# Capítulo 10

# Coordenação entre tarefas

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 a aplicação possam cooperar, elas precisam comunicar informações umas às outras e coordenar suas atividades, para garantir que os resultados obtidos sejam coerentes. Neste capítulo serão estudados os problemas que podem ocorrer quando duas ou mais tarefas acessam os mesmos recursos de forma concorrente; também serão apresentadas algumas técnicas usadas para coordenar os acessos das tarefas aos recursos compartilhados.

## 10.1 O problema da concorrência

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.

### 10.1.1 Uma aplicação concorrente

O código apresentado a seguir implementa de forma simplificada a operação de depósito de um valor em um saldo de conta bancária informado como parâmetro. Para facilitar a compreensão do código de máquina apresentado na sequência, todos os valores manipulados são inteiros.

```
1 void depositar (long * saldo, long valor)
2 {
3    (*saldo) += valor ;
4 }
```

Após a compilação em uma plataforma *Intel* 64 bits, a função `depositar` assume a seguinte forma em *Assembly*:

```
0000000000000000 <depositar>:
       ; inicializa a função
       push  %rbp
       mov   %rsp,%rbp
       mov   %rdi,-0x8(%rbp)
       mov   %esi,-0xc(%rbp)

       ; carrega o conteúdo da memória apontada por "saldo" em EDX
       mov   -0x8(%rbp),%rax       ; saldo → rax (endereço do saldo)
       mov   (%rax),%edx           ; mem[rax] → edx

       ; carrega o conteúdo de "valor" no registrador EAX
       mov   -0xc(%rbp),%eax       ; valor → eax

       ; soma EAX ao valor em EDX
       add   %eax,%edx             ; eax + edx → edx

       ; escreve o resultado em EDX na memória apontada por "saldo"
       mov   -0x8(%rbp),%rax       ; saldo → rax
       mov   %edx,(%rax)           ; edx → mem[rax]

       ; finaliza a função
       nop
       pop   %rbp
       retq
```

Consideremos que a função `depositar` faz parte de um sistema mais amplo de gestão de contas em um banco, que pode ser acessado simultaneamente por centenas ou milhares de usuários em agências e terminais distintos. Caso dois clientes em terminais diferentes tentem depositar valores na mesma conta ao mesmo tempo, existirão duas tarefas $t_1$ e $t_2$ acessando os dados da conta de forma concorrente. A Figura 10.1 ilustra esse cenário.

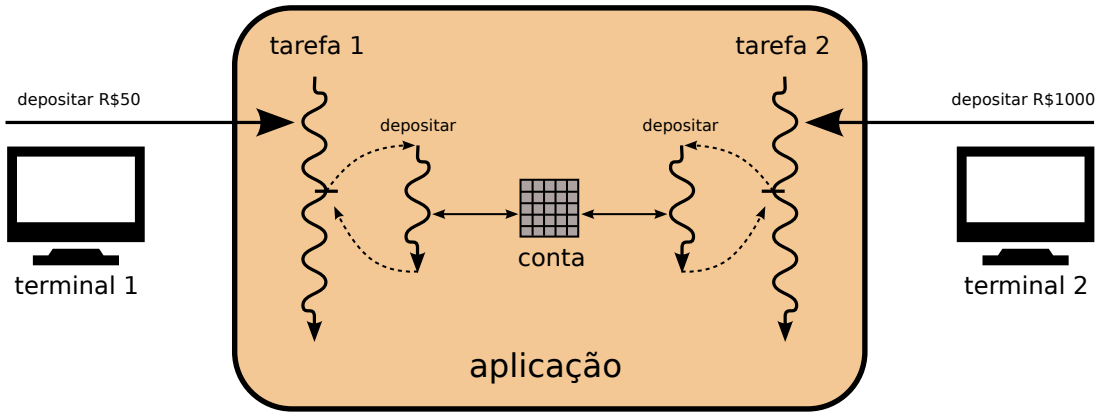


Figura 10.1: Acessos concorrentes a variáveis compartilhadas.

### 10.1.2 Condições de disputa

O comportamento dinâmico da aplicação da Figura 10.1 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

10.2. Em ambas as execuções o saldo inicial da conta passou de R$ 0,00 para R$ 1.050,00, conforme o esperado.

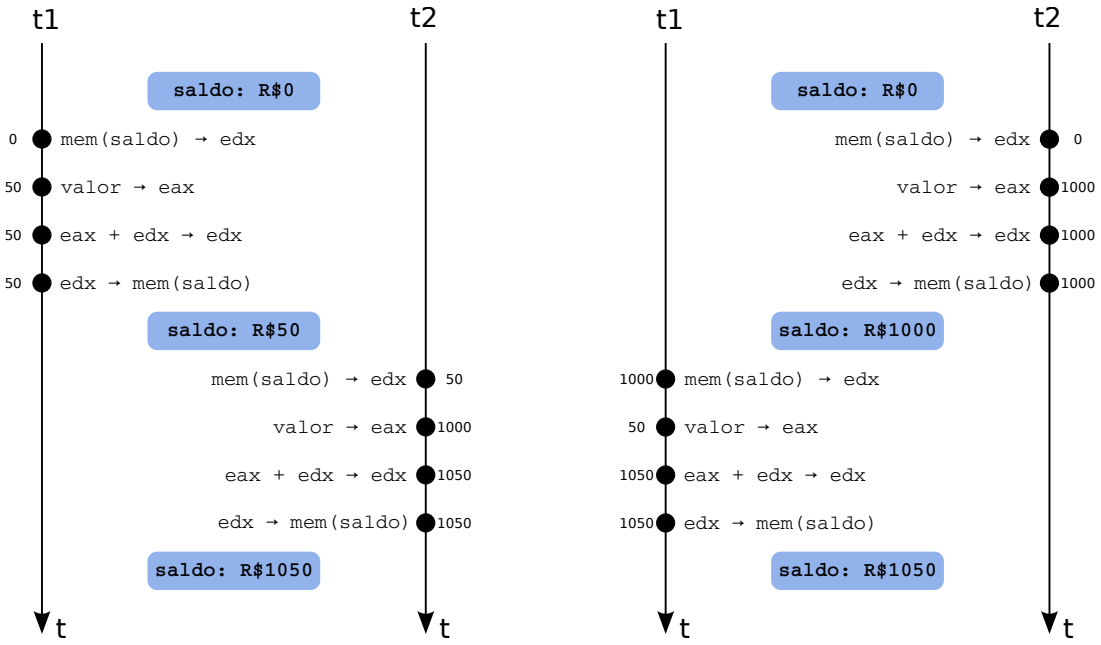


Figura 10.2: Operações de depósitos não-concorrentes.

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 da função `depositar`. Em sistemas multiprocessados a situação é mais complexa, pois cada tarefa poderá estar executando em um processador distinto.

Os diagramas de tempo apresentados na Figura 10.3 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. Pode-se observar que apenas é concretizado o depósito da tarefa que realizou a escrita do resultado na memória por último (operação `edx` → `mem(saldo)`)[1].

Os erros e inconsistências gerados por acessos concorrentes a dados compartilhados, como os ilustrados na Figura 10.3, são denominados **condições de disputa**, ou condições de corrida (do inglês *race conditions*). Condições de disputa podem ocorrer em sistemas onde várias tarefas acessam de forma concorrente recursos compartilhados (variáveis, áreas de memória, arquivos abertos, etc.), sob certas condições.

É importante observar que condições de disputa são erros *dinâmicos*, ou seja, erros que não aparecem no código fonte e que só se manifestam durante a execução. Assim, são dificilmente detectáveis através da simples 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

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

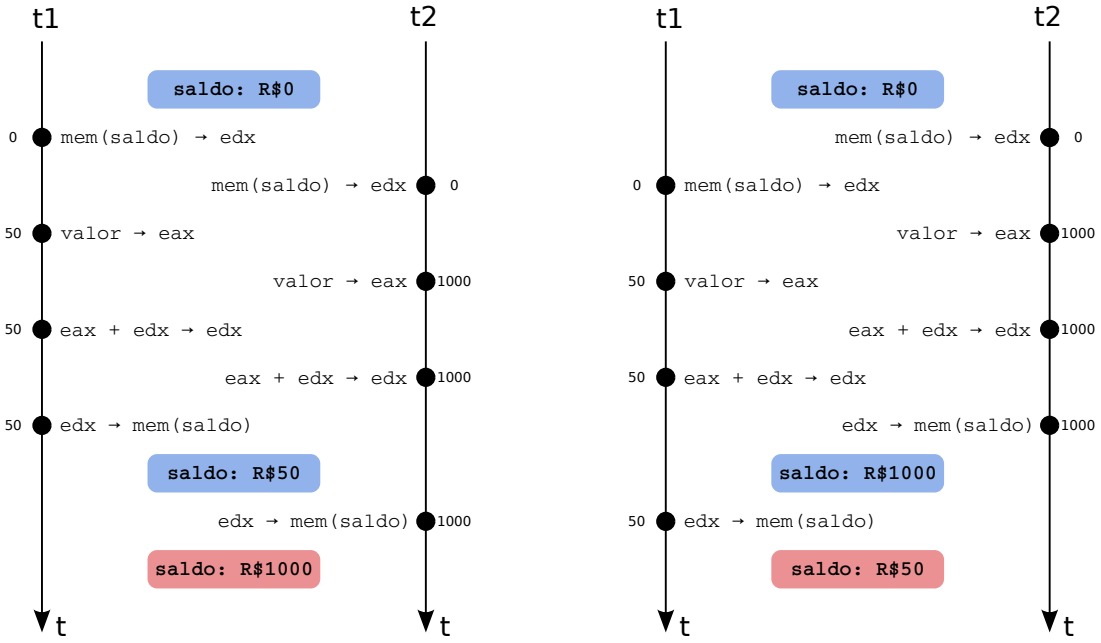


Figura 10.3: Operações de depósito concorrentes.

e ser difícil de reproduzir durante a depuração. Por isso, é importante conhecer técnicas que previnam a ocorrência de condições de disputa.

### 10.1.3 Condições de Bernstein

Condições de disputa entre tarefas paralelas podem ser formalizadas através das chamadas **condições de Bernstein** [Bernstein, 1966], assim definidas: dadas duas tarefas $t_1$ e $t_2$, sendo $\mathcal{R}(t_i)$ o conjunto de variáveis lidas por $t_i$ e $\mathcal{W}(t_i)$ o conjunto de variáveis escritas por $t_i$, essas tarefas podem executar em paralelo sem risco de condição de disputa ($t_1 \parallel t_2$) se e somente se as seguintes condições forem atendidas:

$$t_1 \parallel t_2 \iff \begin{cases} \mathcal{R}(t_1) \cap \mathcal{W}(t_2) = \varnothing & (t_1 \text{ não lê as variáveis escritas por } t_2) \\ \mathcal{R}(t_2) \cap \mathcal{W}(t_1) = \varnothing & (t_2 \text{ não lê as variáveis escritas por } t_1) \\ \mathcal{W}(t_1) \cap \mathcal{W}(t_2) = \varnothing & (t_1 \text{ e } t_2 \text{ não escrevem nas mesmas variáveis}) \end{cases}$$

Percebe-se claramente que as condições de Bernstein não são respeitadas na aplicação bancária usada como exemplo neste texto, pois ambas as tarefas podem ler e escrever simultaneamente na mesma variável (o saldo). Por isso, elas não devem ser executadas em paralelo.

Outro ponto importante evidenciado pelas condições de Bernstein é que as condições de disputa somente ocorrem se pelo menos uma das operações envolvidas for de escrita; acessos de leitura concorrentes às mesmas variáveis respeitam as condições de Bernstein e portanto não geram condições de disputa entre si.

### 10.1.4 Seções críticas

Na seção anterior vimos que tarefas acessando dados compartilhados de forma concorrente podem ocasionar condições de disputa. Os trechos de código que acessam dados compartilhados em cada tarefa são denominados **seções críticas** (ou *regiões*

*críticas*). No caso da Figura 10.1, as seções críticas das tarefas $t_1$ e $t_2$ são idênticas e resumidas à seguinte linha de código:

```
1 (*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: dado um conjunto de regiões críticas relacionadas, apenas uma tarefa pode estar em sua seção crítica a cada instante, excluindo o acesso das demais às suas respectivas regiões críticas. Essa propriedade é conhecida como **exclusão mútua**.

## 10.2 Exclusão mútua

Diversos mecanismos podem ser definidos para garantir a exclusão mútua, impedindo o entrelaçamento de seções críticas. Todos eles exigem que o programador defina os limites (início e o final) de cada seção crítica. Dada uma seção crítica $cs_i$ podem ser definidas as primitivas *enter*($cs_i$), para que uma tarefa indique sua intenção de entrar na seção crítica $cs_i$, e *leave*($cs_i$), para que uma tarefa que está na seção crítica $cs_i$ informe que está saindo da mesma. 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 $cs_i$ através da primitiva *leave*($cs_i$).

Usando as primitivas *enter*() e *leave*(), o código da operação de depósito visto na Seção 10.1 pode ser reescrito como segue:

```
1 void depositar (long conta, long *saldo, long valor)
2 {
3    enter (conta) ;          // entra na seção crítica "conta"
4    (*saldo) += valor ;      // usa as variáveis compartilhadas
5    leave (conta) ;          // sai da seção crítica
6 }
```

Nesta seção serão apresentadas algumas soluções para a implementação das primitivas de exclusão mútua. As soluções propostas devem atender a alguns critérios básicos 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, ou seja, não pode haver inanição.

**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, que no momento não estejam interessadas em entrar na região crítica, não podem influenciar 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.

### 10.2.1 Inibição de interrupções

Uma solução simples para a implementação da exclusão mútua consiste em impedir as trocas de contexto dentro da seção crítica. Ao entrar em uma seção crítica, a tarefa desativa 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. Assim, uma tarefa mal intencionada poderia desativar as interrupções 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.

### 10.2.2 A solução trivial

Uma solução trivial para o problema da seção crítica consiste em usar uma variável *busy* para indicar se a seção crítica 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 ()
4  {
5     while (busy) {} ;    // espera enquanto a seção estiver ocupada
6     busy = 1 ;           // marca a seção como ocupada
7  }
8
9  void leave ()
10 {
11    busy = 0 ;           // libera a seção (marca como livre)
12 }
```

Infelizmente, essa solução 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 (conforme demonstra o diagrama de tempo da Figura 10.4). Em outras palavras, as linhas 5 e 6 da implementação formam uma seção crítica que também deve ser protegida.

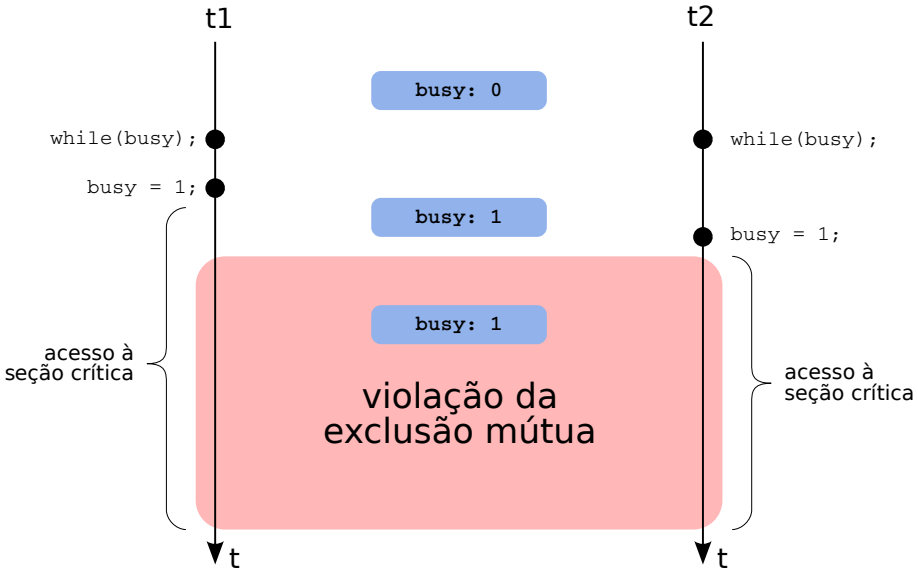


Figura 10.4: Condição de disputa no acesso à variável *busy*.

Outro problema importante com essa solução ocorre no laço da linha 5 do código: o teste contínuo da variável *busy* consome muito processador. Se houverem muitas tarefas tentando entrar em uma seção crítica, muito tempo de processamento será gasto nesse teste. O teste contínuo de uma condição é denominado **espera ocupada** (*busy wait*) e deve ser evitado, por conta de sua ineficiência.

### 10.2.3 Alternância de uso

Outra solução simples para a implementação da exclusão mútua 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:

```
1  int num_tasks ;
2  int turn = 0 ;                    // inicia pela tarefa 0
3
4  void enter (int task)             // task vale 0, 1, ..., num_tasks-1
5  {
6     while (turn != task) {} ;      // a tarefa espera seu turno
7  }
8
9  void leave (int task)
10 {
11    turn = (turn + 1) % num_tasks ; // passa para a próxima tarefa
12 }
```

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.

### 10.2.4 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:

```
1  int turn = 0 ;           // indica de quem éa vez
2  int wants[2] = {0, 0} ; // indica se a tarefa i quer acessar a seção crítica
3
4  void enter (int task)   // task pode valer 0 ou 1
5  {
6     int other = 1 - task ; // indica a outra tarefa
7     wants[task] = 1 ;     // task quer acessar a seção crítica
8     turn = task ;
9     while ((turn == task) && wants[other]) {} ; // espera ocupada
10 }
11
12 void leave (int task)
13 {
14   wants[task] = 0 ;      // task libera a seção crítica
15 }
```

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 $n > 2$ tarefas podem ser encontradas na literatura [Raynal, 1986], sendo a mais conhecida delas o *algoritmo do padeiro*, proposto por Leslie Lamport [Lamport, 1974].

### 10.2.5 Operações atômicas

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 sua atribuição são feitos em momentos distintos do código, permitindo condições de disputa. 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* (indivisível, sem possibilidade de troca de contexto entre essas duas operações). A execução atômica das operações de teste e atribuição impede a ocorrência de condições de disputa sobre a variável *busy*.

Um exemplo de operação atômica simples é a instrução de máquina *Test-and-Set Lock* (TSL), que é executada atomicamente pelo processador e cujo comportamento é descrito pelo seguinte pseudocódigo:

[2] Este algoritmo **pode falhar** em arquiteturas que permitam execução fora de ordem, ou seja, onde a ordem das operações de leitura e de escrita na memória possa ser trocada pelo processador para obter mais desempenho, como é o caso dos processadores Intel x86. Nesse caso, é necessário incluir uma instrução de barreira de memória logo antes do laço `while`.

$$TSL(x) = \begin{cases} x \rightarrow old & \text{// guarda o valor de } x \\ 1 \rightarrow x & \text{// atribui 1 a } x \\ return(old) & \text{// devolve o valor anterior de } x \end{cases}$$

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

```
1  int lock = 0 ;                    // variável de trava
2
3  void enter (int *lock)            // passa o endereço da trava
4  {
5     while ( TSL (*lock) ) {} ;     // espera ocupada sobre a trava
6  }
7
8  void leave (int *lock)
9  {
10   (*lock) = 0 ;                   // libera a seção crítica
11 }
```

A instrução TSL esteve disponível apenas em processadores antigos, como o IBM System/360. Processadores modernos oferecem diversas operações atômicas com o mesmo objetivo, conhecidas coletivamente como **instruções RMW** (de *Read-Modify-Write*, Lê-Modifica-Escreve), como CAS (*Compare-And-Swap*) e XCHG (*Exchange*). A instrução XCHG, disponível nos processadores Intel e AMD, efetua a troca atômica de conteúdo (*swapping*) entre dois registradores, ou entre um registrador e uma posição de memória:

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

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

```
1  int lock ;                        // variável de trava
2
3  enter (int *lock)
4  {
5     int key = 1 ;                  // variável auxiliar (local)
6     while (key)                    // espera ocupada
7       XCHG (lock, &key) ;          // alterna valores de lock e key
8  }
9
10 leave (int *lock)
11 {
12    (*lock) = 0 ;                  // libera a seção crítica
13 }
```

Os mecanismos de exclusão mútua usando instruções atômicas são amplamente usados no interior do sistema operacional, para controlar o acesso a seções críticas dentro 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 na próxima seção.

## 10.3 Problemas

O acesso concorrente de diversas tarefas aos mesmos recursos pode provocar problemas de consistência, as chamadas *condições de disputa*. Uma forma de eliminar esses problemas é forçar o acesso a esses recursos em exclusão mútua, ou seja, uma tarefa por vez. Neste capítulo foram apresentadas algumas formas de implementar a exclusão mútua. Contudo, apesar dessas soluções garantirem a exclusão mútua (com exceção da solução trivial), elas sofrem de 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:** a não ser na solução de alternância, 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.

**Dependência:** na solução por alternância, tarefas desejando acessar a seção crítica podem ser impedidas de fazê-lo por tarefas que não têm interessa na seção crítica naquele momento.

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 e na construção de sistemas de computação dedicados, como controladores embarcados mais simples. O próximo capítulo apresentará estruturas de controle de sincronização mais sofisticadas, que resolvem os problemas indicados acima.

# Exercícios

1. Explique o que são *condições de disputa*, mostrando um exemplo real.
2. Sobre as afirmações a seguir, relativas aos mecanismos de coordenação, indique quais são incorretas, justificando sua resposta:
   (a) A estratégia de inibir interrupções para evitar condições de disputa funciona em sistemas multi-processados.
   (b) Os mecanismos de controle de entrada nas regiões críticas provêem exclusão mútua no acesso às mesmas.
   (c) Os algoritmos de *busy-wait* se baseiam no teste contínuo de uma condição.
   (d) Condições de disputa ocorrem devido às diferenças de velocidade na execução dos processos.
   (e) Condições de disputa ocorrem quando dois processos tentam executar o mesmo código ao mesmo tempo.

(f) Instruções do tipo *Test&Set Lock* devem ser implementadas pelo núcleo do SO.

(g) O algoritmo de Peterson garante justiça no acesso à região crítica.

(h) Os algoritmos com estratégia *busy-wait* otimizam o uso da CPU do sistema.

(i) Uma forma eficiente de resolver os problemas de condição de disputa é introduzir pequenos atrasos nos processos envolvidos.

3. Explique o que é *espera ocupada* e por que os mecanismos que empregam essa técnica são considerados ineficientes.

4. Em que circunstâncias o uso de espera ocupada é inevitável?

5. Considere `ocupado` uma variável inteira compartilhada entre dois processos A e B (inicialmente, `ocupado = 0`). Sendo que ambos os processos executam o trecho de programa abaixo, explique em que situação A e B poderiam entrar simultaneamente nas suas respectivas regiões críticas.

```
1 while (true) {
2       regiao_nao_critica();
3       while (ocupado) {};
4       ocupado = 1;
5       regiao_critica();
6       ocupado = 0;
7 }
```

# Referências

A. J. Bernstein. Analysis of programs for parallel processing. *IEEE Transactions on Electronic Computers*, EC-15(5):757–763, Oct 1966. ISSN 0367-7508.

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

M. Raynal. *Algorithms for Mutual Exclusion*. The MIT Press, 1986.