Sistemas Operacionais

Interação entre tarefas - mecanismos de coordenação

Prof. Carlos Maziero

DInf UFPR, Curitiba PR

Fevereiro de 2019

Conteúdo

Semáforos

Proposto em 1965 por E. Dijkstra, muito usado

Semáforo

Variável *S* que controla o acesso a uma seção crítica

Componentes:

- Contador inteiro *s.counter*
- Fila de tarefas *s.queue*
- Operações de acesso *Up* e *Down*

Operações (atômicas)

Down(s)

- Solicita acesso à seção crítica
- Decrementa o contador do semáforo
- Suspende a tarefa enquanto aguarda

Up(s)

- Libera a seção crítica
- Incrementa o contador do semáforo
- Acordar a primeira tarefa que estiver aguardando

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
```

Semáforos

O código de depósito em conta bancária usando semáforos:

```
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 à conta
6     (*saldo) += valor ;        // seção crítica
7     up (s) ;                   // libera o acesso à conta
8  }
```

Propriedades: eficiência, justiça, independência

Exemplo: controle de estacionamento

```
1  // número de vagas no estacionamento
2  semaphore vagas = 100 ;
3
4  // cancela de entrada, p/ cada carro
5  void obtem_vaga()
6  {
7     // solicita uma vaga
8     down (vagas) ;
9  }
10
11 // cancela de saída, p/ cada carro
12 void libera_vaga ()
13 {
14    // libera uma vaga
15    up (vagas) ;
16 }
```

Semáforos POSIX

Algumas chamadas da API POSIX para manipulação de semáforos:

```
1  #include <semaphore.h>
2
3  // inicializa um semáforo, com valor inicial "value"
4  int sem_init (sem_t *sem, int pshared, unsigned int value);
5
6  // Up(s)
7  int sem_post (sem_t *sem);
8
9  // Down(s)
10 int sem_wait (sem_t *sem);
11
12 // TryDown(s), retorna erro se o semáforo estiver ocupado
13 int sem_trywait (sem_t *sem);
```

Mutexes

Semáforo simplificado: *livre* (1) ou *ocupado* (0)

```c
#include <pthread.h>

// inicializa 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);

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

Variáveis de condição

Definição

- Representa uma condição aguardada por uma tarefa
- A tarefa é suspensa até a condição ficar verdadeira

Componentes:

- Semáforo binário $c.mutex$
- Fila de tarefas $c.queue$
- Operações atômicas $wait(c)$ e $signal(c)$

Variáveis de condição

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

Exemplo: produção/consumo de dados (parte a)

```
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 }
```

Exemplo: produção e consumo de dados (parte b)

```
1  task consume_data ()
2  {
3    while (1)
4    {
5      // aguarda presença de dados no buffer
6      lock (m) ;                          // acesso exclusivo ao buffer
7      while (buffer.size == 0)            // enquanto buffer estiver vazio
8        wait (c, m) ;                     // aguarda a condição
9
10     // retira os dados do buffer e o libera
11     get_data (buffer, data) ;
12     unlock (m) ;
13
14     // trata os dados recebidos
15     process_data (data) ;
16   }
17 }
```

Semânticas de variáveis de condição

Hoare : em `signal` a tarefa perde o *mutex* e a CPU, que são devolvidos à primeira tarefa da fila.

Mesa : `signal` acorda uma das tarefas suspensas, sem suspender a execução da tarefa corrente; esta deve liberar o *mutex* e não alterar a condição.

Em POSIX: semântica Mesa

`pthread_cond_wait`, `pthread_cond_signal`, `pthread_cond_broadcast`

Monitores

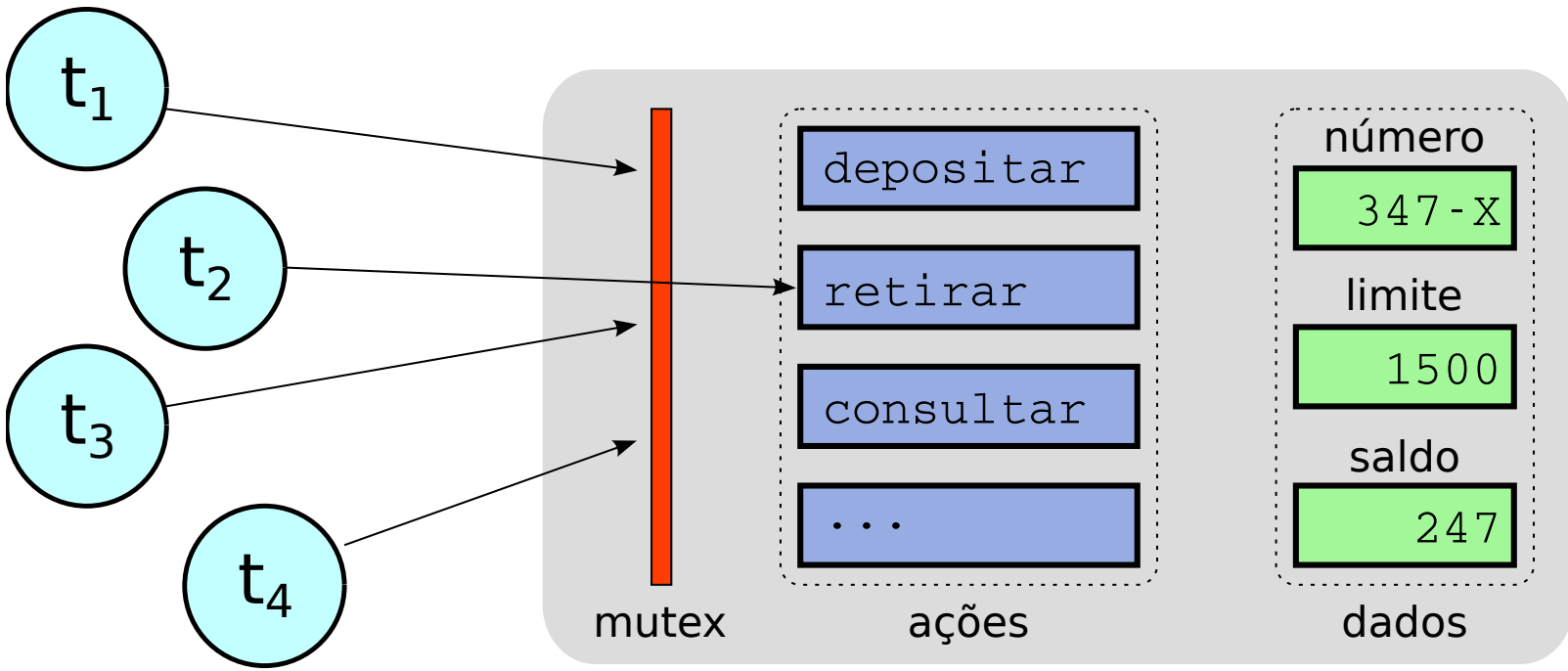
Definição

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.

Componentes:

- Recurso compartilhado (conjunto de variáveis).
- Procedimentos para acessar essas variáveis;
- um *mutex*, usado em cada acesso ao monitor

Estrutura de um monitor



Uso de Monitor

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

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 }
```