SISTEMAS OPERACIONAIS

Comunicação e Sincronização de Processos (2)

Tipos de Soluções (cont.)

- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Dekker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Sleep / Wakeup
  - Semáforos
  - Monitores
  - Passagem de mensagem

As Primitivas Sleep e Wakeup

- A idéia desta abordagem é bloquear a execução dos processos quando a eles não é permitido entrar em suas regiões críticas
- Isto evita o desperdício de tempo de CPU, como nas soluções com *busy wait*.

- Sleep()
  - Bloqueia o processo e espera por uma sinalização, isto é, suspende a execução do processo que fez a chamada até que um outro o acorde.

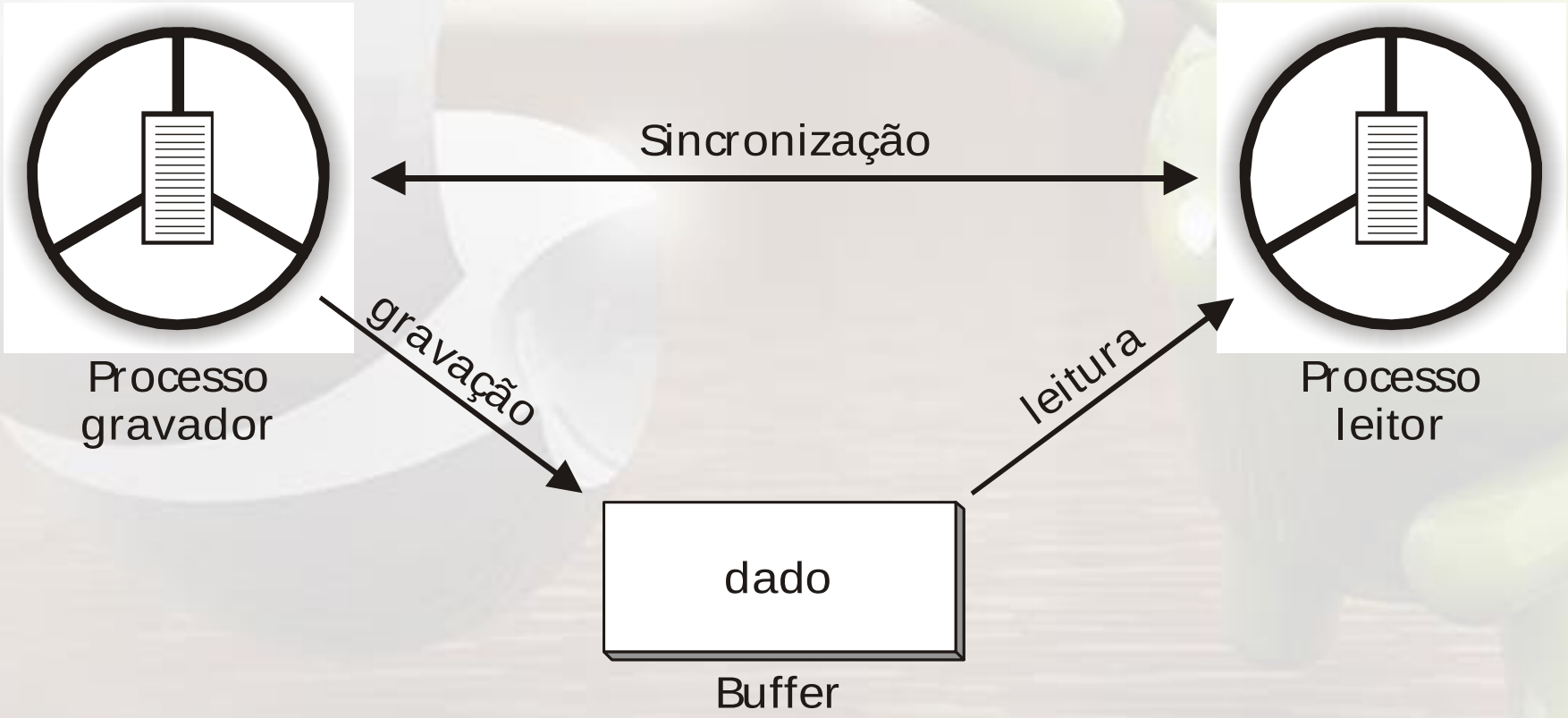
- Wakeup()
  - Sinaliza (acorda) o processo anteriormente bloqueado por Sleep().

O problema do Produtor e Consumidor

- Processo produtor gera dados e os coloca em um *buffer* de tamanho N.
- Processo consumidor retira os dados do buffer, na mesma ordem em que foram colocados, um de cada vez.
  - Se o *buffer* está **cheio**, o produtor deve ser bloqueado
  - Se o *buffer e*stá **vazio**, o consumidor é quem deve ser bloqueado.

- Apenas um único processo, produtor ou consumidor, pode acessar o *buffer* num certo instante.
  - Uso de *Sleep* e *Wakeup* para o problema do Produtor e Consumidor

O problema do Produtor e Consumidor



Comunicação de Processos – Primitivas Sleep/Wakeup

- Buffer: variável `count` controla a quantidade de dados presente no buffer.

- Produtor:
  - Se `count` = valor máximo (buffer cheio)
    - Então processo produtor é colocado para dormir
    - Senão produtor coloca dados no buffer e incrementa `count`

Comunicação de Processos – Primitivas Sleep/Wakeup

- Consumidor:
  - Se count = 0 (buffer vazio)
    - Então processo vai "dormir"
    - Senão retira os dados do buffer e decrementa count

Primitivas Sleep/Wakeup

```
#define N 100                          /* number of slots in the buffer */
int count = 0;                         /* number of items in the buffer */

void producer(void) {
 while (true){
  produce_item();                      /* generate next item */
  if (count == N) sleep();             /* if buffer is full, go to sleep */
  enter_item();                        /* put item in buffer */
  count = count + 1;                   /* increment count of items in buffer*/
  if (count == 1) wakeup(consumer);    /* was buffer empty? */
 }
}

void consumer(void){
 while (true){
  if (count == 0) sleep();             /* if buffer is empty, got to sleep */
  remove_item();                       /* take item out of buffer */
  count = count - 1;                  /* decrement count of items in buffer*/
  if (count == N-1) wakeup(producer); /* was buffer full? */
  consume_item();                      /* print item */
 }
}
```

Exemplo 1

- producerConsumer.cpp

Uma Condição de Corrida ...

- *Buffer* está vazio.
- Consumidor testa o valor de *count*, que é zero, mas não tem tempo de executar *sleep*, pois o escalonador selecionou agora produtor.
- Este produz um item, insere-o no *buffer* e incrementa *count*.
  - Como *count* = 1, produtor chama *wakeup* para acordar consumidor.
- O sinal não tem efeito (é perdido), pois o consumidor ainda não está logicamente adormecido.
- Consumidor ganha a CPU, executa *sleep* e vai dormir.
- Produtor ganha a CPU e, cedo ou tarde, encherá o *buffer*, indo também dormir.
- Ambos dormirão eternamente.

Tipos de Soluções (cont.)

- Soluções de Hardware
  - Inibição de interrupções
  - Instrução TSL (apresenta *busy wait*)
- Soluções de software com *busy wait*
  - Variável de bloqueio
  - Alternância estrita
  - Algoritmo de Dekker
  - Algoritmo de Peterson
- Soluções de software com bloqueio
  - Sleep / Wakeup
  - Semáforos
  - Monitores
  - Passagem de mensagem

Semáforos (1)

- Mecanismo criado pelo matemático holandês E.W. Dijkstra, em 1965.
- O semáforo é uma **variável inteira** que pode ser mudada por apenas duas operações primitivas (atômicas): ***P*** e ***V***.
  - *P* = *proberen* (testar)
  - V = *verhogen* (incrementar).
- Quando um processo executa uma operação ***P***, o valor do semáforo é **decrementado** (se o semáforo for maior que 0). O processo pode ser eventualmente bloqueado (semáforo for igual a 0) e inserido na **fila de espera** do semáforo.
- Numa operação *V*, o semáforo é **incrementado** e, eventualmente, um processo que aguarda na **fila de espera** deste semáforo é acordado.

Semáforos (2)

- P também é comumente referenciada como:
  - *down* ou *wait*
- V também é comumente referenciada
  - *up* ou *signal*
- Um semáforo pode ter valor 0 ou menor que 0 quando não recursos disponível (região crítica bloqueada) ou um valor positivo referente ao número recursos disponíveis;
- Semáforos que assumem somente os valores 0 e 1 são denominados semáforos binários ou *mutex*. Neste caso, P e V são também chamadas de *LOCK* e *UNLOCK*, respectivamente.

Semáforos (3)


```
down(S):
        If S > 0
                Then S := S – 1
        Else bloqueia processo (coloca-o na fila de S)

up(S):
        If algum processo dorme na fila de S
                Then acorda processo
        Else S := S + 1
```

Comunicação de Processos – Semáforos

- ***Down(wait)***: verifica se o valor do semáforo é maior do que 0; se for, o semáforo é decrementado e o processo entra na região crítica (existe recurso disponível); se o valor for 0 ou menor que 0, o semáforo é decrementado e o processo é colocado para dormir sem completar sua operação de ***down***;
  - Todas essas ações são chamadas de **ações atômicas**;
  - **Ações atômicas** garantem que quando uma operação no semáforo está sendo executada, nenhum processo pode acessar o semáforo até que a operação seja finalizada ou bloqueada;
    - Desativa interrupções e usa TSL para garantir que apenas uma CPU acesse um semáforo por vez.

Comunicação de Processos – Semáforos

- ***Up(signal)***: incrementa o valor do semáforo. Se o novo valor for 1 faz com que algum processo que esteja dormindo possa terminar de executar sua operação ***down***. Se o novo valor ainda for menor que 0 continua preempção.

- ***Semáforo Mutex***: garante a exclusão mútua, não permitindo que os processos acessem uma região crítica ao mesmo tempo
  - Também chamado de **semáforo binário**

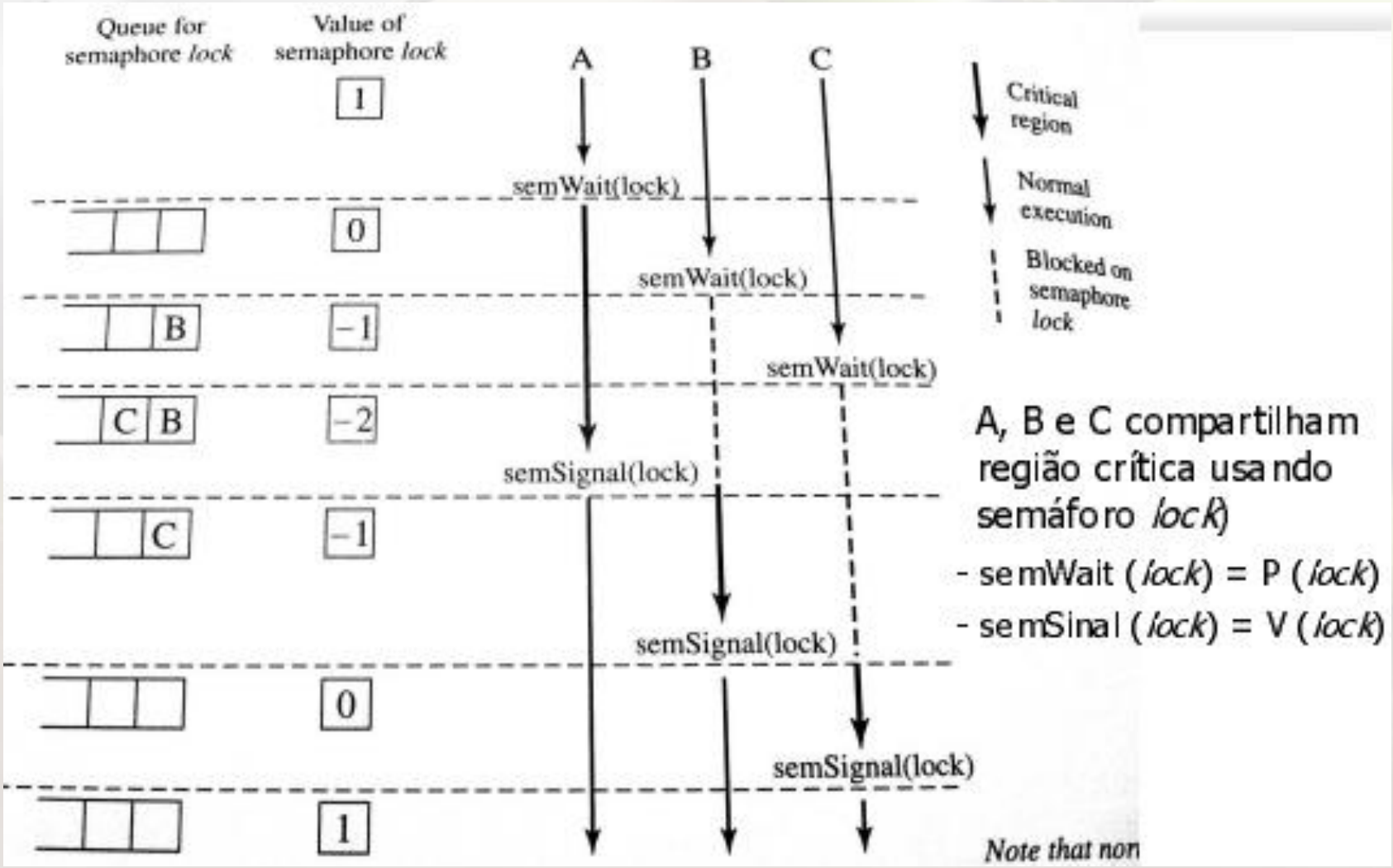
Uso de Semáforos (1)

- Exclusão mútua (semáforos binários):

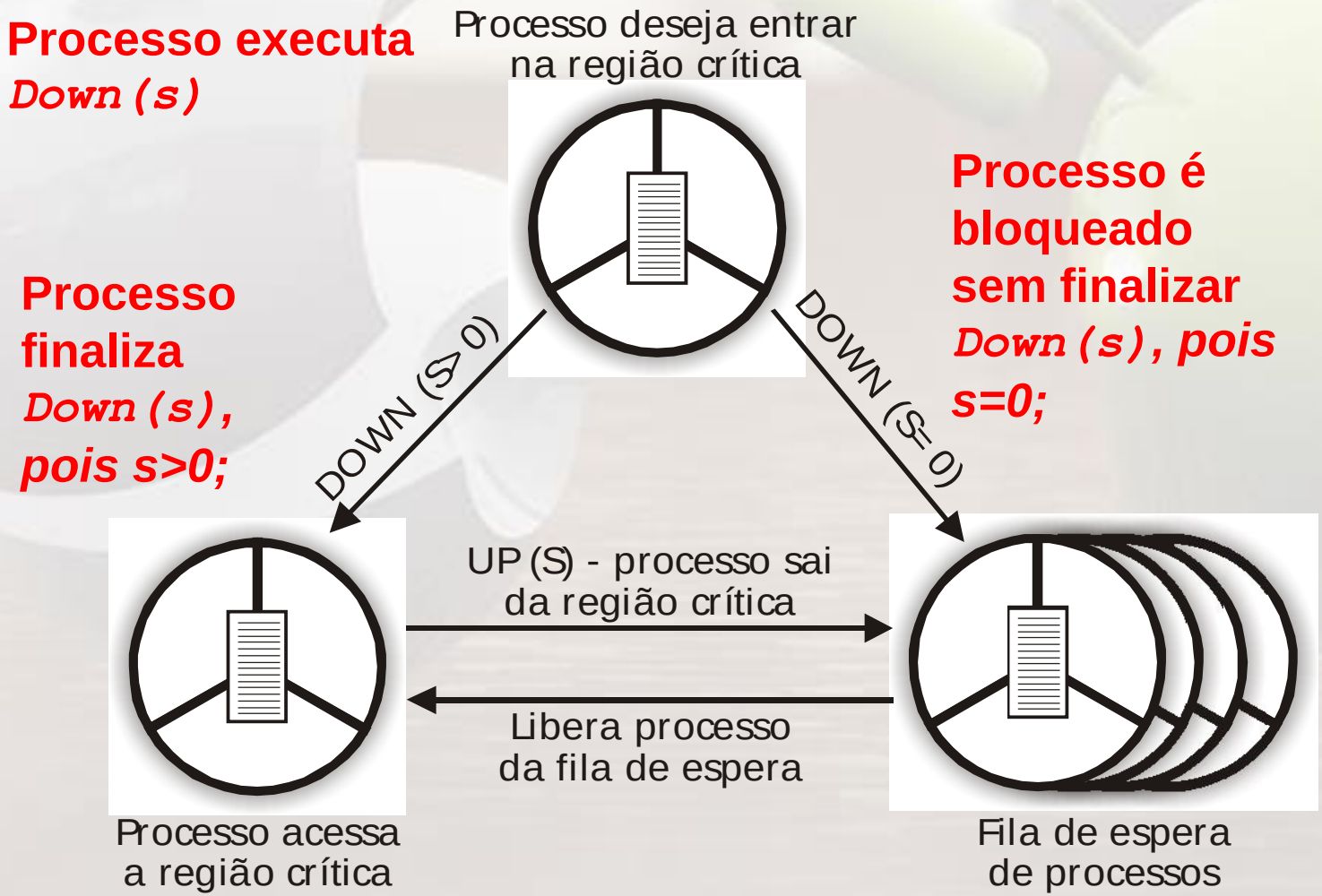
```
...
Semaphore mutex = 1;              /*var.semáforo,
                                  iniciado com 1*/

Processo P1     Processo P2   ...   Processo Pn
...             ...                 ...
P(mutex)        P(mutex)            P(mutex)
// R.C.         // R.C.              // R.C.
V(mutex)        V(mutex)            V(mutex)
...             ...                 ...
```

Uso de Semáforos (1)



Uso de Semáforos (1)



Uso de Semáforos (2)

- Alocação de Recursos (semáforos contadores):

```
...
Semaphore S := 3;    /*var. semáforo, iniciado com
                       qualquer valor inteiro */

Processo P1          Processo P2          Processo P3
...                  ...                       ...
P(S)                 P(S)                 P(S)
//usa recurso        //usa recurso        //usa recurso
V(S)                 V(S)                 V(S)
...                  ...                  ...
```

Uso de Semáforos (3)

- Relação de precedência entre processos:
  - Ex: executar p1_rot2 somente depois de p0_rot1

```
semaphore S :=0  ;
parbegin
      begin                          /* processo P0*/
            p0_rot1()
            V(S)
            p0_rot2()
      end
      begin                          /* processo P1*/
            p1_rot1()
            P(S)
            p1_rot2()
      end
parend
```

Comunicação de Processos – Semáforos

- 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; também chamado de **semáforo binário**; Permite a exclusão mútua;

```
#define N 100                                   /* número de lugares no buffer */
typedef int semaphore;                          /* semáforos são um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso à região crítica */
semaphore empty = N;                            /* conta os lugares vazios no buffer */
semaphore full = 0;                             /* conta os lugares preenchidos no buffer */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE é a constante 1 */
          item = produce_item( );               /* gera algo para pôr no buffer */
          down(&empty);                         /* decresce o contador empty */
          down(&mutex);                         /* entra na região crítica */
          insert_item(item);                    /* põe novo item no buffer */
          up(&mutex);                           /* sai da região crítica */
          up(&full);                            /* incrementa o contador de lugares preenchidos */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* laço infinito */
          down(&full);                          /* decresce o contador full */
          down(&mutex);                         /* entra na região crítica */
          item = remove_item( );                /* pega item do buffer */
          up(&mutex);                           /* sai da região crítica */
          up(&empty);                           /* incrementa o contador de lugares vazios */
          consume_item(item);                   /* faz algo com o item */
     }
```

Exemplo 2

- producerConsumerSemaphores.cpp

Deficiência dos Semáforos (1)

- Exemplo: suponha que os dois down do código do produtor estivessem invertidos.
  - Neste caso, mutex seria diminuído antes de empty. Se o buffer estivesse completamente cheio, o produtor bloquearia com mutex = 0.
  - Portanto, da próxima vez que o consumidor tentasse acessar o buffer ele faria um down em mutex, agora zero, e também bloquearia.
  - Os dois processos ficariam bloqueados eternamente.
- Conclusão: erros de programação com semáforos podem levar a resultados imprevisíveis.

Referências

- A. S. Tanenbaum, "Sistemas Operacionais Modernos", 3a. Edição, Editora Prentice-Hall, 2010.
  - Seções 2.3.4 a 2.3.6
- Silberschatz A. G.; Galvin P. B.; Gagne G.; "Fundamentos de Sistemas Operacionais", 6a. Edição, Editora LTC, 2004.
  - Seção 7.4
- Deitel H. M.; Deitel P. J.; Choffnes D. R.; "Sistemas Operacionais", 3ª. Edição, Editora Prentice-Hall, 2005
  - Seção 5.6