Sistemas Operacionais

Prof. Jó Ueyama

Apresentação baseada nos slides da Profa. Dra. Kalinka Castelo Branco, do Prof. Dr. Antônio Carlos Sementille e da Profa. Dra. Luciana A. F. Martimiano e nas transparências fornecidas no site de compra do livro "Sistemas Operacionais Modernos"

Aula de Hoje (conteúdo detalhado)

1. **Comunicação interprocessos**
   - **2.1 Formas de especificar uma execução paralela**
2. **Condições de corrida e Exclusão Mútua**
3. **Soluções de exclusão mútua**

Comunicação de Processos

- Processos precisam se comunicar;
  - ex.: aplicação de passagem aérea
- 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;
    - a = b + c; x = a + y;

Comunicação de Processos

Especificação de Execução Concorrente

Questão importante na estruturação de Algoritmos paralelos

Como decompor um problema em um conjunto de processos paralelos

Algumas formas de se expressar uma execução concorrente (usadas em algumas linguagens e sistemas operacionais)

- **Co-rotinas**
- **Declarações FORK/JOIN**
- **Declarações COBEGIN/COEND**
- **Declarações de Processos Concorrentes**

Comunicação de Processos

Co-Rotinas

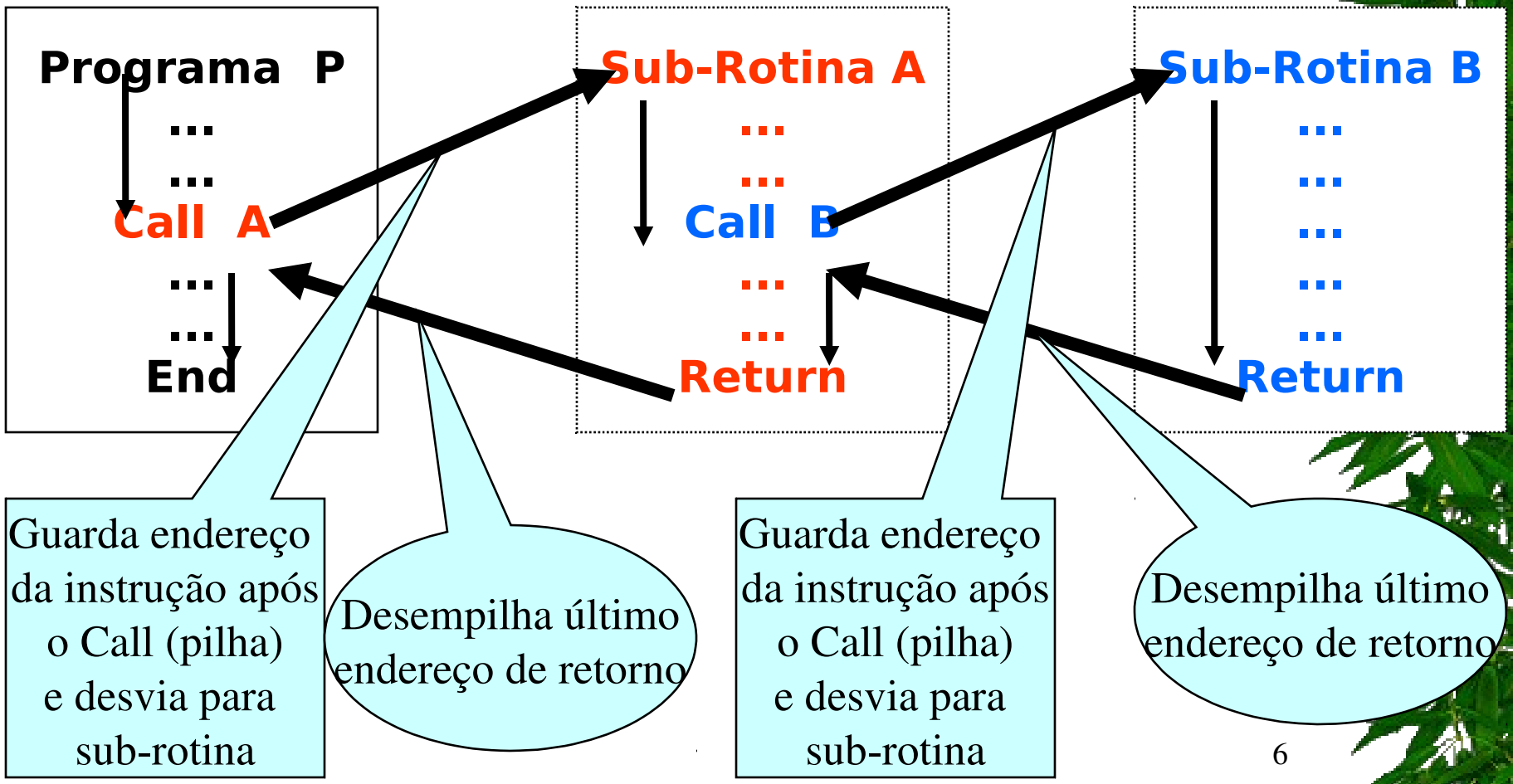
- As co-rotinas são parecidas com sub-rotinas (ou procedimentos), diferindo apenas na forma de transferência de controle, realizada na chamada e no retorno

- As co-rotinas possuem um ponto de entrada, mas pode representar diversos pontos intermediários de entrada e saída

- A transferência de controle entre eles é realizada através do endereçamento explícito e de livre escolha do programador (através de comandos do tipo TRANSFER, do Modula-2)

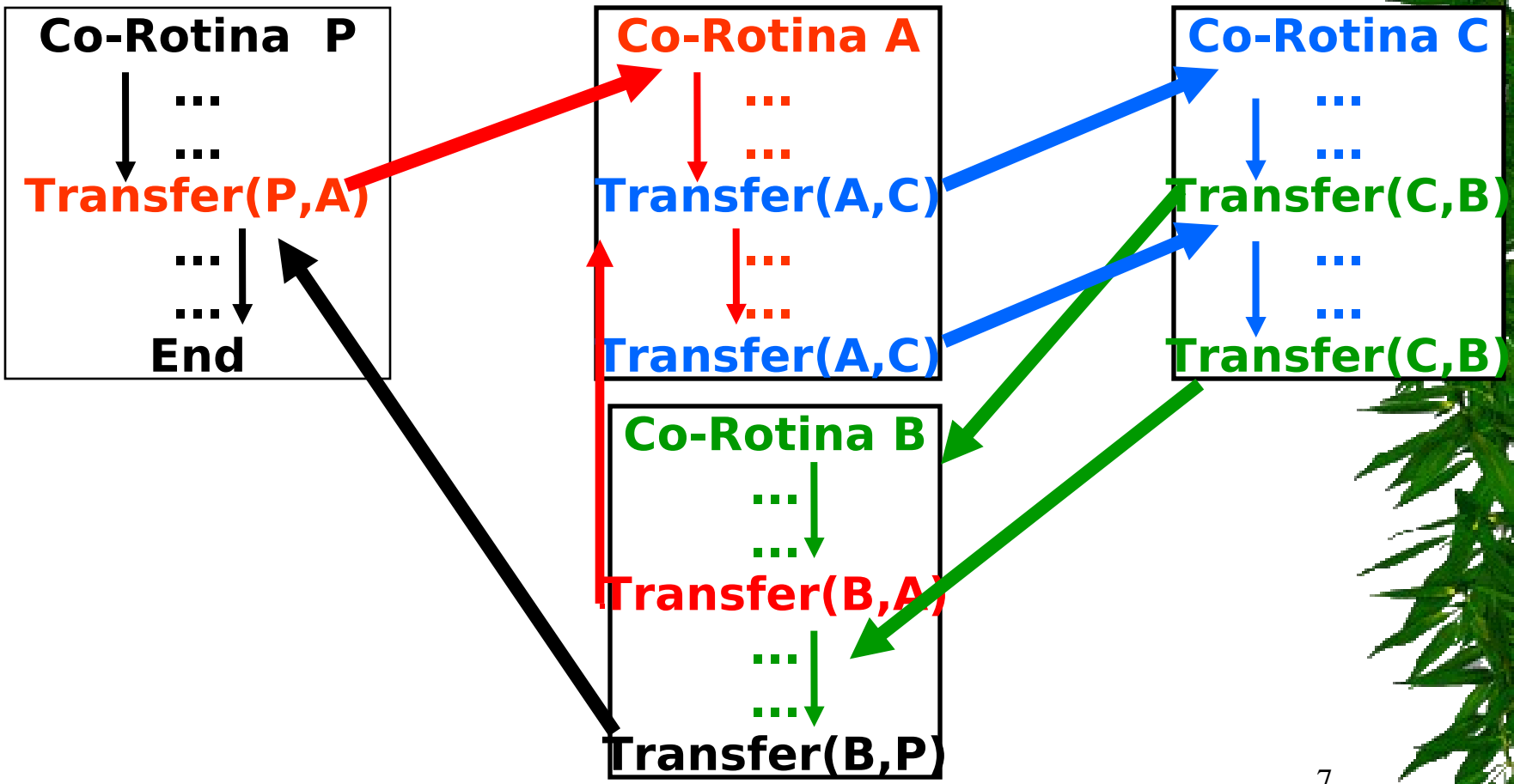
Comunicação de Processos

Funcionamento das Sub-rotinas comuns



Comunicação de Processos

Funcionamento das Co-Rotinas

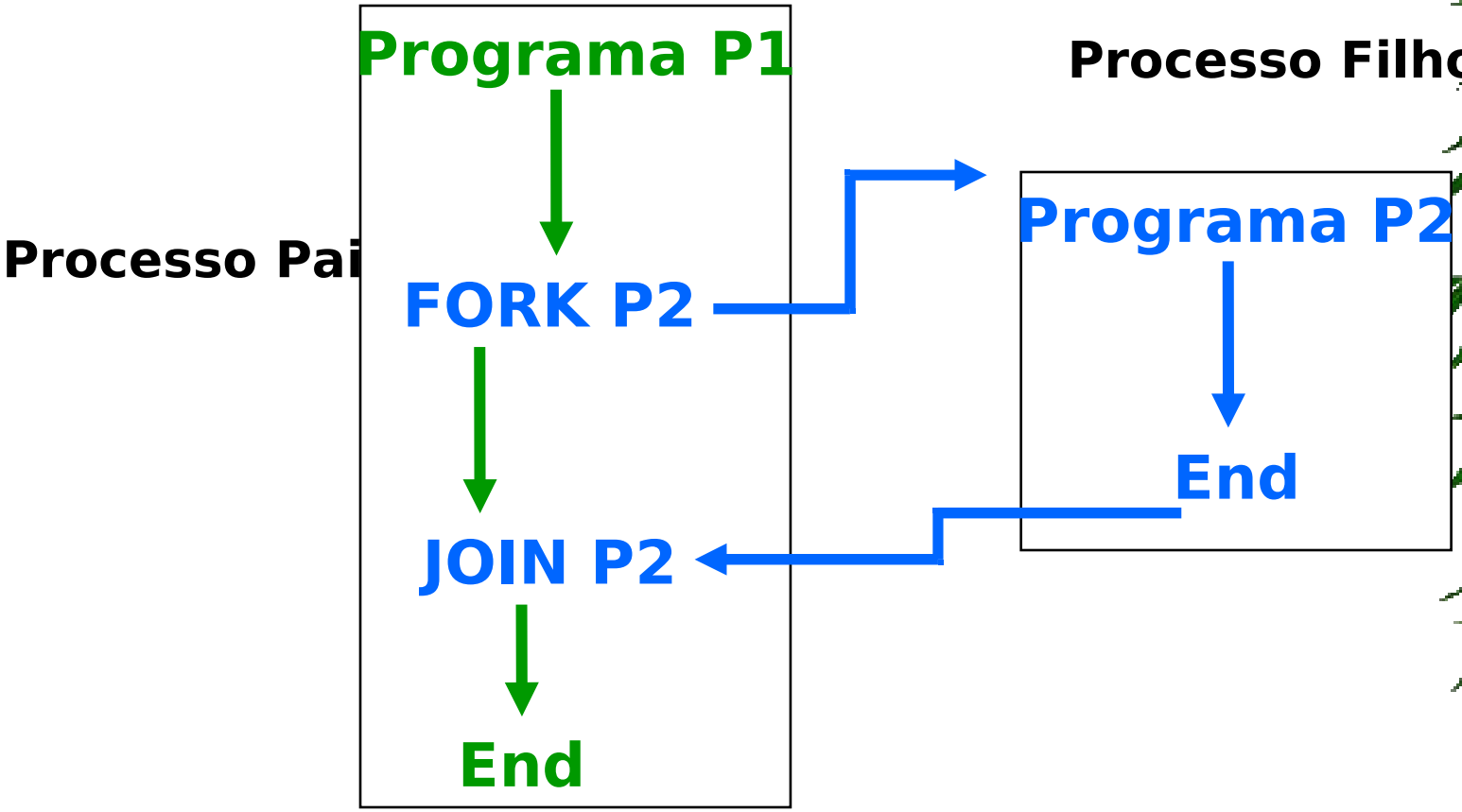


Comunicação de Processos

- **Declarações FORK/JOIN**
  - **A declaração `FORK <nome do programa>` determina o início de execução de um determinado programa, de forma concorrente com o programa sendo executado.**
  - **Para sincronizar-se com o término do programa chamado, o programa chamador deve executar a declaração `JOIN <nome do programa chamado>`.**
  - **O uso do `FORK/JOIN` permite a concorrência e um mecanismo de criação dinâmica entre processos (criação de múltiplas versões de um mesmo programa -> processo-filho), como no sistema UNIX**

Comunicação de Processos

Declarações FORK/JOIN



Comunicação de Processos

- **Declarações COBEGIN/COEND**
  - **Constituem uma forma estruturada de especificar execução concorrente ou paralela de um conjunto de declarações agrupadas da seguinte maneira:**

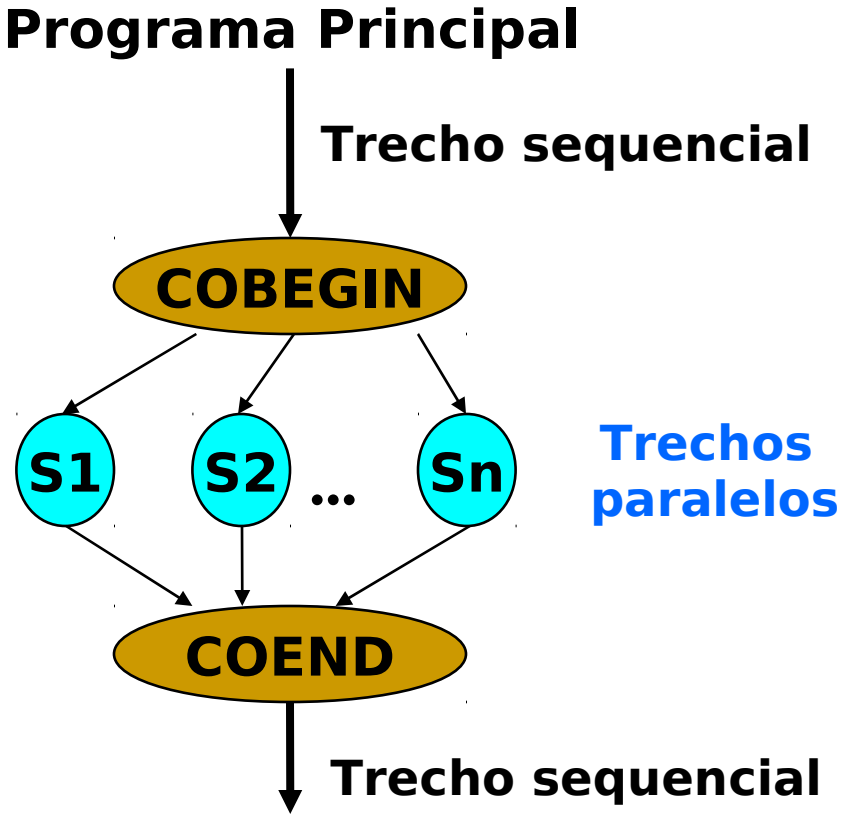
```
COBEGIN
   S1//S2//...//Sn
COEND
```

Onde:

- **A execução deste trecho de programa provoca a execução concorrente das declarações S1, S2, ..., Sn.**
- **Esta execução só termina, quando todas as declarações Si terminarem.**

Comunicação de Processos

Declarações COBEGIN/COEND




```
Program Paralelo;
/* declaração de var.e const. globais */
Begin
   /* trecho sequencial */
  ...
  COBEGIN /* trechos   paralelos */
       Begin  /* S1 */
          ...
       End;
       ...
       Begin /* Sn */
          ...
       End;
  COEND
  /* trecho sequencial */
  ...
End.
```

Aula de Hoje (conteúdo detalhado)

1. Comunicação interprocessos
   2.1 Formas de especificar uma execução paralela
2. **Condições de corrida e Exclusão Mútua**
3. **Soluções de exclusão mútua**

Comunicação de Processos

Mecanismos Simples de Comunicação e Sincronização entre Processos

- **Num sistema de multiprocessamento ou multiprogramação, os processos geralmente precisam se comunicar com outros processos.**

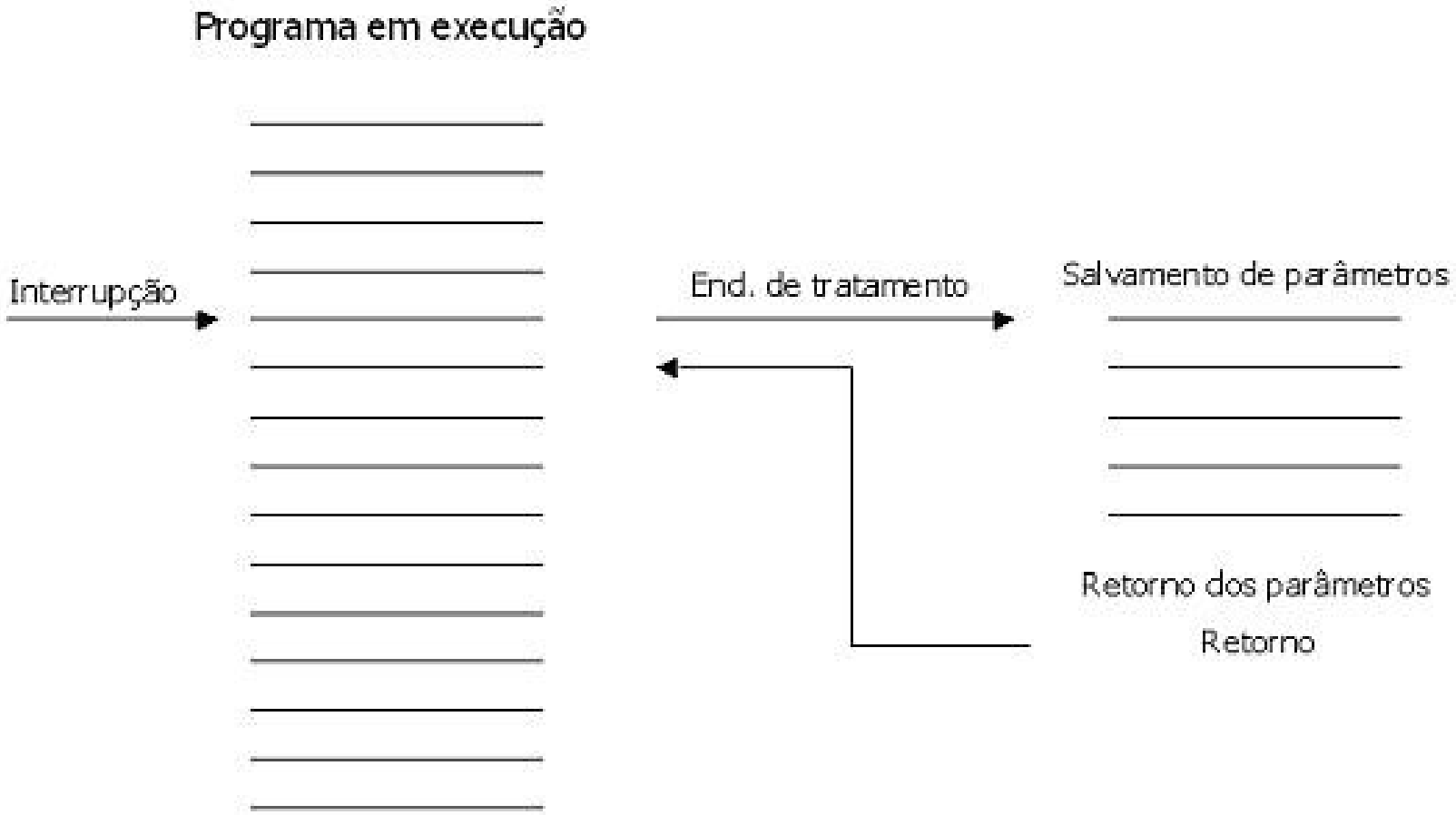
- **A comunicação entre processos é mais eficiente se for estruturada e não utilizar interrupções.**

- **A seguir, serão vistos alguns destes mecanismos e problemas da comunicação inter-processos.**

Comunicação de Processos

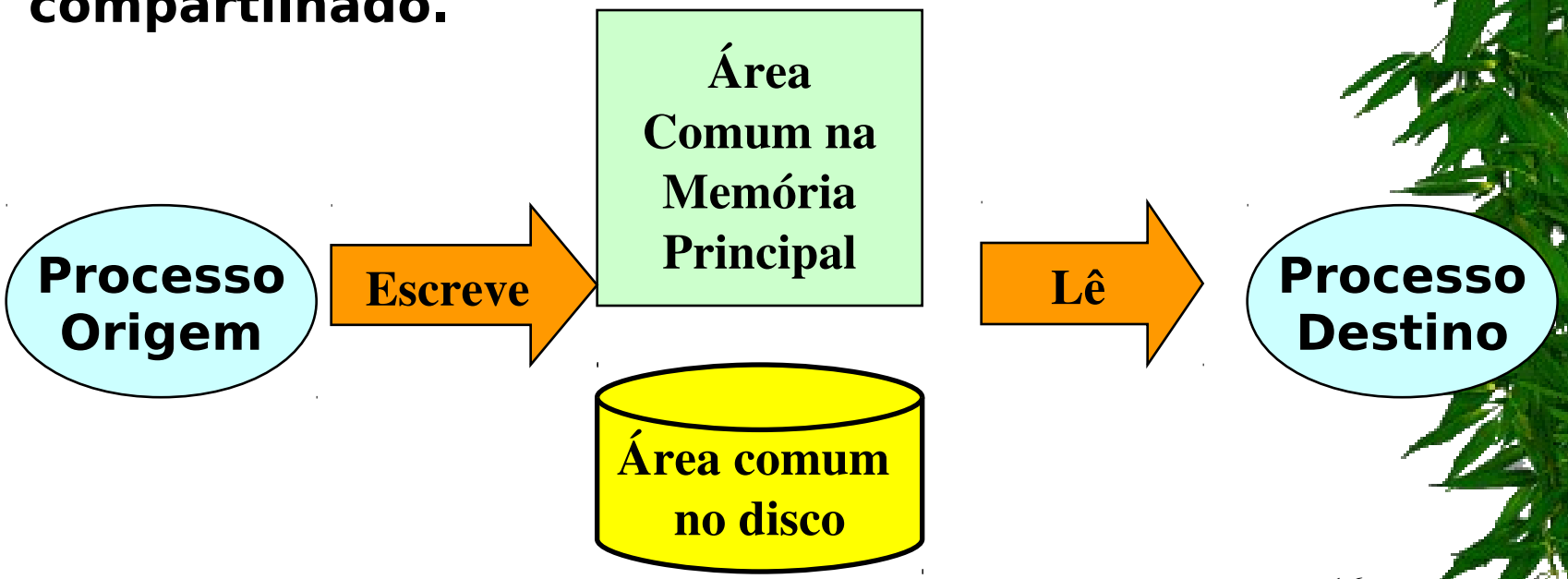
- ***O que são interrupções?***
  - Uma interrupção é um evento externo que faz com que o processador pare a execução do programa corrente e desvie a execução para um bloco de código chamado rotina de interrupção (normalmente são decorrentes de operações de E/S).
  - Ao terminar o tratamento de interrupção o controle retorna ao programa interrompido exatamente no mesmo estado em que estava quando ocorreu a interrupção.

Comunicação de Processos



Comunicação de Processos

- **Condições de Corrida**
  - **Em alguns Sistemas Operacionais: os processos se comunicam através de alguma área de armazenamento comum. Esta área pode estar na memória principal ou pode ser um arquivo compartilhado.**



Comunicação de Processos

Condições de Corrida

- **Definição de condições de corrida: situações onde dois os mais processos estão lendo ou escrevendo algum dado compartilhado e o resultado depende de quem processa no momento propício.**
- **Exemplo: a = b + c; x = a + y;**

- **Depurar programas que contém condições de corrida não é fácil, pois não é possível prever quando o processo será suspenso.**

Comunicação de Processos

Condições de Corrida

- **Um exemplo: Print Spooler**

- **Quando um processo deseja imprimir um arquivo, ele coloca o nome do arquivo em uma lista de impressão (*spooler directory*).**

- **Um processo chamado "printer daemon", verifica a lista periodicamente para ver se existe algum arquivo para ser impresso, e se existir, ele os imprime e remove seus nomes da lista.**

Comunicação de Processos

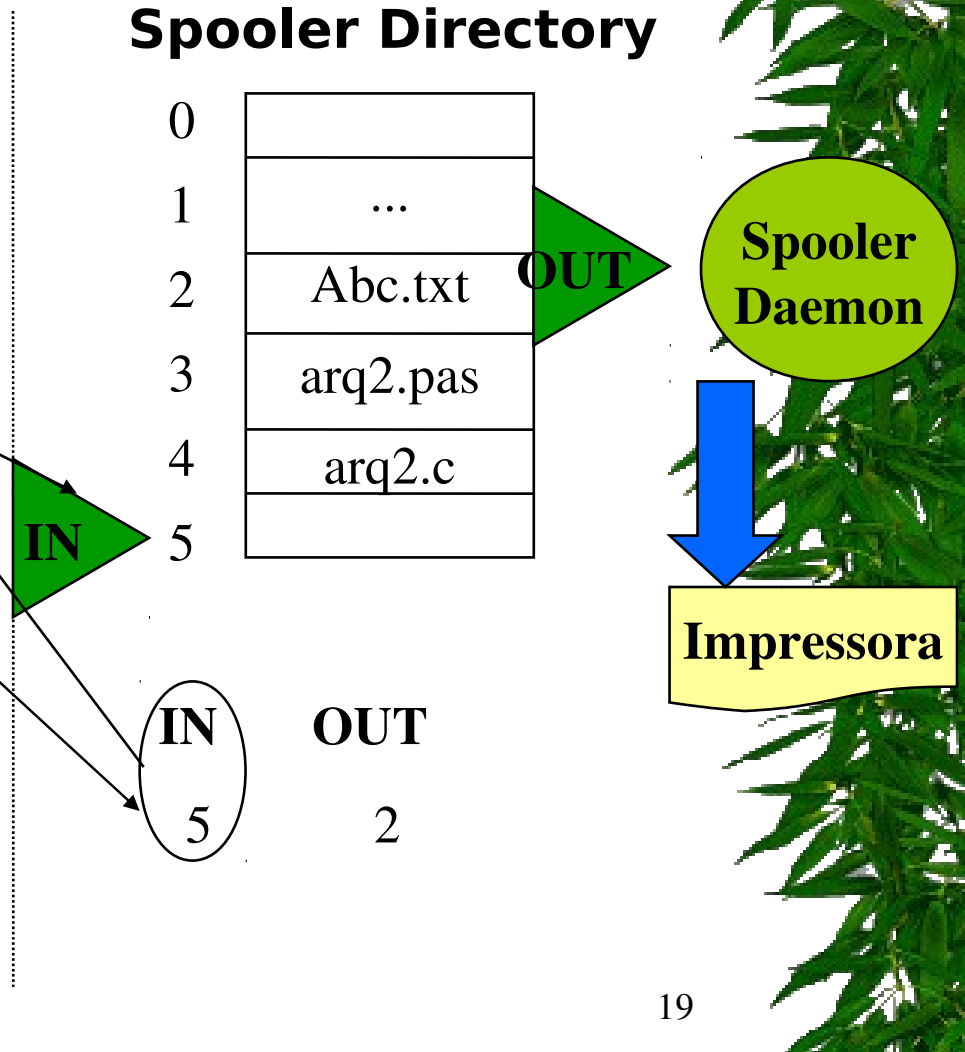
Condições de Corrida

Processo A

```
...
get_prox_slot(slot);
set_nomearq(slot,nomearq);
slot++;
set_prox_slot(slot);
...
```

Processo B

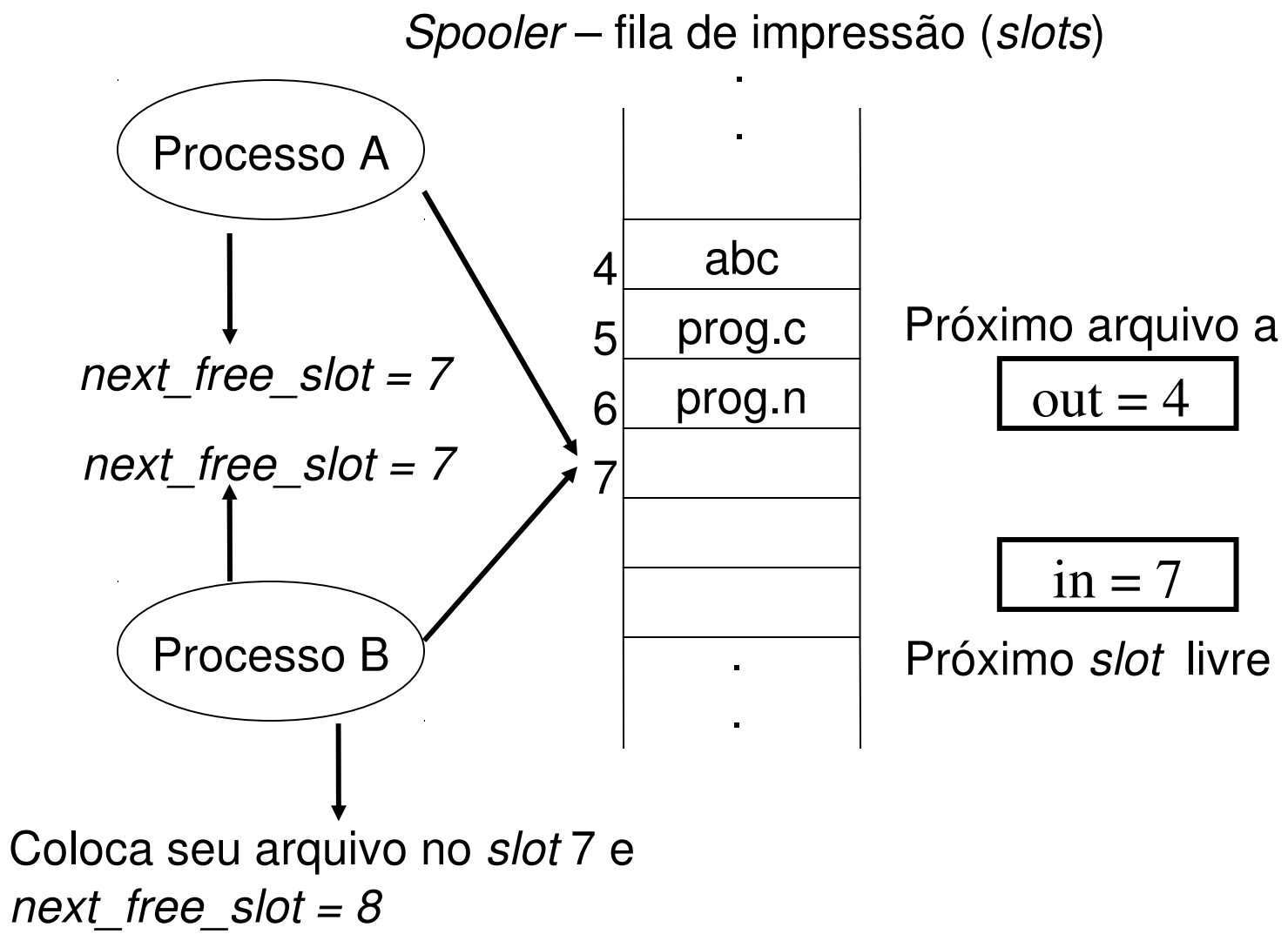
```
...
get_prox_slot(sl);
set_nomearq(sl,nomearq);
slot++;
set_prox_slot(slot);
...
```



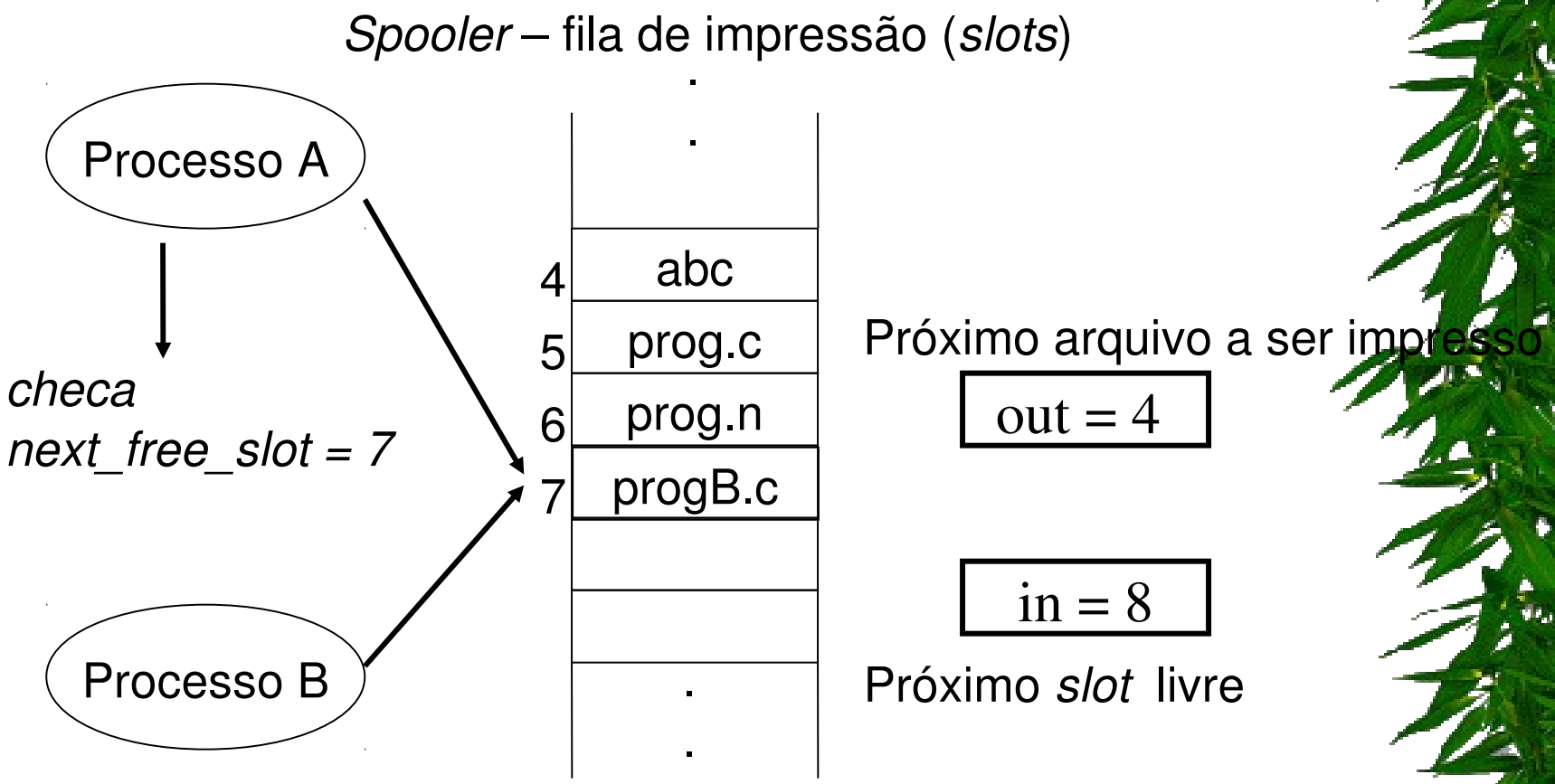
Comunicação de Processos
Race Conditions

- *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...

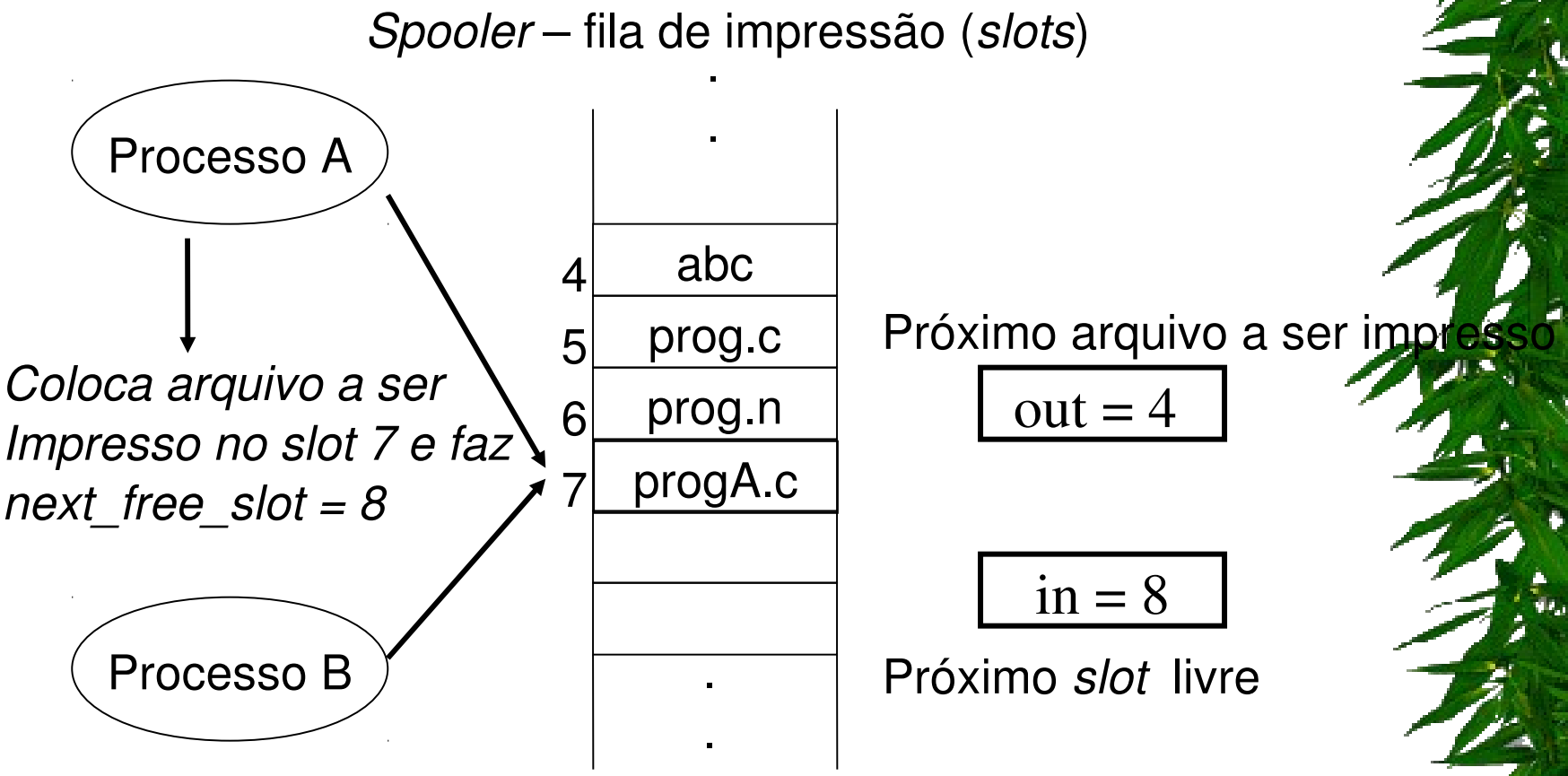
Comunicação de Processos - *Race Conditions*



Comunicação de Processos - *Race Conditions*



Comunicação de Processos - *Race Conditions*



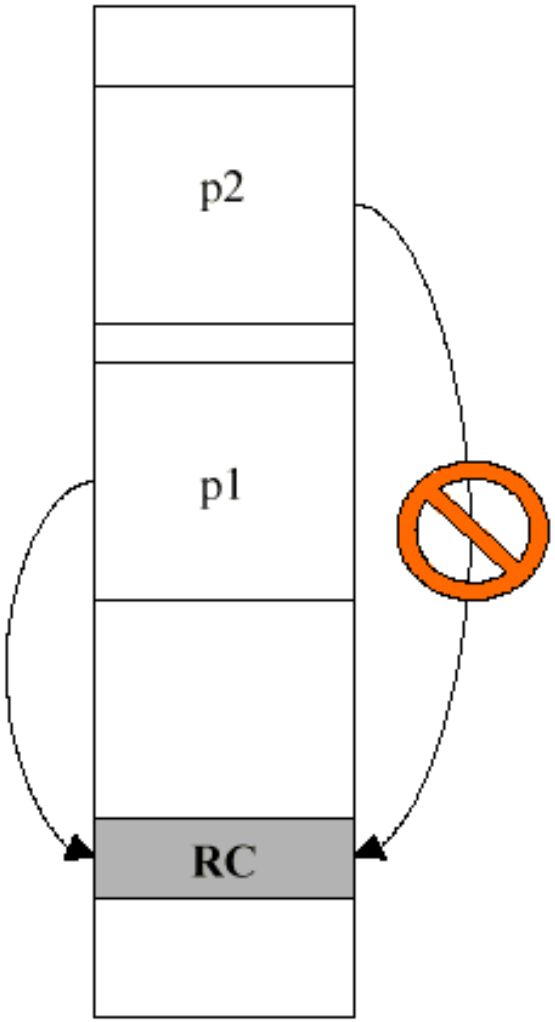
Processo B nunca receberá sua impressão!!!!!

Comunicação de Processos – *Regiões Críticas*

- **Regiões Críticas**
  - **Uma solução para as condições de corrida é proibir que mais de um processo leia ou escreva em uma variável compartilhada ao mesmo tempo.**
  - **Esta restrição é conhecida como exclusão mútua, e os trechos de programa de cada processo que usam um recurso compartilhado e são executados um por vez, são denominados seções críticas ou regiões críticas (R.C.).**

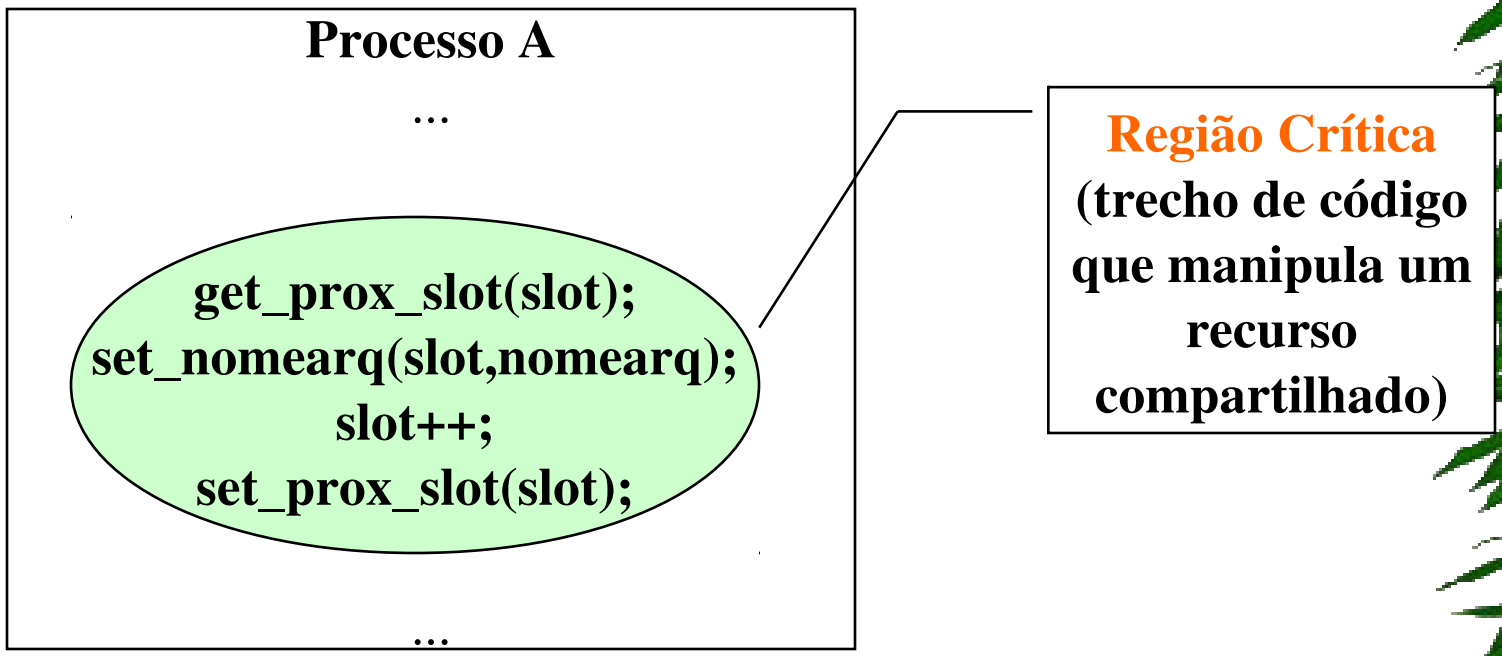
Comunicação de Processos – *Regiões Críticas*

Regiões Críticas



Comunicação de Processos – *Regiões Críticas*

Regiões Críticas



Regiões Críticas e Exclusão Mútua

- **Região crítica**
  - seção do programa onde são efetuados acessos (para leitura e escrita) a recursos partilhados por dois ou mais processos
  - é necessário assegurar que dois ou mais processos não se encontrem simultaneamente na região crítica

Comunicação de Processos – *Regiões Críticas*

Pergunta: isso quer dizer que uma máquina no Brasil e outra no Japão, cada uma com processos que se comunicam, nunca terão Condições de Disputa?

Ex.: Vaga em avião

1. Operador OP1 (no Brasil) lê Cadeira1 vaga;
2. Operador OP2 (no Japão) lê Cadeira1 vaga;
3. Operador OP1 compra Cadeira1;
4. Operador OP2 compra Cadeira1;

Solução simples para exclusão mútua

- Caso de venda no avião:
  - apenas um operador pode estar vendendo em um determinado momento;
- Isso gera uma fila de clientes nos computadores;
- Problema: ineficiência!

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;

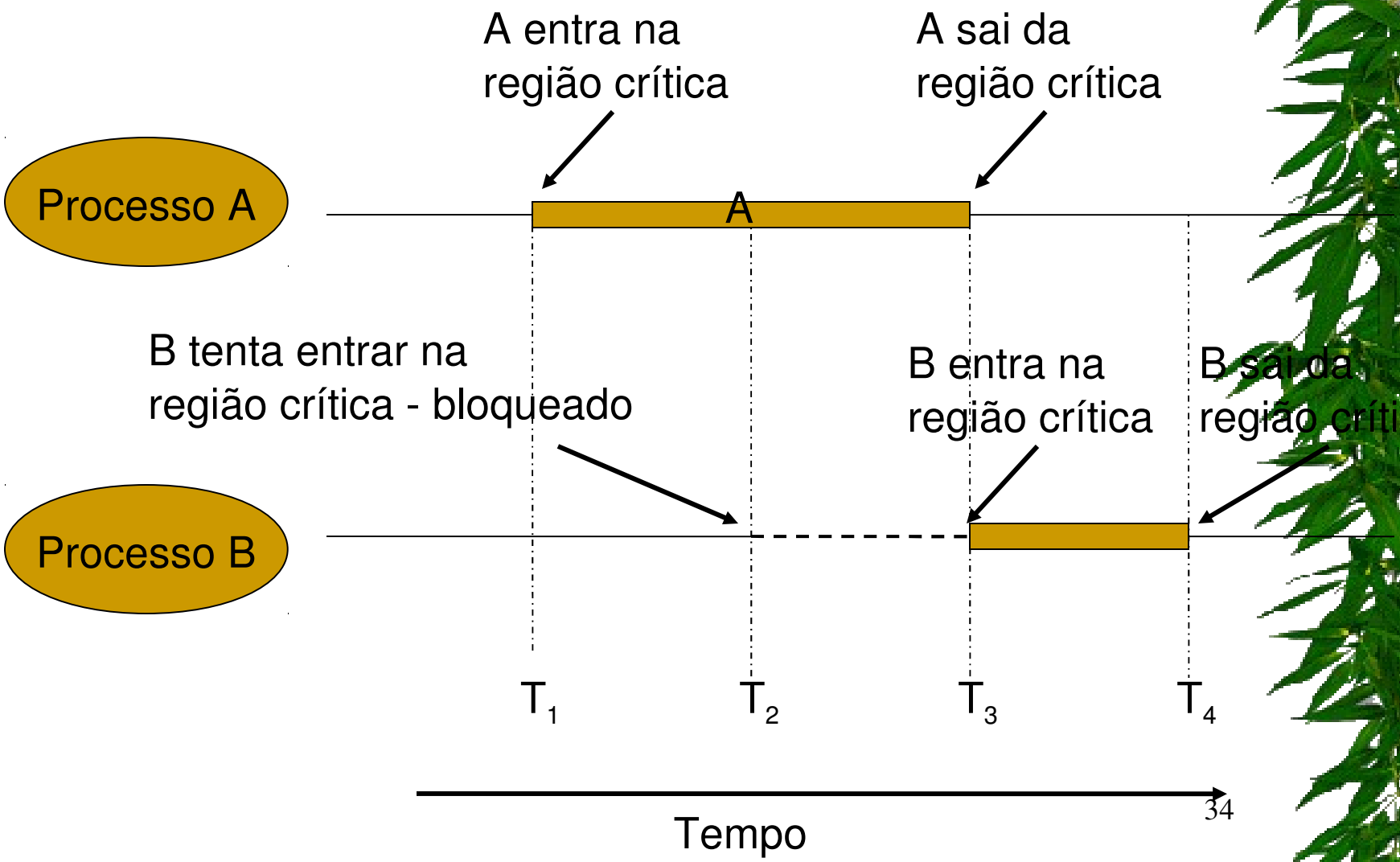
Comunicação de Processos – Exclusão Mútua

- assegura-se a exclusão mútua recorrendo aos mecanismos de sincronização fornecidos pelo SO
- Estas afirmações são válidas também para as *threads* (é ainda mais crítico, pois todas as *threads* dentro do mesmo processo partilham os mesmos recursos)

Regiões Críticas e Exclusão Mútua

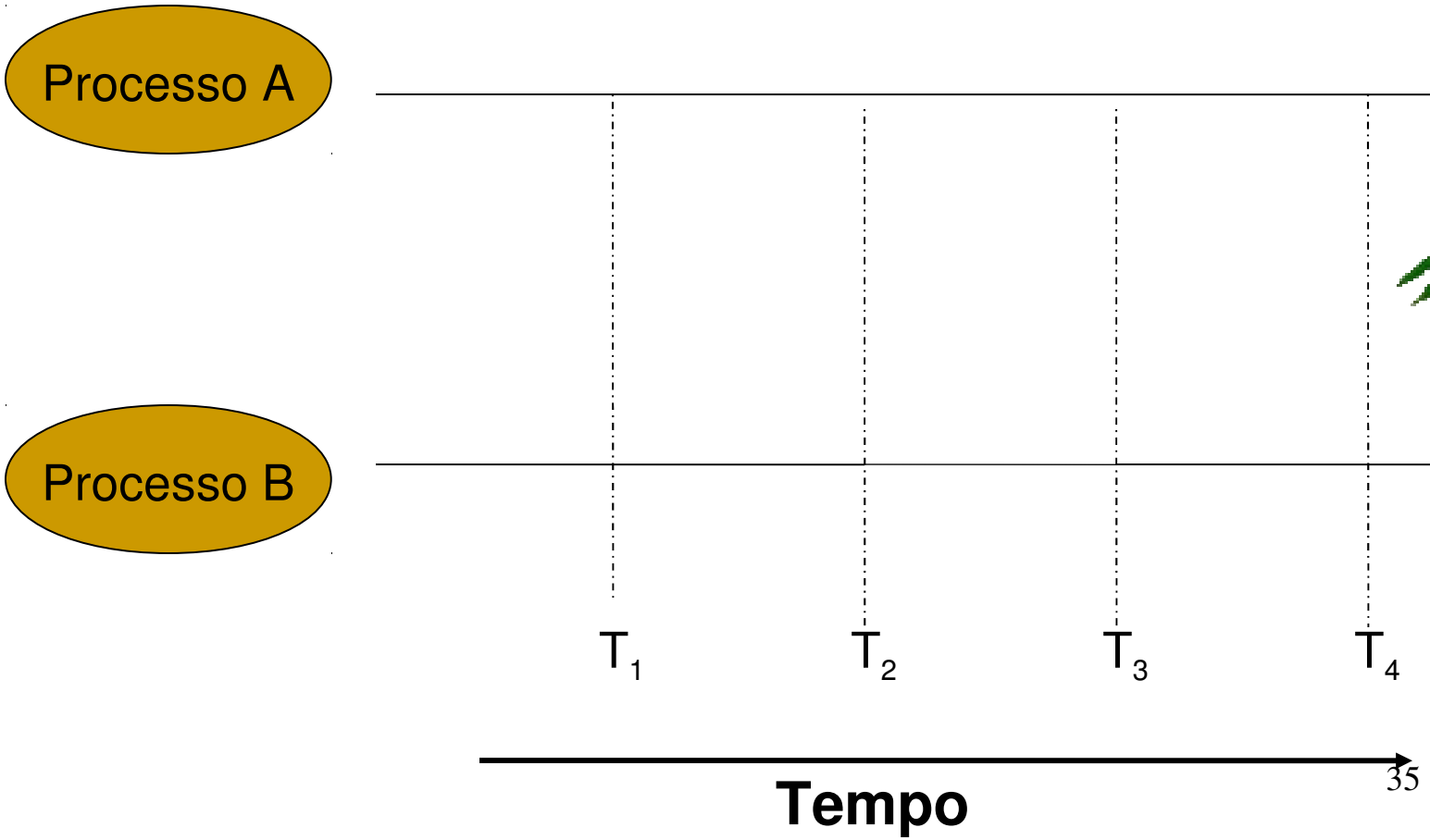
- Regras para programação concorrente (condições para uma boa solução)
  - 1) Dois ou mais processos não podem estar simultaneamente dentro de uma região crítica
  - 2) Não se podem fazer afirmações em relação à velocidade e ao número de CPUs
  - 3) Um processo fora da região crítica não deve causar bloqueio a outro processo
  - 4) Um processo não pode esperar infinitamente para entrar na região crítica

Comunicação de Processos – Exclusão Mútua



Comunicação de Processos – Exclusão Mútua

Volta a situação inicial!!!!



Aula de Hoje (conteúdo detalhado)

1. Comunicação interprocessos
   2.1 Formas de especificar uma execução paralela
2. Condições de corrida e Exclusão Mútua
3. **Soluções de 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 (funciona bem para monoprocessador);
    - Viola condição 2;
  - Não é uma solução segura, pois um processo pode não habilitar novamente suas interrupções e não ser finalizado;
    - Viola condição 4;

Comunicação de Processos
Exclusão Mútua

- **Exclusão Mútua com Espera Ocupada**
- **Desabilitando as Interrupções**

- **SOLUÇÃO MAIS SIMPLES:** cada processo desabilita todas as interrupções (inclusive a do relógio) após entrar em sua região crítica, e as reabilita antes de deixá-la.

- **DESVANTAGENS:**
  - Processo pode esquecer de reabilitar as interrupções;
  - Em sistemas com várias UCPs, desabilitar interrupções em uma UCP não evita que as outras acessem a memória compartilhada.

- **CONCLUSÃO:** é útil que o kernel tenha o poder de desabilitar interrupções, mas não é apropriado que os processos de usuário usem este método de exclusão mútua.

Comunicação de Processos – Exclusão Mútua

- Variáveis *Lock*:
  - O processo que deseja utilizar uma região crítica atribui 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;

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;
    non-critical_region();
}
```

Processo A

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

Processo B

Comunicação de Processos – Exclusão Mútua

- Problema do *Strict Alternation:*

1. Suponha que o Processo B é mais rápido e saí 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;

Comunicação de Processos – Exclusão Mútua

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

Aula de Hoje (conteúdo detalhado)

1. Soluções de exclusão mútua
   - 1.1 Soluções com espera ocupada
     - 1.1.1 Desabilitando interrup
     - 1.1.2 Locks
     - 1.1.3 Strict Alternation
     - **1.1.4 TSL**
   - **1.2 Primitivas Sleep/Wakeup**
   - **1.3 Semáforos**

Comunicação de Processos – Sincronização

O Problema de Espaço na Geladeira

Hora	Pessoa A	Pessoa B
6:00	Olha a geladeira: sem leite	-
6:05	Sai para a padaria	-
6:10	Chega na padaria	Olha a geladeira: sem leite
6:15	Sai da padaria	Sai para a padaria
6:20	Chega em casa: guarda o leite	Chega na padaria
6:25	-	Sai da padaria
6:30	-	Chega em casa: Ops!

Comunicação de Processos – Sincronização - Solução

Regra	Exemplo da geladeira
1. Trancar antes de utilizar	Deixar aviso
2. Destrancar quando terminar	Retirar o aviso
3. Esperar se estiver trancado	Não sai para comprar se houver aviso

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

- Instrução TSL: utiliza registradores do hardware;
  - TSL RX, LOCK; (lê o conteúdo de ***lock*** e armazena em RX; na sequência 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
```

Comunicação de Processos – Exclusão Mútua

- **Instrução TSL (Test and Set Lock)**

- Esta solução é implementada com **uso do hardware**.

- Muitos computadores possuem uma instrução especial, chamada **TSL (test and set lock)**, que funciona assim: ela lê o conteúdo de uma palavra de memória e armazena um valor diferente de zero naquela posição.

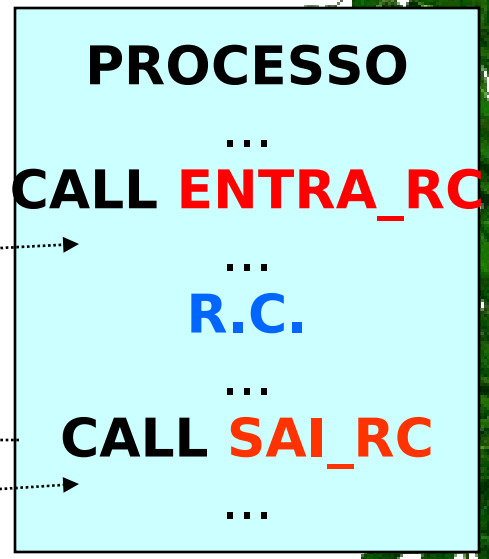
- **Em sistemas multiprocessados:** esta instrução trava o barramento de memória, proibindo outras UCPs de acessar a memória até ela terminar.

Comunicação de Processos – Exclusão Mútua

- **Instrução TSL (Test and Set Lock) - Exemplo**

```
ENTRA_RC:
    TSL reg, flag ; copia flag para reg
                  ; e coloca 1 em flag
    CMP reg,0     ; flag era zero?
    JNZ ENTRA_RC  ; se a trava não
                  ; estava ligada,
                  ; volta ao laço
    RET

SAI_RC:
    MOV flag,0  ;   desliga flag
    RET
```



Comunicação de Processos – Exclusão Mútua

- **Exclusão Mútua com Espera Ocupada**

Considerações Finais

- **Espera Ocupada:** quando um processo deseja entrar na sua região crítica, ele verifica se a entrada é permitida. Se não for, o processo ficará em um laço de espera, até entrar.

- **Desvantagens:**
  - desperdiça tempo de UCP;
  - pode provocar "**bloqueio perpétuo**" (deadlock) em sistemas com prioridades.

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;

Comunicação de Processos – Primitivas *Sleep/Wakeup*

- Problemas que podem ser solucionados com o uso dessas primitivas:
  - Problema do Produtor/Consumidor (*bounded buffer*): 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;

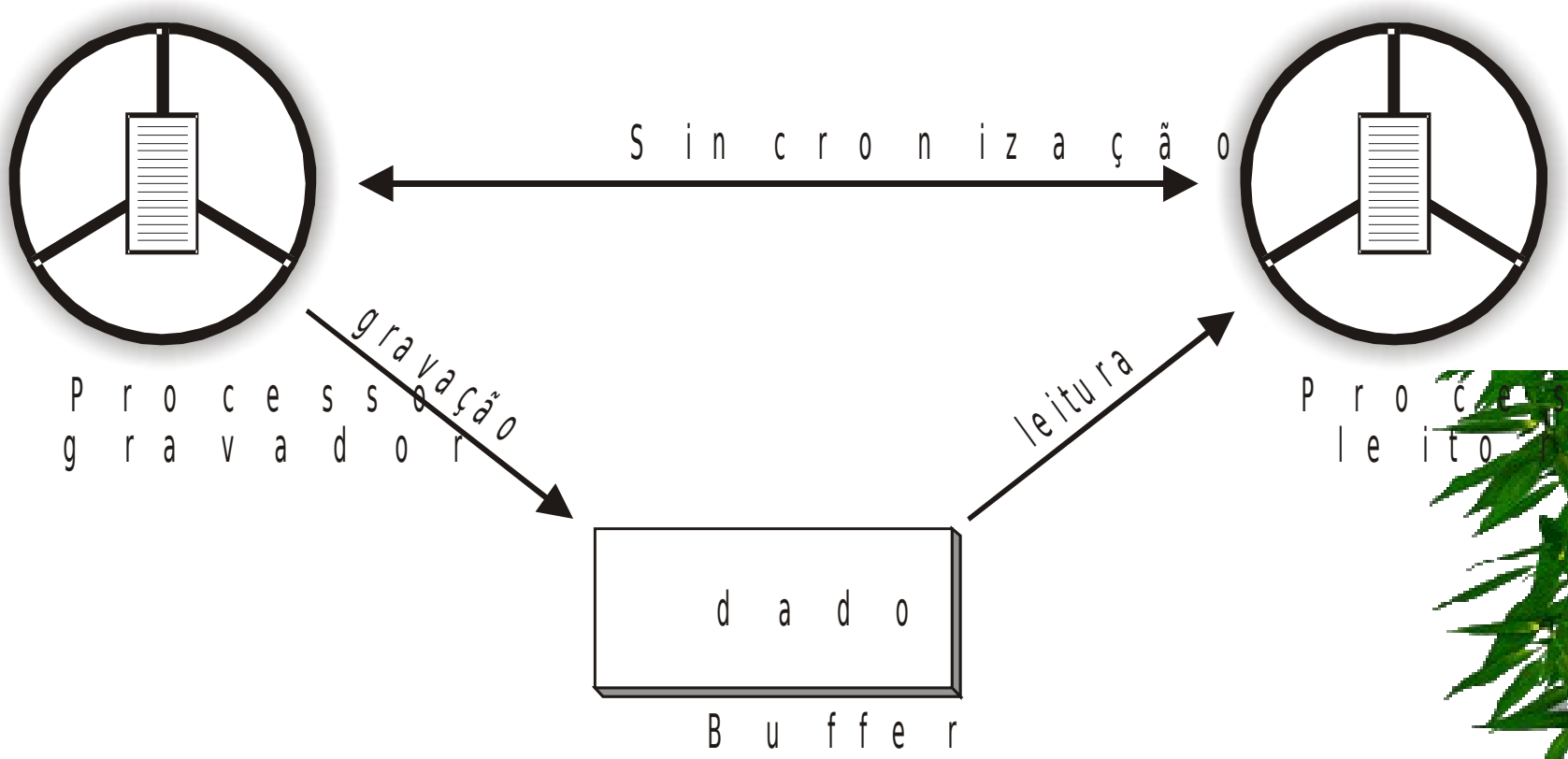
Comunicação de Processos – Primitivas *Sleep/Wakeup*

- *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
Sincronização Produtor-Consumidor

- **Exemplo do Problema do Produtor/ Consumidor usando Sleep e Wakeup**

Para os casos extremos de ocupação do buffer (cheio/vazio), deverão funcionar as seguintes **regras de sincronização**:

- se o produtor tentar depositar uma mensagem no **buffer cheio**, ele será suspenso até que o consumidor retire pelo menos uma mensagem do buffer;
- se o consumidor tenta retirar uma mensagem do **buffer vazio**, ele será suspenso até que o produtor deposite pelo menos uma mensagem no buffer.

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*

- **Exemplo do Problema do Produtor/ Consumidor usando Sleep e Wakeup**

```
#define N 100
int contador = 0;
```

```
produtor()
{
  while(TRUE)
   {
     produz_item();
     if (contador==N)
        Sleep();
     deposita_item();
     contador + = 1;
     if (contador==1)
        Wakeup(consumidor);
    }
}
```

interrupção

```
consumidor()
{
  while(TRUE)
   {
     if (contador==0)
        Sleep();
     retira_item();
     contador - = 1;
     if (contador==N-1)
        Wakeup(produtor);
    consome_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 outros itens 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!!!!

Comunicação de Processos – Primitivas *Sleep/Wakeup*

- **Exemplo do Problema do Produtor/ Consumidor usando Sleep e Wakeup**

Problema: pode ocorrer uma condição de corrida, se a variável contador for utilizada sem restrições.

Solução: Criar-se um **"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.

Soluções

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

Comunicação de Processos – Semáforos

- Variável utilizada para controlar o acesso a recursos compartilhados
  - sincronizar o uso de recursos em grande quantidade
  - exclusão mútua
  - semáforo=0 → recurso está sendo utilizado
  - semáforo>0 → recurso livre
- Operações sobre semáforos
  - down → executada sempre que um processo deseja usar um recurso compartilhado
  - up → executada sempre que um processo liberar o recurso

Comunicação de Processos – Semáforos

- down(semáforo)
  - Verifica se o valor do semáforo é maior que 0
  - Se for, semáforo=semáforo – 1
  - Se não for, o processo que executou o down bloqueia
- up(semáforo)
  - semáforo=semáforo + 1
  - Se há processos bloqueados nesse semáforo, escolhe um deles e o desbloqueia
    - Nesse caso, o valor do semáforo permanece o mesmo

Operações sobre semáforos são atômicas.

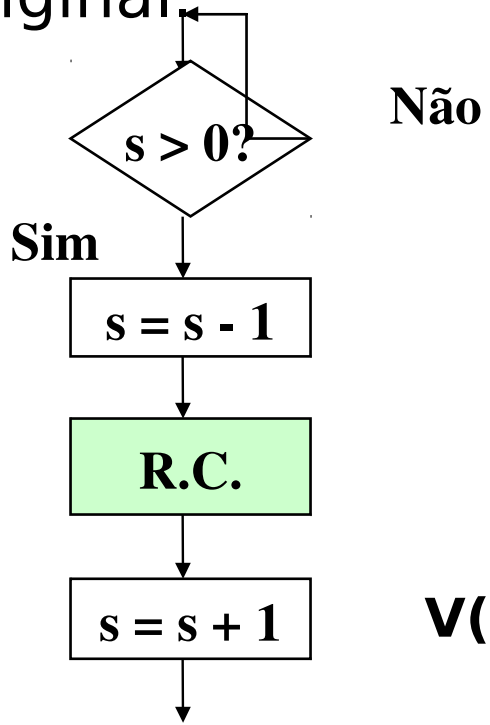
Comunicação de Processos – Semáforos

- Semáforos usados para implementar exclusão mútua são chamados de ***mutex*** (*mutual exclusion semaphor*) ou binários, por apenas assumirem os valores 0 e 1
  - Recurso é a própria região crítica

- Duas primitivas de chamadas de sistema: *down (sleep)* e *up (wake);*
- Originalmente P (*down*) e V (*up*) em holandês*;*

Comunicação de Processos – Semáforos

- **SEMÁFOROS**
- **1ª. Implementação – Espera ocupada**
- Esta implementação é através da espera ocupada: não é a melhor, apesar de ser fiel à definição original.



P(s): Espera até que s>0 e então decrementa s;

V(s): Incrementa s;

Comunicação de Processos – Semáforos

- **SEMÁFOROS**
- **2ª. Implementação – Associando uma fila Qi a cada semáforo si**
- Quando se utiliza este tipo de implementação, o que é muito comum, as primitivas P e V apresentam o seguinte significado:

P(si): se si>0 e então decrementa si (e o processo continua) e não bloqueia o processo;

V(si): se a fila Qi está vazia então incrementa si senão acorda processo da fila Qi;

Comunicação de Processos – Semáforos

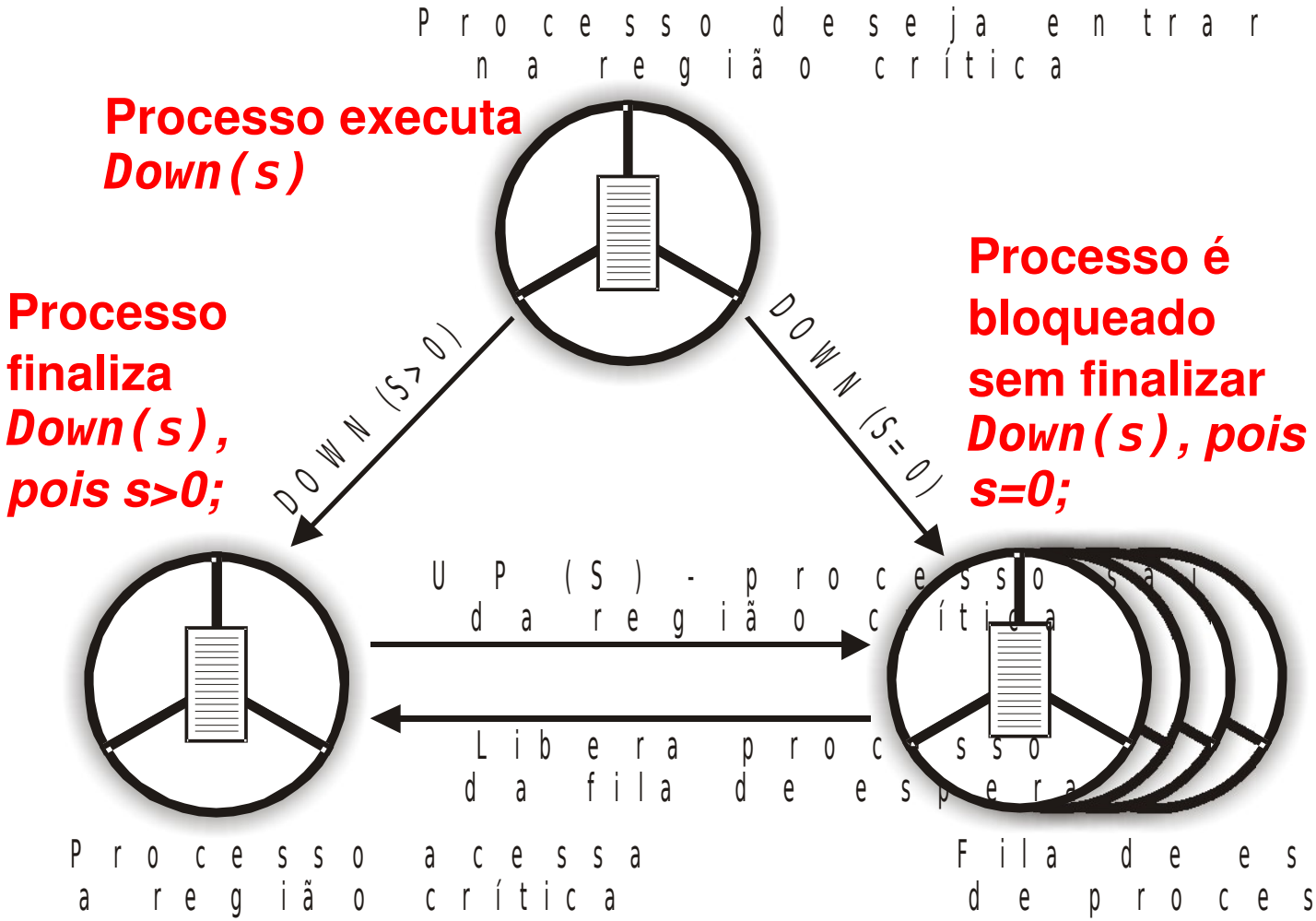
SEMÁFOROS

O semáforo é um mecanismo bastante geral para resolver problemas de sincronismo e exclusão mútua.

Tipos de Semáforos

- **Semáforo geral:** se o semáforo puder tomar qualquer valor inteiro não negativo;
- **Semáforo binário (booleano):** só pode tomar os valores 0 e 1.

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);
    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);
  }
}
```

Soluções

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

Comunicação de Processos – Monitores

- Idealizado por Hoare (1974) e Brinch Hansen (1975)
- **Monitor**: primitiva (unidade básica de sincronização) 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 dentro do monitor em um mesmo instante; outros processos ficam bloqueados até que possam estar ativos no monitor;

Comunicação de Processos – Monitores

```
monitor example
   int i;
   condition c;

   procedure A();
   .
   end;
   procedure B();
   .
   end;
end monitor;
```

Estrutura básica de um Monitor

Dependem da linguagem de programação → Compilador é que garante a exclusão mútua.

- JAVA

Todos os recursos compartilhados entre processos devem estar implementados **dentro** do **Monitor**;

Comunicação de Processos – Monitores

- 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**;

Comunicação de Processos – Monitores

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count – 1;
                if count = N – 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

Comunicação de Processos – Monitores

- Limitações de semáforos e monitores:
  - 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;