Sistemas Operacionais

Interação entre tarefas - mecanismos de coordenação

Prof. Carlos Maziero

DInf UFPR, Curitiba PR

Julho de 2020

Conteúdo

Semáforo

Mecanismo proposto por Esdger **Dijkstra** em 1965.

Usado em várias situações de coordenação entre tarefas.

O semáforo provê:

- Eficiência: baixo consumo de CPU.
- Justiça: respeita a ordem das requisições.
- Independência de outras tarefas.

Base de muitos mecanismos de comunicação e coordenação.

Estrutura de um semáforo

Um semáforo s é uma **variável composta** por:

- Contador inteiro $s.counter$
- Fila de tarefas $s.queue$ (inicia vazia)
- Operações de acesso $down(s)$ e $up(s)$

O conteúdo interno do semáforo não é acessivel diretamente.

Imagine um "objeto semáforo" com atributos e métodos.

Acesso ao semáforo

É feito usando **operações atômicas**:

Operação *Down(s)* ou *P(s)*:

- Decrementa o contador do semáforo.
- Se < 0, **suspende** a tarefa e a põe na fila de s.

Operação *Up(s)* ou *V(s)*:

- Incrementa o contador do semáforo.
- Se ≤ 0, **acorda** uma tarefa da fila.

Operações

```
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

15: procedure INIT(s, v)
16:     s.counter ← v                    ▷ valor inicial do contador
17:     s.queue ← [ ]                    ▷ a fila inicia vazia
18: end procedure
```

Exclusão mútua usando semáforo

Cada recurso é representado por um semáforo:

- Operações *down*(s) para obter e *up*(s) para liberar.
- Iniciar o semáforo com o contador em 1.
- Somente uma tarefa obtém o semáforo por vez.

Código do depósito em conta bancária usando semáforos:

```
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  }
```

Exemplo: estacionamento

Estacionamento com acesso por cancelas de entrada e de saída.

```
1  // inicia o semáforo (100 vagas)
2  init (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 semáforos:

```c
#include <semaphore.h>

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

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

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

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

Mutex

Semáforo simplificado: *livre* ou *ocupado*.

```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);
```

Variável de condição

Definição

- Representa uma condição aguardada por uma tarefa.
- Uma tarefa **espera** a condição ficar verdadeira.
- Outra tarefa **sinaliza** que a condição é verdadeira.

Componentes da variável de condição c:

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

Variável 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 e consumo de dados

Tarefa `produce_data`:

- Obtém dados de alguma fonte (arquivo, etc)
- Deposita os dados em um *buffer* compartilhado
- Sinaliza que o *buffer* contém dados

Tarefa `consume_data`:

- Aguarda a presença de dados no *buffer*
- Retira os dados do *buffer*
- Usa os dados obtidos

Condição: *há dados no buffer compartilhado*

Exemplo: produção e consumo de dados

```
1  int buffer[...] ;              // buffer de dados
2  condition c ;                  // indica presença de dados no buffer
3  mutex m ;                      // mutex associado à condição c
```

```
1  task produce_data ()
2  {
3    while (1)
4    {
5      // obtem dados de alguma fonte (rede, disco, etc)
6      retrieve_data (...) ;
7
8      // insere dados no buffer
9      lock (m) ;                          // acesso exclusivo ao buffer
10     put_data (buffer, ...) ;            // poe dados no buffer
11     signal (c) ;                        // sinaliza c (buffer tem dados)
12     unlock (m) ;                        // libera o buffer
13   }
14 }
```

Exemplo: produção e consumo de dados

```
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 (size (buffer) == 0)          // enquanto buffer estiver vazio
8        wait (c, m) ;                     // aguarda sinal da condição c
9
10     get_data (buffer, ...) ;            // retira os dados do buffer
11     unlock (m) ;                        // libera o buffer
12
13     process_data (...) ;                // trata os dados recebidos
14   }
15 }
```

Semânticas de variáveis de condição

Definem o comportamento da operação *signal(c)*:

Hoare : em *signal(c)* a tarefa perde o *mutex* e a CPU,
que são entregues à primeira tarefa da fila de *c*.

Mesa : *signal(c)* acorda a primeira tarefa da fila de *c*,
sem suspender a execução da tarefa corrente;
esta deve liberar o *mutex* e não alterar a condição.

A implementação POSIX usa a semântica Mesa:

```
pthread_cond_wait (cond, mutex)
pthread_cond_signal (cond)
pthread_cond_broadcast (cond)
```

Monitor

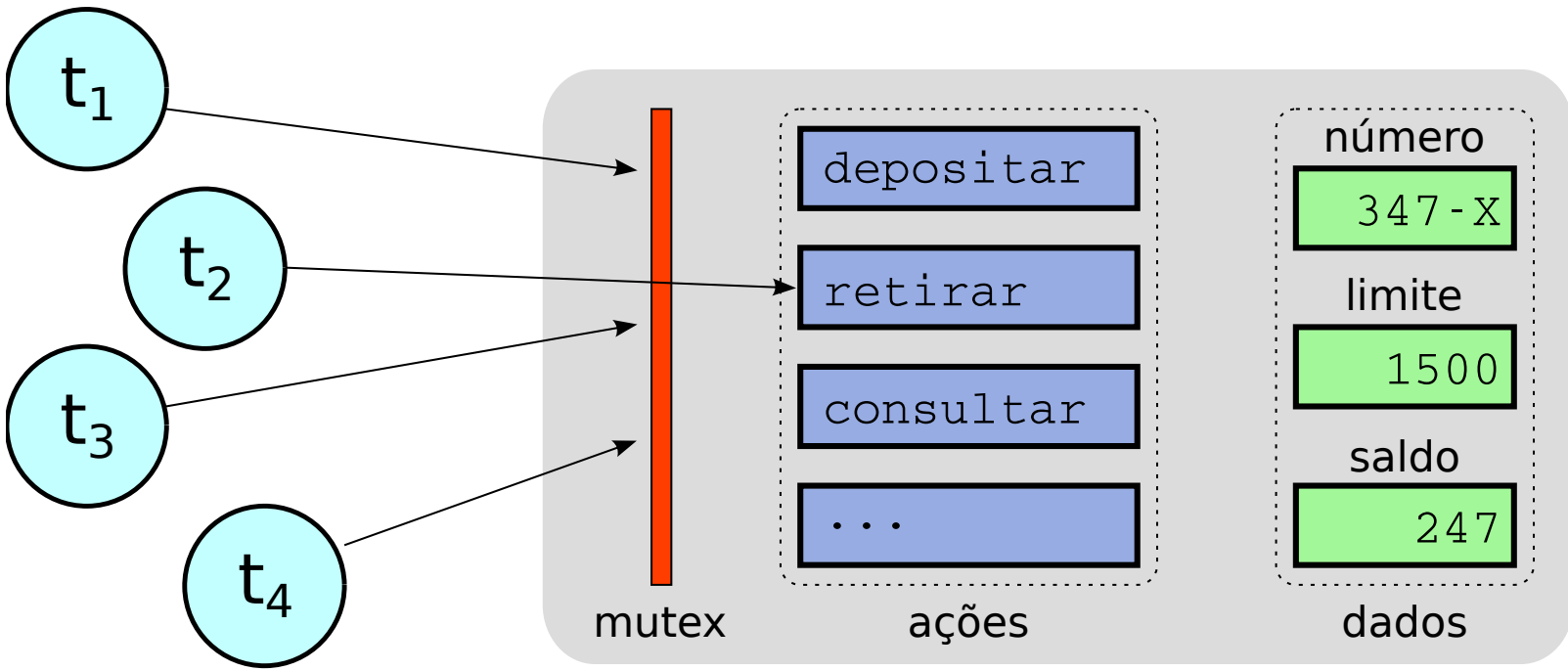
Definição

- Estrutura de sincronização associada a um recurso.
- Requer e libera a seção crítica de forma **transparente**.
- O programador não precisa mais se preocupar com isso.

Componentes de um monitor:

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

Estrutura de um monitor



Pense em um monitor como um “objeto sincronizado”.

Estrutura de um Monitor

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

Um 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 }
```