadas estão indicadas a seguir:

[4]Algumas implementações de semáforos acordam uma tarefa aleatória da fila, não necessariamente a primeira tarefa. Essas implementações são chamadas de *semáforos fracos*, por não garantirem a justiça no acesso à seção crítica nem a ausência de inanição (*starvation*) de tarefas.

```
#include <semaphore.h>

// inicializa um semáforo apontado por "sem", com valor inicial "value"
int sem_init(sem_t *sem, int pshared, unsigned int value);

// Operação Up(s)
int sem_post(sem_t *sem);

// Operação Down(s)
int sem_wait(sem_t *sem);

// Operação TryDown(s), retorna erro se o semáforo estiver ocupado
int sem_trywait(sem_t *sem);
```

Os semáforos nos quais o contador inteiro pode assumir qualquer valor são denominados **semáforos genéricos** e constituem um mecanismo de coordenação muito poderoso. No entanto, Muitos ambientes de programação, bibliotecas de threads e até mesmo núcleos de sistema proveem uma versão simplificada de semáforos, na qual o contador só assume dois valores possíveis: *livre* (1) ou *ocupado* (0). Esses semáforos simplificados são chamados de **mutexes** (uma abreviação de *mutual exclusion*) ou semáforos binários. Por exemplo, algumas das funções definidas pelo padrão POSIX [Gallmeister, 1994, Barney, 2005] para criar e usar *mutexes* são:

```
#include <pthread.h>

// inicializa uma variável do tipo mutex, usando um struct de atributos
int pthread_mutex_init (pthread_mutex_t *restrict mutex,
                        const pthread_mutexattr_t *restrict attr);

// destrói uma variável do tipo mutex
int pthread_mutex_destroy (pthread_mutex_t *mutex);

// solicita acesso à seção crítica protegida pelo mutex;
// se a seção estiver ocupada, bloqueia a tarefa
int pthread_mutex_lock (pthread_mutex_t *mutex);

// solicita acesso à seção crítica protegida pelo mutex;
// se a seção estiver ocupada, retorna com status de erro
int pthread_mutex_trylock (pthread_mutex_t *mutex);

// libera o acesso à seção crítica protegida pelo mutex
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

7 Variáveis de condição

Além dos semáforos, outro mecanismo de sincronização de uso frequente são as *variáveis de condição*, ou variáveis condicionais. Uma variável de condição não representa um valor, mas uma condição, que pode ser aguardada por uma tarefa. Quando uma tarefa aguarda uma condição, ela é colocada para dormir até que a condição seja

verdadeira. Assim, a tarefa não precisa testar continuamente aquela condição, evitando uma espera ocupada.

Uma tarefa aguardando uma condição representada pela variável de condição *c* pode ficar suspensa através do operador *wait(c)*, para ser notificada mais tarde, quando a condição se tornar verdadeira. Essa notificação ocorre quando outra tarefa chamar o operador *notify(c)* (também chamado *signal(c)*). Por definição, uma variável de condição *c* está sempre associada a um semáforo binário *c.mutex* e a uma fila *c.queue*. O *mutex* garante a exclusão mútua sobre a condição representada pela variável de condição, enquanto a fila serve para armazenar em ordem as tarefas que aguardam aquela condição.

Uma implementação hipotética[5] para as operações *wait*, *notify* e *broadcast* (que notifica todas as tarefas na espera da condição) para uma tarefa *t*, seria a seguinte:

```
1  wait (c):
2     c.queue ← t              // coloca a tarefa t no fim de c.queue
3     unlock (c.mutex)         // libera o mutex
4     suspend (t)              // põe a tarefa atual para dormir
5     lock (c.mutex)           // quando acordar, obtém o mutex imediatamente
6
7  notify (c):
8     awake (first (c.queue)) // acorda a primeira tarefa da fila c.queue
9
10 broadcast (c):
11    awake (c.queue)          // acorda todas as tarefas da fila c.queue
```

No exemplo a seguir, a tarefa *A* espera por uma condição que será sinalizada pela tarefa *B*. A condição de espera pode ser qualquer: um novo dado em um buffer de entrada, a conclusão de um procedimento externo, a liberação de espaço em disco, etc.

[5] Assim como os operadores sobre semáforos, os operadores sobre variáveis de condição também devem ser implementados de forma atômica.

```
1  Task A ()
2  {
3     ...
4     lock (c.mutex)
5     while (not condition)
6        wait (c) ;
7     ...
8     unlock (c.mutex)
9     ...
10 }
11
12 Task B ()
13 {
14    ...
15    lock (c.mutex)
16    condition = true
17    notify (c)
18    unlock (c.mutex)
19    ...
20 }
```

É importante observar que na definição original de variáveis de condição (conhecida como *semântica de Hoare*), a operação *notify(c)* fazia com que a tarefa notificadora perdesse imediatamente o semáforo e o controle do processador, que eram devolvidos à primeira tarefa da fila de *c*. Como essa semântica é complexa de implementar e interfere diretamente no escalonador de processos, as implementações modernas de variáveis de condição normalmente adotam a *semântica Mesa* [Lampson and Redell, 1980], proposta na linguagem de mesmo nome. Nessa semântica, a operação *notify(c)* apenas "acorda" as tarefas que esperam pela condição, sem suspender a execução da tarefa corrente. Cabe ao programador garantir que a tarefa corrente vai liberar o *mutex* e não vai alterar o estado associado à variável de condição.

As variáveis de condição estão presentes no padrão POSIX, através de operadores como `pthread_cond_wait`, `pthread_cond_signal` e `pthread_cond_broadcast`. O padrão POSIX adota a semântica *Mesa*.

8 Monitores

Ao usar semáforos, um programador define explicitamente os pontos de sincronização necessários em seu programa. Essa abordagem é eficaz para programas pequenos e problemas de sincronização simples, mas se torna inviável e suscetível a erros em sistemas mais complexos. Por exemplo, se o programador esquecer de liberar um semáforo previamente alocado, o programa pode entrar em um impasse (vide Seção 10). Por outro lado, se ele esquecer de requisitar um semáforo, a exclusão mútua sobre um recurso pode ser violada.

Em 1972, os cientistas Per Brinch Hansen e Charles Hoare definiram o conceito de *monitor* [Lampson and Redell, 1980]. Um monitor é uma estrutura de sincronização que requisita e libera a seção crítica associada a um recurso de forma transparente, sem que o programador tenha de se preocupar com isso. Um monitor consiste de:

- um recurso compartilhado, visto como um conjunto de variáveis internas ao monitor.
- um conjunto de procedimentos que permitem o acesso a essas variáveis;
- um *mutex* ou semáforo para controle de exclusão mútua; cada procedimento de acesso ao recurso deve obter o semáforo antes de iniciar e liberar o semáforo ao concluir;
- um invariante sobre o estado interno do recurso.

O pseudocódigo a seguir define um monitor para operações sobre uma conta bancária (observe sua semelhança com a definição de uma classe em programação orientada a objetos):

```
1  monitor conta
2  {
3     float saldo = 0.0 ;
4  
5     void depositar (float valor)
6     {
7        if (valor >= 0)
8           conta->saldo += valor ;
9        else
10          error ("erro: valor negativo\n") ;
11    }
12 
13    void retirar (float saldo)
14    {
15       if (valor >= 0)
16          conta->saldo -= valor ;
17       else
18          error ("erro: valor negativo\n") ;
19    }
20 }
```

A definição formal de monitor prevê e existência de um *invariante*, ou seja, uma condição sobre as variáveis internas do monitor que deve ser sempre verdadeira. No caso da conta bancária, esse invariante poderia ser o seguinte: *"O saldo atual deve ser a soma de todos os depósitos efetuados e todas as retiradas efetuadas (com sinal negativo)"*. Entretanto, a maioria das implementações de monitor não suporta a definição de invariantes, com exceção da linguagem Eiffel.

De certa forma, um monitor pode ser visto como um objeto que encapsula o recurso compartilhado, com procedimentos (métodos) para acessá-lo. No entanto, a execução dos procedimentos é feita com exclusão mútua entre eles. As operações de obtenção e liberação do semáforo são inseridas automaticamente pelo compilador do programa em todos os pontos de entrada e saída do monitor (no início e final de cada procedimento), liberando o programador dessa tarefa e assim evitando erros. Monitores estão presentes em várias linguagens, como Ada, C#, Eiffel, Java e Modula-3. O código a seguir mostra um exemplo simplificado de uso de monitor em Java:

```
1  class Conta
2  {
3     private float saldo = 0;
4
5     public synchronized void depositar (float valor)
6     {
7        if (valor >= 0)
8           saldo += valor ;
9        else
10          System.err.println("valor negativo");
11    }
12
13    public synchronized void retirar (float valor)
14    {
15       if (valor >= 0)
16          saldo -= valor ;
17       else
18          System.err.println("valor negativo");
19    }
20 }
```

Em Java, a cláusula synchronized faz com que um semáforo seja associado aos métodos indicados, para cada objeto (ou para cada classe, se forem métodos de classe). No exemplo anterior, apenas um depósito ou retirada de cada vez poderá ser feito sobre cada objeto da classe Conta.

Variáveis de condição podem ser usadas no interior de monitores (na verdade, os dois conceitos nasceram juntos). Todavia, devido às restrições da semântica Mesa, um procedimento que executa a operação *notify* em uma variável de condição deve concluir e sair imediatamente do monitor, para garantir que o invariante associado ao estado interno do monitor seja respeitado [Birrell, 2004].

9 Problemas clássicos de coordenação

Algumas situações de coordenação entre atividades ocorrem com muita frequência na programação de sistemas complexos. Os *problemas clássicos de coordenação* retratam muitas dessas situações e permitem compreender como podem ser implementadas suas soluções. Nesta seção serão estudados três problemas clássicos: o problema dos *produtores/consumidores*, o problema dos *leitores/escritores* e o *jantar dos filósofos*. Diversos outros problemas clássicos são frequentemente descritos na literatura, como o *problema dos fumantes* e o do *barbeiro dorminhoco*, entre outros [Raynal, 1986, Ben-Ari, 1990]. Uma extensa coletânea de problemas de coordenação (e suas soluções) é apresentada em [Downey, 2008] (disponível *online*).

9.1 O problema dos produtores/consumidores

Este problema também é conhecido como o problema do *buffer limitado*, e consiste em coordenar o acesso de tarefas (processos ou threads) a um buffer compartilhado com capacidade de armazenamento limitada a N itens (que podem ser inteiros, registros,

mensagens, etc.). São considerados dois tipos de processos com comportamentos simétricos:

Produtor : periodicamente produz e deposita um item no buffer, caso o mesmo tenha uma vaga livre. Caso contrário, deve esperar até que surja uma vaga no buffer. Ao depositar um item, o produtor "consome" uma vaga livre.

Consumidor : continuamente retira um item do buffer e o consome; caso o buffer esteja vazio, aguarda que novos itens sejam depositados pelos produtores. Ao consumir um item, o consumidor "produz" uma vaga livre.

Deve-se observar que o acesso ao buffer é bloqueante, ou seja, cada processo fica bloqueado até conseguir fazer seu acesso, seja para produzir ou para consumir um item. A Figura 5 ilustra esse problema, envolvendo vários produtores e consumidores acessando um buffer com capacidade para 12 entradas. É interessante observar a forte similaridade dessa figura com a Figura **??**; na prática, a implementação de *mailboxes* e de *pipes* é geralmente feita usando um esquema de sincronização produtor/consumidor.

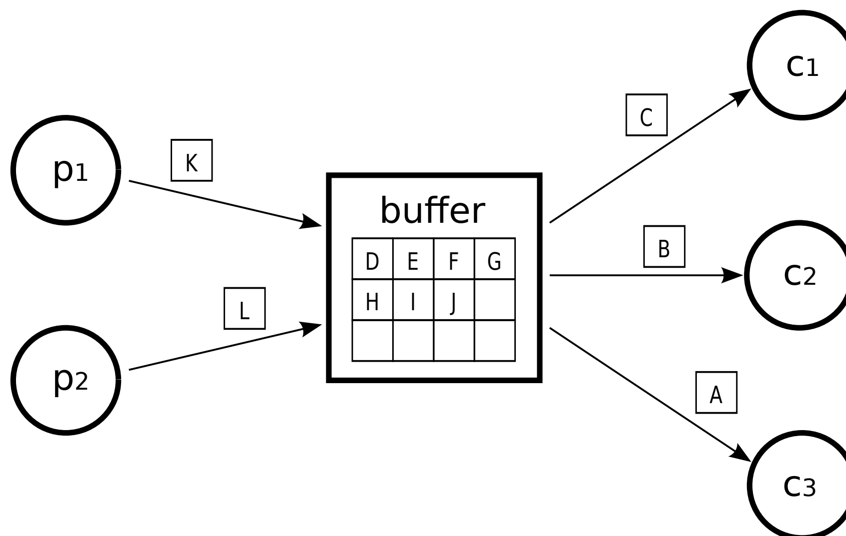

Figura 5: O problema dos produtores/consumidores.

A solução do problema dos produtores/consumidores envolve três aspectos de coordenação distintos:

- A exclusão mútua no acesso ao buffer, para evitar condições de disputa entre produtores e/ou consumidores que poderiam corromper seu conteúdo.
- O bloqueio dos produtores no caso do buffer estar cheio: os produtores devem aguardar até surjam vagas livres no buffer.
- O bloqueio dos consumidores no caso do buffer estar vazio: os consumidores devem aguardar até surjam novos itens a consumir no buffer.

A solução para esse problema exige três semáforos, um para atender cada aspecto de coordenação acima descrito. O código a seguir ilustra de forma simplificada uma solução para esse problema, considerando um buffer com capacidade para N itens, inicialmente vazio:

```
sem_t mutex ; // controla o acesso ao buffer (inicia em 1)
sem_t item ;  // número de itens no buffer (inicia em 0)
sem_t vaga ;  // número de vagas livres no buffer (inicia em N)

produtor () {
   while (1) {
      ...                     // produz um item
      sem_down(&vaga) ;       // aguarda uma vaga no buffer
      sem_down(&mutex) ;      // aguarda acesso exclusivo ao buffer
      ...                     // deposita o item no buffer
      sem_up(&mutex) ;        // libera o acesso ao buffer
      sem_up(&item) ;         // indica a presença de um novo item no buffer
   }
}

consumidor () {
   while (1) {
      sem_down(&item) ;       // aguarda um novo item no buffer
      sem_down(&mutex) ;      // aguarda acesso exclusivo ao buffer
      ...                     // retira o item do buffer
      sem_up(&mutex) ;        // libera o acesso ao buffer
      sem_up(&vaga) ;         // indica a liberação de uma vaga no buffer
      ...                     // consome o item retirado do buffer
   }
}
```

É importante observar que essa solução é genérica, pois não depende do tamanho do buffer, do número de produtores ou do número de consumidores.

9.2 O problema dos leitores/escritores

Outra situação que ocorre com frequência em sistemas concorrentes é o problema dos *leitores/escritores*. Neste caso, um conjunto de processos ou threads acessam de forma concorrente uma área de memória comum (compartilhada), na qual podem fazer leituras ou escritas de valores. As leituras podem ser feitas simultaneamente, pois não interferem umas com as outras, mas as escritas têm de ser feitas com acesso exclusivo à área compartilhada, para evitar condições de disputa. No exemplo da Figura 6, os leitores e escritores acessam de forma concorrente uma matriz de inteiros M.

Uma solução trivial para esse problema consistiria em proteger o acesso à área compartilhada com um semáforo inicializado em 1; assim, somente um processo por vez poderia acessar a área, garantindo a integridade de todas as operações. O código a seguir ilustra essa abordagem simplista:

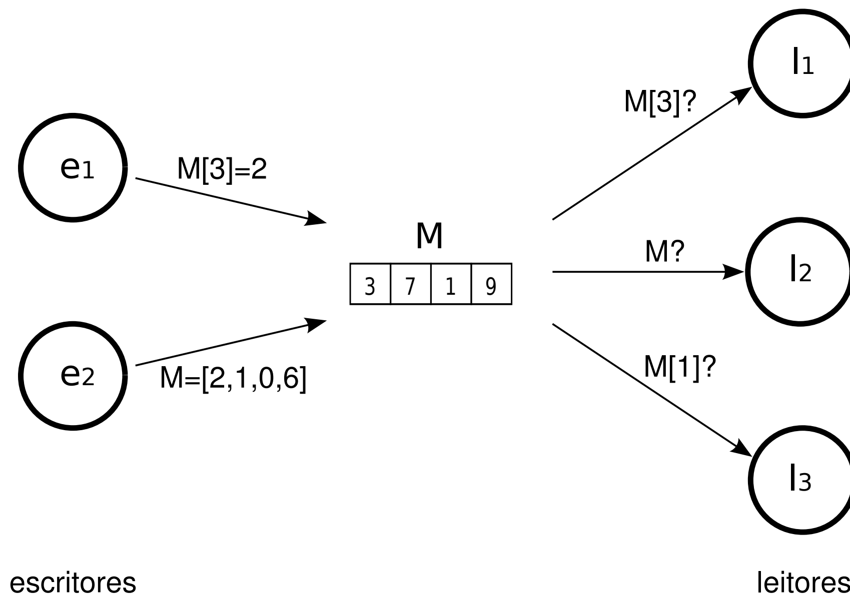

Figura 6: O problema dos leitores/escritores.

```
sem_t mutex_area ;              // controla o acesso à área (inicia em 1)

leitor () {
   while (1) {
      sem_down (&mutex_area) ;      // requer acesso exclusivo à área
      ...                           // lê dados da área compartilhada
      sem_up (&mutex_area) ;        // libera o acesso à área
      ...
   }
}

escritor () {
   while (1) {
      sem_down (&mutex_area) ;      // requer acesso exclusivo à área
      ...                           // escreve dados na área compartilhada
      sem_up (&mutex_area) ;        // libera o acesso à área
      ...
   }
}
```

Todavia, essa solução deixa a desejar em termos de desempenho, porque restringe desnecessariamente o acesso dos leitores à área compartilhada: como a operação de leitura não altera os valores armazenados, não haveria problema em permitir o acesso simultâneo de vários leitores à área compartilhada, desde que as escritas continuem sendo feitas de forma exclusiva. Uma nova solução para o problema, considerando a possibilidade de acesso simultâneo pelos leitores, seria a seguinte:

```
sem_t mutex_area ;           // controla o acesso à área (inicia em 1)
int conta_leitores = 0 ;     // número de leitores acessando a área
sem_t mutex_conta ;          // controla o acesso ao contador (inicia em 1)

leitor () {
  while (1) {
    sem_down (&mutex_conta) ;    // requer acesso exclusivo ao contador
    conta_leitores++ ;           // incrementa contador de leitores
    if (conta_leitores == 1)     // sou o primeiro leitor a entrar?
      sem_down (&mutex_area) ;   // requer acesso à área
    sem_up (&mutex_conta) ;      // libera o contador

    ...                          // lê dados da área compartilhada

    sem_down (&mutex_conta) ;    // requer acesso exclusivo ao contador
    conta_leitores-- ;           // decrementa contador de leitores
    if (conta_leitores == 0)     // sou o último leitor a sair?
      sem_up (&mutex_area) ;     // libera o acesso à área
    sem_up (&mutex_conta) ;      // libera o contador
    ...
  }
}

escritor () {
  while (1) {
    sem_down(&mutex_area) ;      // requer acesso exclusivo à área
    ...                          // escreve dados na área compartilhada
    sem_up(&mutex_area) ;        // libera o acesso à área
    ...
  }
}
```

Essa solução melhora o desempenho das operações de leitura, mas introduz um novo problema: a *priorização dos leitores*. De fato, sempre que algum leitor estiver acessando a área compartilhada, outros leitores também podem acessá-la, enquanto eventuais escritores têm de esperar até a área ficar livre (sem leitores). Caso existam muito leitores em atividade, os escritores podem ficar impedidos de acessar a área, pois ela nunca ficará vazia. Soluções com priorização para os escritores e soluções equitativas entre ambos podem ser facilmente encontradas na literatura [Raynal, 1986, Ben-Ari, 1990].

O relacionamento de sincronização *leitor/escritor* é encontrado com muita frequência em aplicações com múltiplas threads. O padrão POSIX define mecanismos para a criação e uso de travas com essa funcionalidade (com priorização de escritores), acessíveis através de chamadas como `pthread_rwlock_init`, entre outras.

9.3 O jantar dos filósofos

Um dos problemas clássicos de coordenação mais conhecidos é o *jantar dos filósofos*, que foi inicialmente proposto por Dijkstra [Raynal, 1986, Ben-Ari, 1990]. Neste problema, um grupo de cinco filósofos chineses alterna suas vidas entre meditar e comer. Na mesa há um lugar fixo para cada filósofo, com um prato, cinco palitos (*hashis* ou *chopsticks*)

compartilhados e um grande prato de comida ao meio (na versão inicial de Dijkstra, os filósofos compartilhavam garfos e comiam spaguetti). A Figura 7 ilustra essa situação.

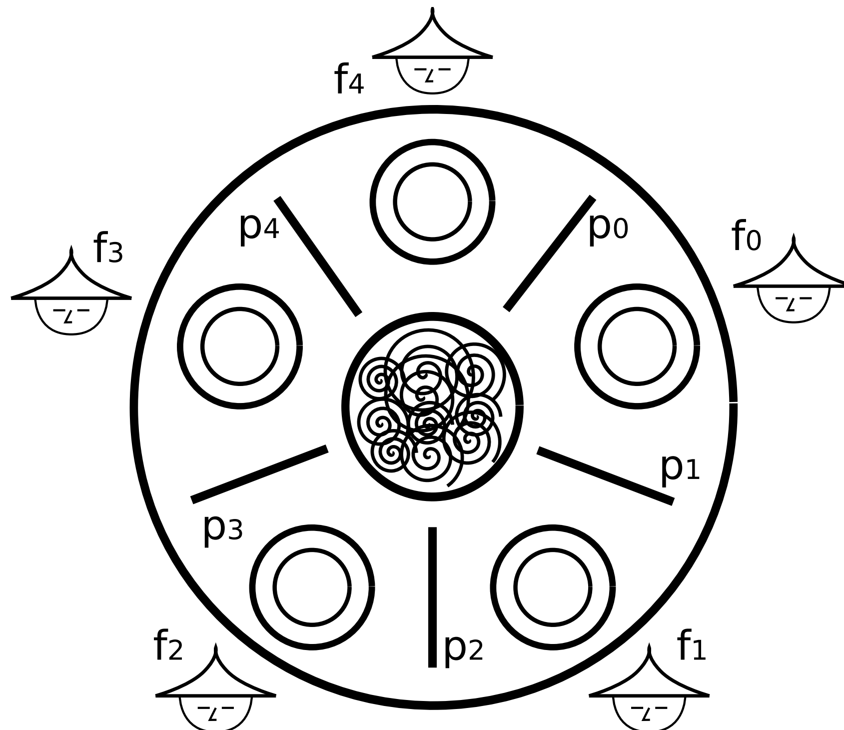

Figura 7: O jantar dos filósofos chineses.

Para comer, um filósofo f_i precisa pegar os palitos à sua direita (p_i) e à sua esquerda (p_{i+1}), um de cada vez. Como os palitos são compartilhados, dois filósofos vizinhos nunca podem comer ao mesmo tempo. Os filósofos não conversam entre si nem podem observar os estados uns dos outros. O problema do jantar dos filósofos é representativo de uma grande classe de problemas de sincronização entre vários processos e vários recursos sem usar um coordenador central. A listagem a seguir representa uma implementação do comportamento básico dos filósofos, na qual cada palito é representado por um semáforo:

```
1  #define NUMFILO 5
2  sem_t hashi [NUMFILO] ; // um semáforo para cada palito (iniciam em 1)
3
4  filosofo (int i) {
5     while (1) {
6        medita () ;
7        sem_down (&hashi [i]) ;                  // obtem palito i
8        sem_down (&hashi [(i+1) % NUMFILO]) ;    // obtem palito i+1
9        come () ;
10       sem_up (&hashi [i]) ;                    // devolve palito i
11       sem_up (&hashi [(i+1) % NUMFILO]) ;      // devolve palito i+1
12    }
13 }
```

Resolver o problema do jantar dos filósofos consiste em encontrar uma forma de coordenar suas atividades de maneira que todos os filósofos consigam meditar e comer. As soluções mais simples para esse problema podem provocar impasses, nos quais todos os filósofos ficam bloqueados (impasses serão estudados na Seção 10). Outras soluções podem provocar inanição (*starvation*), ou seja, alguns dos filósofos nunca conseguem comer. A Figura 8 apresenta os filósofos em uma situação de impasse: cada filósofo obteve o palito à sua direita e está aguardando o palito à sua esquerda (indicado pelas setas tracejadas). Como todos os filósofos estão aguardando, ninguém mais consegue executar.

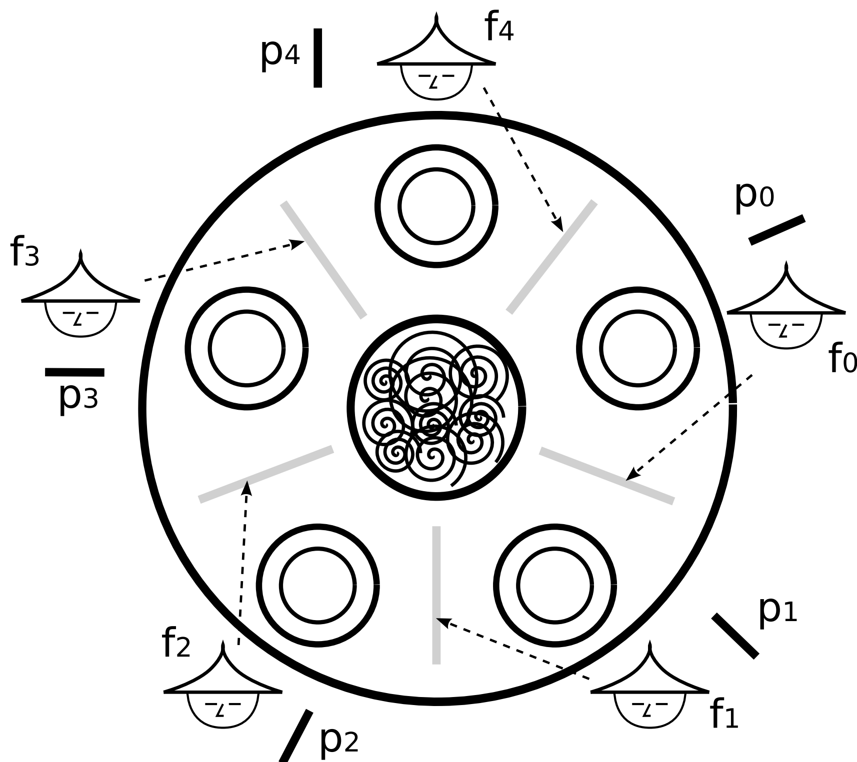

Figura 8: Um impasse no jantar dos filósofos chineses.

Uma solução trivial para o problema do jantar dos filósofos consiste em colocar um "saleiro" hipotético sobre a mesa: quando um filósofo deseja comer, ele deve pegar o saleiro antes de obter os palitos; assim que tiver ambos os palitos, ele devolve o saleiro à mesa e pode comer. Obviamente, essa solução serializa o acesso aos palitos e por isso tem baixo desempenho. Imagine como essa solução funcionaria em uma situação com 1000 filósofos e 1000 palitos? Diversas outras soluções podem ser encontradas na literatura [Tanenbaum, 2003, Silberschatz et al., 2001].

10 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 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. Para ilustrar uma situação de impasse, será utilizado o exemplo de acesso a uma conta bancária apresentado na Seção 2. O código a seguir implementa uma operação de transferência de fundos entre duas contas bancárias. A cada conta está associado um semáforo, 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     sem_t lock ;                  // semáforo 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    sem_down (contaDeb->lock) ;     // obtém acesso a contaDeb
11    sem_down (contaCred->lock) ;    // 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    sem_up (contaDeb->lock) ;       // libera acesso a contaDeb
19    sem_up (contaCred->lock) ;      // 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 9.

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

Impasses são situações muito frequentes em programas concorrentes, mas também podem ocorrer em sistemas distribuídos. 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.

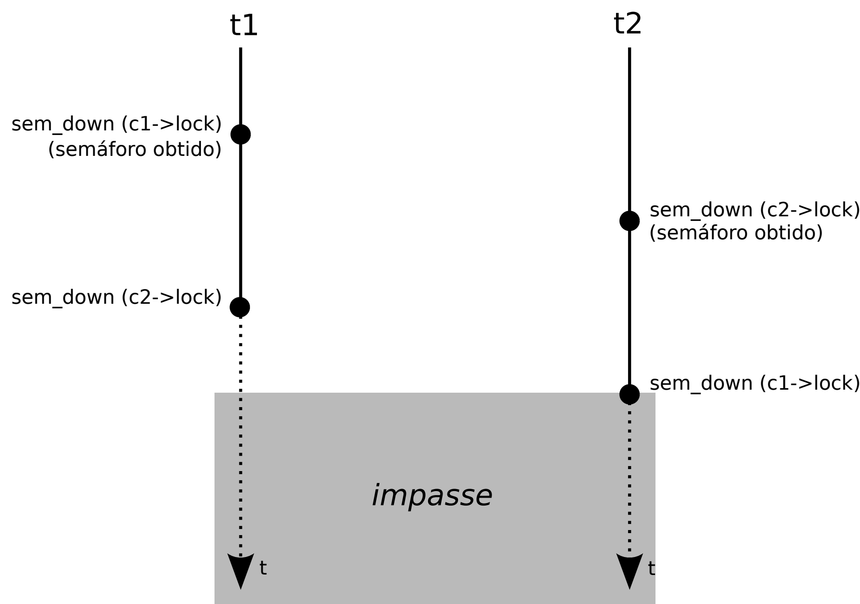

Figura 9: Impasse entre duas transferências.

10.1 Caracterização de 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 requisitá-los 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]:

C1 – 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.

C2 – 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.

C3 – Não-preempção : uma tarefa somente libera os recursos que detém quando assim o decidir, e não pode perdê-los contra a sua vontade (ou seja, o sistema operacional não retira os recursos já alocados às tarefas). No exemplo da conta corrente, cada tarefa detém indefinidamente os semáforos que já obteve.

C4 – 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 existirão 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.

10.2 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 □ → ◯, enquanto a requisição de um recurso por uma tarefa é indicada por ◯ → □.

A Figura 10 apresenta o grafo de alocação de recursos da situação de impasse ocorrida na transferência de valores entre contas bancárias da Figura 9. Nessa figura percebe-se claramente a dependência cíclica entre tarefas e recursos $t_1 \rightarrow c_2 \rightarrow t_2 \rightarrow c_1 \rightarrow t_1$, 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.

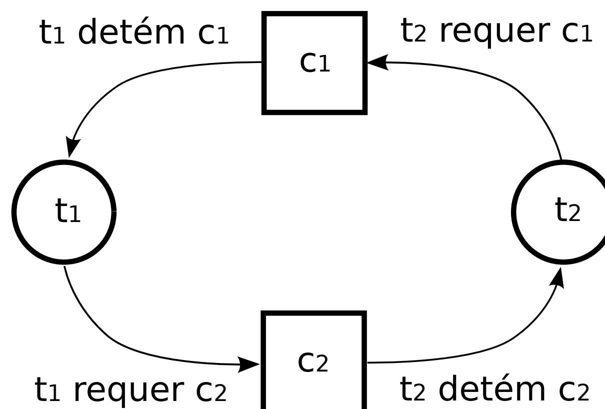

Figura 10: Grafo de alocação de recursos com impasse.

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 11 indica apresenta um grafo de alocação de recursos considerando alguns recursos com múltiplas instâncias.

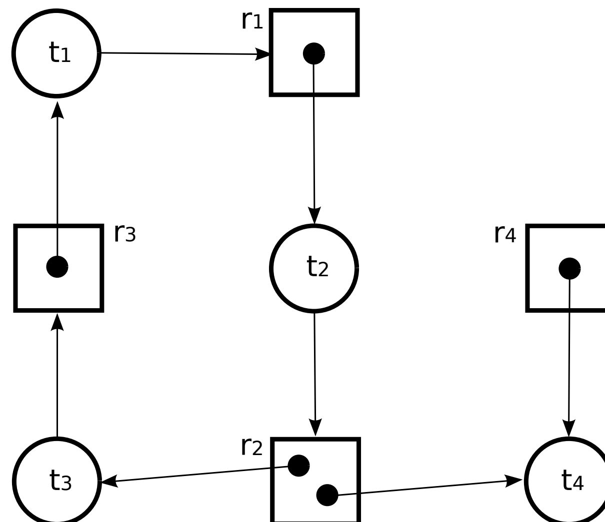

Figura 11: Grafo de alocação com múltiplas instâncias de recursos.

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 $t_1 \rightarrow r_1 \rightarrow t_2 \rightarrow r_2 \rightarrow t_3 \rightarrow r_3 \rightarrow t_1$ presente no diagrama da Figura 11 não representa um impasse, porque a qualquer momento a tarefa t_4 pode liberar uma instância do recurso r_2, solicitado por t_2, desfazendo assim o ciclo. Um algoritmo de detecção de impasses envolvendo recursos com múltiplas instâncias é apresentado em [Tanenbaum, 2003].

10.3 Técnicas de tratamento de impasses

Como os impasses paralisam tarefas que detêm recursos, sua ocorrência pode incorrer em 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, Solaris, 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.

10.3.1 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 10.1, nunca sejam 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 esse recurso. A técnica de *spooling* previne impasses envolvendo as impressoras, mas não é facilmente aplicável a outros 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 9, 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, e devolvê-lo mais tarde, 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 contexto do processador. No entanto, é de difícil aplicação sobre recursos como arquivos ou áreas de memória compartilhada, porque a preempção viola a exclusão mútua e pode deixar 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 9, o número de conta bancária poderia definir uma ordem global. 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.

10.3.2 Impedimento de impasses

Uma 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 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 **inseguro**. As técnicas de impedimento de impasses devem portanto manter o sistema sempre em um estado seguro, evitando entrar em estados inseguros.

A Figura 12 ilustra o grafo de estados do sistema de transferência de valores com duas tarefas discutido anteriormente. Nesse grafo, cada estado é a combinação dos estados individuais das duas tarefas. Pode-se observar no grafo que o estado E_{13} 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_{12}, E_{14} e E_{15} são considerados estados inseguros, pois levam invariavelmente na direção do impasse. Os demais estados são considerados seguros, pois a partir de cada um deles é possível continuar a execução e retornar ao estado inicial E_0. Obviamente, transições que levem de um estado seguro a um inseguro devem ser evitadas, como $E_9 \rightarrow E_{14}$ ou $E_{10} \rightarrow E_{12}$.

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 poderem operar. Normalmente é necessário conhecer com antecedência que recursos serão acessados por cada tarefa, quantas

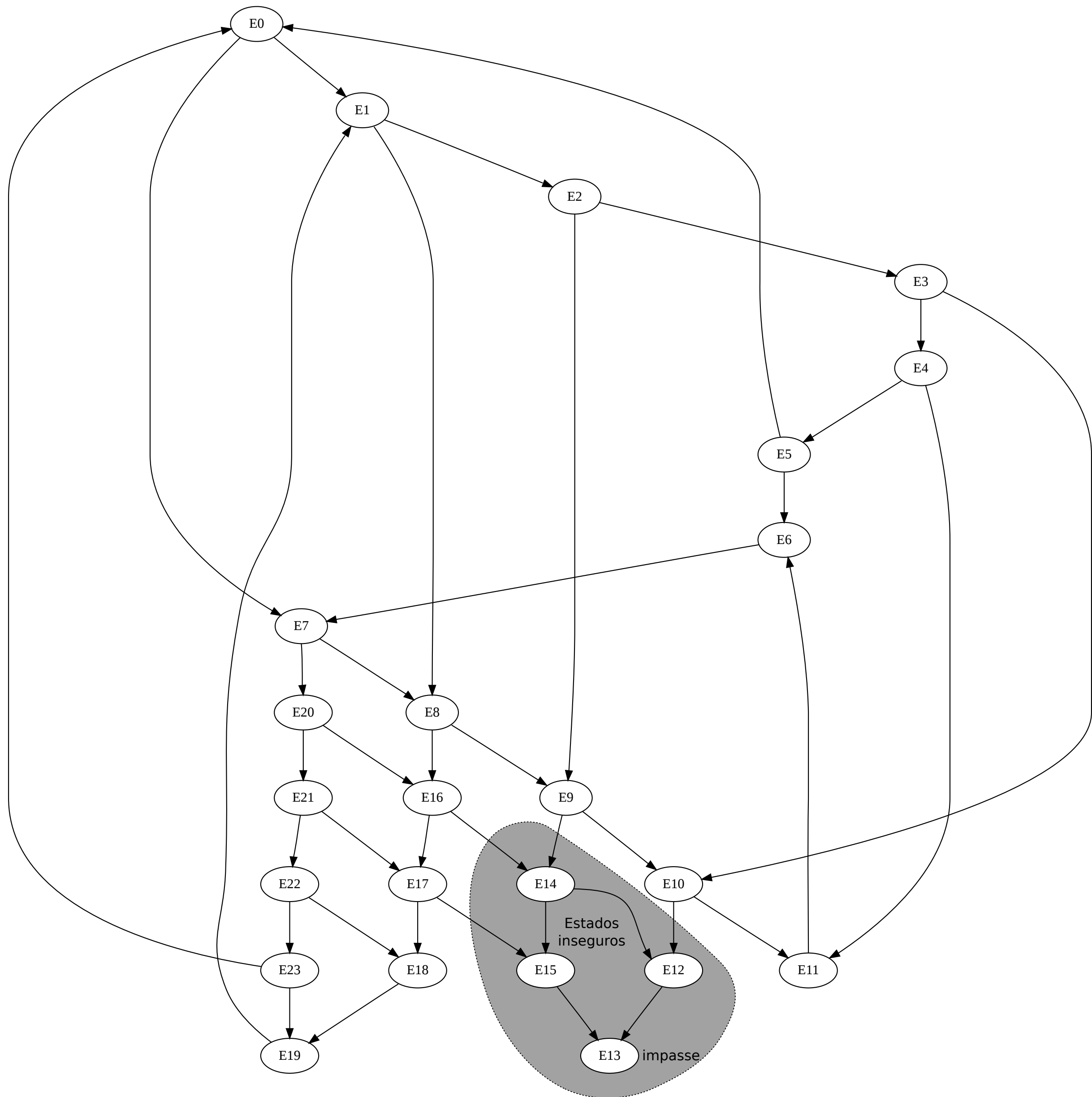

Figura 12: Grafo de estados do sistema de transferências com duas tarefas.

instâncias de cada um serão necessárias e qual a ordem de acesso aos recursos. Por essa razão, são pouco utilizadas na prática.

10.3.3 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 o detecta, determina quais as tarefas e recursos envolvidos e toma 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 10.2), 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.

Uma vez detectado um impasse e identificadas as tarefas e recursos envolvidos, o sistema deve proceder à sua resolução, 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, etc.

Retroceder tarefas : uma ou mais tarefas envolvidas no impasse têm sua execução parcialmente desfeita, 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[6]. 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?

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.

Referências

[Barney, 2005] Barney, B. (2005). POSIX threads programming. http://www.llnl.gov/computing/tutorials/pthreads.

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

[Birrell, 2004] Birrell, A. (2004). Implementing condition variables with semaphores. *Computer Systems Theory, Technology, and Applications*, pages 29–37.

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

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

[Downey, 2008] Downey, A. (2008). *The Little Book of Semaphores*. Green Tea Press.

[Gallmeister, 1994] Gallmeister, B. (1994). *POSIX.4: Programming for the Real World*. O'Reilly Media, Inc.

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

[Lamport, 1974] Lamport, L. (1974). A new solution of Dijkstra's concurrent programming problem. *Communications of the ACM*, 17(8):453–455.

[Lampson and Redell, 1980] Lampson, B. and Redell, D. (1980). Experience with processes and monitors in Mesa. *Communications of the ACM*.

[Raynal, 1986] Raynal, M. (1986). *Algorithms for Mutual Exclusion*. The MIT Press.

[Silberschatz et al., 2001] Silberschatz, A., Galvin, P., and Gagne, G. (2001). *Sistemas Operacionais – Conceitos e Aplicações*. Campus.

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