Universidade Federal da Bahia

Sistemas Operacionais

MATA58

Prof. Maycon Leone M. Peixoto

mayconleone@dcc.ufba.br

Processos

- Introdução
- Escalonamento de Processos
- **Comunicação entre Processos**
  - **Condição de Disputa**
  - **Região Crítica**
  - **Formas de Exclusão Mútua**
  - **Problemas Clássicos**
- Threads
- Deadlock

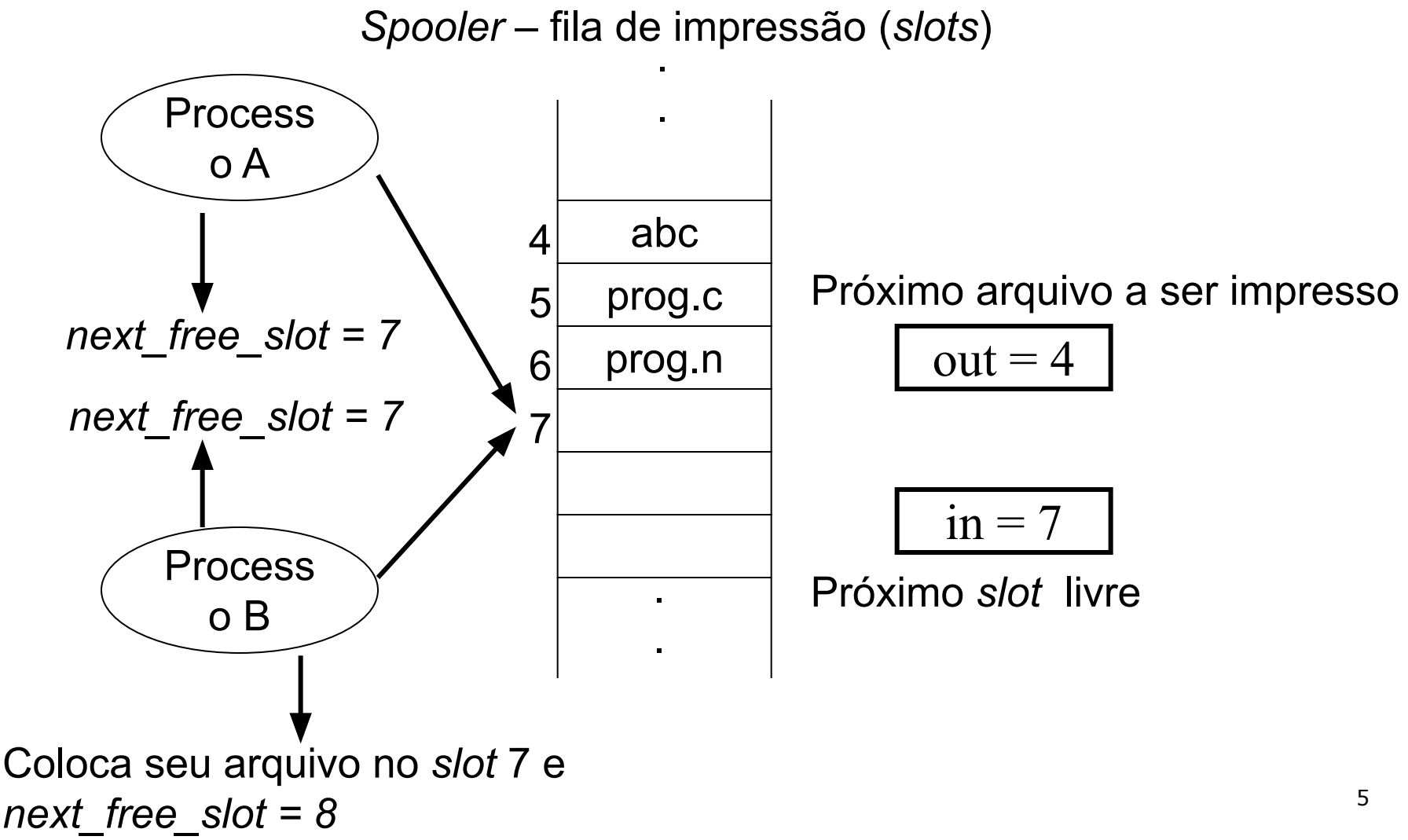
Comunicação de Processos

- Processos precisam se comunicar;
- Processos competem por recursos
- Três aspectos importantes:
  - Como um processo passa informação para outro processo;
  - Como garantir que processos não invadam espaços uns dos outros;
  - Dependência entre processos: seqüência adequada;

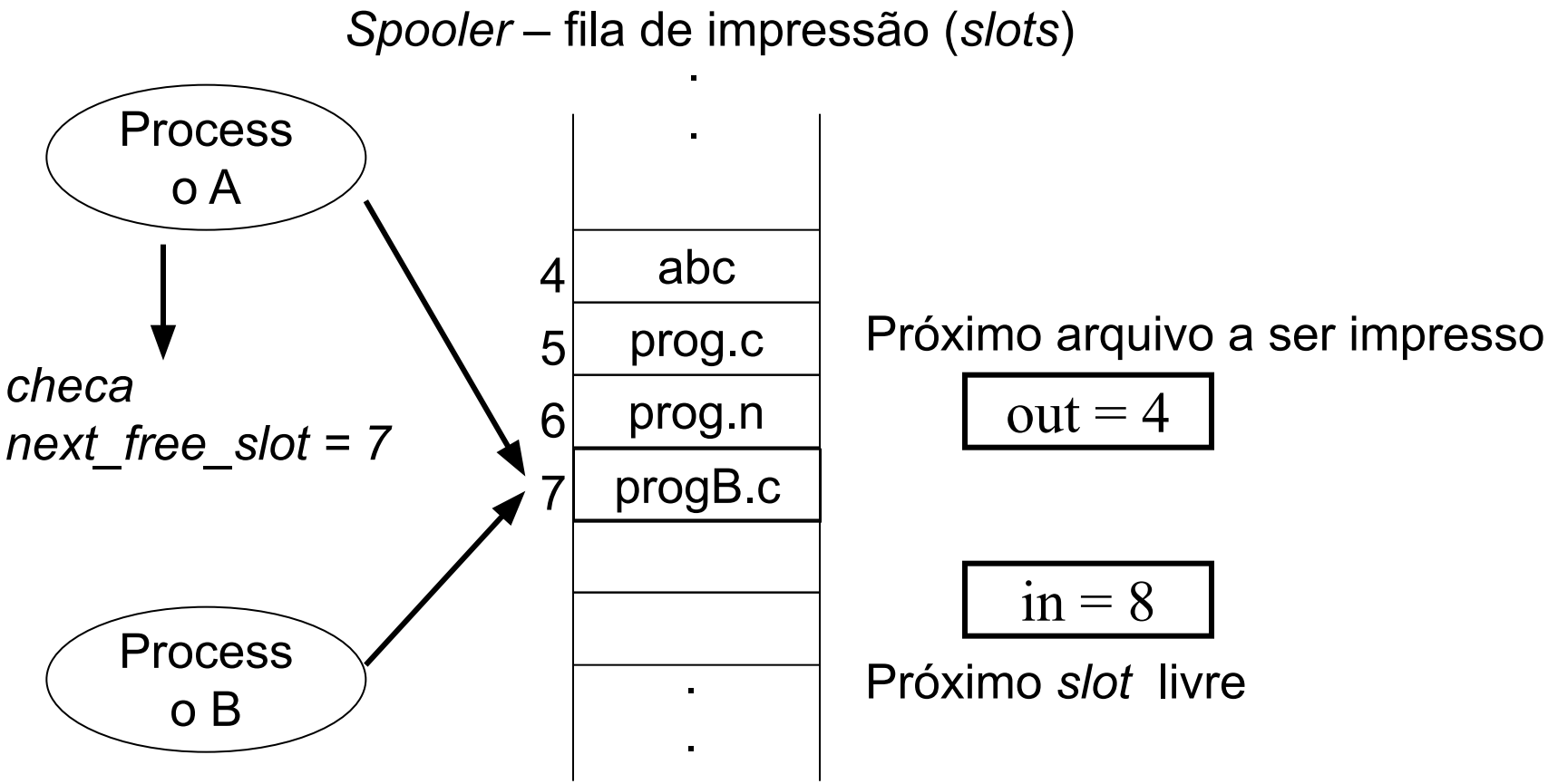
Comunicação de Processos – *Race Conditions*

- *Race Conditions*: processos acessam recursos compartilhados concorrentemente;
  - Recursos: memória, arquivos, impressoras, discos, variáveis;
- Ex.: Impressão: quando um processo deseja imprimir um arquivo, ele coloca o arquivo em um local especial chamado ***spooler*** (tabela). Um outro processo, chamado ***printer spooler***, checa se existe algum arquivo a ser impresso. Se existe, esse arquivo é impresso e retirado do *spooler*. Imagine dois processos que desejam ao mesmo tempo imprimir um arquivo...

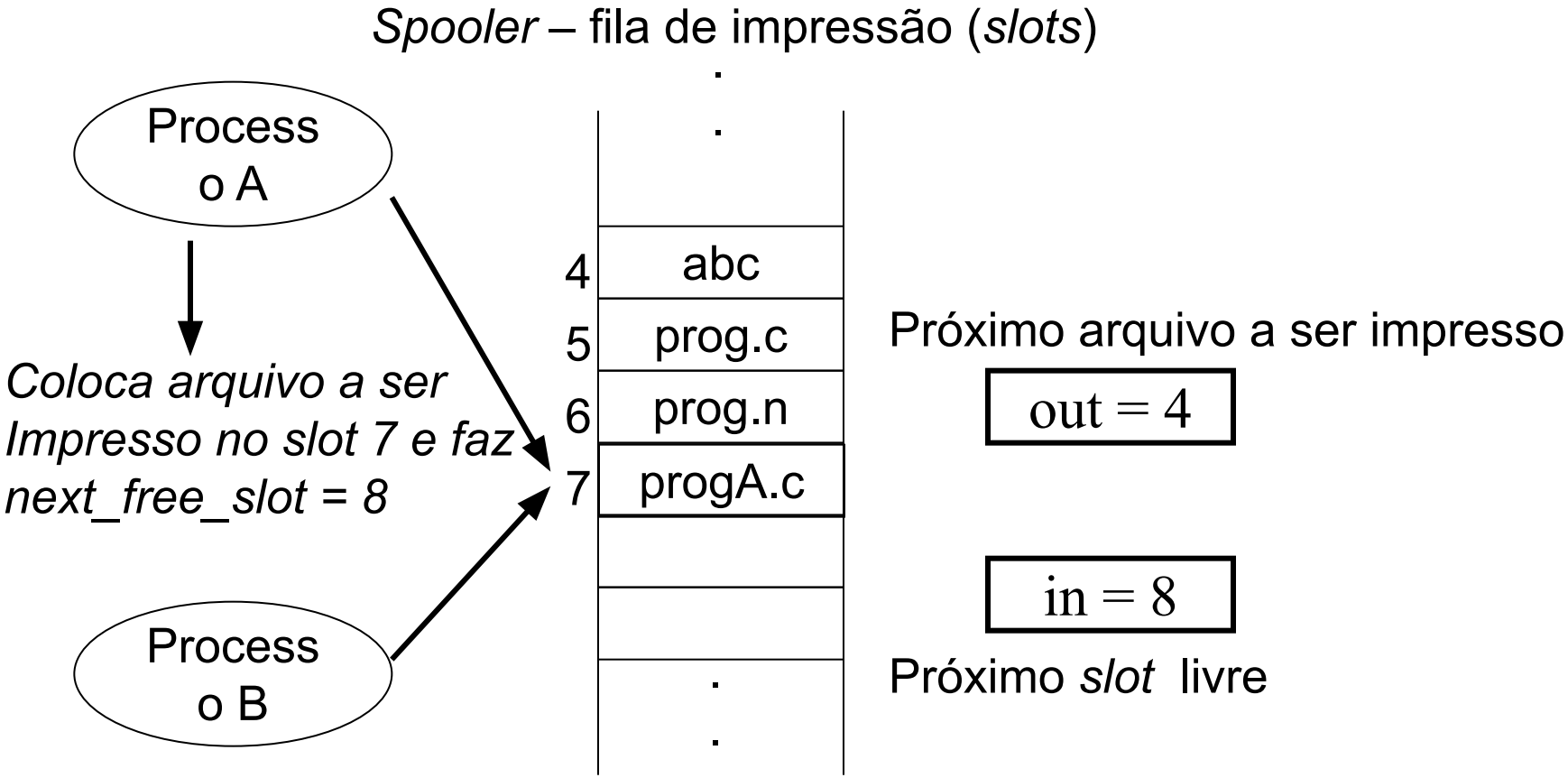
Comunicação de Processos - *Race Conditions*



Comunicação de Processos - *Race Conditions*

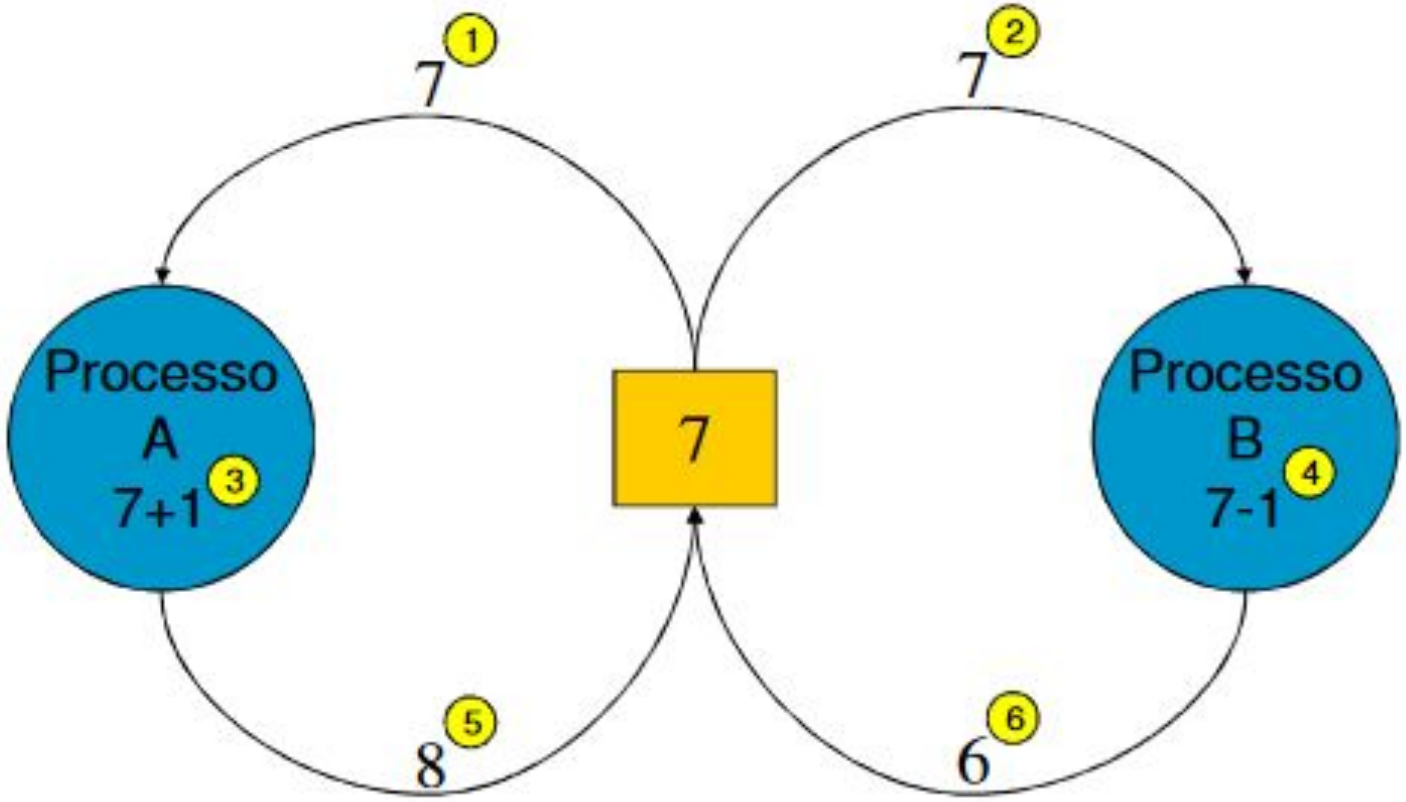


Comunicação de Processos - *Race Conditions*



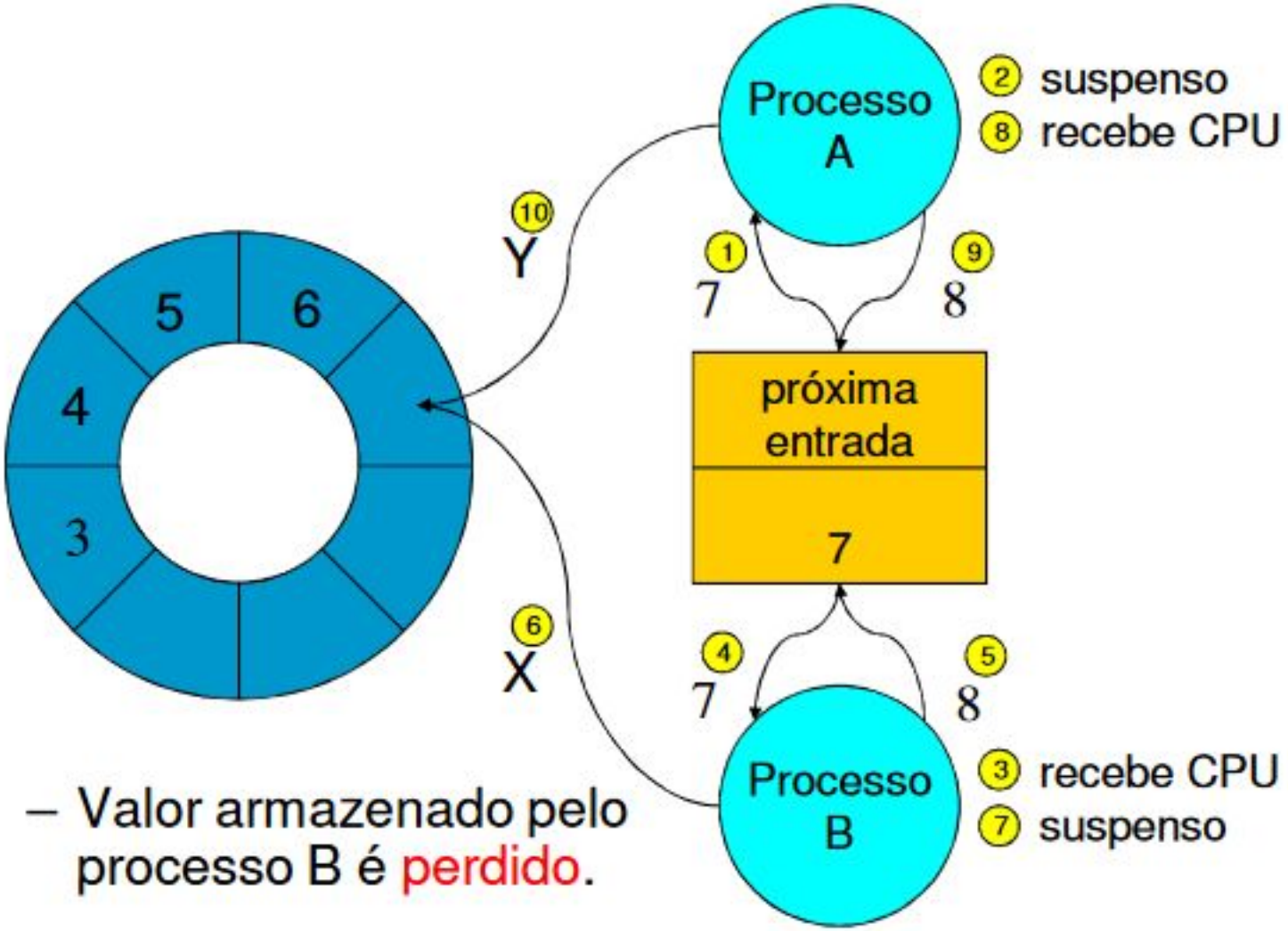
Processo B nunca receberá sua impressão!!!!!

Comunicação de Processos - *Race*



– Resultado Final: Contador = 6 (ERRO!)

Comunicação de Processos - *Race Conditions*



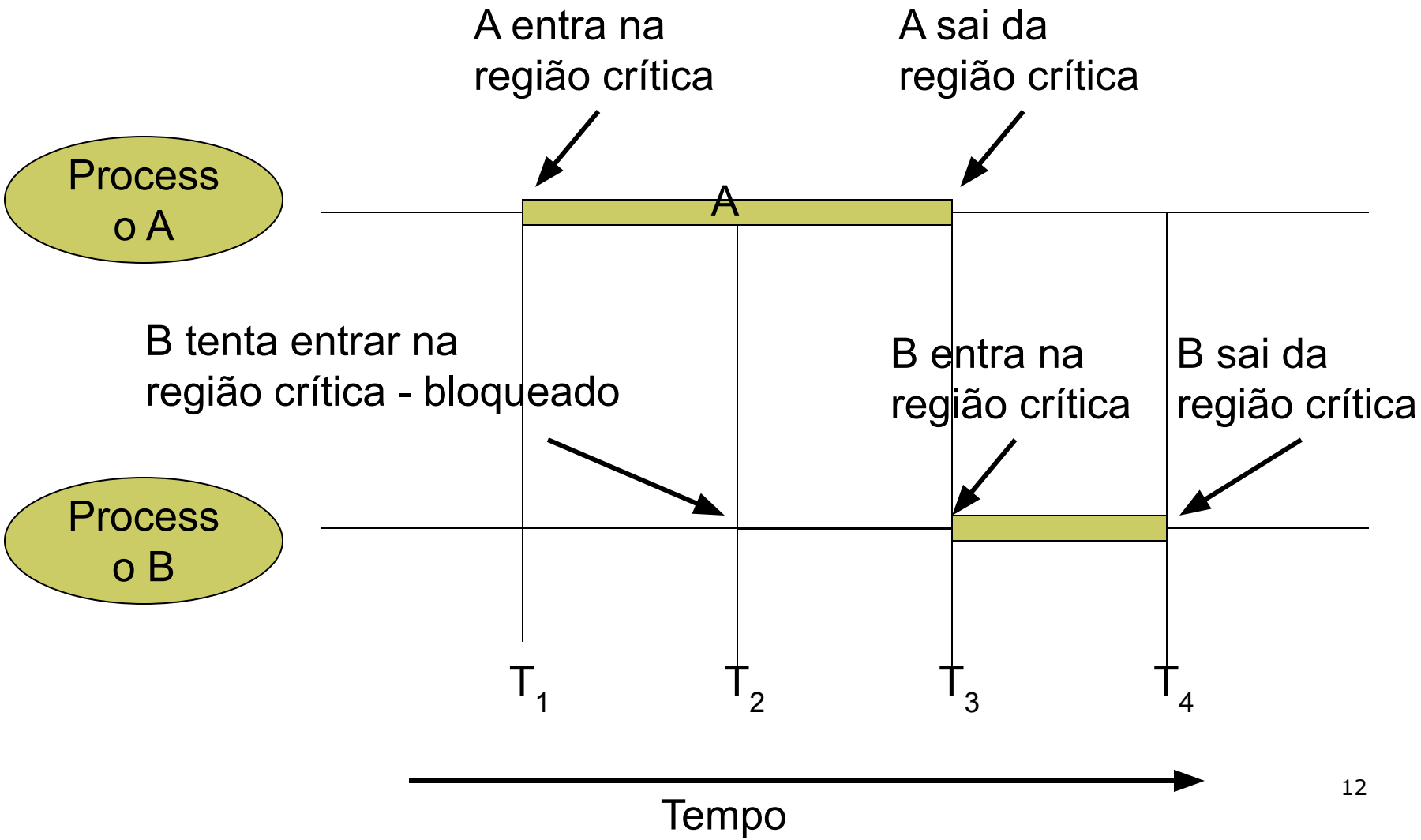
Comunicação de Processos – Regiões Críticas

- Como solucionar problemas de *Race Conditions*???
  - Proibir que mais de um processo leia ou escreva em recursos compartilhados concorrentemente (ao "mesmo tempo")
    - **Recursos compartilhados □ regiões críticas**;
  - Exclusão mútua: garantir que um processo não terá acesso à uma região crítica quando outro processo está utilizando essa região;

Comunicação de Processos – Exclusão Mútua

- Quatro condições para uma boa solução:
  1. Dois processos não podem estar simultaneamente em regiões críticas;
  2. Nenhuma restrição deve ser feita com relação à CPU;
  3. Processos que não estão em regiões críticas não podem bloquear outros processos que desejam utilizar regiões críticas;
  4. Processos não podem esperar para sempre para acessarem regiões críticas;

Comunicação de Processos – Exclusão Mútua



Soluções

- Exclusão Mútua:
  - **Espera Ocupada**;
  - Primitivas *Sleep/Wakeup*;
  - Semáforos;
  - Monitores;
  - Passagem de Mensagem;

Comunicação de Processos – Exclusão Mútua

- Espera Ocupada (*Busy Waiting)*: constante checagem por algum valor;
- Algumas soluções para Exclusão Mútua com Espera Ocupada:
  - Desabilitar interrupções;
  - Variáveis de Travamento (*Lock)*;
  - Estrita Alternância (*Strict Alternation)*;
  - Solução de Peterson e Instrução TSL;

Comunicação de Processos – Exclusão Mútua

- Desabilitar interrupções:
  - Processo desabilita todas as suas interrupções ao entrar na região crítica e habilita essas interrupções ao sair da região crítica;
  - Com as interrupções desabilitadas, a CPU não realiza chaveamento entre os processos;
    - Viola condição 2 (Nenhuma restrição deve ser feita com relação à CPU);
  - Não é uma solução segura, pois um processo pode não habilitar novamente suas interrupções e não ser finalizado;
    - Viola condição 4 (Processos não podem esperar para sempre para acessarem regiões críticas);

Comunicação de Processos – Exclusão Mútua

- Variáveis *Lock*:
  - O processo que deseja utilizar uma região crítica atribuí um valor a uma variável chamada *lock*;
  - Se a variável está com valor 0 (zero) significa que nenhum processo está na região crítica; Se a variável está com valor 1 (um) significa que existe um processo na região crítica;
  - Apresenta o mesmo problema do exemplo do *spooler de impressão*;

Comunicação de Processos – Exclusão Mútua

- Variáveis *Lock* - Problema:
  - Suponha que um processo A leia a variável *lock* com valor 0;
  - Antes que o processo A posso alterar a variável para o valor 1, um processo B é escalonado e altera o valor de *lock* para 1;
  - Quando o processo A for escalonado novamente, ele altera o valor de *lock* para 1, e ambos os processos estão na região crítica;
    - Viola condição 1 (Dois processos não podem estar simultaneamente em regiões críticas);

Comunicação de Processos – Exclusão Mútua

- Variáveis *Lock:* ***lock==0;***

```
while(true){
    while(lock!=0); //loop
    lock=1;
    critical_region();
    lock=0;
    non-critical_region();
}
```

Processo A

```
while(true){
    while(lock!=0); //loop
    lock=1;
    critical_region();
    lock=0;
    non-critical_region();
}
```

Processo B

Comunicação de Processos – Exclusão Mútua

- *Strict Alternation*:
  - Fragmentos de programa controlam o acesso às regiões críticas;
  - Variável `turn`, inicialmente em `0`, estabelece qual processo pode entrar na região crítica;

```
while (TRUE) {
    while (turn!=0); //loop
        critical_region();
        turn = 1;
        noncritical region();}
```

(Processo A)

`turn` **0**

```
while (TRUE){
    while (turn!=1); //loop
        critical_region();
        turn = 0;
        noncritical region();}
```

(Processo B)

`turn` **1**

Comunicação de Processos – Exclusão Mútua

- Problema do *Strict Alternation:*

1. Suponha que o Processo B é mais rápido e sai da região crítica;
2. Ambos os processos estão fora da região crítica e `turn` com valor 0;
3. O processo A termina antes de executar sua região não crítica e retorna ao início do *loop*; Como o `turn` está com valor zero, o processo A entra novamente na região crítica, enquanto o processo B ainda está na região não crítica;
4. Ao sair da região crítica, o processo A atribui o valor 1 à variável `turn` e entra na sua região não crítica;

```
while (TRUE) {
    while (turn!=0); //loop
        critical_region();
        turn = 1;
        noncritical region();}
```

(Processo A)

`turn` **0**

```
while (TRUE){
    while (turn!=1); //loop
        critical_region();
        turn = 0;
        noncritical region();}
```

(Processo B)

`turn` **1**

Comunicação de Processos – Exclusão Mútua

- Problema do *Strict Alternation:*
  5. Novamente ambos os processos estão na região não crítica e a variável `turn` está com valor 1;
  6. Quando o processo A tenta novamente entrar na região crítica, não consegue, pois `turn` ainda está com valor 1;
  7. **Assim, o processo A fica bloqueado pelo processo B que NÃO está na sua região crítica, violando a condição 3**;

```
while (TRUE) {
    while (turn!=0); //loop
        critical_region();
        turn = 1;
        noncritical region();}
```

(Processo A)

`turn` **0**

```
while (TRUE){
    while (turn!=1); //loop
        critical_region();
        turn = 0;
        noncritical region();}
```

(Processo B)

`turn` **1**

Comunicação de Processos – Exclusão Mútua

- Solução de Peterson e Instrução TSL (*Test and Set Lock*):
  - Uma variável (ou programa) é utilizada para bloquear a entrada de um processo na região crítica quando um outro processo está na região;
  - Essa variável é compartilhada pelos processos que concorrem pelo uso da região crítica;
  - Ambas as soluções possuem fragmentos de programas que controlam a entrada e a saída da região crítica;

Comunicação de Processos – Exclusão Mútua

- Solução de Peterson

```
//flag[2] é booleana; e turn é um inteiro
flag[0]   = 0;
flag[1]   = 0;
turn;
```

```
P0: flag[0] = 1;
    turn = 1;
    while (flag[1] == 1 && turn == 1)
    {
        // ocupado, espere
    }
    // parte crítica
    ...
    // fim da parte crítica
    flag[0] = 0;
```

```
P1: flag[1] = 1;
    turn = 0;
    while (flag[0] == 1 && turn == 0)
    {
        // ocupado, espere
    }
    // parte crítica
    ...
    // fim da parte crítica
    flag[1] = 0;
```

Comunicação de Processos – Exclusão Mútua

- Instrução TSL: utiliza registradores do hardware;
  - TSL RX, LOCK; (lê o conteúdo de ***lock*** em RX, e armazena um valor diferente de zero (0) em ***lock*** – operação indivisível);
  - ***Lock*** é compartilhada
    - Se ***lock***==0, então região crítica "liberada".
    - Se ***lock***<>0, então região crítica "ocupada".

```
enter_region:
    TSL REGISTER, LOCK          | Copia lock para reg. e lock=1
    CMP REGISTER, #0            | lock valia zero?
    JNE enter_region        | Se sim, entra na região crítica,
                                | Se não, continua no laço
    RET                 | Retorna para o processo chamador

leave_region
     MOVE LOCK, #0              | lock=0
     RET                        | Retorna para o processo chamador
```

Soluções

- Exclusão Mútua:
  - Espera Ocupada;
  - **Primitivas *Sleep/Wakeup***;
  - Semáforos;
  - Monitores;
  - Passagem de Mensagem;

Comunicação de Processos – Primitivas *Sleep/Wakeup*

- Todas as soluções apresentadas utilizam espera ocupada ☐ processos ficam em estado de espera (*looping)* até que possam utilizar a região crítica:
  - Tempo de processamento da CPU;
  - Situações inesperadas;

Comunicação de Processos – Primitivas *Sleep/Wakeup*

- Para solucionar esse problema de espera, um par de primitivas *Sleep* e *Wakeup* é utilizado ☐ BLOQUEIO E DESBLOQUEIO de processos.
- 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";
- A primitiva *Wakeup* é uma chamada de sistema que "acorda" um determinado processo;
- Ambas as primitivas possuem dois parâmetros: o processo sendo manipulado e um endereço de memória para realizar a correspondência entre uma primitiva *Sleep* com sua correspondente *Wakeup*;

Comunicação de Processos – Primitivas *Sleep/Wakeup*

- Problemas que podem ser solucionados com o uso dessas primitivas:
  - Problema do Produtor/Consumidor (*bounded buffer* ou *buffer* limitado): dois processos compartilham um *buffer* de tamanho fixo. 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;

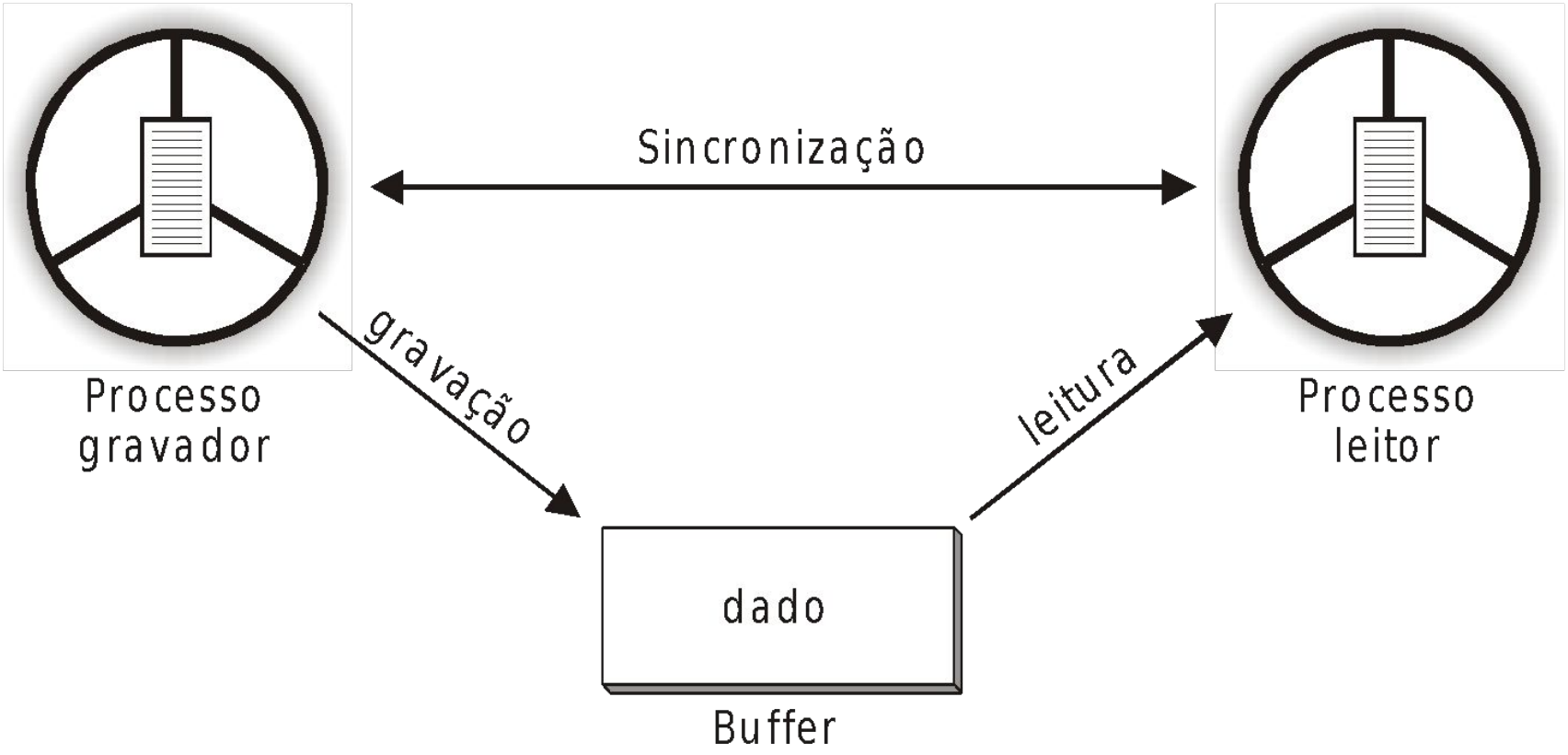
Comunicação de Processos – Primitivas *Sleep/Wakeup*

- *Buffer*: uma variável `count` controla a quantidade de dados presente no *buffer*.
- Produtor: Antes de colocar dados no *buffer*, o processo produtor checa o valor dessa variável. Se a variável está com valor máximo, o processo produtor é colocado para dormir. Caso contrário, o produtor coloca dados no *buffer* e o incrementa.

Comunicação de Processos – Primitivas *Sleep/Wakeup*

- Consumidor: Antes de retirar dados no *buffer*, o processo consumidor checa o valor da variável `count` para saber se ela está com 0 (zero). Se está, o processo vai "dormir", senão ele retira os dados do *buffer* e decrementa a variável;

Comunicação de Processos
Sincronização Produtor-Consumidor



Comunicação de Processos – Primitivas *Sleep/Wakeup*

```
# define N 100
int 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);
   }
}
```

Comunicação de Processos – Primitivas *Sleep/Wakeup*

- Problemas desta solução: Acesso à variável `count` é irrestrita
  - O *buffer* está vazio e o consumidor acabou de checar a variável `count` com valor 0;
  - O escalonador (por meio de uma interrupção) decide que o processo produtor será executado; Então o processo produtor insere um item no *buffer* e incrementa a variável `count` com valor 1; Imaginando que o processo consumidor está dormindo, o processo produtor envia um sinal de *wakeup* para o consumidor;
  - No entanto, o processo consumidor não está dormindo, e não recebe o sinal de *wakeup*;

Comunicação de Processos – Primitivas *Sleep/Wakeup*

- Assim que o processo consumidor é executado novamente, a variável `count` já tem o valor zero; Nesse instante, o consumidor é colocado para dormir, pois acha que não existem informações a serem lidas no *buffer*;
- Assim que o processo produtor acordar, ele insere outro item no *buffer e* volta a dormir. Ambos os processos dormem para sempre...

- Solução: *bit* de controle recebe um valor `true` quando um sinal é enviado para um processo que não está dormindo. No entanto, no caso de vários pares de processos, vários *bits* devem ser criados sobrecarregando o sistema!!!!

Soluções

- Exclusão Mútua:
  - Espera Ocupada;
  - Primitivas *Sleep/Wakeup*;
  - **Semáforos**;
  - Monitores;
  - Passagem de Mensagem;

Comunicação de Processos – Semáforos

- Idealizados por E. W. Dijkstra (1965);
- Variável inteira que armazena o número de sinais *wakeups* enviados;
- Um semáforo pode ter valor 0 quando não há sinal armazenado ou um valor positivo referente ao número de sinais armazenados;
- Duas primitivas de chamadas de sistema: `down` *(sleep)* e `up` *(wake);*
- Originalmente P (`down`) e V (`up`) em holandês*;*

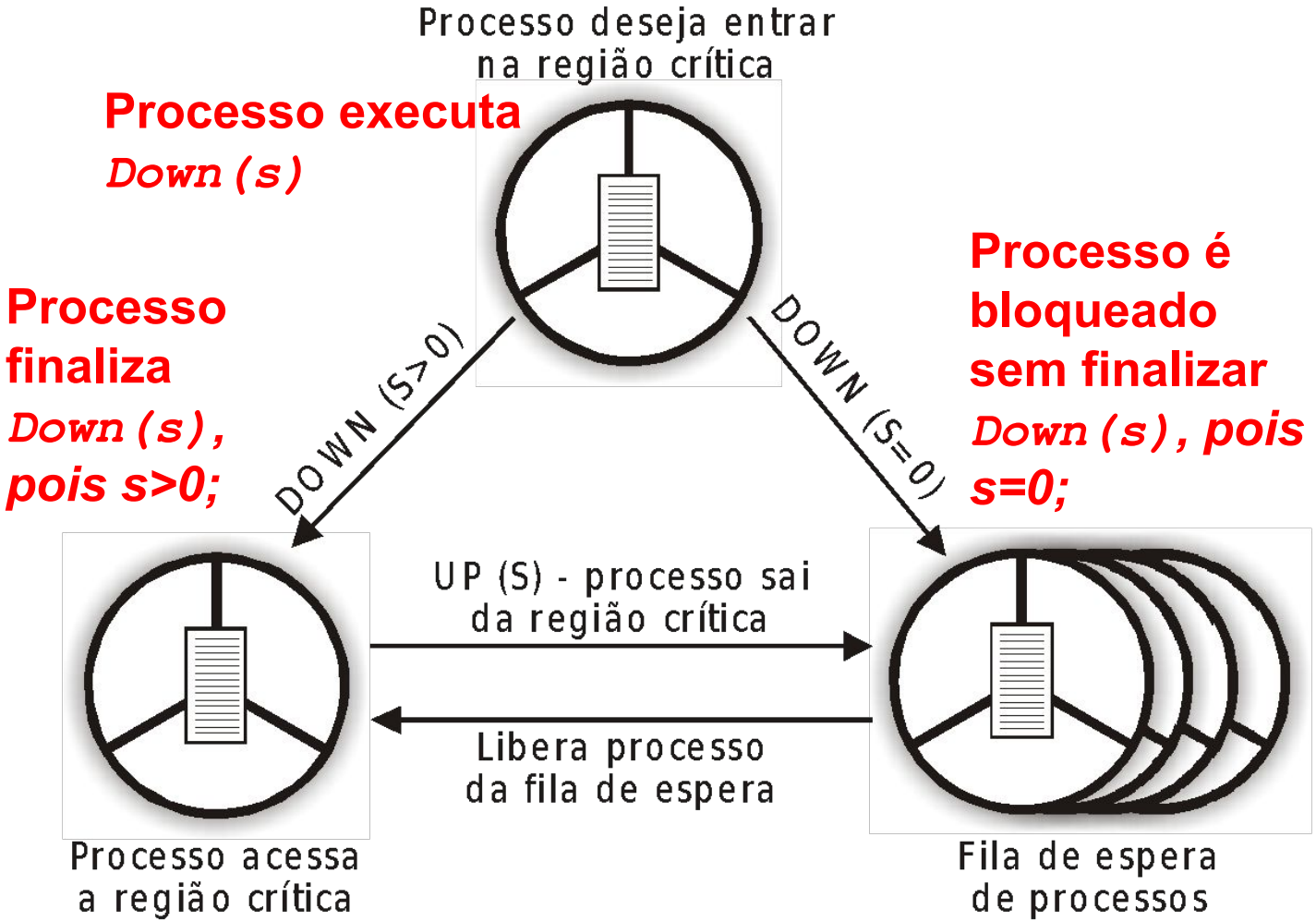
Comunicação de Processos – Semáforos

- ***Down***: verifica se o valor do semáforo é maior do que 0; se for, o semáforo é decrementado; se o valor for 0, 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;

Comunicação de Processos – Semáforos

- ***Up***: incrementa o valor do semáforo, fazendo com que algum processo que esteja dormindo possa terminar de executar sua operação ***down***;
- ***Semáforo Mutex***: garante a exclusão mútua, não permitindo que os processos acessem uma região crítica ao mesmo tempo
  - Também chamado de **semáforo binário**

Comunicação de Processos – Semáforo Binário



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;

Comunicação de Processos – Semáforos

```
# 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);
    enter_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);
  }
}
```

Comunicação de Processos – Semáforos

- Problema: erro de programação pode gerar um *deadlock*;
  - Suponha que o código seja trocado no processo produtor;

```
..                      ..
down(&empty);           down(&mutex);
down(&mutex);           down(&empty);
..                      ..
```

  - Se o *buffer* estiver cheio, o produtor será bloqueado com `mutex` = *0;* Assim, a próxima vez que o consumidor tentar acessar o *buffer*, ele tenta executar um `down` sobre o `mutex`, ficando também bloqueado.

Processos

- Introdução
- Escalonamento de Processos
- **Comunicação entre Processos**
  - **Condição de Disputa**
  - **Região Crítica**
  - **Formas de Exclusão Mútua**
  - **Problemas Clássicos**
- Threads
- Deadlock