Capítulo 11

Mecanismos de coordenação

O capítulo anterior descreveu o problema das condições de disputa entre tarefas concorrentes e a necessidade de executar operações em exclusão mútua para evitá-las. No entanto, as soluções apresentadas não são adequadas para a construção de aplicações, devido à ineficiência e à falta de justiça na distribuição do acesso à seção crítica pelas tarefas.

Este capítulo apresenta mecanismos de sincronização mais sofisticados, como os semáforos e *mutexes*, que atendem os requisitos de eficiência e justiça. Esses mecanismos são amplamente usados na construção de aplicações concorrentes.

11.1 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 como base na construção de mecanismos de coordenação mais abstratos, como os monitores.

Um semáforo pode ser visto como uma variável s, que representa uma determinada seção crítica e cujo conteúdo interno não é acessível ao programador. Internamente, cada semáforo contém um contador inteiro *s.counter* e uma fila de tarefas *s.queue*, inicialmente vazia. As tarefas podem invocar **operações atômicas** sobre os semáforos, descritas a seguir:

down(s)**:** solicita acesso à seção crítica associada ao semáforo s, equivalendo à primitiva *enter*() discutida na Seção 10.2. Caso a seção crítica esteja livre, a chamada retorna imediatamente e a tarefa continua sua execução, entrando na seção crítica. Caso contrário, a tarefa solicitante é suspensa e adicionada à fila do semáforo[1]; o contador associado ao semáforo é decrementado. Dijkstra denominou essa operação $P(s)$ (do holandês *probeer*, que significa *tentar*).

up(s)**:** libera a seção crítica associada ao semáforo s, de forma similar à primitiva *leave*(). 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

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

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.

As operações $down(s)$ e $up(s)$ estão especificadas no Algoritmo 1.

Algoritmo 1 Operações sobre semáforos

Require: as operações devem executar atomicamente

t: tarefa que invocou a operação
s: semáforo, contendo um contador e uma fila

```
 1: procedure DOWN(t, s)
 2:     s.counter ← s.counter − 1
 3:     if s.counter < 0 then
 4:         append (t, s.queue)              ▹ põe t no final de s.queue
 5:         suspend (t)                      ▹ a tarefa t perde o processador
 6:     end if
 7: end procedure

 8: procedure UP(s)
 9:     s.counter ← s.counter + 1
10:     if s.counter ≤ 0 then
11:         u = first (s.queue)              ▹ retira a primeira tarefa de s.queue
12:         awake(u)                         ▹ devolve u à fila de tarefas prontas
13:     end if
14: end procedure
```

As operações de acesso aos semáforos são geralmente implementadas pelo núcleo do sistema operacional, como chamadas de sistema. É importante observar que a execução dessas operações deve ser **atômica**, para evitar condições de disputa sobre as variáveis internas do semáforo. Para garantir a atomicidade dessas operações em um sistema monoprocessador, seria suficiente inibir as interrupções durante a execução das mesmas; no caso de sistemas multiprocessados, devem ser usados outros mecanismos de controle de concorrência, como as operações atômicas estudadas na Seção 10.2.5, para proteger a integridade do semáforo. Neste caso, 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 10.1 poderia ser reescrito da seguinte forma:

```
1 // s: semáforo associado à conta, inicializado em 1 (livre)
2
3 void depositar (semaphore s, int *saldo, int valor)
4 {
5   down (s) ;              // solicita acesso a conta
6   (*saldo) += valor ;     // seção crítica
7   up (s) ;                // libera o acesso a conta
8 }
```

Por sua forma de funcionamento, os semáforos resolvem os problemas encontrados nas soluções vistas no Capítulo 10:

Eficiência: as tarefas que aguardam o semáforos são suspensas e não consomem processador; quando o semáforo é liberado, somente a primeira tarefa da fila de semáforos é acordada.

Justiça: a fila de tarefas do semáforo obedece uma política FIFO, garantindo que as tarefas receberão o semáforo na ordem das solicitações[2].

Independência: somente as tarefas que solicitaram o semáforo através da operação *down*(*s*) são consideradas na decisão de quem irá acessá-lo.

O semáforo é um mecanismo de sincronização muito poderoso, seu uso vai muito além de controlar a exclusão mútua no acesso a seções críticas. Por exemplo, 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 aquele recurso. Seu valor inicial permite expressar diferentes situações de sincronização, como será visto no Capítulo 12.

Um semáforo pode ser usado, por exemplo, para gerenciar a entrada de veículos em um estacionamento controlado por cancelas. O valor inicial do semáforo representa o número de total de vagas no estacionamento. 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, o carro ficará aguardando o semáforo até que uma vaga seja liberada, o que ocorre quando outro carro sair do estacionamento. A listagem a seguir representa o princípio de funcionamento dessa solução. Observa-se que essa solução funciona para um número qualquer de cancelas de entrada e de saída do estacionamento.

```
1  semaphore vagas = 100 ;     // estacionamento tem 100 vagas
2
3  // cancela de entrada invoca esta operacao para cada carro
4  void obtem_vaga()
5  {
6     down (vagas) ;            // solicita uma vaga
7  }
8
9  // cancela de saída invoca esta operacao para cada carro
10 void libera_vaga ()
11 {
12    up (vagas) ;              // libera uma vaga
13 }
```

Semáforos estão disponíveis na maioria dos sistemas operacionais e linguagens de programação. O padrão POSIX define várias chamadas para a criação e manipulação de semáforos, sendo estas as mais frequentemente utilizadas:

[2] 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.

```c
#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 try_down(s), retorna erro se o semáforo estiver ocupado
int sem_trywait (sem_t *sem) ;
```

11.2 Mutexes

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* ou *ocupado*. Esses semáforos simplificados são chamados de **mutexes** (uma abreviação de *mutual exclusion*), semáforos binários ou simplesmente **locks** (travas). Algumas das funções definidas pelo padrão POSIX [Gallmeister, 1994; Barney, 2005] para criar e usar *mutexes* são:

```c
#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) ;
```

Os sistemas Windows oferecem chamadas em C/C++ para gerenciar *mutexes*, como `CreateMutex`, `WaitForSingleObject` e `ReleaseMutex`. *Mutexes* estão disponíveis na maior parte das linguagens de programação de uso geral, como C, C++, Python, Java, C#, etc.

11.3 Variáveis de condição

Outro mecanismo de sincronização de uso frequente são as *variáveis de condição*, ou variáveis condicionais. Uma variável de condição está associada a uma condição lógica que pode ser aguardada por uma tarefa, como a conclusão de uma operação, a chegada de um pacote de rede ou o preenchimento de um *buffer*. Quando uma tarefa aguarda uma condição, ela é colocada para dormir até que outra tarefa a avise que aquela condição se tornou verdadeira. Assim, a tarefa não precisa testar continuamente uma condição, evitando esperas ocupadas.

O uso de variáveis de condição é simples: a condição desejada é associada a uma variável de condição c. Uma tarefa aguarda essa condição através do operador *wait*(c), ficando suspensa enquanto espera. A tarefa em espera será acordada quando outra tarefa sinalizar que a condição se tornou verdadeira, através do operador *signal*(c) (ou *notify*(c)).

Internamente, uma variável de condição possui uma fila de tarefas *c.queue* que aguardam a condição. Além disso, a variável de condição deve ser usada em conjunto com um *mutex*, para garantir a exclusão mútua sobre o estado da condição representada por c.

O Algoritmo 2 descreve o funcionamento das operações *wait*, *signal* e *broadcast* (que sinaliza todas as tarefas que estão aguardando a condição c). Assim como os operadores sobre semáforos, os operadores sobre variáveis de condição também devem ser executados de forma atômica.

Algoritmo 2 Operadores sobre variáveis de condição

Require: as operações devem executar atomicamente

```
t: tarefa que invocou a operação
c: variável de condição
m: mutex associado à condição

procedure WAIT(t, c, m)
    append (t, c.queue)              ▷ põe t no final de c.queue
    unlock (m)                       ▷ libera o mutex
    suspend (t)                      ▷ a tarefa t é suspensa
    lock (m)                         ▷ ao acordar, requer o mutex
end procedure

procedure SIGNAL(c)
    u = first (c.queue)              ▷ retira a primeira tarefa de c.queue
    awake(u)                         ▷ devolve u à fila de tarefas prontas
end procedure

procedure BROADCAST(c)
    while c.queue ≠ ∅ do             ▷ acorda todas as tarefas de c.queue
        u = first (c.queue)
        awake (u)
    end while
end procedure
```

Deve-se terem mente que a variável de condição **não contém** a condição propriamente dita, apenas permite efetuar a sincronização sobre essa condição. Por exemplo, se em um dado programa a condição a testar for um *buffer* ficar vazio (`buffer==0`), a variável de condição apenas permite esperar que essa condição seja verdadeira e sinalizar quando isso ocorre. As operações sobre o buffer (`buffer++`, etc) e os testes (`if (buffer == 0) {...}`) devem ser feitas pelo próprio programa.

No exemplo a seguir, a tarefa `produce_data` obtém dados de alguma fonte (rede, disco, etc) e os deposita em um *buffer* compartilhado. Enquanto isso, a tarefa `consume_data` aguarda por novos dados nesse *buffer* para consumi-los. Uma variável de condição é usada para a tarefa produtora sinalizar a presença de novos dados no *buffer*. Por sua vez, o *mutex* protege o *buffer* de condições de disputa.

```
1  condition c ;
2  mutex m ;
3
4  task produce_data ()
5  {
6    while (1)
7    {
8      // obtem dados de alguma fonte (rede, disco, etc)
9      retrieve_data (data) ;
10
11     // insere dados no buffer
12     lock (m) ;                      // acesso exclusivo ao buffer
13     put_data (buffer, data) ;       // poe dados no buffer
14     signal (c) ;                    // sinaliza que o buffer tem dados
15     unlock (m) ;                    // libera o buffer
16   }
17 }
18
19 task consume_data ()
20 {
21   while (1)
22   {
23     // aguarda presença de dados no buffer
24     lock (m) ;                      // acesso exclusivo ao buffer
25     while (buffer.size == 0)        // enquanto buffer estiver vazio
26       wait (c, m) ;                 // aguarda a condição
27
28     // retira os dados do buffer e o libera
29     get_data (buffer, data) ;
30     unlock (m) ;
31
32     // trata os dados recebidos
33     process_data (data) ;
34   }
35 }
```

É importante observar que na definição original de variáveis de condição, a operação *signal(c)* fazia com que a tarefa sinalizadora perdesse imediatamente o *mutex* e o processador, que eram entregues à primeira tarefa da fila de *c*. Esse comportamento, conhecido como *semântica de Hoare* [Lampson and Redell, 1980], interfere diretamente no escalonador de processos, sendo indesejável em sistemas operacionais de uso geral.

As implementações modernas de variáveis de condição adotam outro comportamento, denominado *semântica Mesa*, que foi inicialmente proposto na linguagem programação concorrente *Mesa*. Nessa semântica, a operação *signal(c)* apenas "acorda" uma tarefa que espera pela condição, sem suspender a execução da tarefa corrente. Cabe ao programador garantir que a tarefa corrente vai liberar o *mutex* logo em seguida e que não vai alterar a condição representada pela 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*.

11.4 Monitores

Ao usar semáforos ou *mutexes*, um programador precisa identificar 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 13). 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 dos seguintes elementos:

- um recurso compartilhado, visto como um conjunto de variáveis internas ao monitor.
- um conjunto de procedimentos e funções 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 *mutex* antes de iniciar e liberá-lo 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). Esse exemplo está também ilustrado na Figura 11.1.

```
1  monitor conta
2  {
3     string numero ;
4     float saldo = 0.0 ;
5     float limite ;
6
7     void depositar (float valor)
8     {
9        if (valor >= 0)
10          conta->saldo += valor ;
11       else
12          error ("erro: valor negativo\n") ;
13    }
14
15    void retirar (float saldo)
16    {
17       if (valor >= 0)
18          conta->saldo -= valor ;
19       else
20          error ("erro: valor negativo\n") ;
21    }
22 }
```

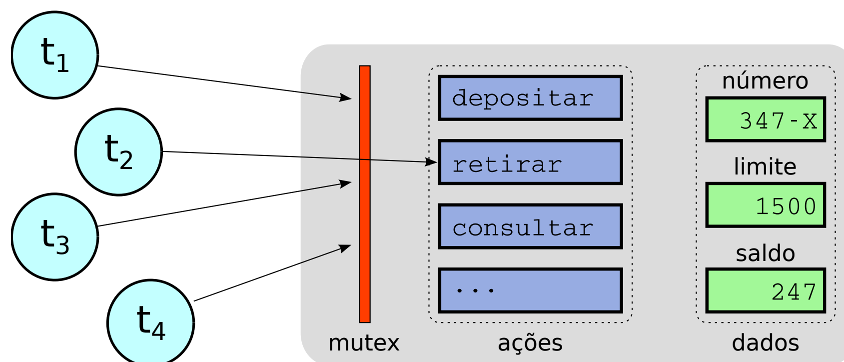

Figura 11.1: Estrutura básica de um monitor de sincronização.

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 menos todas as retiradas efetuadas*". 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 monitor, a execução dos procedimentos é feita com exclusão mútua entre eles. As operações de obtenção e liberação do *mutex* 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. Em Java, a cláusula `synchronized` faz com que um semáforo seja associado aos métodos de um objeto (ou de uma classe, se forem métodos de classe). O código a

seguir mostra um exemplo simplificado de uso de monitor em Java, no qual apenas um depósito ou retirada de cada vez poderá ser feito sobre cada objeto da classe `Conta`.

```
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 }
```

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 *signal* 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].

Exercícios

1. Por que não existem operações *read*(s) e *write*(s) para ler ou ajustar o valor atual de um semáforo?

2. Mostre como pode ocorrer violação da condição de exclusão mútua se as operações *down*(s) e *up*(s) sobre semáforos não forem implementadas de forma atômica.

3. Em que situações um semáforo deve ser inicializado em 0, 1 ou $n > 1$?

4. A implementação das operações *down*(s) e *up*(s) sobre semáforos deve ser atômica, para evitar condições de disputa sobre as variáveis internas do semáforo. Escreva, em pseudo-código, a implementação dessas duas operações, usando instruções TSL para evitar as condições de disputa. A estrutura interna do semáforo é indicada a seguir. Não é necessário detalhar as operações de ponteiros envolvendo a fila *task_queue*.

```
1 struct semaphore
2 {
3   int lock = false ;
4   int count ;
5   task_t *queue ;
6 }
```

5. Desenhe o diagrama de tempo da execução e indique as possíveis saídas para a execução concorrente das duas threads cujos pseudo-códigos são descritos a seguir. Os semáforos s_1 e s_2 estão inicializados com zero (0).

```
thread1 ()
{
   down (s1) ;
   printf ("A") ;
   up (s2) ;
   printf ("B") ;
}
```

```
thread2 ()
{
   printf ("X") ;
   up (s1) ;
   down (s2) ;
   printf ("Y") ;
}
```

Referências

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

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

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

B. Lampson and D. Redell. Experience with processes and monitors in Mesa. *Communications of the ACM*, February 1980.

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