

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 Edsger Dijkstra propôs o **semáforo**, um mecanismo de coordenação eficiente e flexível para o controle da exclusão mútua entre n tarefas, entre outros usos [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 composta s que contém uma fila de tarefas $s.queue$, inicialmente vazia, e um contador inteiro $s.counter$, cujo valor inicial depende de como o semáforo será usado. O conteúdo interno do semáforo não é diretamente acessível ao programador; para manipulá-lo devem ser usadas as seguintes **operações atômicas**:

down(s): **decrementa** o contador interno $s.counter$ e o testa: se ele for negativo, a tarefa solicitante é adicionada à fila do semáforo ($s.queue$) e suspensa¹. Caso contrário, a chamada *down(s)* retorna e a tarefa pode continuar sua execução. Dijkstra denominou essa operação $P(s)$ (do holandês *probeer*, que significa *tentar*).

up(s): **incrementa** o contador interno $s.counter$ e o testa: um contador negativo ou nulo indica que há tarefa(s) suspensa(s) naquele semáforo. A primeira tarefa da fila $s.queue$ é então devolvida à fila de tarefas prontas, para retomar sua execução assim que possível. Deve-se observar que esta chamada não é bloqueante: a

¹Alguns sistemas implementam também a chamada *try_down(s)*, com semântica não-bloqueante: caso o contador esteja negativo, a chamada retorna imediatamente com um código de erro.

tarefa solicitante não é suspensa ao executá-la. Essa operação foi inicialmente denominada $V(s)$ (do holandês *verhoog*, que significa *incrementar*).

Além das operações *down* e *up*, deve existir uma operação *init* para inicializar o semáforo, que defina o valor inicial do contador (a fila inicia vazia). Essas operações estão descritas no Algoritmo 1 a seguir.

Algoritmo 1 Operações sobre semáforos

Require: as operações devem executar atômicamente

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

```

1: procedure DOWN( $t, s$ )
2:    $s.counter \leftarrow s.counter - 1$ 
3:   if  $s.counter < 0$  then
4:      $append(t, s.queue)$ 
5:      $suspend(t)$ 
6:   end if
7: end procedure

8: procedure UP( $s$ )
9:    $s.counter \leftarrow s.counter + 1$ 
10:  if  $s.counter \leq 0$  then
11:     $u = first(s.queue)$ 
12:     $awake(u)$ 
13:  end if
14: end procedure

15: procedure INIT( $s, v$ )
16:    $s.counter \leftarrow v$ 
17:    $s.queue \leftarrow [ ]$ 
18: end procedure

```

▷ põe t no final de $s.queue$
 ▷ a tarefa t perde o processador

▷ retira a primeira tarefa de $s.queue$
 ▷ devolve a tarefa u à fila de tarefas prontas

▷ valor inicial do contador
 ▷ a fila inicia vazia

As operações de acesso aos semáforos são geralmente implementadas pelo núcleo do sistema operacional e oferecidas como chamadas de sistema. É importante observar que elas devem ser **atômicas**, para evitar condições de disputa sobre as variáveis internas do semáforo e proteger sua integridade. Para garantir a atomicidade dessas operações em um sistema monoprocessoador seria suficiente inibir as interrupções durante sua execução; no caso de sistemas multiprocessoadores, devem ser usados outros mecanismos de controle de concorrência, como as operações atômicas estudadas na Seção 10.2.5. Neste caso, a espera ocupada não constitui um problema, pois a execução dessas operações é muito rápida.

Semáforos podem ser usados para o controle da exclusão mútua em seções críticas. Para tal, basta usar $down(s)$ para solicitar acesso a uma seção crítica e $up(s)$ para liberá-la. O semáforo s deve ser inicializado em 1, para que somente uma tarefa consiga

entrar na seção crítica de cada vez. Usando semáforos, o código de depósito em conta bancária apresentado na Seção 10.1 poderia ser reescrito da seguinte forma:

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

Por sua forma de funcionamento, os semáforos oferecem uma solução adequada aos 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 solicitações².

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

Um exemplo interessante do uso de semáforos é o controle de vagas em um estacionamento controlado por cancelas. O valor inicial do semáforo *s* representa o número de total de vagas no estacionamento. Quando um carro deseja entrar no estacionamento, ele solicita uma vaga usando *down(s)*; enquanto o semáforo for positivo não haverá bloqueios, pois há vagas livres. Caso não existam mais vagas livres, o carro ficará aguardando no semáforo até que uma vaga seja liberada, o que ocorre quando outro carro sair do estacionamento e invocar *up(s)*. A listagem a seguir representa o princípio de funcionamento dessa solução. Observa-se que essa solução é genérica: ela funciona para qualquer número de vagas e quantidade de cancelas de entrada ou saída do estacionamento.

²Algumas 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.

```
1 init (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 funções em C para a criação e manipulação de semáforos, sendo estas as mais frequentemente utilizadas:

```
1 #include <semaphore.h>
2
3 // inicializa um semáforo apontado por "sem", com valor inicial "value"
4 int sem_init (sem_t *sem, int pshared, unsigned int value) ;
5
6 // Operação up(s)
7 int sem_post (sem_t *sem) ;
8
9 // Operação down(s)
10 int sem_wait (sem_t *sem) ;
11
12 // Operação try_down(s), retorna erro se o semáforo estiver ocupado
13 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* (1) ou *ocupado* (0). 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:

```
1 #include <pthread.h>
2
3 // inicializa uma variável do tipo mutex, usando um struct de atributos
4 int pthread_mutex_init (pthread_mutex_t *restrict mutex,
5                          const pthread_mutexattr_t *restrict attr);
6
7 // destrói uma variável do tipo mutex
8 int pthread_mutex_destroy (pthread_mutex_t *mutex) ;
9
10 // solicita acesso à seção crítica protegida pelo mutex;
11 // se a seção estiver ocupada, bloqueia a tarefa
12 int pthread_mutex_lock (pthread_mutex_t *mutex) ;
13
14 // solicita acesso à seção crítica protegida pelo mutex;
15 // se a seção estiver ocupada, retorna com status de erro
16 int pthread_mutex_trylock (pthread_mutex_t *mutex) ;
17
18 // libera o acesso à seção crítica protegida pelo mutex
19 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 de 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 perceber que a condição se tornou verdadeira e informar isso através do operador *signal(c)* (ou *notify(c)*).

Internamente, uma variável de condição possui uma fila de tarefas *c.queue* que aguardam a condição *c*. 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*) unlock (*m*) suspend (*t*) lock (*m*)**end procedure** ▷ põe a tarefa *t* no final de *c.queue* ▷ libera o *mutex* ▷ a tarefa *t* é suspensa ▷ ao acordar, *t* requer o *mutex***procedure** SIGNAL(*c*) *u* = first (*c.queue*) awake(*u*)**end procedure** ▷ retira a primeira tarefa de *c.queue* ▷ devolve *u* à fila de tarefas prontas**procedure** BROADCAST(*c*) **while** *c.queue* ≠ [] **do** *u* = first (*c.queue*) awake (*u*) **end while****end procedure** ▷ acorda todas as tarefas de *c.queue*

Deve-se ter em mente que a variável de condição **não contém** a condição propriamente dita, apenas permite efetuar a sincronização sobre uma 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 int buffer[...] ;           // buffer de dados
2 condition c ;              // indica presença de dados no buffer
3 mutex m ;                  // mutex associado à condição c
4
5 task produce_data ()
6 {
7     while (1)
8     {
9         retrieve_data (...) ; // obtém dados de alguma fonte
10        lock (m) ;            // pede acesso exclusivo ao buffer
11        put_data (buffer, ...) ; // insere dados no buffer
12        signal (c) ;         // sinaliza que o buffer tem dados
13        unlock (m) ;        // libera o buffer
14    }
15 }
16
17 task consume_data ()
18 {
19     while (1)
20     {
21         // aguarda presença de dados no buffer
22         lock (m) ;          // pede acesso exclusivo ao buffer
23         while (size (buffer) == 0) // enquanto buffer estiver vazio
24             wait (c, m) ;   // aguarda a condição (e libera o buffer)
25
26         // ao acordar, a tarefa detém o buffer (vide "wait")
27
28         get_data (buffer, ...) ; // retira os dados do buffer
29         unlock (m) ;           // libera o buffer
30         process_data (...) ;   // trata os dados recebidos
31     }
32 }
```

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 de 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 (cond, mutex)`, `pthread_cond_signal (cond)` e `pthread_cond_broadcast (cond)`. 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 }
```

A definição formal de monitor prevê a 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

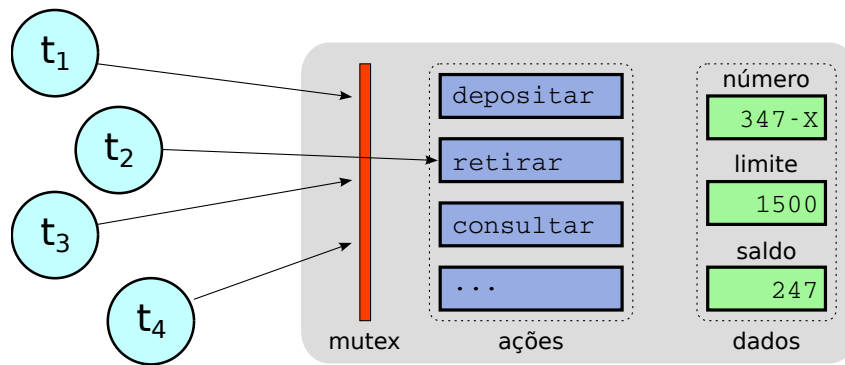


Figura 11.1: Estrutura básica de um monitor de sincronização.

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 de programação, 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 ()                                thread2 ()
{                                           {
    down (s1) ;                               printf ("X") ;
    printf ("A") ;                             up (s1) ;
    up (s2) ;                                   down (s2) ;
    printf ("B") ;                             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.