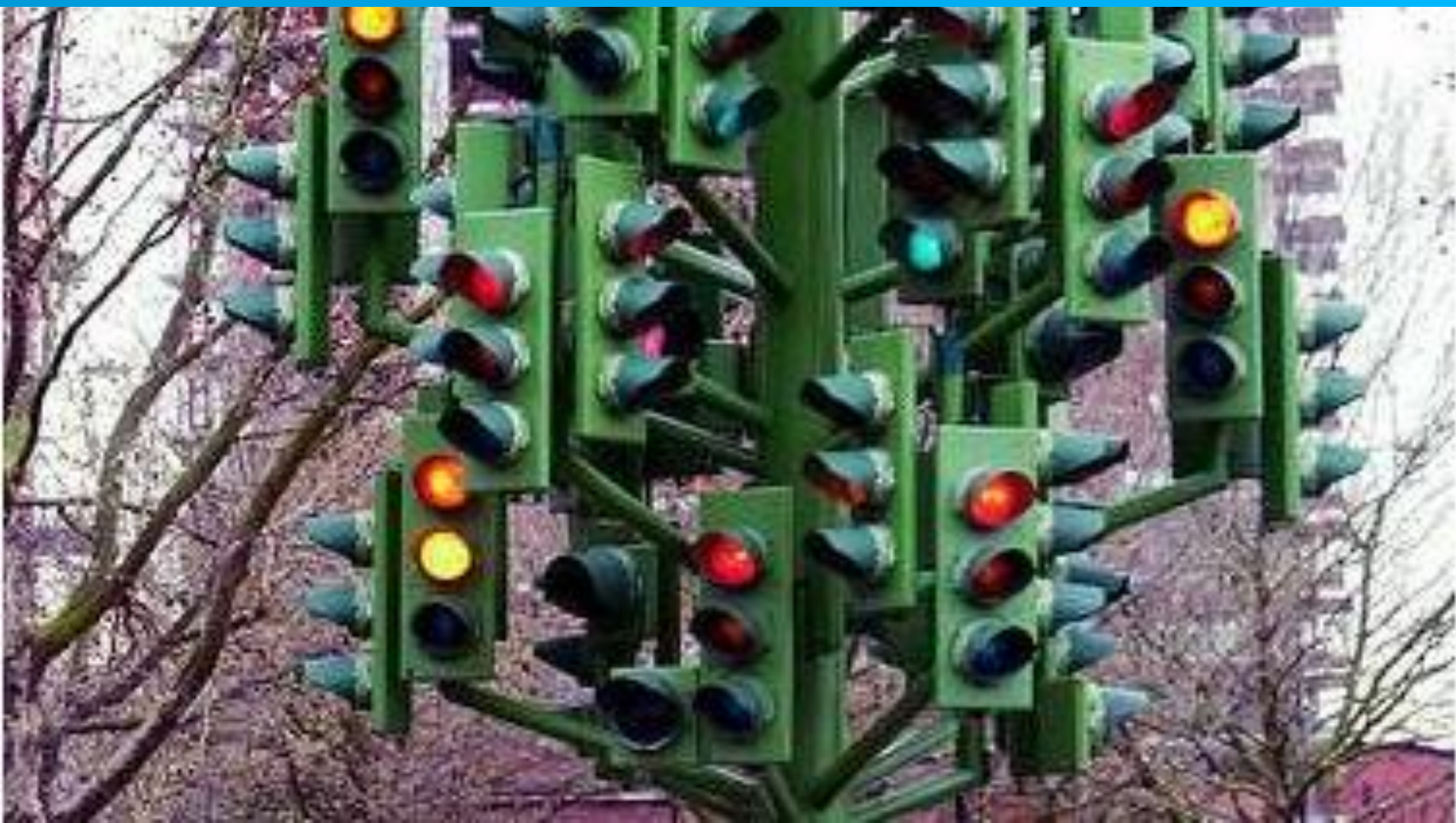


Semáforos



Índice

- Definição
- Primitivas: incrementa() e decrementa()
- Implementação de semáforos
- API POSIX para semáforos
- Monitores

Espera Ocupada - Problemas

Ineficiência : tarefas aguardam o acesso a uma seção crítica **testando continuamente** uma condição, consumindo tempo de processador.

O procedimento adequado seria **suspender estas tarefas** até que a seção crítica solicitada seja liberada.

Injustiça : não há garantia de **ordem** no acesso à seção crítica.

Dependendo da duração de quantum e da política de escalonamento, uma tarefa pode entrar e sair da seção crítica várias vezes, **antes** que outras tarefas consigam acessá-la.

Espera Ocupada - Problemas

Soluções com **espera ocupada** são pouco usadas na construção de **aplicações**.

PORÉM, seu uso é mais comum na programação de estruturas de controle de concorrência dentro do **núcleo** do sistema operacional (*Spinlock*).

Semáforos

Edsger Dijkstra em 1965 → Mecanismo de coordenação eficiente e flexível para o controle da exclusão mútua entre n tarefas.

Utilizado com muita frequência atualmente.

É uma ferramenta do SO.

Semáforos – definição

É como uma variável s especial:

1. Pode ser inicializado com qualquer valor inteiro;
2. Após a inicialização: apenas incremento ou decremento (± 1);
3. Não é necessário ler o valor de um semáforo*
4. Se uma tarefa **decrementa** o semáforo e o resultado é negativo a tarefa é suspensa até que outra tarefa incremente o semáforo.
5. Quando uma tarefa **incrementa** o semáforo e há outras tarefas esperando, uma delas é levada à fila de prontas, ou seja, é acordada.

Semáforos – definição

2 primitivas:

Semáforo

decrementa (s)

incrementa (s)

Internamente, cada semáforo contém um contador inteiro *s.counter* e uma fila de tarefas *s.queue*

Semáforos – entrada

decrementa (s) - usado para **solicitar acesso** à seção crítica associada a s

decrementa (s):

s.counter \leftarrow *s.counter* - 1

if *s.counter* < 0 **then**

 //põe a tarefa *t* no final de *s.queue*

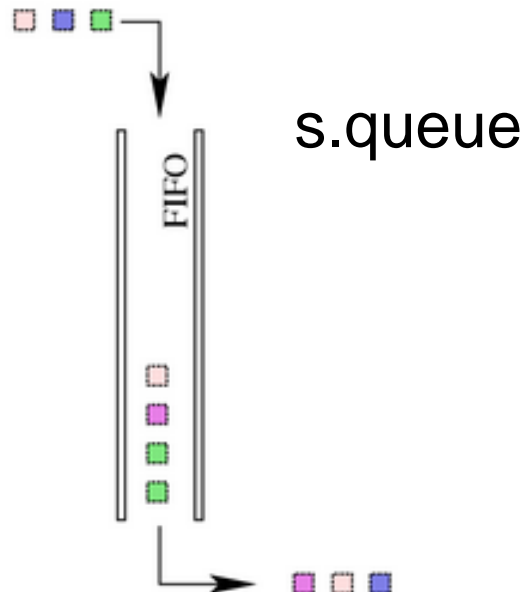
 //suspende a tarefa

end if

Fila (semáforo) de tarefas suspensas

Se a fila obedece uma política FIFO, garante-se a também a justiça no acesso à seção crítica

First-in First-out (FIFO)



Semáforos – saída

incrementa (s) - invocado para **liberar** a seção crítica associada a s

incrementa (s):

s.counter \leftarrow *s.counter* + 1

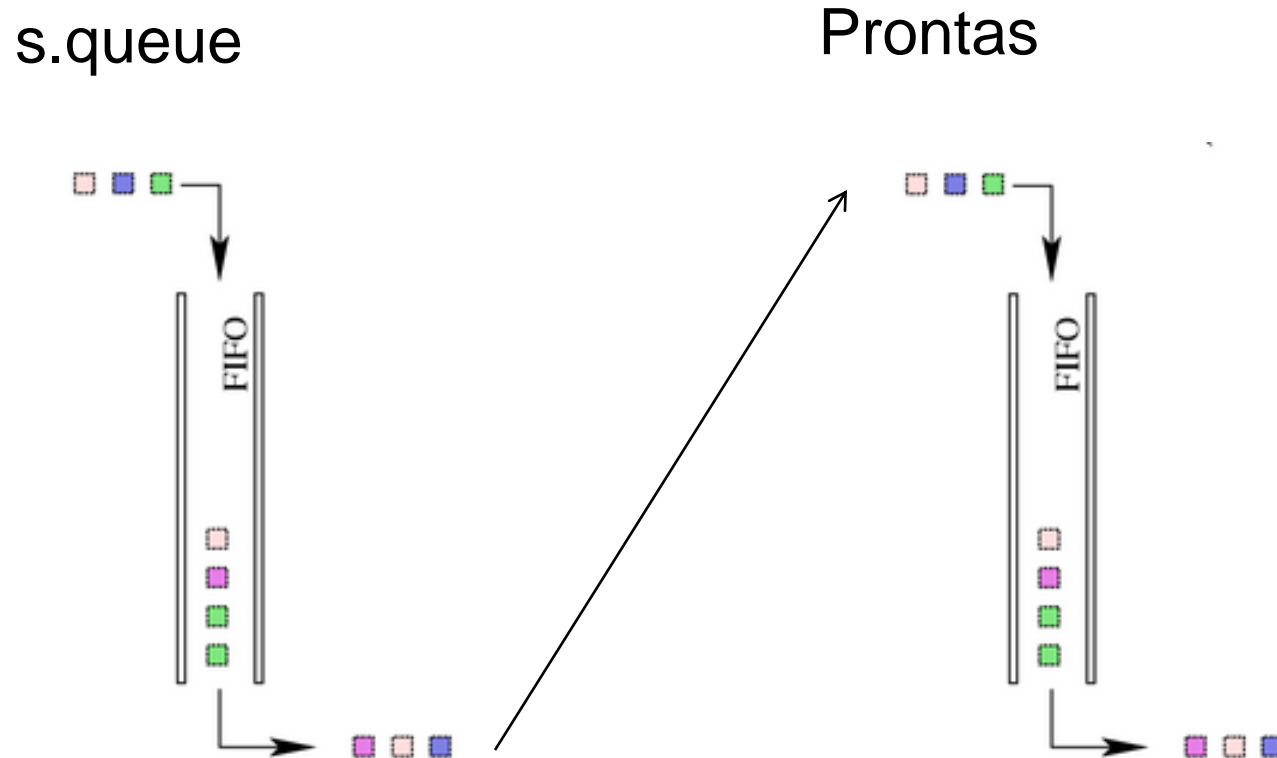
if *s.counter* \leq 0 **then**

 //retira a primeira tarefa *t* de *s.queue*

 //devolve *t* à fila de tarefas prontas (acorda *t*)

end if

Fim da suspensão



Obs.: O grupo de tarefas suspensas é diferente da fila *s.queue*.

Entrada na seção crítica

decrementa();

→ *Seção crítica;*

incrementa();

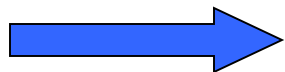
decrementa (s):

s.counter ← *s.counter* - 1

if *s.counter* < 0 **then**

//põe a tarefa no final de s.queue

//suspende a tarefa



end if

Semáforos – implementação

As duas primitivas (*acesso ao contador*) devem ser executadas de forma “atômica”!

Solução:

Semáforos – implementação

As duas primitivas (*acesso ao contador*) devem ser executadas de forma “atômica”!

Solução:

```
enter();
```

```
acesso a s.counter
```

```
leave();
```

Bloqueante!!!??? → *Spinlock*

Semáforos – nomenclatura

Dijkstra denominou a operação decrementar de **$P(s)$** (do holandês *proberen*, que significa *testar*)

E a operação incrementar foi inicialmente denominada **$V(s)$** (do holandês *verhogen*, que significa *incrementar*).

Outras formas:

- wait(); signal(); test(); post()...

Ex.: Transação bancária

```
typedef struct conta_t
{
    int saldo ;           // saldo atual da conta
    sem_t s = 1;         // semáforo associado à conta, valor
                          // inicial 1
    ...                   // outras informações da conta
} conta_t ;

void depositar (conta_t * conta, int valor)
{
    decrementa (conta->s) ;           // solicita acesso à conta
    conta->saldo += valor ;           // seção crítica
    incrementa (conta->s) ;           // libera o acesso à conta
}
```


Ex.: Transação bancária

```
typedef struct conta_t
{
    int saldo ;           // saldo atual da conta
    sem_t s = 1;         // semáforo associado à conta, valor
                          // inicial 1
    ...                   // outras informações da conta
} conta_t ;

void depositar (conta_t * conta, int valor)
{
    decrementa (conta->s) ;           // solicita acesso à conta
    conta->saldo += valor ;           // seção crítica
    incrementa (conta->s) ;           // libera o acesso à conta
}
```

Implementação?

Ex.: implementação de um semáforo

```
task_t *current ;           //aponta para a tarefa em execução no
                             //momento

*****

typedef struct sem_t        // estrutura que representa um semáforo
{
    int count ;             // contador do semáforo
    task_t *queue ;         // fila de tarefas em espera (no semáforo)
} sem_t ;

*****

// inicializa o semáforo sem com o valor indicado e uma fila vazia
void sem_init (sem_t *sem, int init_value)
{
    sem->count = init_value ;
    sem->queue = NULL ;
}
```

Ex.: implementação de um semáforo

```
void sem_down (sem_t *sem)
```

```
{
```

```
    // esta função deve ser implementada de forma atômica
```

```
    sem->count-- ;
```

```
    if (sem->count < 0)
```

```
    {
```

```
        current->status = SUSPENDED ; // ajusta status da  
                                         // tarefa atual
```

```
        append (sem->queue, current) ; // poe a tarefa atual  
                                         //no fim da fila do  
                                         //semáforo
```

```
        sched_yield () ;                // retorna ao  
                                         //dispatcher
```

```
    }
```

```
}
```

Ex.: implementação de um semáforo

```
void sem_up (sem_t *sem)
{
    task_t *task ;
    // esta função deve ser implementada de forma atômica
    sem->count++ ;
    if (sem->count <= 0)
    {
        task = first (sem->queue) ; // primeira tarefa da fila
        remove (sem_queue, task) ; // retira a tarefa da fila do
                                   // semáforo
        task->status = READY ;     // ajusta o status da tarefa
        append (ready_queue, task) ; // põe a tarefa de volta na fila de
                                   // tarefas prontas
    }
}
```

Mutex

Versão simplificada de semáforos, que só assume dois valores possíveis: *livre* (1) ou *ocupado* (0).


Mutual **Ex**clusion

<http://greenteapress.com/semaphores/>

Allen B. Downey

Semáforos - POSIX

A API POSIX define várias chamadas para a criação e manipulação de semáforos:

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

Monitores

Deficiências dos semáforos:

- Ao usar semáforos, um **programador** define explicitamente os pontos de sincronização necessários em seu programa;
- Abordagem é eficaz apenas para **programas pequenos** e problemas de sincronização simples;
- Se torna **inviável** e **suscetível** a erros em sistemas mais **complexos**.

Exemplos de erros comuns

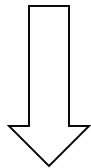
incrementa (s)

...

seção crítica

...

decrementa (s)



Viola exclusão mútua

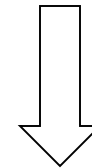
decrementa (s)

...

seção crítica

...

decrementa (s)



Pode causar impasse

Monitores

Em 1972, os cientistas Per Brinch Hansen e Charles Hoare definiram o conceito de **monitor**:

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.

Monitores

Um monitor consiste de:

- um **recurso compartilhado**, visto como um conjunto de variáveis internas ao monitor;
- um **conjunto de procedimentos** 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 semáforo antes de iniciar e liberar o semáforo ao concluir.

Monitores

De certa forma, um **monitor** pode ser visto como um **objeto** que encapsula o **recurso compartilhado**, com procedimentos (métodos) para acessá-lo.

OOP!

Monitores – pseudo-código

monitor conta

```
{  
    float saldo = 0.0 ;  
    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") ;  
}  
}
```

Monitores

- A execução dos procedimentos é feita com exclusão mútua entre eles.
- As operações de obtenção e liberação do semáforo são inseridas **automaticamente** pelo **compilador** do programa:
 - *Em todos os pontos de entrada e saída do monitor*

Monitores

- Java é um exemplo de linguagem que permite o uso de monitor
 - Java suporta threads de usuário e também permite que métodos sejam agrupados em classes
 - Adicionando-se a palavra-chave **synchronized** à declaração de um método, Java garante que, uma vez iniciado qualquer *thread* executando esse método, **nenhuma** outra *thread* será permitida executar qualquer outro método **synchronized** naquela classe

```
package br.concorrencia.produtorConsumidor;
```

```
public class Buffer {
```

```
    private int conteudo;
```

```
    private boolean disponivel;
```

```
    public synchronized void set(int idProdutor, int valor) {
```

```
        while (disponivel == true) {
```

```
            try {
```

```
                System.out.println("Produtor #" + idProdutor + " esperando...");
```

```
                wait();
```

```
            } catch (Exception e) {
```

```
                e.printStackTrace();
```

```
            }
```

```
        }
```

```
        conteudo = valor;
```

```
        System.out.println("Produtor #" + idProdutor + " colocou " + conteudo);
```

```
        disponivel = true;
```

```
        notifyAll();
```

```
    }
```

```
    public synchronized int get(int idConsumidor) {
```

```
        while (disponivel == false) {
```

```
            try {
```

```
                System.out.println("Consumidor #" + idConsumidor
```

```
                    + " esperando...");
```

```
                wait();
```

```
            } catch (Exception e) {
```

```
                e.printStackTrace();
```

```
            }
```

```
        }
```

```
        System.out.println("Consumidor #" + idConsumidor + " consumiu: "
```

```
            + conteudo);
```

```
        disponivel = false;
```

```
        notifyAll();
```

```
        return conteudo;
```

```
    }
```

```
}
```