

Sistemas Operacionais

Sincronização de Processos

Parte 2

Prof. Otávio Gomes

otavio.gomes@unifei.edu.br



- Espera ocupada (*busy waiting*)
- *Sleep / WakeUp* (primitivas - chamadas de sistema)
- Semáforos (variável de controle)
- Monitores (primitiva de alto nível)
- Troca de Mensagens

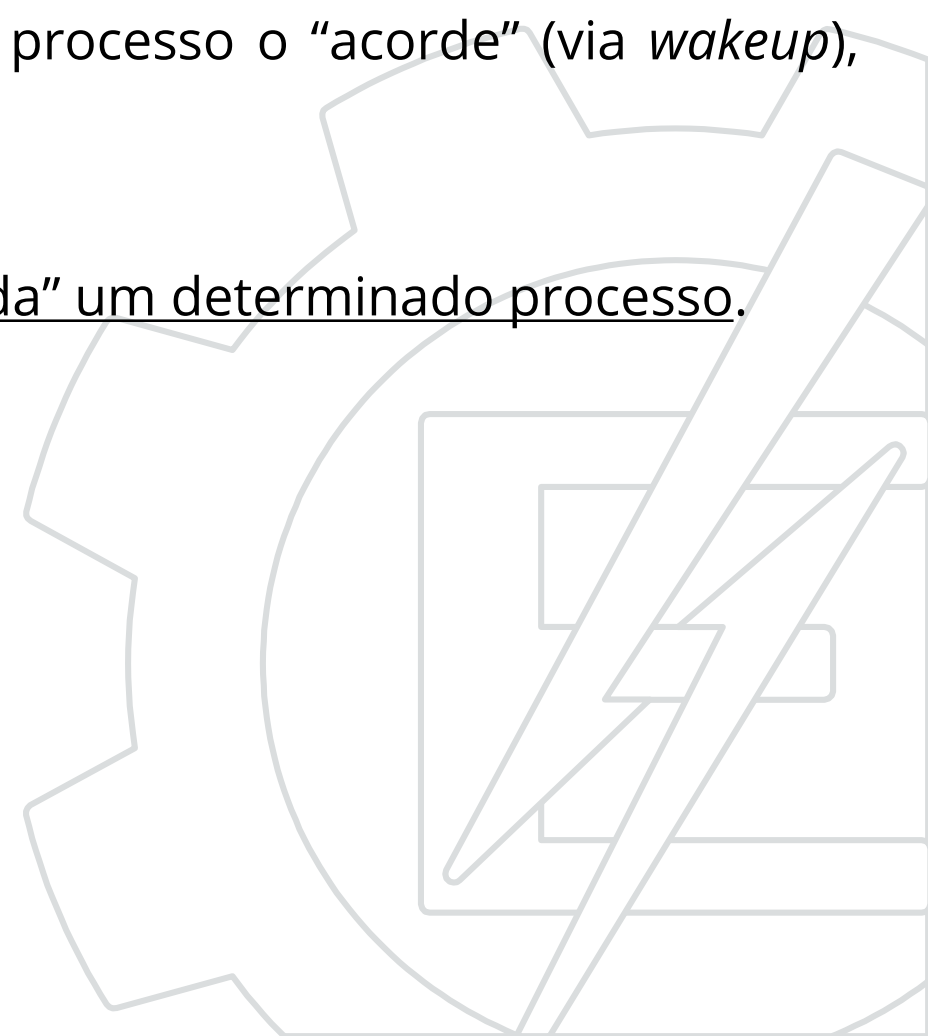


- Espera ocupada (*busy waiting*)
- ***Sleep / WakeUp*** (primitivas - chamadas de sistema)
- Semáforos (variáveis de controle)
- Monitores (primitiva de alto nível)
- Troca de Mensagens

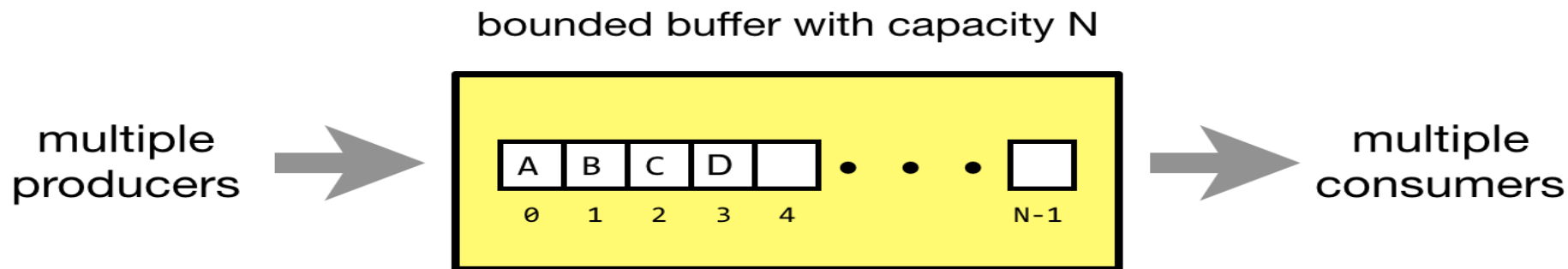


Sleep e WakeUp

- A primitiva ***sleep()*** é uma chamada de sistema que bloqueia o processo que a chamou, ou seja, suspende a execução de tal processo até que outro processo o “acorde” (via *wakeup*), **evitando a espera ocupada**;
- A primitiva ***wakeup()*** é uma chamada de sistema que “acorda” um determinado processo.



- Problema que pode ser solucionado com o uso dessas primitivas:
 - Problema do **Produtor/Consumidor** (*bounded buffer*): dois processos compartilham um *buffer* de tamanho fixo (limitado).
 - O processo produtor coloca dados no *buffer* e o processo consumidor retira dados do *buffer*.



- Problema que pode ser solucionado com o uso dessas primitivas:
 - Problema do Produtor/Consumidor (***bounded buffer***): dois processos compartilham um *buffer* de tamanho fixo (limitado).
 - O processo produtor coloca dados no *buffer* e o processo consumidor retira dados do *buffer*;
 - **Problemas:**
 - Produtor deseja colocar dados quando o *buffer* ainda está cheio;
 - Consumidor deseja retirar dados quando o *buffer* está vazio;
 - Solução: colocar os processos para “dormir”, até que eles possam ser executados.

- **Buffer:**

- Uma variável *count* controla a quantidade de dados presente no *buffer*.

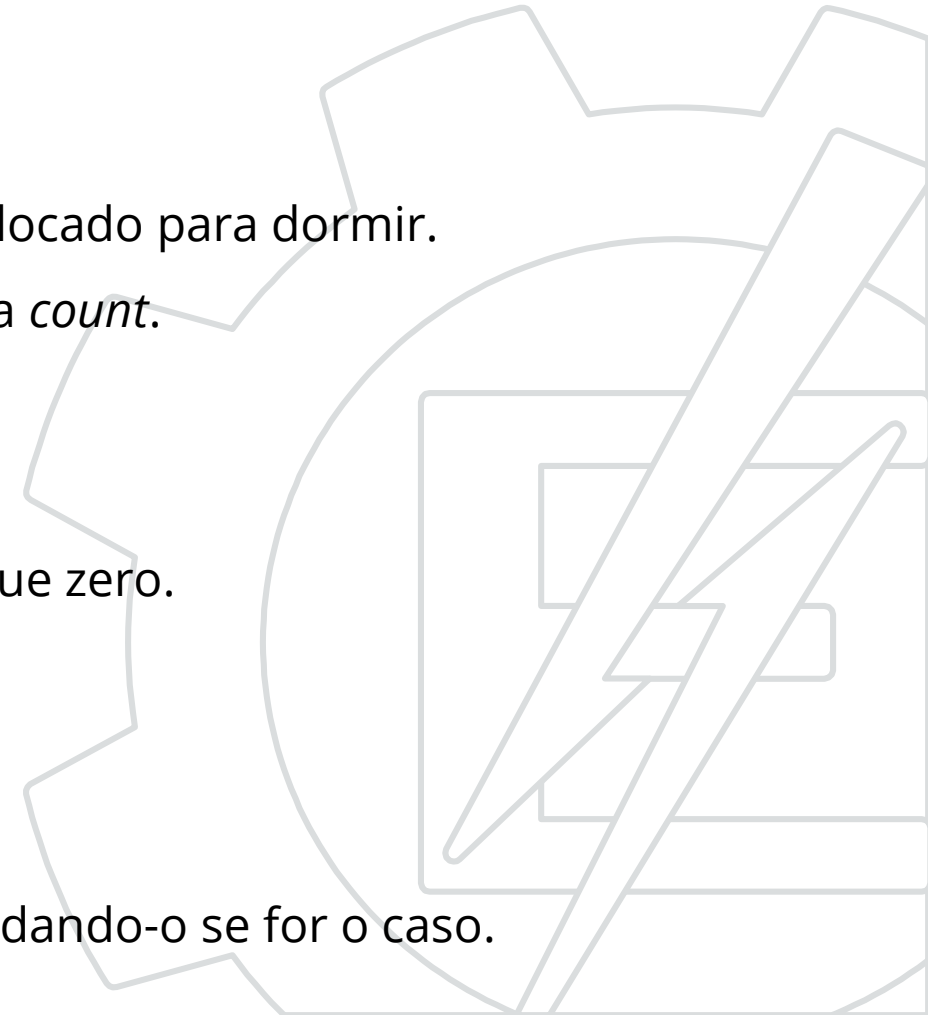
- **Produtor:**

- Antes de colocar dados no *buffer*, verifica o valor de *count*.
- Se a variável está com valor máximo, o processo produtor é colocado para dormir.
- Caso contrário, o produtor coloca dados no *buffer* e incrementa *count*.

- **Consumidor:**

- Antes de retirar dados do *buffer*, verifica se *count* é maior do que zero.
- Se for, ele retira os dados e decrementa *count*.
- Caso contrário, o processo vai dormir.

Sempre testam para verificar se o outro processo está acordado, acordando-o se for o caso.



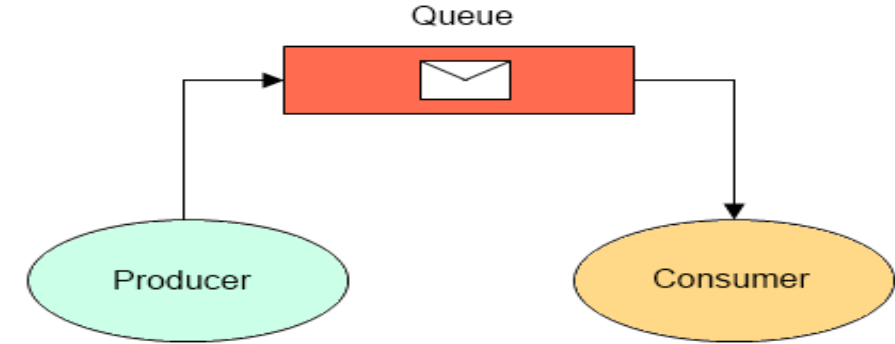
Sleep e WakeUp

- Chamadas de sistema para bloqueio e desbloqueio.
- Envolva o problema clássico do Produtor-Consumidor.

```
#define N 100
#define count=0

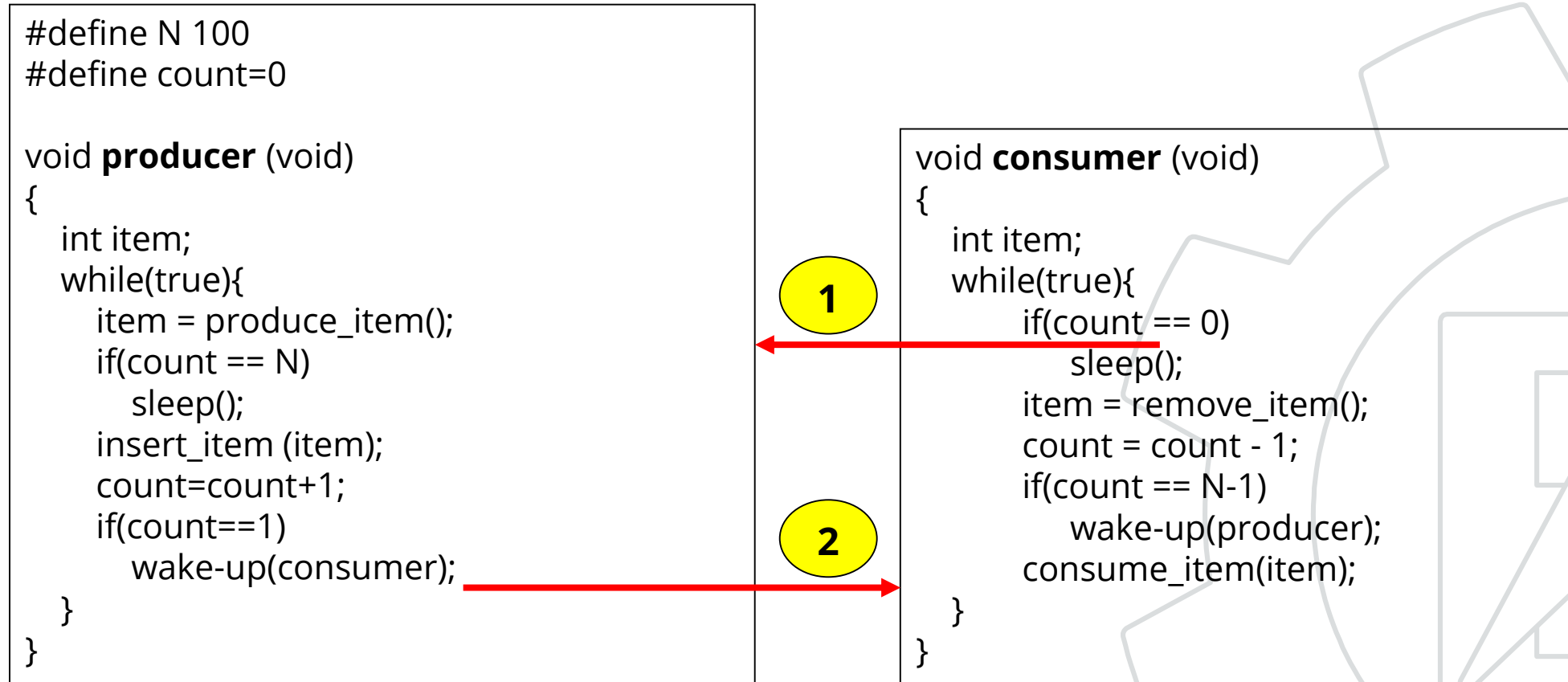
void producer (void)
{
    int item;
    while(true){
        item = produce_item();
        if(count == N)
            sleep();
        insert_item (item);
        count=count+1;
        if(count==1)
            wakeup(consumer);
    }
}
```

```
void consumer (void)
{
    int item;
    while(true){
        if(count == 0)
            sleep();
        item = remove_item();
        count = count - 1;
        if(count == N-1)
            wakeup(producer);
        consume_item(item);
    }
}
```

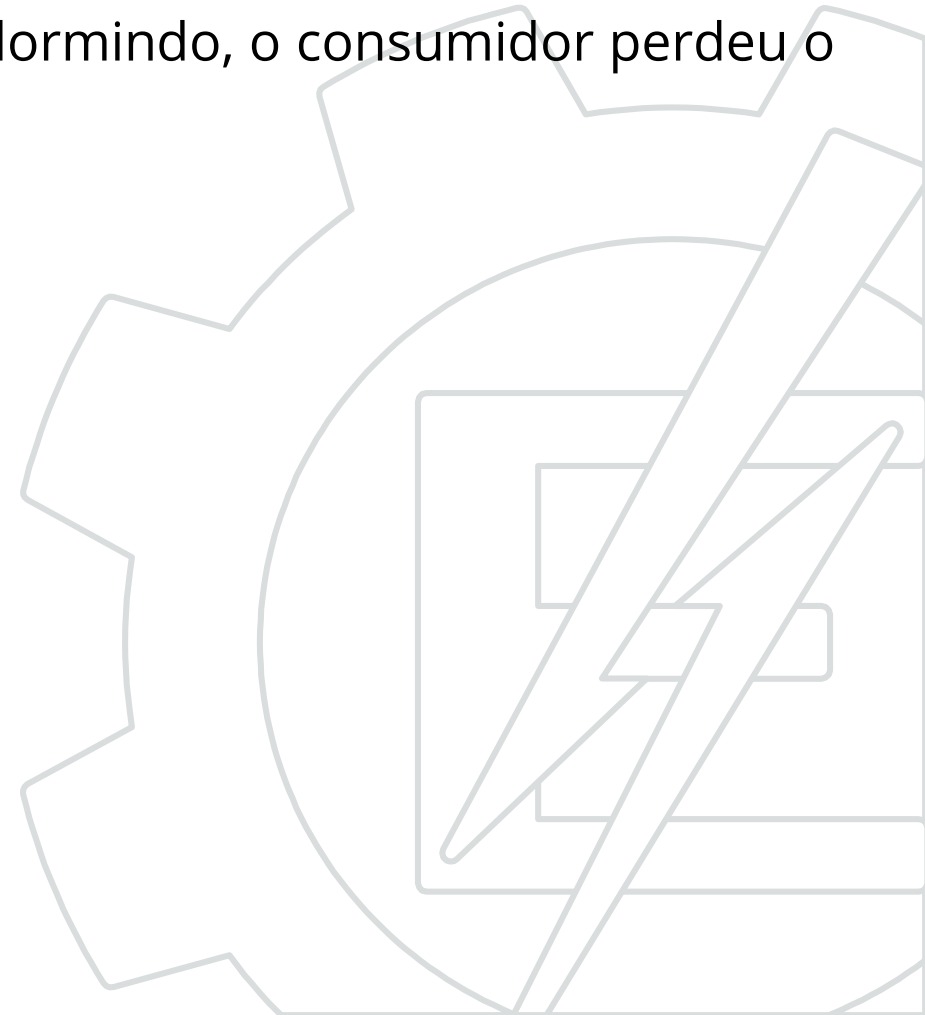


Sleep e WakeUp

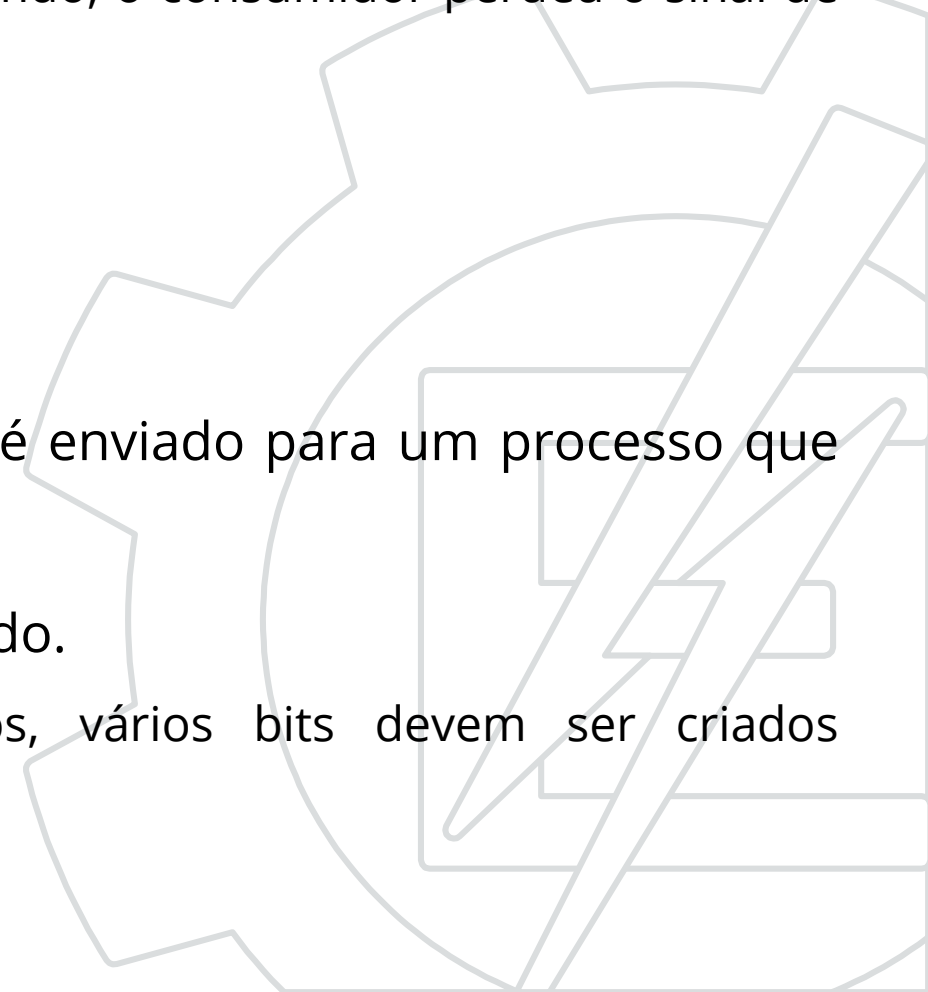
- Situação-Problema relacionada à preempção:



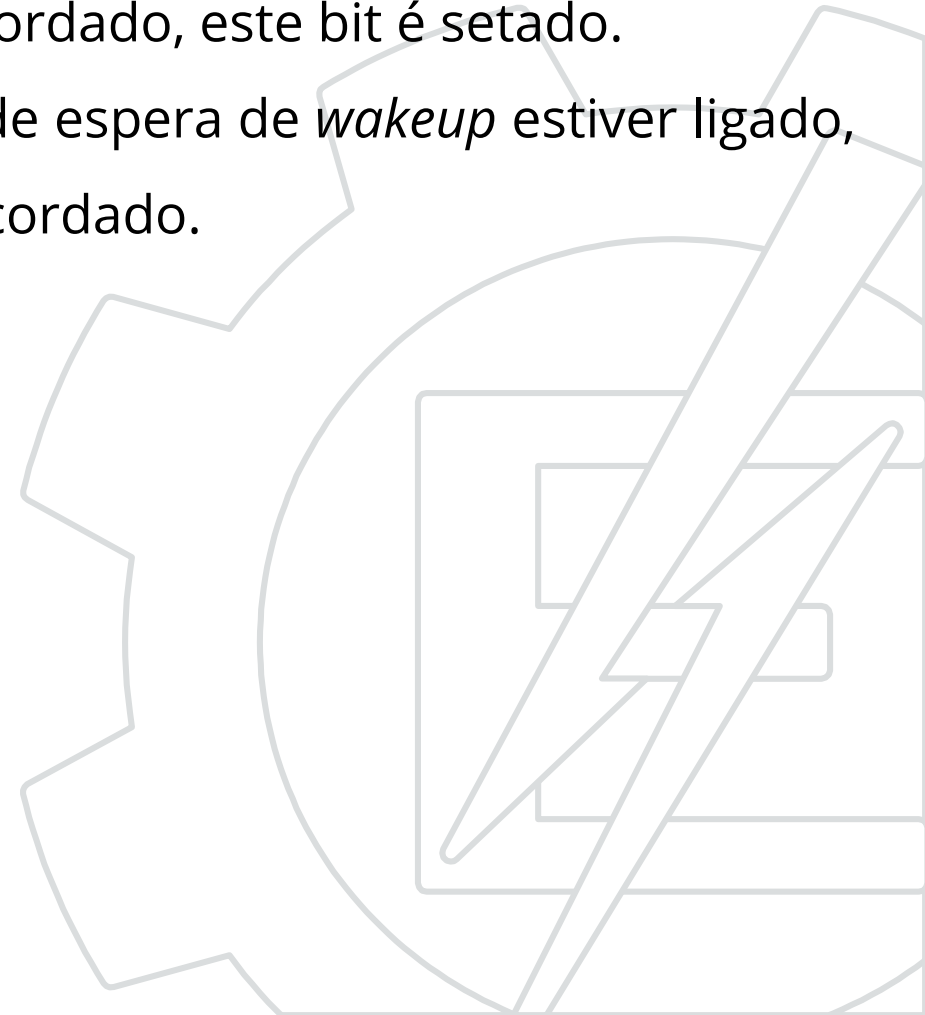
- Problema relacionado à preempção:
 - Interrupção no consumidor após a comparação *count == 0*
 - A essência do problema está no fato de, por não estar dormindo, o consumidor perdeu o sinal de *wakeup*.
 - Ambos os processos dormem para sempre.



- Problema relacionado à preempção:
 - Interrupção no consumidor após a comparação *count == 0*
 - A essência do problema está no fato de, por não estar dormindo, o consumidor perdeu o sinal de *wakeup*.
 - Ambos os processos dormem para sempre.
- Solução – *bit de wakeup*:
 - *bit* de controle recebe um valor *true* quando um sinal é enviado para um processo que não está dormindo.
 - Quando o processo é posto para dormir, o bit é verificado.
 - No entanto, no caso de vários pares de processos, vários bits devem ser criados sobrecarregando o sistema.



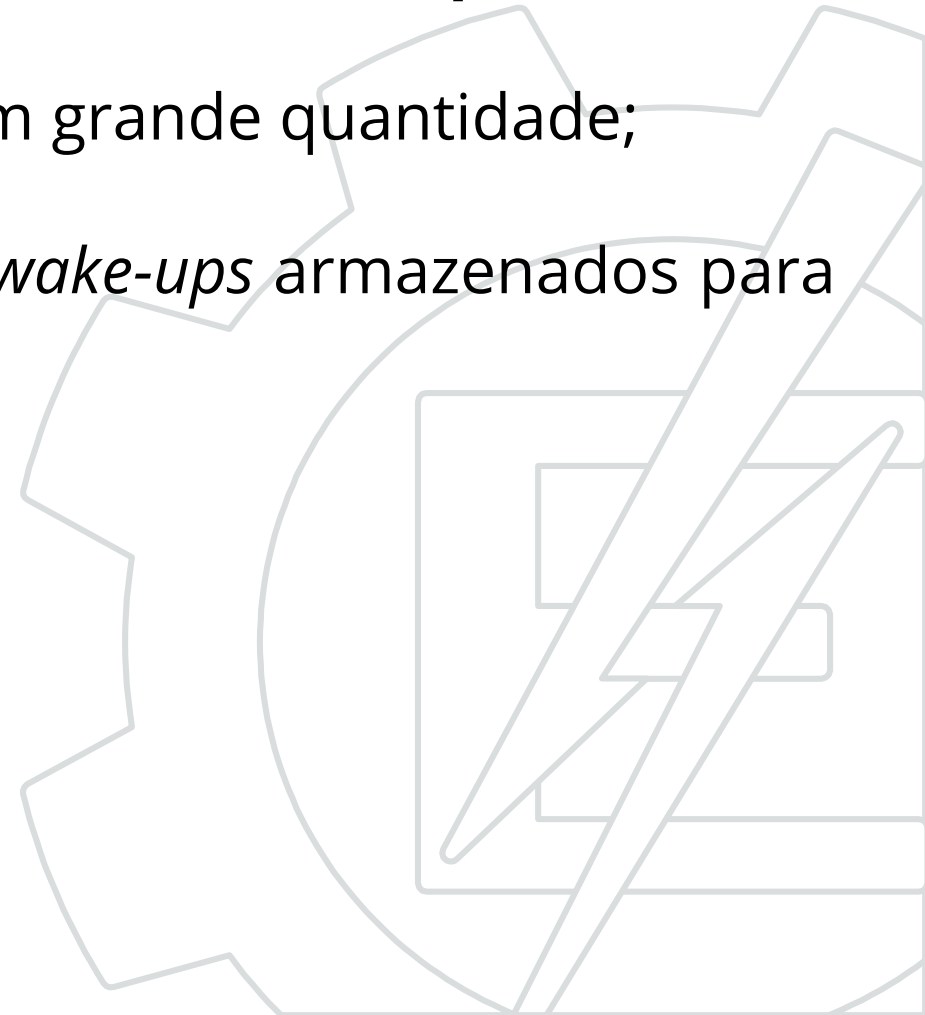
- “*bit de wakeup*”.
 - Quando um *wakeup* é mandado à um processo já acordado, este bit é setado.
 - Depois, quando o processo tenta ir dormir, se o bit de espera de *wakeup* estiver ligado, este bit será desligado, e o processo será mantido acordado.



- Espera ocupada (*busy waiting*)
- *Sleep / WakeUp* (primitivas - chamadas de sistema)
- **Semáforos** (variáveis de controle)
- Monitores (primitiva de alto nível)
- Troca de Mensagens

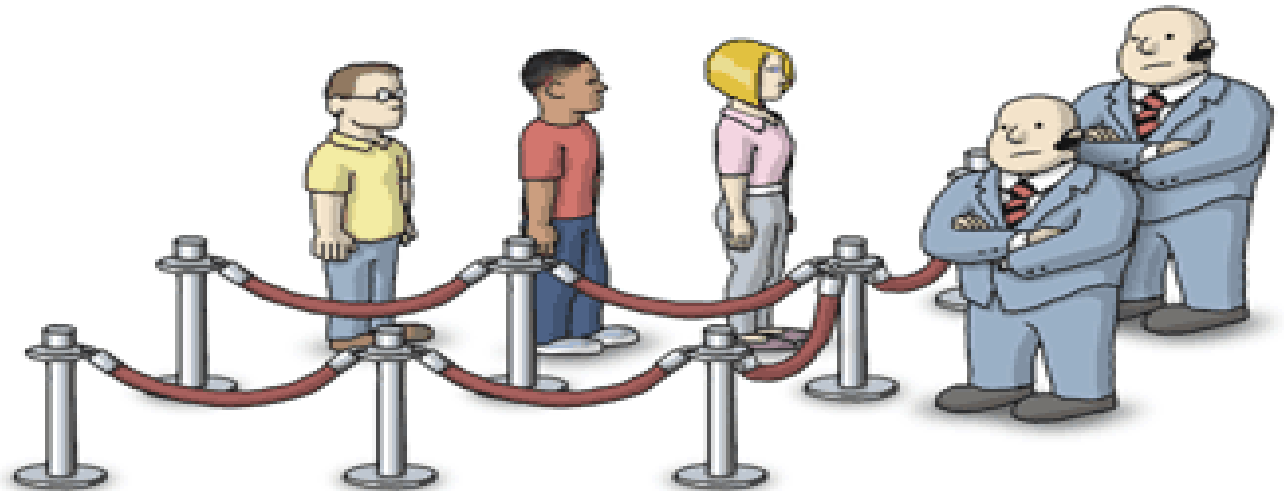
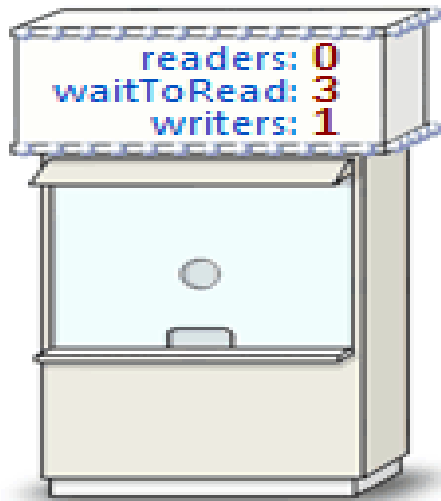


- É uma variável utilizada para **controlar o acesso a recursos compartilhados**;
- Tem o objetivo de sincronizar o uso de recursos em grande quantidade;
- Nasceu como proposta para contar o número de *wake-ups* armazenados para uso futuro.

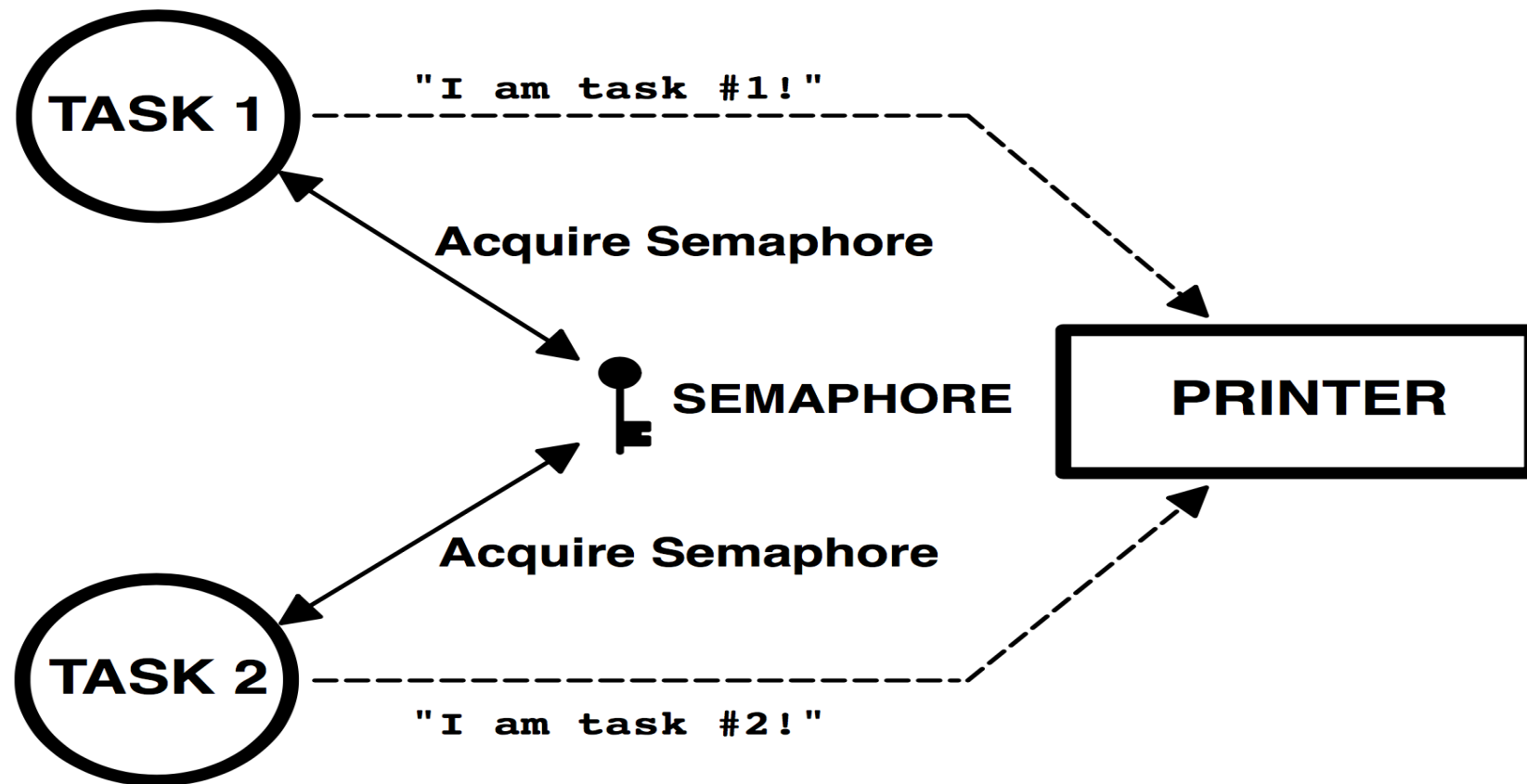


Semáforos

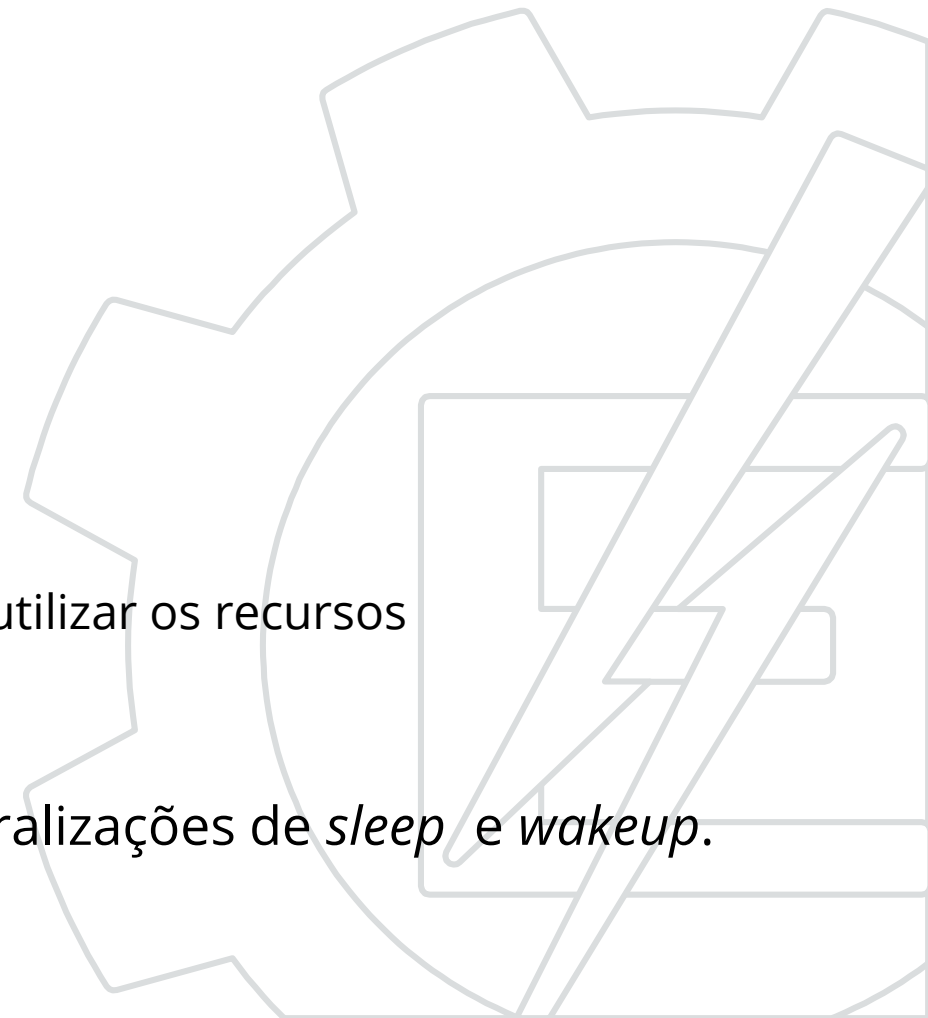
- Pode ser do tipo **geral** ou de **contagem** (*counting*) ou **binário** (*mutex / booleano*);
- Pode ser implementado através de **espera ocupada** ou através da associação de uma **fila** a cada semáforo.



- É uma variável utilizada para controlar o acesso a recursos compartilhados:

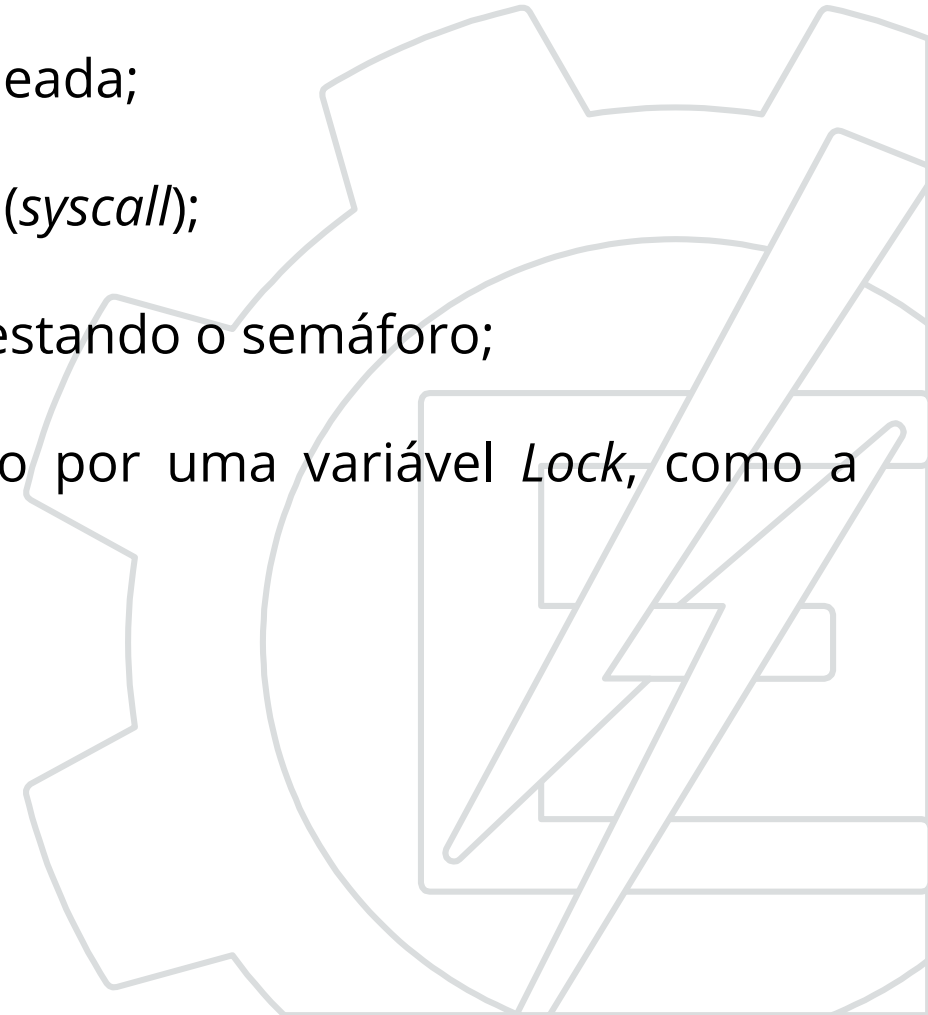


- **Semáforo = 0**
 - Não há recurso livre
 - Nenhum *wakeup* está armazenado
- **Semáforo > 0**
 - Recurso livre
 - Um ou mais *wakeups* estão pendentes >> Devem utilizar os recursos
- Dijkstra propôs 2 operações sobre semáforos, que são generalizações de *sleep* e *wakeup*.



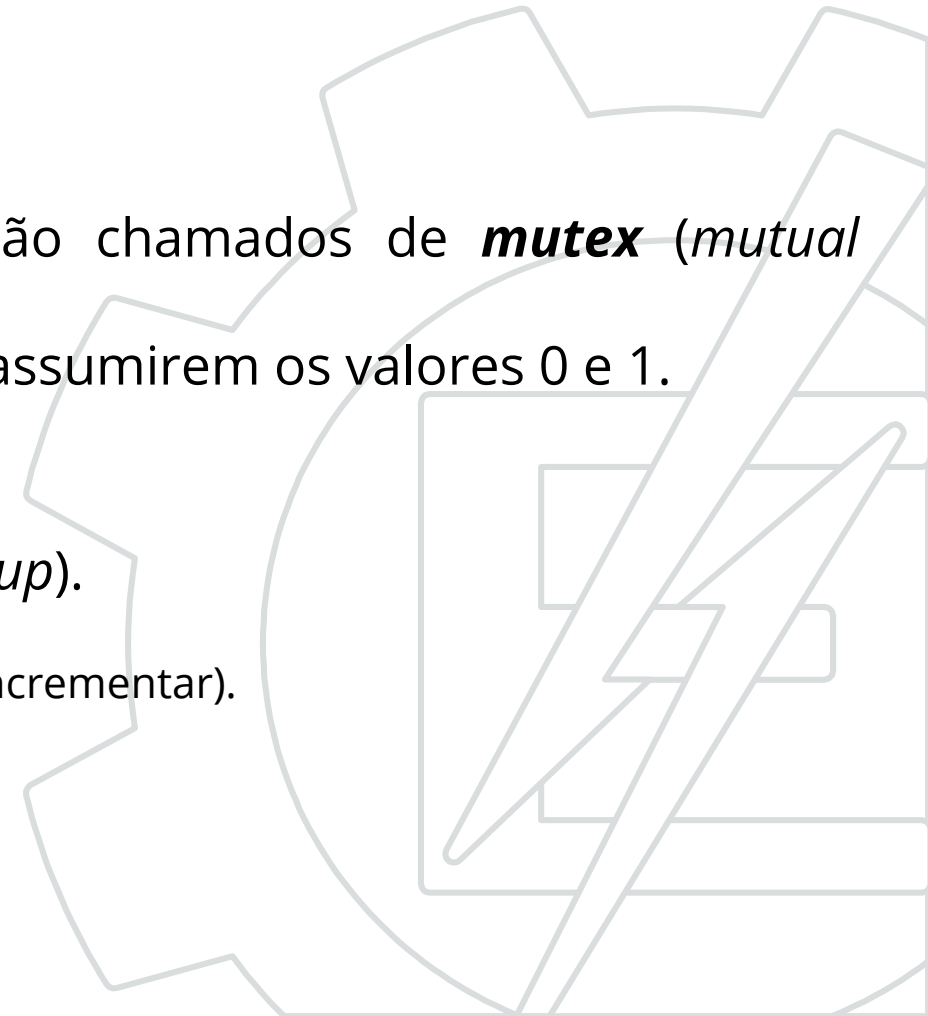
- Operações sobre semáforos (**atômicas**)
 - **down** - executada sempre que um processo deseja usar um recurso compartilhado:
 - Verifica se o valor do semáforo é maior que 0;
 - Se for, $\text{semáforo} = \text{semáforo} - 1$, e continua a operação;
 - Se não for, o processo que executou o *down* fica bloqueado, sem completar o *down*, pois não há recurso disponível.
 - **up** - executada sempre que um processo liberar o recurso.
 - $\text{semáforo} = \text{semáforo} + 1$;
 - Se há processos bloqueados nesse semáforo, escolhe um deles e o desbloqueia - finaliza a execução do *down* (Neste último caso, o valor do semáforo permanece o mesmo).

- Operações **atômicas**
 - Uma vez que uma operação semáforo iniciou, nenhum outro processo pode acessar o semáforo até que a operação seja completada ou bloqueada;
 - Geralmente implementadas como chamade de sistema (*syscall*);
 - O S.O. desabilita todas as interrupções enquanto está testando o semáforo;
 - Se houver múltiplas CPUs, cada semáforo é protegido por uma variável *Lock*, como a instrução TSL.

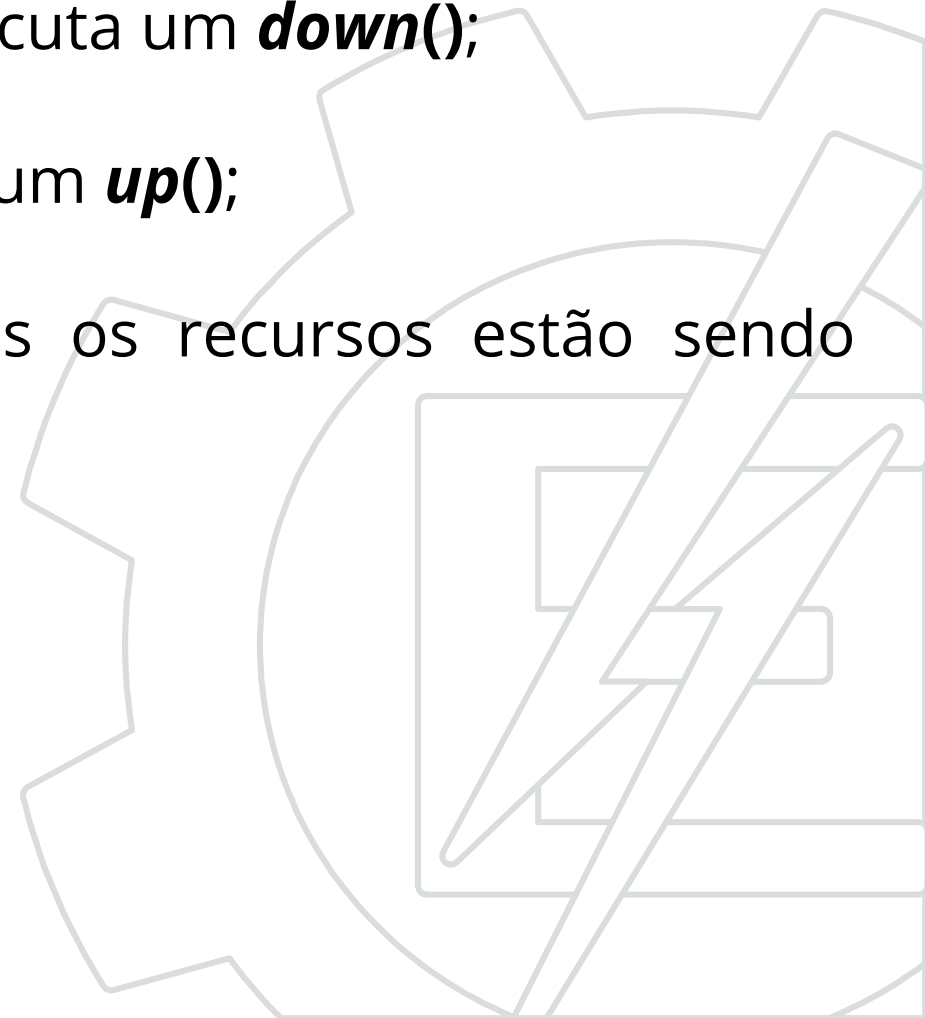


- Semáforo geral (*counting semaphore*) - usados para controlar acessos a um determinado recurso com um número finito de instâncias.
- Semáforos usados para implementar exclusão mútua são chamados de ***mutex*** (*mutual exclusion semaphore*) ou binários ou *booleanos*, por apenas assumirem os valores 0 e 1.
- Primitivas de chamadas de sistema: ***down*** (*sleep*) e ***up*** (*wakeup*).

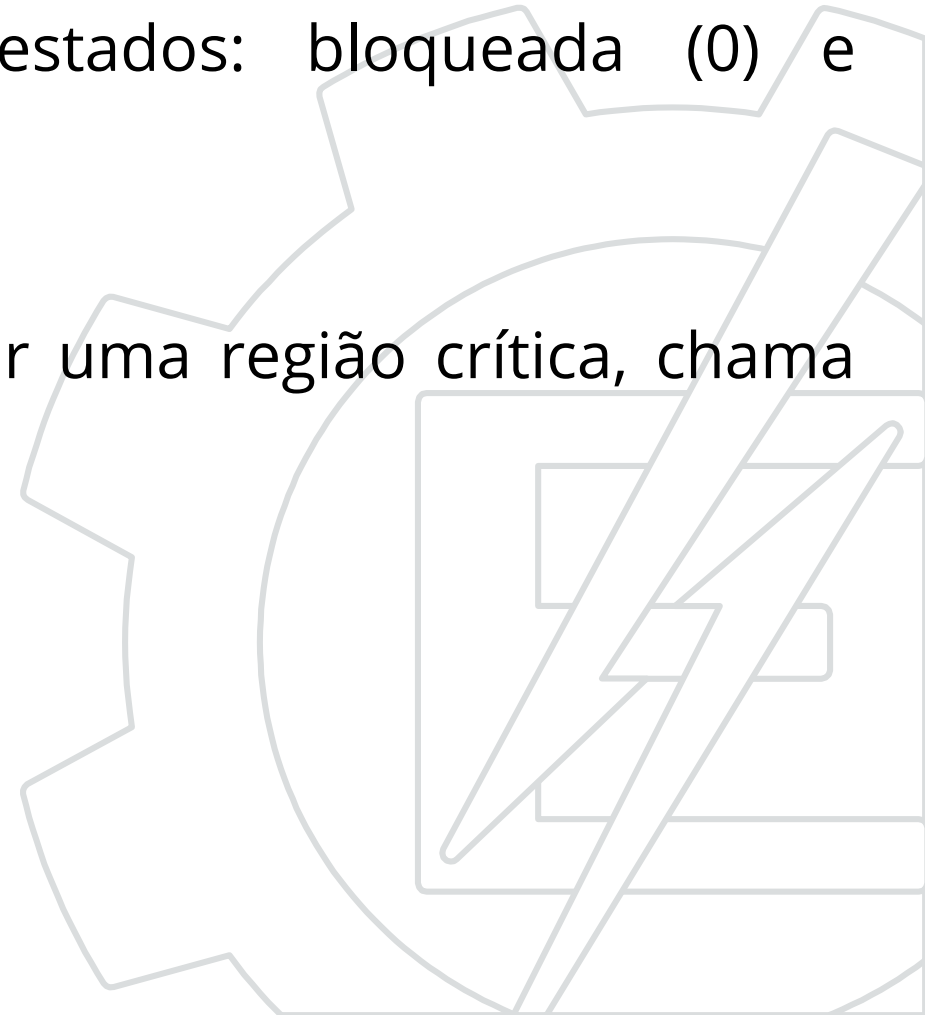
Originalmente, em holandês, **P** (***down*** - *proberen* - testar) e **V** (***up*** - *verhogen* - incrementar).



- Inicializa com o número de recursos disponíveis;
- Cada processo que deseja utilizar um recurso executa um ***down()***;
- Quando um processo libera um recurso, executa um ***up()***;
- Quando o contador do semáforo vale 0, todos os recursos estão sendo usados – o processo é posto para dormir.



- Serve para conceder entrada à região crítica;
- ***Mutex*** é uma variável com apenas dois estados: bloqueada (0) e desbloqueada (1);
- Quando uma *thread* ou processo precisa acessar uma região crítica, chama **mutex_lock**;
- Quando termina, chama **mutex_unlock**.



- Problema produtor/consumidor: Resolve o problema de perda de sinais enviados.
- Solução utiliza três semáforos:
 - **Full**: conta o número de *slots* no *buffer* que estão ocupados; iniciado com 0 - resolve sincronização;
 - **Empty**: conta o número de *slots* no *buffer* que estão vazios; iniciado com o número total de *slots* no *buffer* - resolve sincronização;
 - **Mutex**: garante que os processos produtor e consumidor não acessem o *buffer* ao mesmo tempo; iniciado com 1 - permite a exclusão mútua;

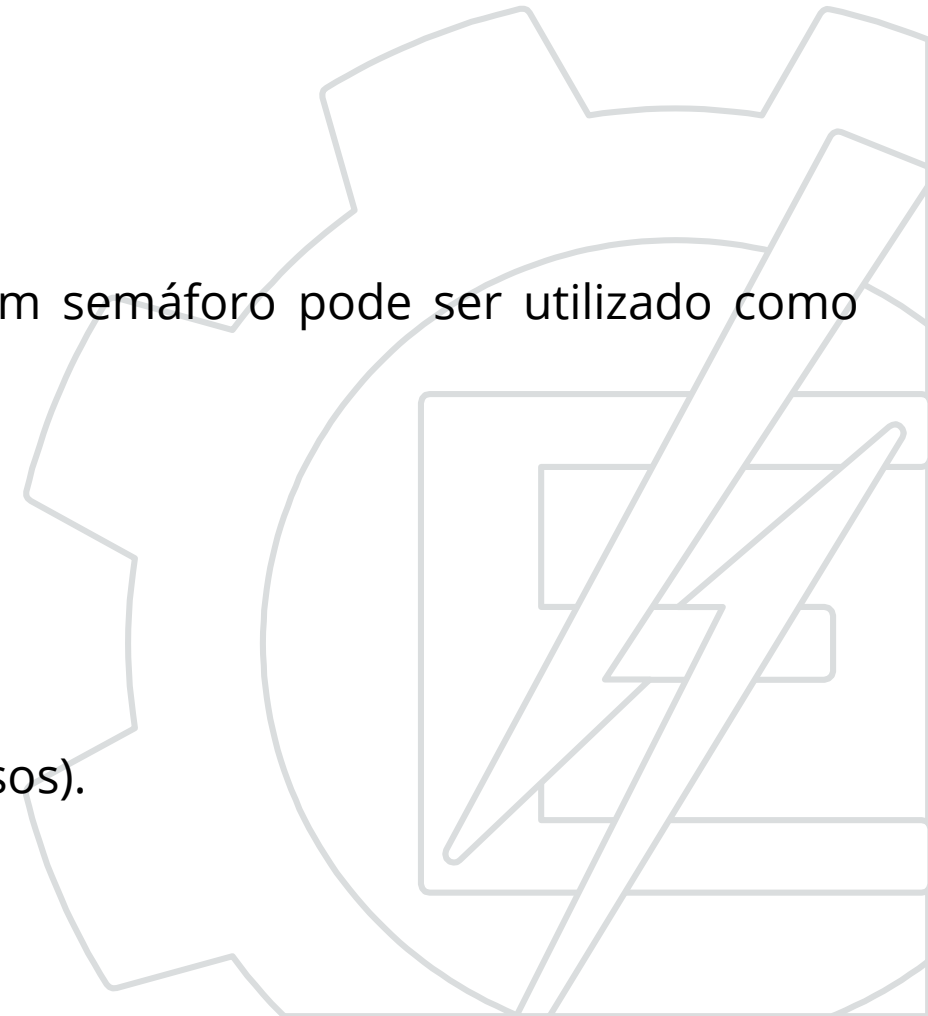
```
# include "prototypes.h"
# define N 100

typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer (void){
    int item;
    while (TRUE){
        produce_item(&item);
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
void consumer (void){
    int item;
    while (TRUE){
        down(&full);
        down(&mutex);
        remove_item(item);
        up(&mutex);
        up(&empty);
        consume_item(item);
    }
}
```


- Quando utilizar um ***mutex***?
 - É um mecanismo de trava para garantir o acesso a um recurso compartilhado.
 - Somente uma tarefa (*task*) pode manter a trava e liberá-la.
- Quando utilizar um **semáforo binário**?
 - É similar ao *mutex*, mas é um mecanismo de sinalização. Um semáforo pode ser utilizado como *mutex*, mas o contrário não é válido.
- Quando utilizar um **semáforo**?
 - Pode gerenciar mais de um dispositivo do mesmo tipo;
 - Pode ser utilizado como uma fila (compartilhamento de recursos).



Mutex vs. Semáforos

Mutex (**Mutual Exclusion**)

```
wait (mutex);  
.....  
Critical Section  
.....  
signal (mutex);
```

Semáforo

A operação *wait* decrementa o valor do argumento *S*, caso ele seja positivo. Caso ele seja negativo ou igual a zero, nenhuma operação é realizada.

```
wait(S)  
{  
    while (S<=0);  
    S--;  
}
```

A operação *signal* incrementa o valor de *S*.

```
signal(S)  
{  
    S++;  
}
```

- Espera ocupada (*busy waiting*)
- *Sleep / WakeUp* (primitivas - chamadas de sistema)
- Semáforos (variáveis de controle)
- **Monitores** (primitiva de alto nível)
- Troca de Mensagens



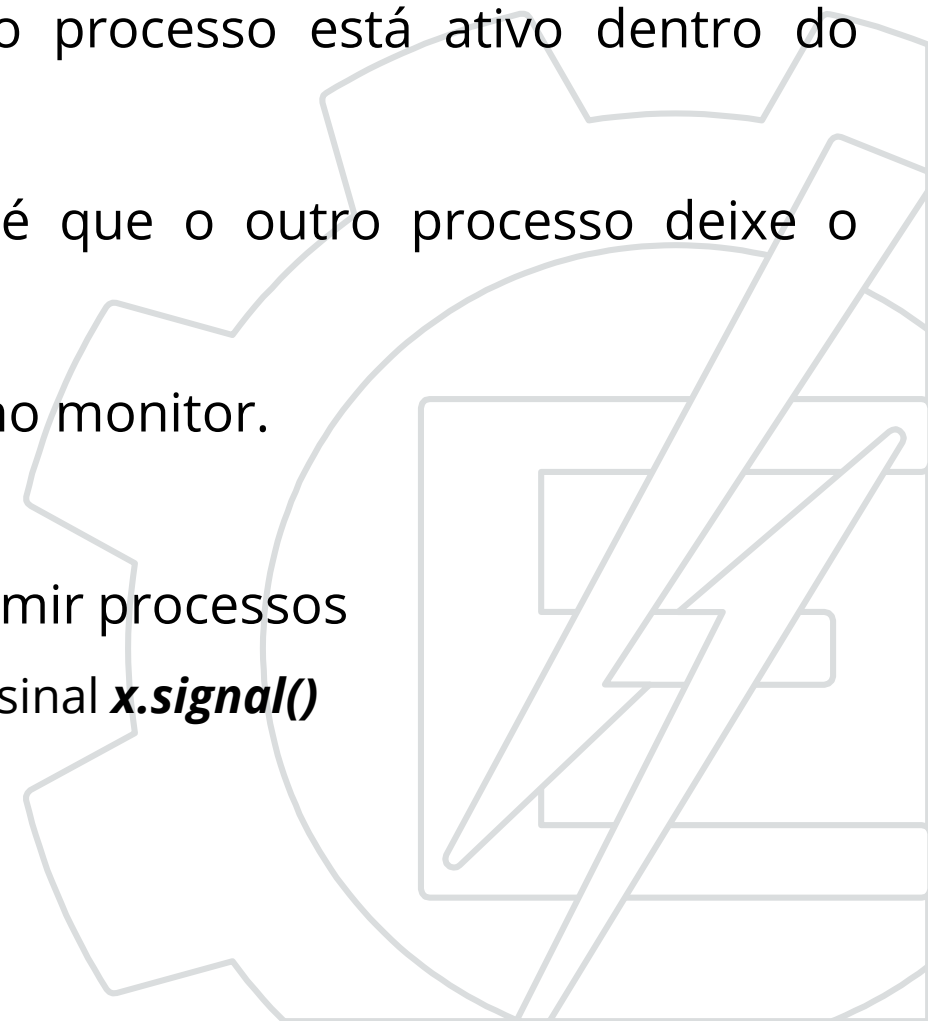
- Idealizado por Hoare (1974) e Brinch Hansen (1975).
- Primitiva de alto nível para sincronizar processos, conjunto de procedimentos, variáveis e estruturas de dados agrupados em um único módulo ou pacote;
- Somente um processo pode estar ativo no monitor ao mesmo tempo;
- Não são boas soluções para sistemas distribuídos, pois não provém sincronização para máquinas diferentes;
- Todos os recursos compartilhados entre processos devem estar implementados dentro do Monitor;
- Depende da linguagem de programação e o compilador é quem garante a exclusão mútua;
- Existente em Java. Não existe em C.

monitor example

```
int i;  
condition c;  
procedure A();  
.  
end;  
procedure B();  
.  
end;  
end monitor;
```

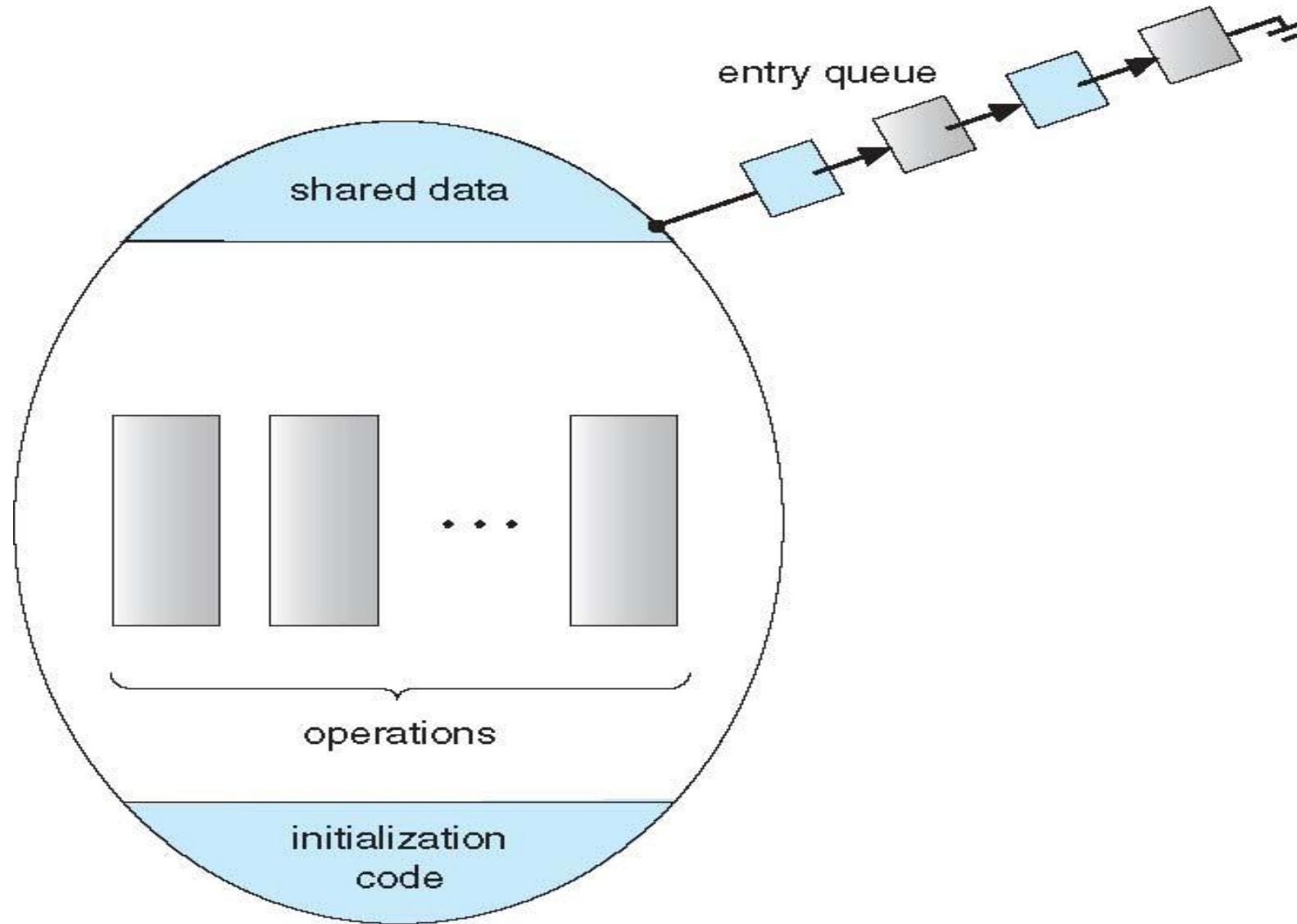
- Um **tipo de dado abstrato** — ou ADT (*abstract data type*) — encapsula dados com um conjunto de funções para operarem sobre esses dados, que são independentes de qualquer implementação específica do ADT.
- Um tipo monitor é um ADT que inclui um conjunto de operações definidas pelo programador e que são dotadas de **exclusão mútua** dentro do monitor.
- O tipo monitor também declara as variáveis cujos valores definem o estado de uma instância desse tipo, além dos corpos de funções que operam sobre essas variáveis.

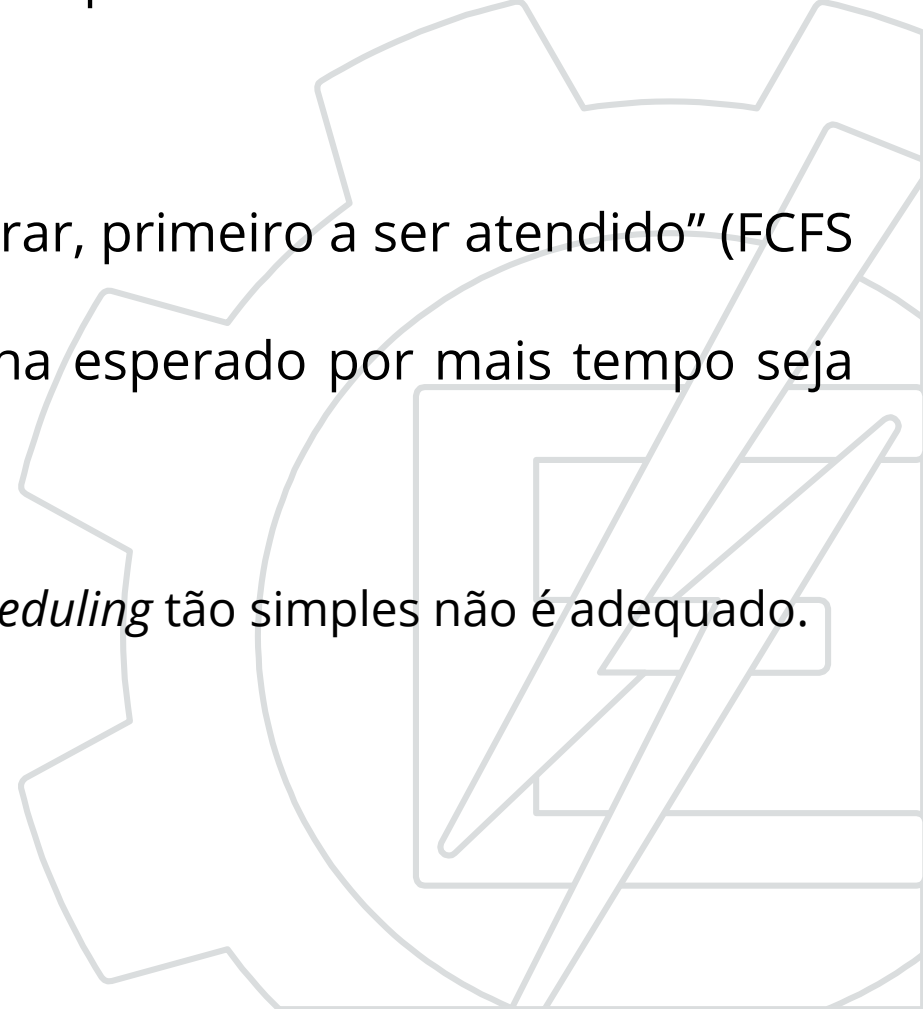
- Execução:
 - Chamada a uma rotina do monitor;
 - Instruções iniciais - teste para detectar se um outro processo está ativo dentro do monitor;
 - Se positivo, o processo novo ficará bloqueado até que o outro processo deixe o monitor;
 - Caso contrário, o processo novo executa as rotinas no monitor.
- Pode utilizar variáveis de condição para suspender e reassumir processos
 - ***x.wait()*** – o processo que invoca a operação é suspenso até o sinal ***x.signal()***
 - ***x.signal()*** – retoma um dos processos que invocou ***x.wait()***



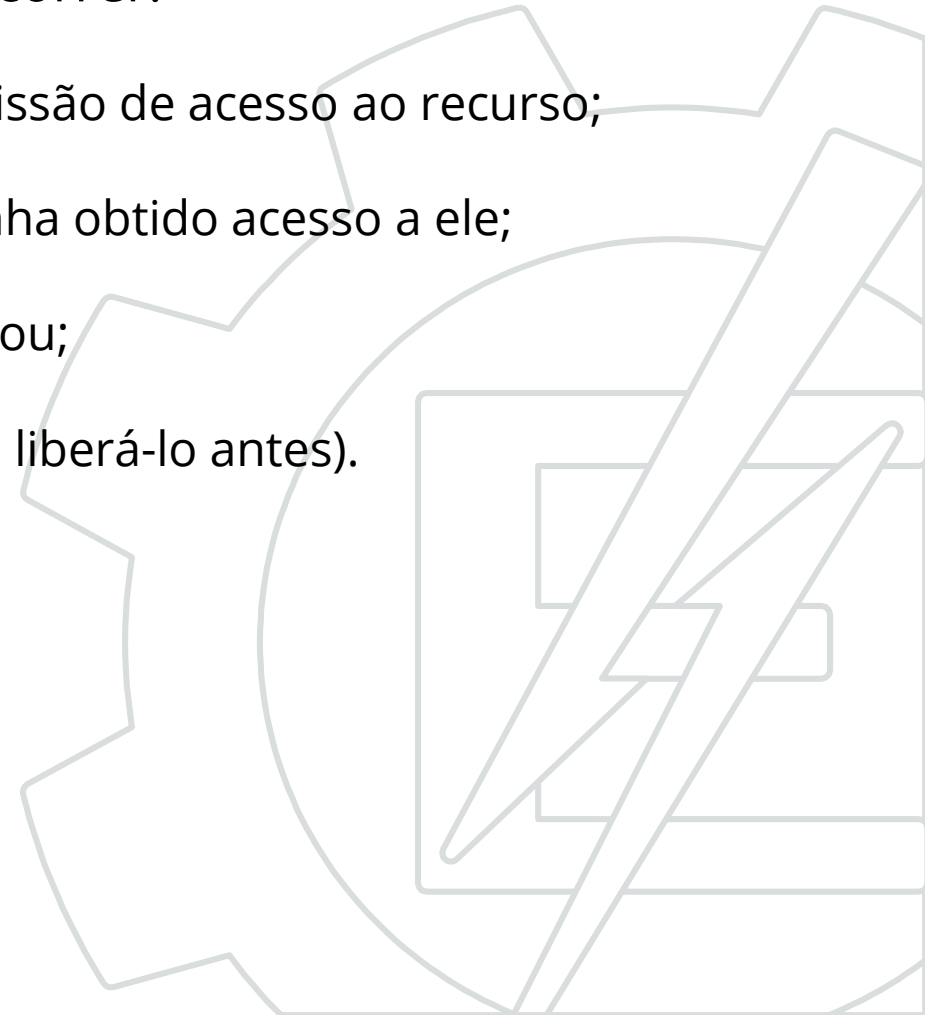
Monitores

Esquemático/Estrutura

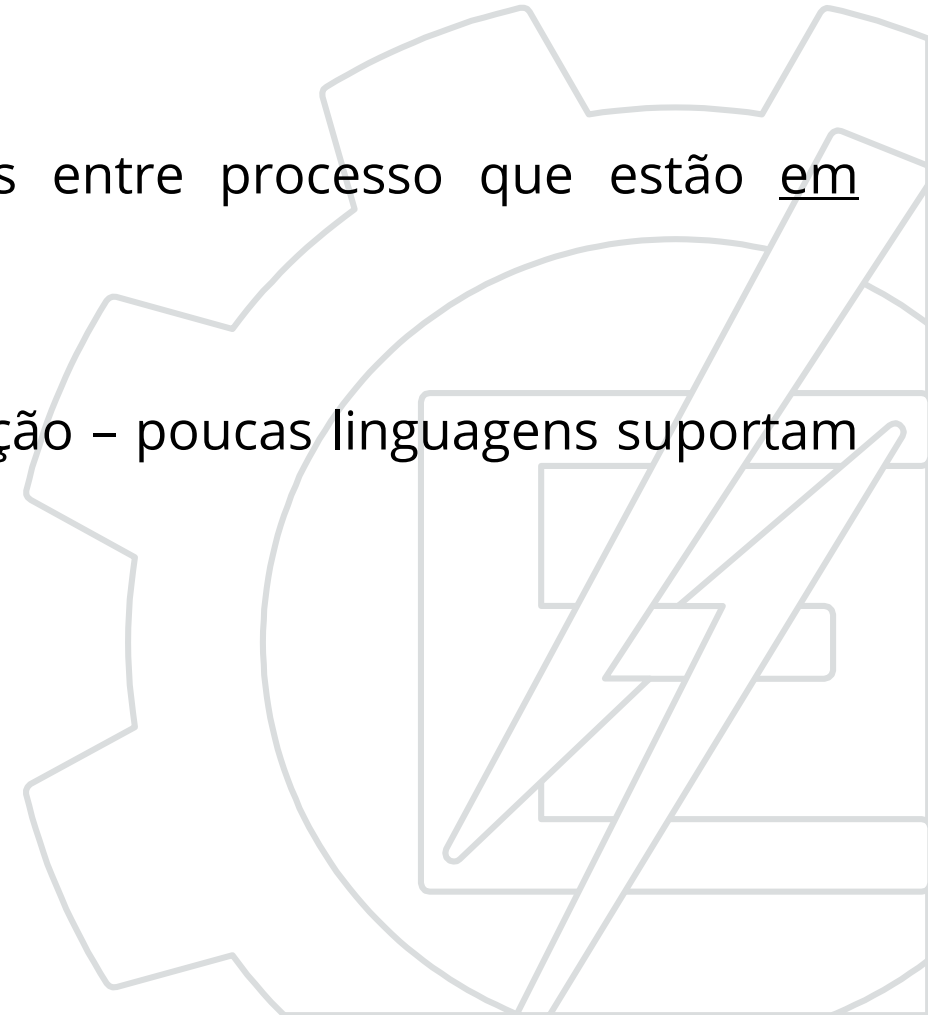


- Se vários processos estão suspensos na condição x , e uma operação $x.signal()$ é executada por algum processo, como determinar qual dos processos suspensos deve ser retomado em seguida?
 - Uma solução simples é usar uma ordem “primeiro a entrar, primeiro a ser atendido” (FCFS — *first-come, first-served*) para que o processo que tenha esperado por mais tempo seja retomado primeiro.
 - Em muitas circunstâncias, no entanto, um esquema de *scheduling* tão simples não é adequado.
- 

- Infelizmente, o conceito de monitor não garante que a sequência de acesso anterior seja seguida. Especificamente, os seguintes problemas podem ocorrer:
 - Um processo pode acessar um recurso sem antes obter permissão de acesso ao recurso;
 - Um processo pode nunca liberar um recurso, uma vez que tenha obtido acesso a ele;
 - Um processo pode tentar liberar um recurso que nunca solicitou;
 - Um processo pode solicitar o mesmo recurso duas vezes (sem liberá-lo antes).



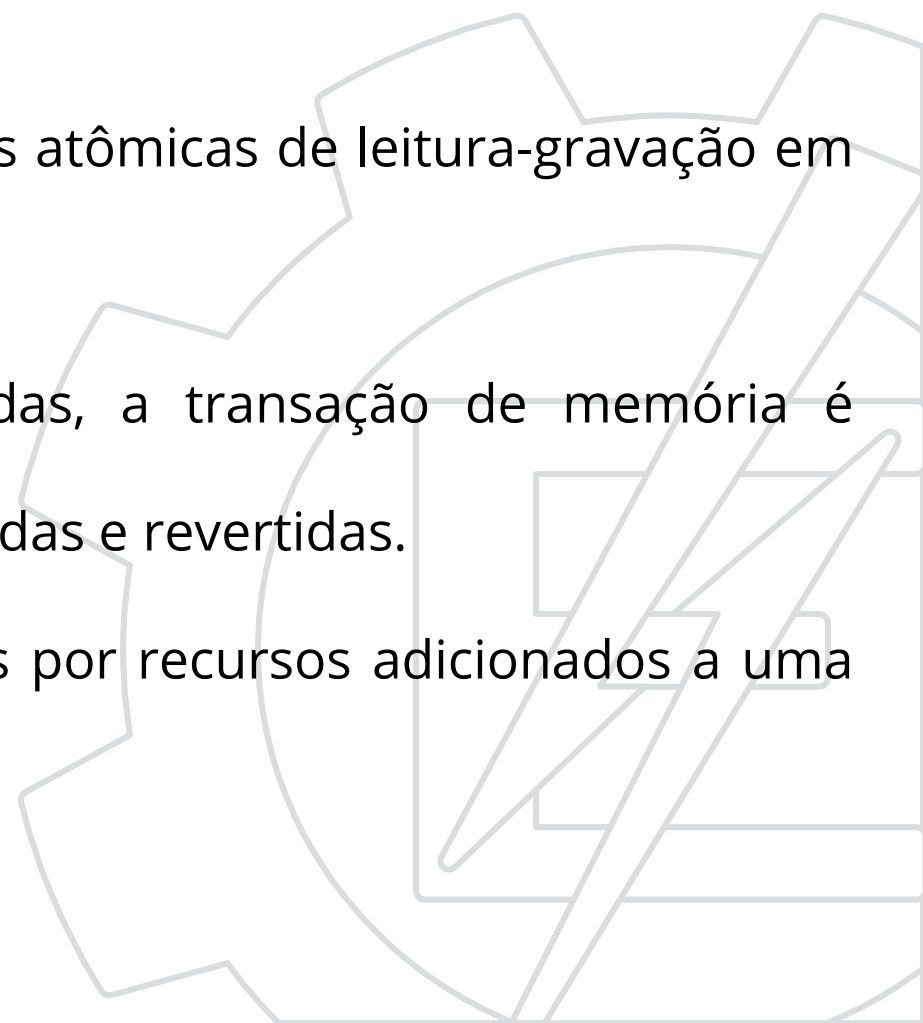
- Ambos são boas soluções somente para CPUs com memória compartilhada. Não são boas soluções para sistema distribuídos;
- Nenhuma das soluções provê troca de informações entre processo que estão em diferentes máquinas;
- Monitores dependem de uma linguagem de programação – poucas linguagens suportam Monitores.



- Espera ocupada (*busy waiting*)
- *Sleep / WakeUp* (primitivas - chamadas de sistema)
- Semáforos (variáveis de controle)
- Monitores (primitiva de alto nível)
- Troca de Mensagens



Memória Transacional

- O conceito de memória transacional teve origem na teoria de bancos de dados e fornece uma estratégia para a sincronização de processos.
 - Uma transação de memória é uma sequência de operações atômicas de leitura-gravação em memória.
 - Se todas as operações de uma transação são concluídas, a transação de memória é confirmada. Caso contrário, as operações devem ser abortadas e revertidas.
 - Os benefícios da memória transacional podem ser obtidos por recursos adicionados a uma linguagem de programação.
- 

Bibliografia

- TANENBAUM, Andrew S; BOS, Herbert. Sistemas operacionais modernos. 4a ed. São Paulo: Pearson Education do Brasil, 2016.

Capítulo 2.

<https://plataforma.bvirtual.com.br/Acervo/Publicacao/1233>

- DEITEL, H.M; DEITEL, P.J; CHOFFNES,D.R. Sistemas Operacionais. 3a ed. São Paulo: Pearson Prentice Hall, 2005. **Capítulos 5 e 6.**

<https://plataforma.bvirtual.com.br/Acervo/Publicacao/315>



Sistemas Operacionais

Prof. Otávio Gomes

otavio.gomes@unifei.edu.br

