

Arquiteturas Paralelas e Distribuídas

# Programação de processos, sincronização e monitores

Eduardo Furlan Miranda

Baseado em: MAZIERO, C. A. Sistemas Operacionais:  
Conceitos e Mecanismos. 2019.

- Tarefa: é a unidade básica de atividade dentro de um sistema operacional, e podem ser implementadas de várias formas, ex.: processos, e threads
- Processos: são contêineres de recursos isolados entre si pelos mecanismos de proteção do hardware, e possuem áreas de memória, contexto e descritores, separados
- Threads: um processo pode conter uma ou mais threads, cada uma executando seu próprio código e compartilhando recursos com as demais threads localizadas no mesmo processo

- Semáforos
- Mutexes
- Variáveis de condição
- Monitores

- Mecanismo de coordenação eficiente e flexível para o controle da exclusão mútua entre **n** tarefas, entre outros usos
- É o mais utilizado na construção de aplicações concorrentes
  - De forma explícita, ou como base na construção de mecanismos de coordenação mais abstratos, como os monitores
- Pode ser visto como uma variável composta **s** que contém uma fila de tarefas, inicialmente vazia, e um contador inteiro
- Operações atômicas (apenas uma thread executa a operação por vez, evita condições de disputa): **down(s)**, **up(s)**, **init**

- 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 com um elemento para que somente uma tarefa consiga entrar na seção crítica de cada vez

- Ex.:

```
init(s, 1) ;           // semáforo que representa uma conta com saldo

void depositar (semaphore s, int *saldo, int valor) {
    down(s) ;           // solicita acesso à conta
    (*saldo) += valor ; // seção crítica
    up(s) ;             // libera o acesso à conta
}
```

os demais threads rodando este código, fazem a chamada a `down(s)` e ficam aguardando

```
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:     $u = \text{first}(s.queue)$ 
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
```

**t**: tarefa que invocou a  
operação

**s**: semáforo, contendo um  
contador ( $s.counter$ ) e  
uma fila ( $s.queue$ )

**v**: inicializa o contador  
(ex.: 1)

- **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
- **Ordem**: a fila de tarefas do semáforo obedece uma política FIFO, garantindo que as tarefas receberão o semáforo na ordem das solicitações
- **Independência**: somente as tarefas que solicitaram o semáforo através da operação **down(s)** são consideradas na decisão de quem irá obtê-lo

- São **semáforos simplificados** na qual o contador só assume dois valores possíveis: **livre (1)** ou **ocupado (0)**

```
#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);
```

POSIX



# Mutexes- variáveis de condição

9/11

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

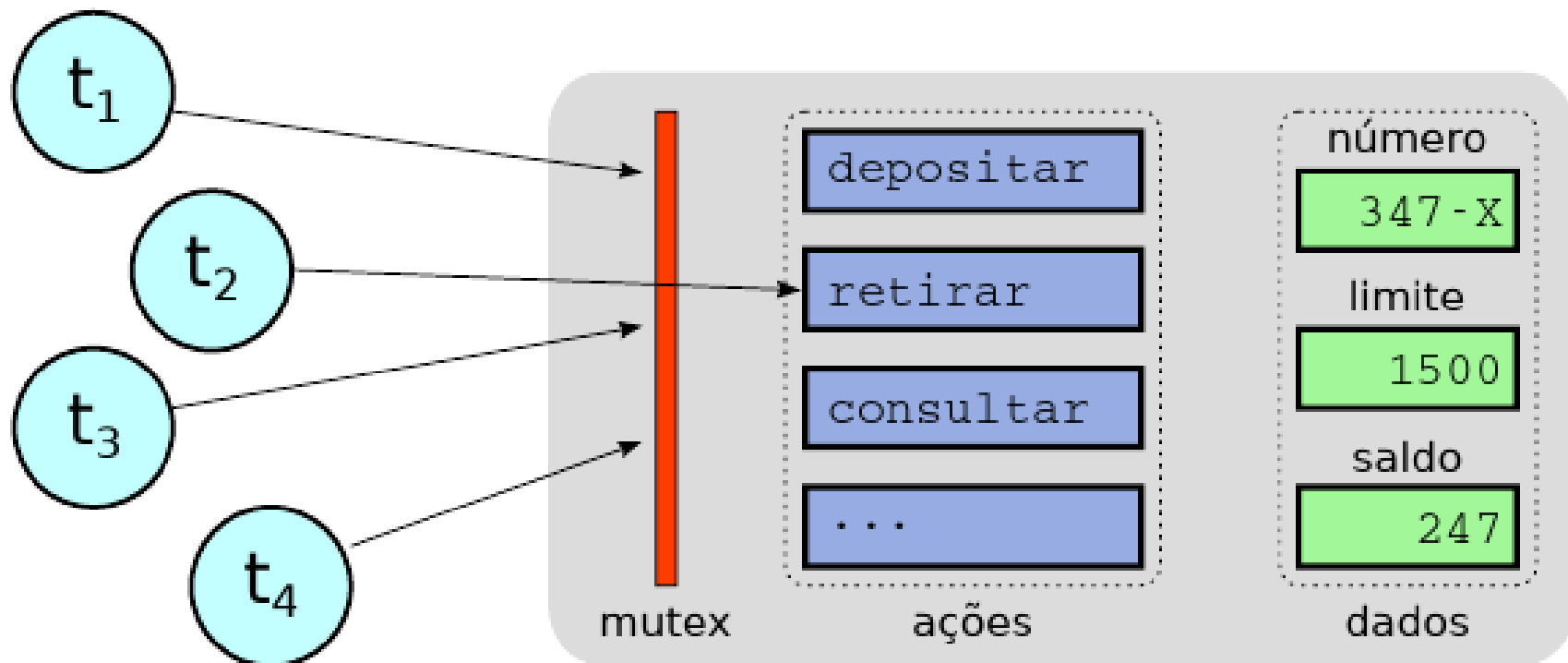
- Uma tarefa aguarda uma condição através do operador **wait(c)**, ficando suspensa enquanto espera
- Outra tarefa usa **signal(c)** para acordar a primeira
- **broadcast(c)**: acorda todas que estão esperando
- **unlock(m)**: libera o mutex, permitindo que outras tarefas acessem o recurso compartilhado

```
procedure WAIT(t, c, m)  
    append (t, c.queue)  
    unlock (m)  
    suspend (t)  
    lock (m)  
end procedure
```

```
procedure SIGNAL(c)  
    u = first (c.queue)  
    awake(u)  
end procedure
```

```
procedure BROADCAST(c)  
    while c.queue ≠ [ ] do  
        u = first (c.queue)  
        awake (u)  
    end while  
end procedure
```


- Ao usar semáforos ou mutexes, é necessário identificar explicitamente os pontos de sincronização em seu programa
  - Uma rotina esquecendo de liberar um semáforo, cria um problema
  - Se esquecer de requisitar um semáforo, também cria problema
- **Monitor**: requisita e libera a seção crítica de forma transparente



# Monitores (C++)

encapsula variáveis e  
**métodos** relacionados

11/11



```
monitor conta {  
    string numero ;  
    float saldo = 0.0 ;  
    float limite ;  
  
    void depositar (float valor) {  
        if (valor >= 0)  
            conta->saldo += valor;  
        else  
            error ("erro: valor negativo\n");  
    }  
  
    void retirar (float saldo) {  
        if (valor >= 0)  
            conta->saldo -= valor;  
        else  
            error ("erro: valor negativo\n") ;  
    }  
}
```