Departamento de Computação

SISTEMAS OPERACIONAIS 1
21270 A

Departamento de Computação
Prof. Kelen Cristiane Teixeira Vivaldini

Departamento de Computação

Apresentação baseada nos slides
do Prof. Dr. Antônio Carlos Sementille e Prof.
Kalinka C. Branco e nas transparências
fornecidas no site de compra do livro
“Sistemas Operacionais Modernos”

Departamento de Computação

Súmario

- Comunicação entre Processos
  - Race Conditions – Conidções de Corrida
  - Regiões críticas
  - Exclusão mútua
  - Dormir e acordar
  - Semáforos
  - Mutexes
  - Monitores
  - Troca de Mensagens
  - Barreiras

Departamento de Computação

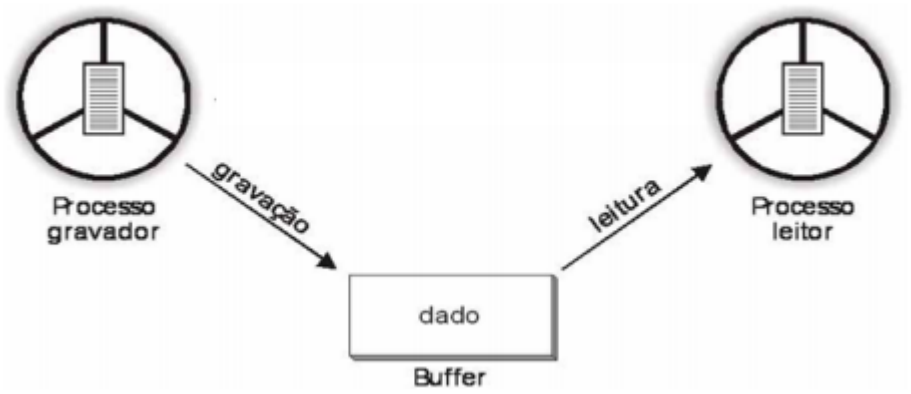
Comunicação entre Processos

- 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 entre Processos

- **Comunicação e Sincronização**

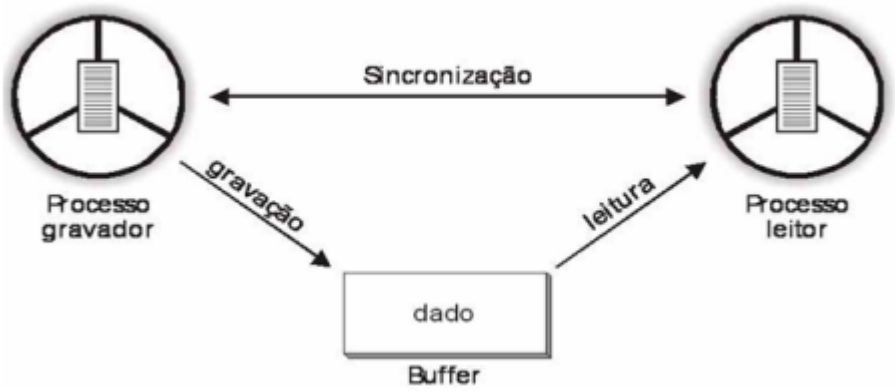
Por exemplo, podemos ter dois processos concorrentes compartilhando um buffer para trocar informações através de operações de gravação e leitura, onde ambos os processos necessitam aguardar que o buffer esteja pronto para realizar as respectivas operações, quer seja de leitura, quer seja de gravação.



Comunicação entre Processos

- **Comunicação e Sincronização**

Mas cada um dos processos podem ser executados com velocidades diferentes. Para cooperar, processos precisam se comunicar e se sincronizar.



Comunicação entre Processos

- **Problemas de Simultaneidade**

Um problema de concorrência simples para ser analisado, é a situação onde dois processos (A e B) executam um comando de atribuição.

O Processo A soma 1 a variável X e o Processo B diminui 1 da mesma variável que está sendo compartilhada.

Inicialmente, a variável X possui o valor 2,

Processo A X: = X + 1; Processo B X: = X – 1;

Seria razoável pensar que ao final da execução dos dois processos, o resultado da variável X continuasse 2, porém isto nem sempre será verdade!

Comunicação entre Processos

- **Problemas de Simultaneidade**

Os comandos de atribuição, em uma linguagem de alto nível, podem ser decompostos em comandos mais elementares.

Processo A X: = X + 1;

Processo A
LOAD X, Ra
ADD 1, Ra
STORE Ra, X

Processo B X: = X - 1;

Processo B
LOAD X, Rb
Sub 1, Rb
STORE Rb, X

Comunicação entre Processos

- **Problemas de Simultaneidade**

Considere que o Processo A carregue o valor de X no registrador Ra, some 1 e, no momento em que vai armazenar o valor em X, seja interrompido.

Nesse instante o Processo B inicia a sua execução, carrega o valor de X em Rb e subtrai 1.

Dessa vez, o Processo B é interrompido e o Processo A volta a ser processado, atribuindo o valor 3 à variável X e concluindo sua execução.

Finalmente, o Processo B retorna a execução, atribui o valor 1 a X, e sobrepõe o valor anteriormente gravado pelo Processo A.

O valor final da variável X é inconsistente em função da forma concorrente com que os dois processos executaram.

Comunicação entre 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

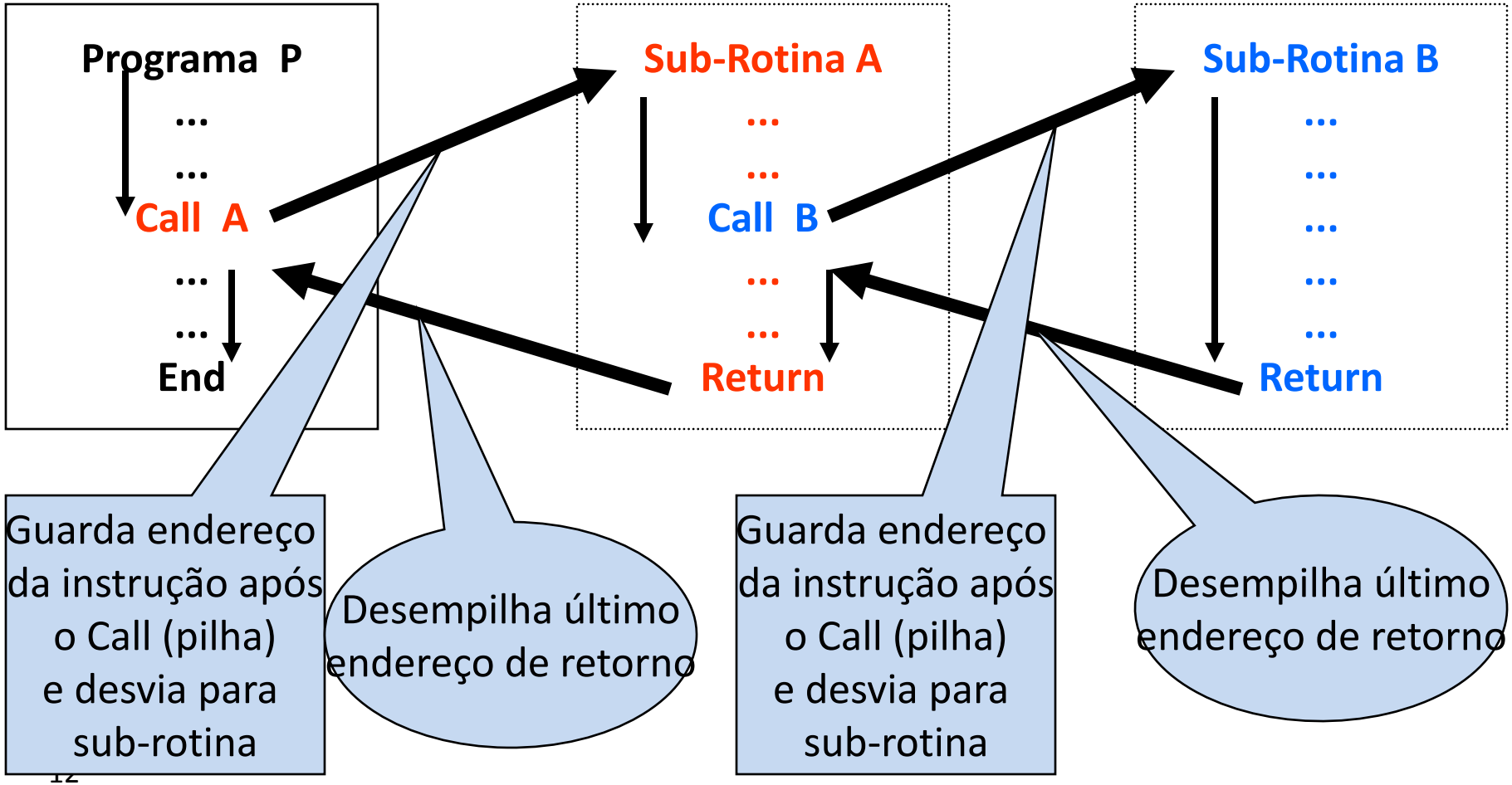
Comunicação entre Processos

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.

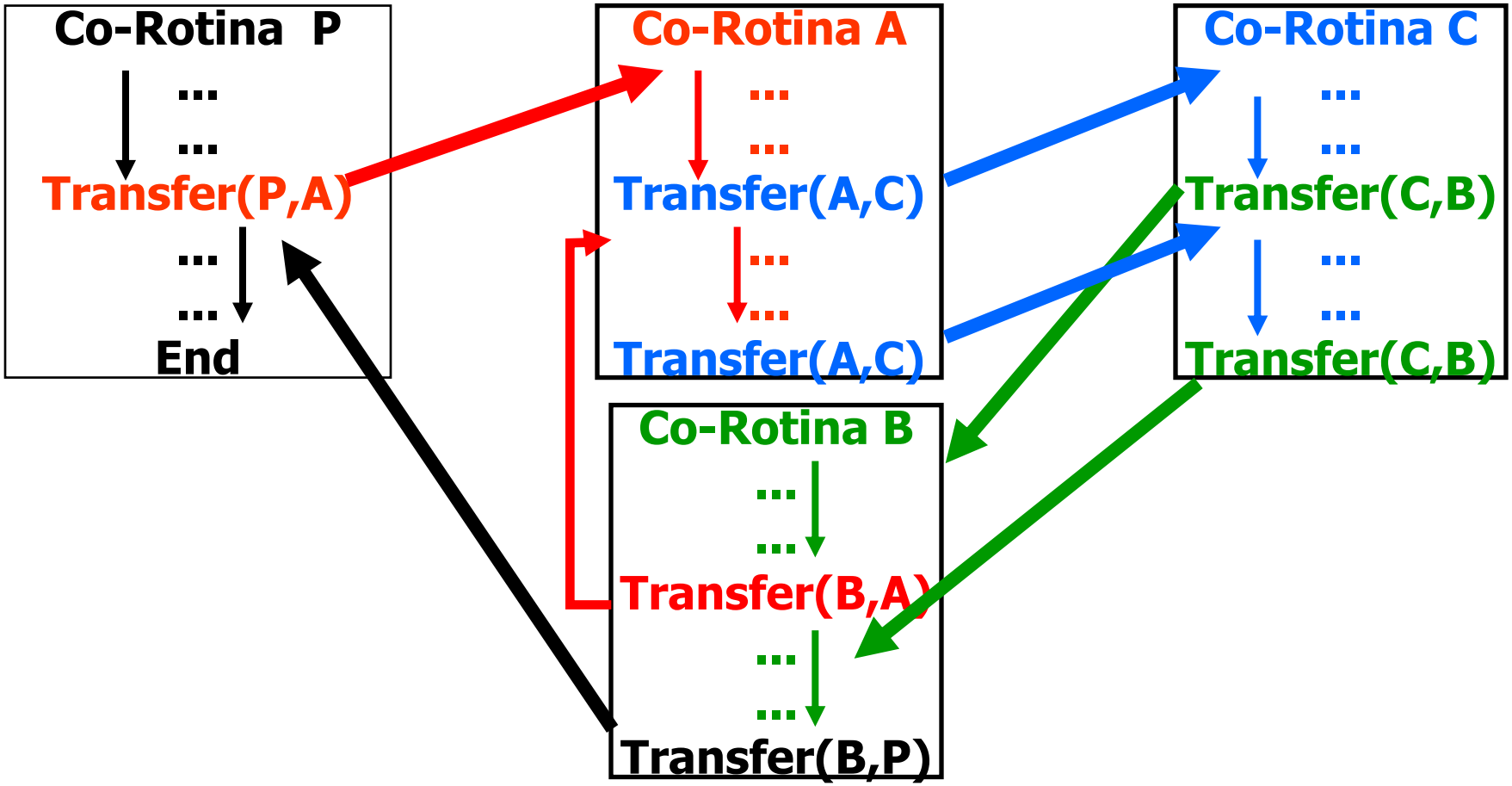
Comunicação entre Processos

Funcionamento das Sub-rotinas comuns



Comunicação entre Processos

Funcionamento das Co-Rotinas



Comunicação entre Processos

- **Declarações FORK/JOIN**

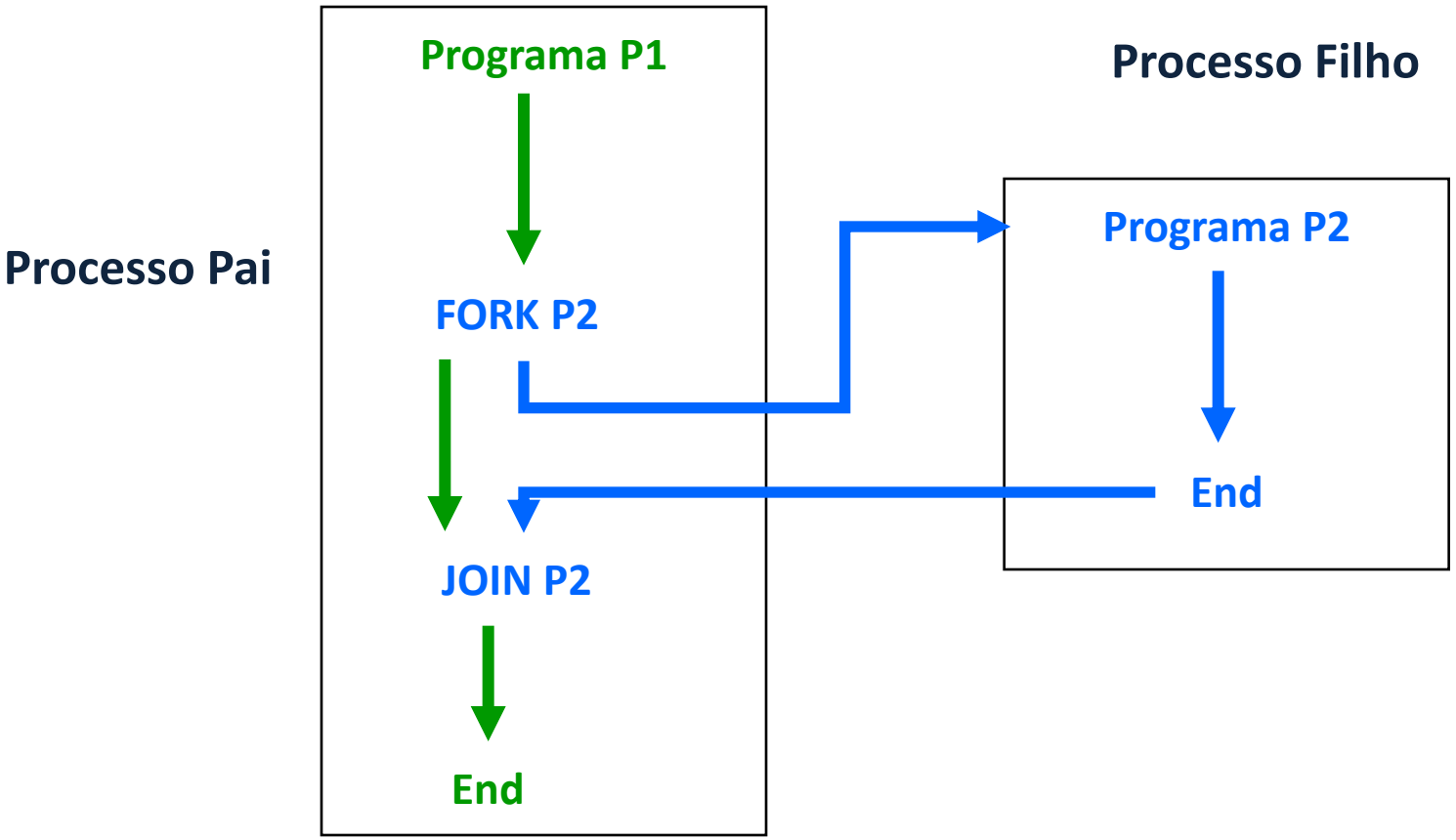
  - A declaração FORK <nome do programa> determina o início de execução de um determinado programa, de forma concorrente com o programa sendo executado.

  - 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 entre Processos

- **Declarações FORK/JOIN**



Comunicação entre Processos

- **Declarações COBEGIN/COEND**

- Constituem uma forma estruturada de especificar execução concorrente ou paralela de um conjunto de declarações agrupadas da seguinte maneira:

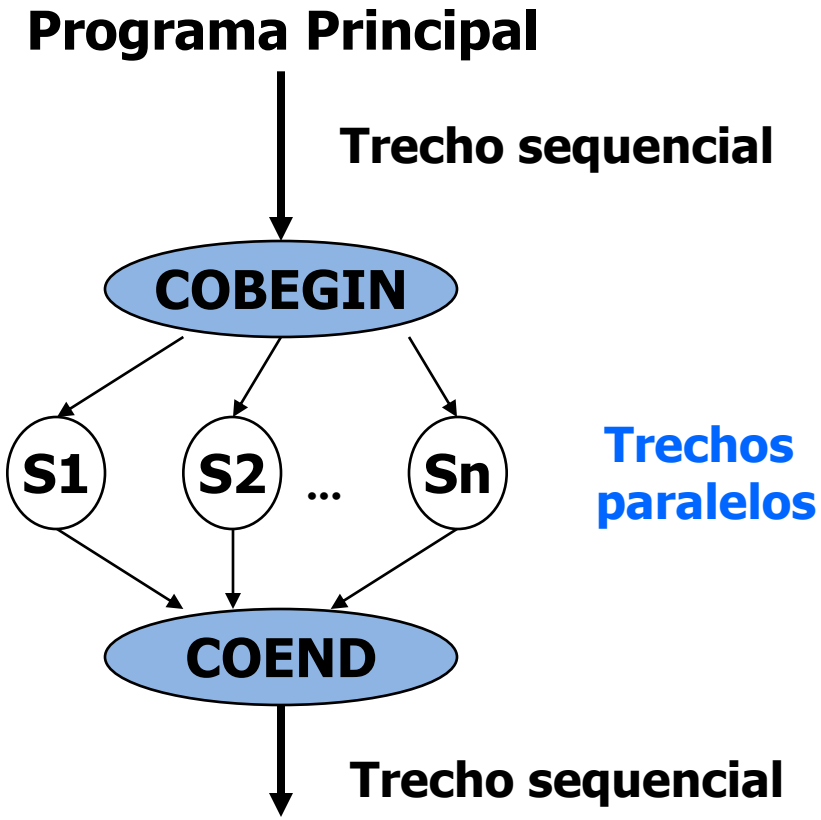
COBEGIN
S1//S2//...//Sn
COEND

Onde:

- **A execução deste trecho de programa provoca a execução concorrente das declarações S1, S2, ..., Sn.**
- **Declarações Si podem ser qualquer declaração, incluindo o para COBEGIN/COEND, ou um bloco de declarações locais.**
- **Esta execução só termina, quando todas as declarações Si terminarem.**

Comunicação entre 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.
```

Comunicação entre Processos

- **Declarações de Processos Concorrentes**

- Geralmente, programas de grande porte são estruturados como conjunto de trechos seqüenciais de programa, que são executados concorrentemente.

- Poderiam ser utilizadas as co-rotinas, FORK/JOIN, ou Cobegin/Coend, porém a estrutura de um programa será mais clara se a especificação dessas rotinas explicitar que as mesmas são executadas concorrentemente.

Comunicação entre Processos

Declarações de Processos Concorrentes

```
Program Conjunto_Processos;
/* declaração de var.e const. globais */
  ...
Processo Pi;
  /* declaração de var.e const. locais */
   ...
  End;
/* Outros  processos */
Processo Pn;
  /* declaração de var.e const. locais */
   ...
  End;
End.
```

Departamento de Computação

Comunicação entre 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 seguir, serão vistos alguns destes mecanismos e problemas da comunicação interprocessos.

Comunicação entre Processos
Condições de Corrida

- Analisando o exemplo apresentado, é possível concluir que em qualquer situação, onde dois ou mais processos compartilham um mesmo recurso, devem existir mecanismos de controle para evitar esse tipo de problema, conhecidos como **condições de corrida** (race conditions).

- Ocorre quando?

Quando dois ou mais processos estão lendo ou escrevendo dados compartilhados e o resultado final **depende** de **qual processo executa** e quando (em que ordem) este executa.

Comunicação entre Processos
Condições de Corrida

Um exemplo: Print Spooler

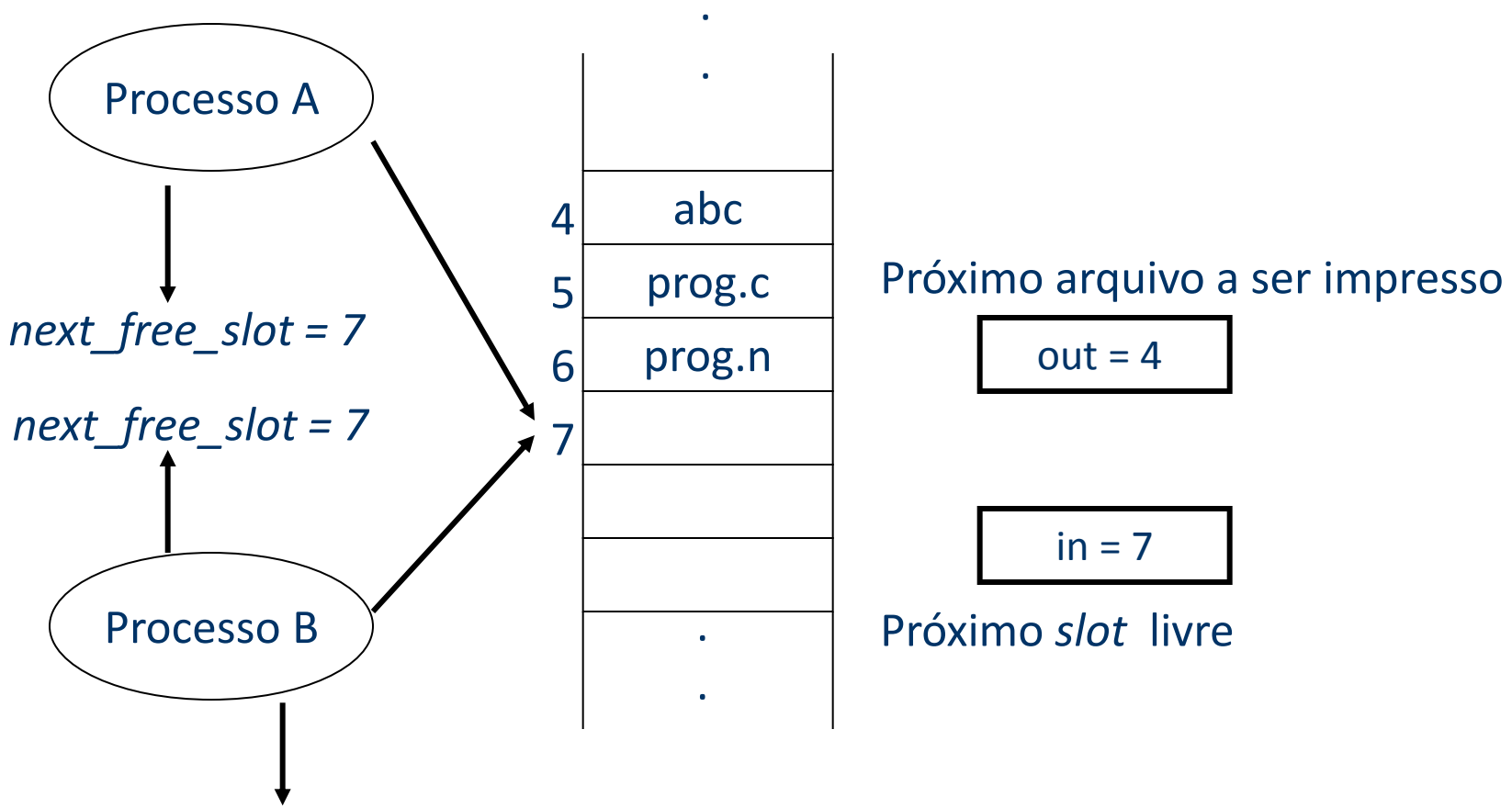
Quando um processo deseja imprimir um arquivo, ele coloca o nome do arquivo em uma lista de impressão (*spooler directory*).

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.

Ex.: Imagine dois processos que desejam ao mesmo tempo imprimir um arquivo...

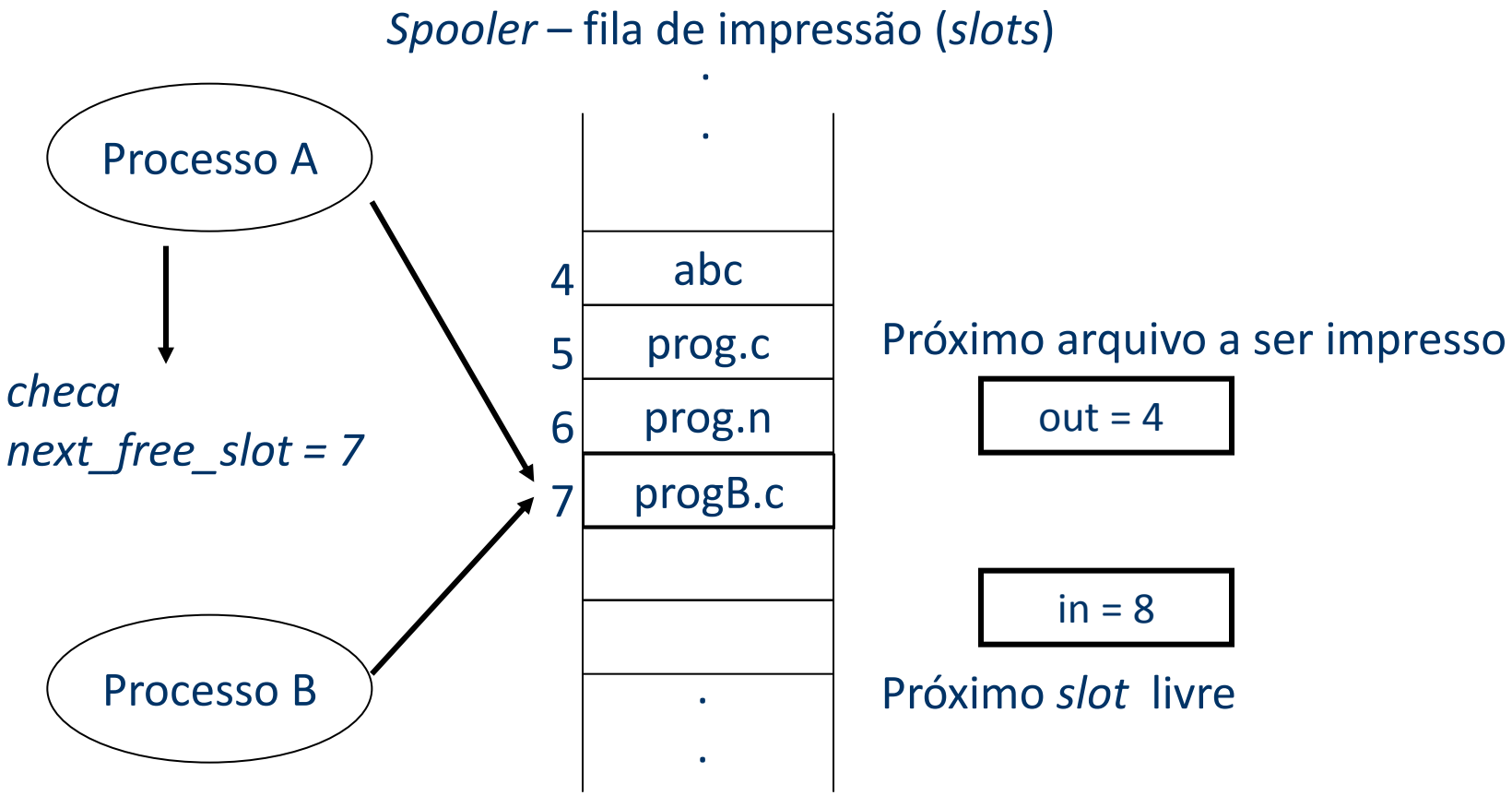
Comunicação entre Processos
Condições de Corrida

Spooler – fila de impressão (*slots*)

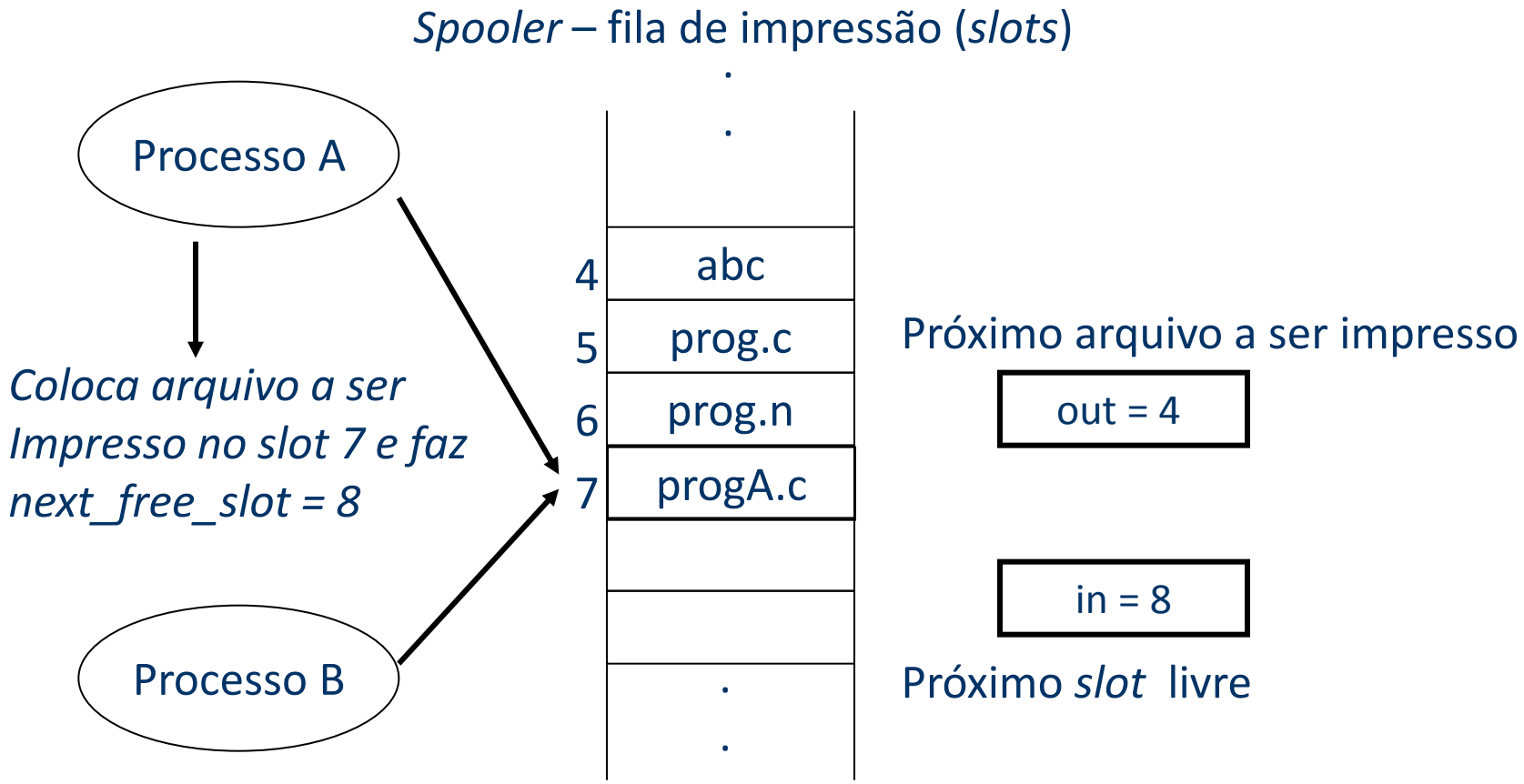


Coloca seu arquivo no *slot* 7 e
next_free_slot = 8

Comunicação entre Processos
Condições de Corrida



Comunicação entre Processos
Condições de Corrida



Processo B nunca receberá sua impressão!!!!!

Departamento de Computação

Comunicação entre Processos
Exclusão Mútua

- A solução mais simples para evitar os problemas de compartilhamento apresentados é impedir que dois ou mais processos acessem um mesmo recurso simultaneamente.
- Para isso, enquanto um processo estiver acessando determinado recurso, todos os demais processos que queiram acessá-lo deverão esperar pelo término da utilização do recurso.
- Essa ideia de exclusividade de acesso é chamada de exclusão mútua (*mutual exclusion*).

Comunicação entre 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)

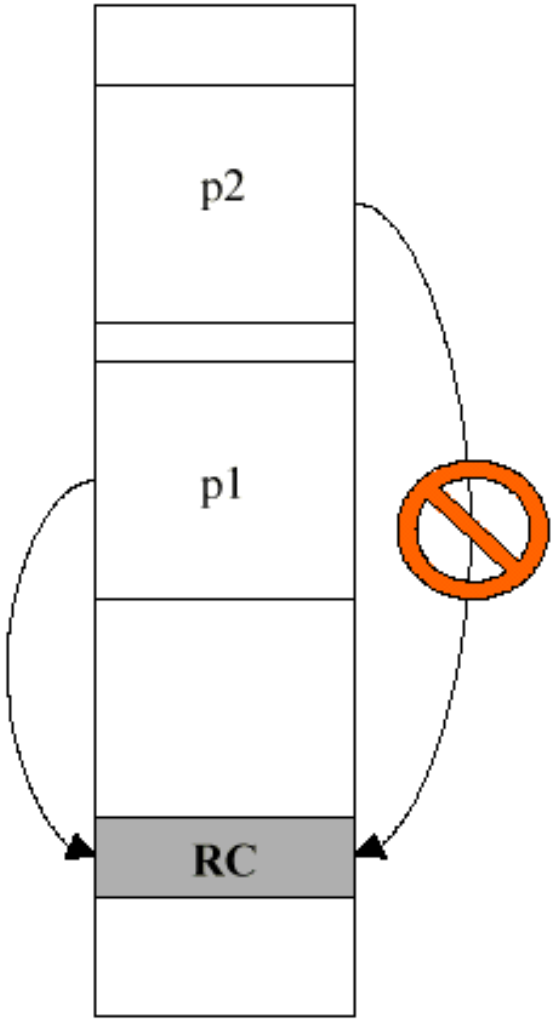
Departamento de Computação

Comunicação entre Processos
Regiões Críticas

- A exclusão mútua deve afetar apenas os processos concorrentes somente quando um deles estiver fazendo acesso ao recurso compartilhado.
- A parte do código do programa onde é feito o acesso ao recurso compartilhado é denominada região crítica (*critical region*).
- Se for possível evitar que dois processos entre em suas regiões críticas ao mesmo tempo, ou seja, se for garantida a execução mutuamente exclusiva das regiões críticas, os problemas decorrentes do compartilhamento serão evitados.

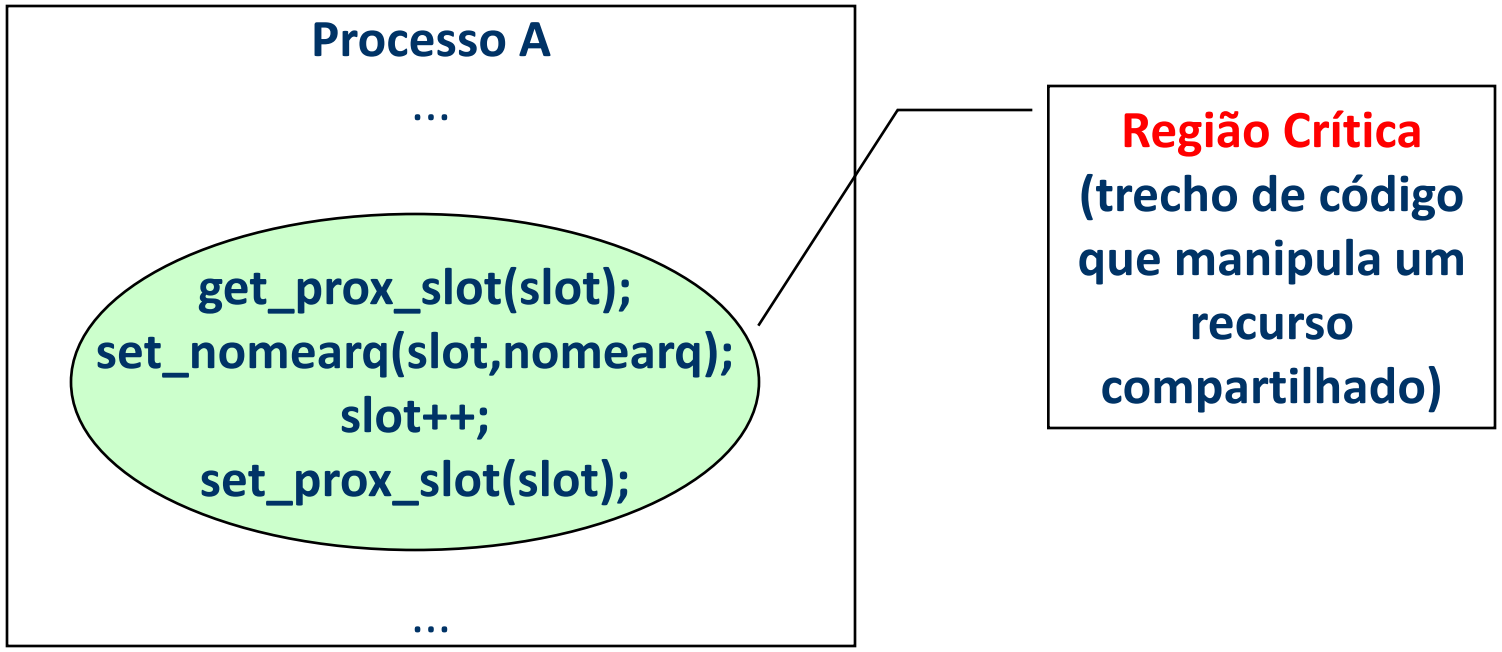
Comunicação entre Processos
Regiões Críticas

Regiões Críticas



Comunicação entre Processos
Regiões Críticas

Regiões Críticas



Departamento de Computação

Comunicação entre Processos
Regiões Críticas

- Os mecanismos que implementam a exclusão mútua utilizam protocolos de acesso à região crítica.
- Toda vez que um processo desejar executar uma instrução de sua região crítica, deverá obrigatoriamente executar antes um protocolo de entrada nesta região.
- Da mesma forma, ao sair da região crítica um protocolo de saída deverá ser executado.
- Estes protocolos garantem a exclusão mútua da região crítica do programa.

Comunicação entre 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?

Departamento de Computação

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;

Departamento de Computação

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 entre Processos
Regiões Críticas

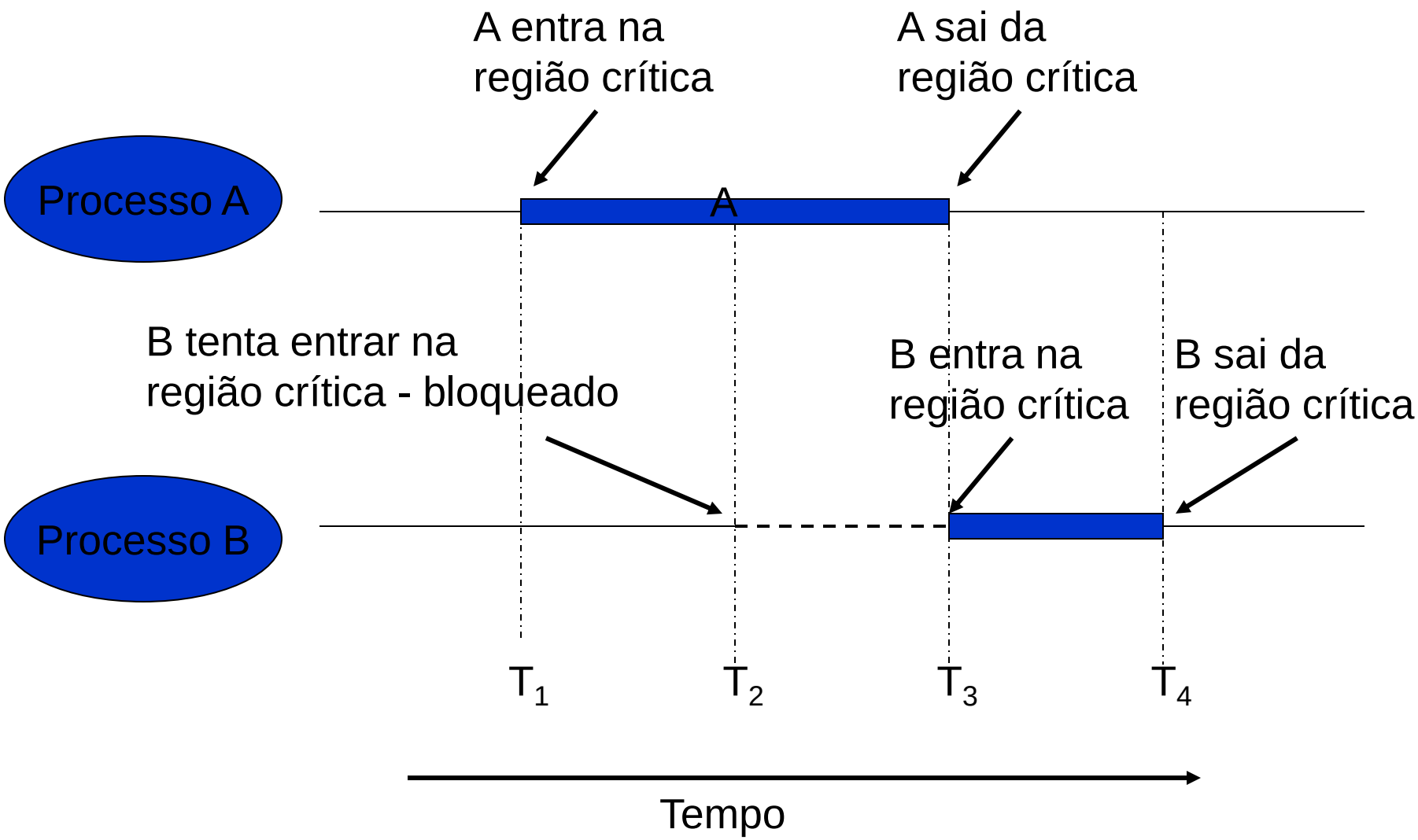
- Como solucionar problemas de *Condições de corrida*???
  - Proibir que mais de um processo leia ou escreva em recursos compartilhados concorrentemente (ao “mesmo tempo”)
    - **Recursos compartilhados → regiões críticas**;
  - Exclusão mútua: garantir que um processo não terá acesso à uma região crítica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

Departamento de Computação

Regiões Críticas e Exclusão Mútua

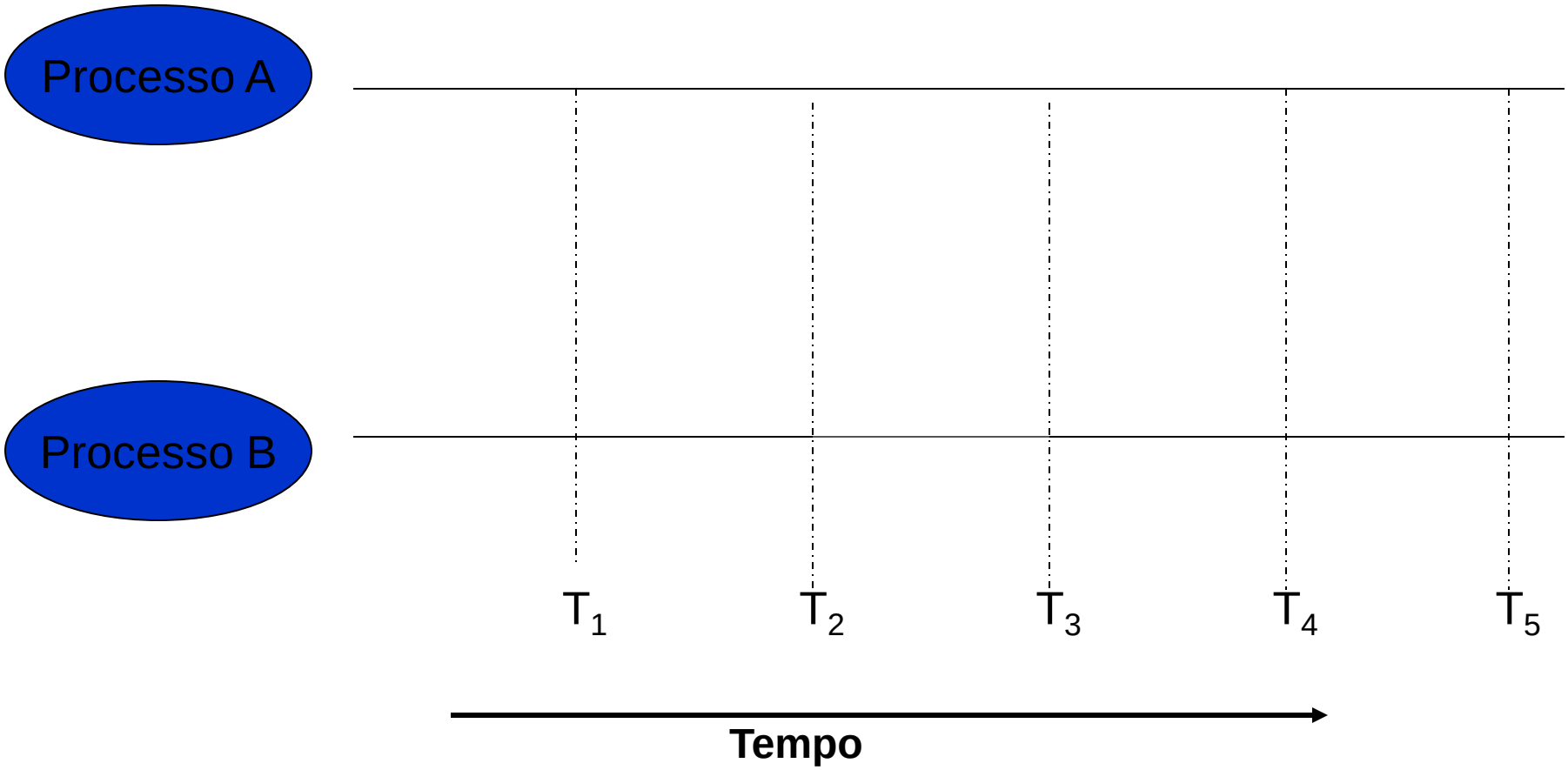
- Quatro condições para uma boa solução:
  - Dois ou mais processos não podem estar simultaneamente dentro de uma região crítica
  - Não se podem fazer assunções em relação à velocidade e ao número de CPUs
  - Um processo fora da região crítica não deve causar bloqueio a outro processo
  - Um processo não pode esperar infinitamente para entrar na região crítica

Comunicação entre Processos
Exclusão Mútua



Comunicação entre Processos
Exclusão Mútua

Volta a situação inicial!!!!



Comunicação entre Processos
Exclusão Mútua

- Exclusão Mútua com espera ociosa

Soluções

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

Departamento de Computação

Comunicação entre 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 entre Processos
Exclusão Mútua com espera ociosa

- ***Desabilitar interrupções:***

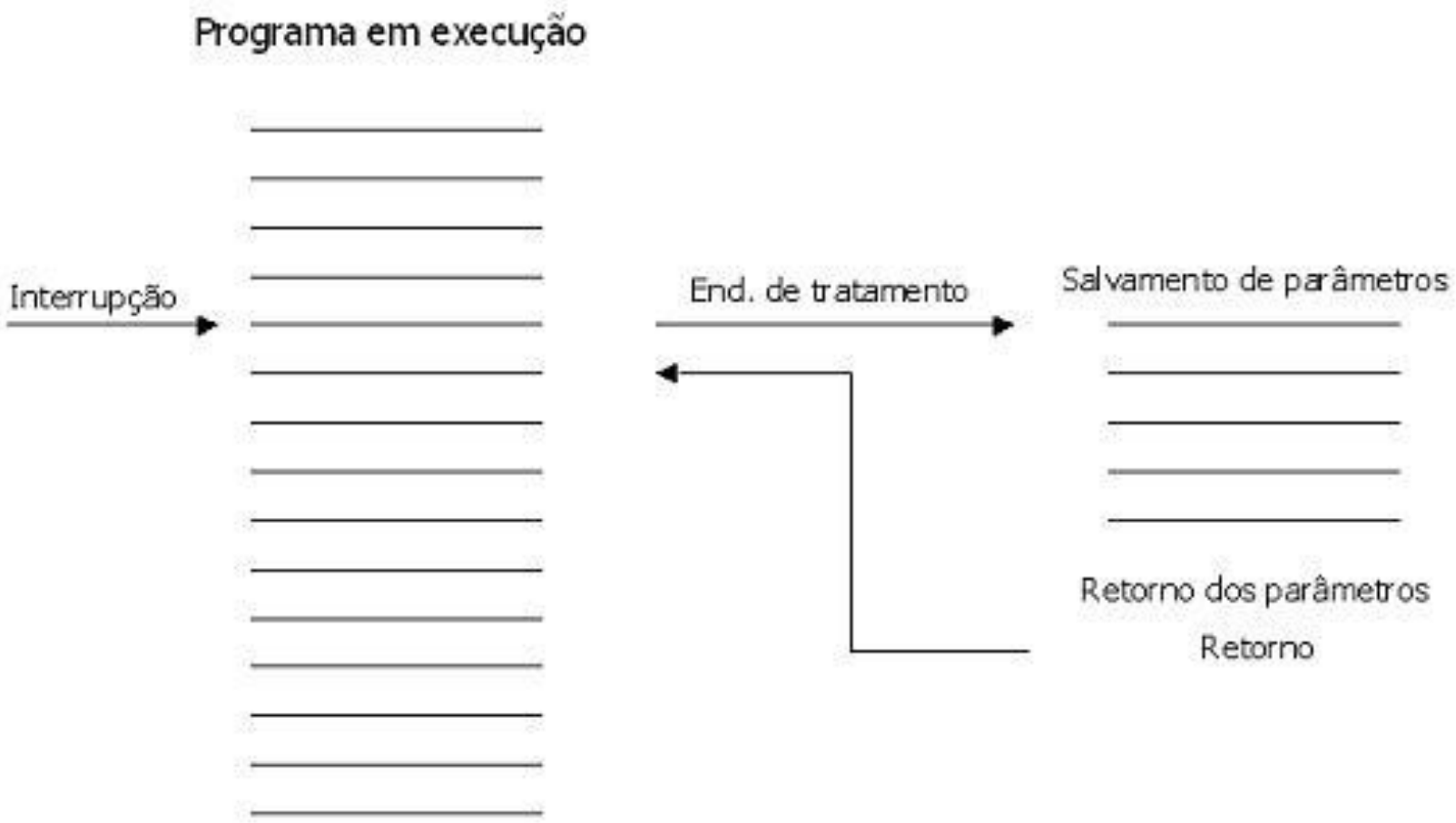
O que são interrupções?

  - 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 entre Processos
Exclusão Mútua com espera ociosa

- **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;
  - Não é uma solução segura, pois um processo pode não habilitar novamente suas interrupções e não ser finalizado;
    - Viola condição 4;

Comunicação entre Processos
Exclusão Mútua com espera ociosa



Comunicação entre Processos
Exclusão Mútua com espera ociosa

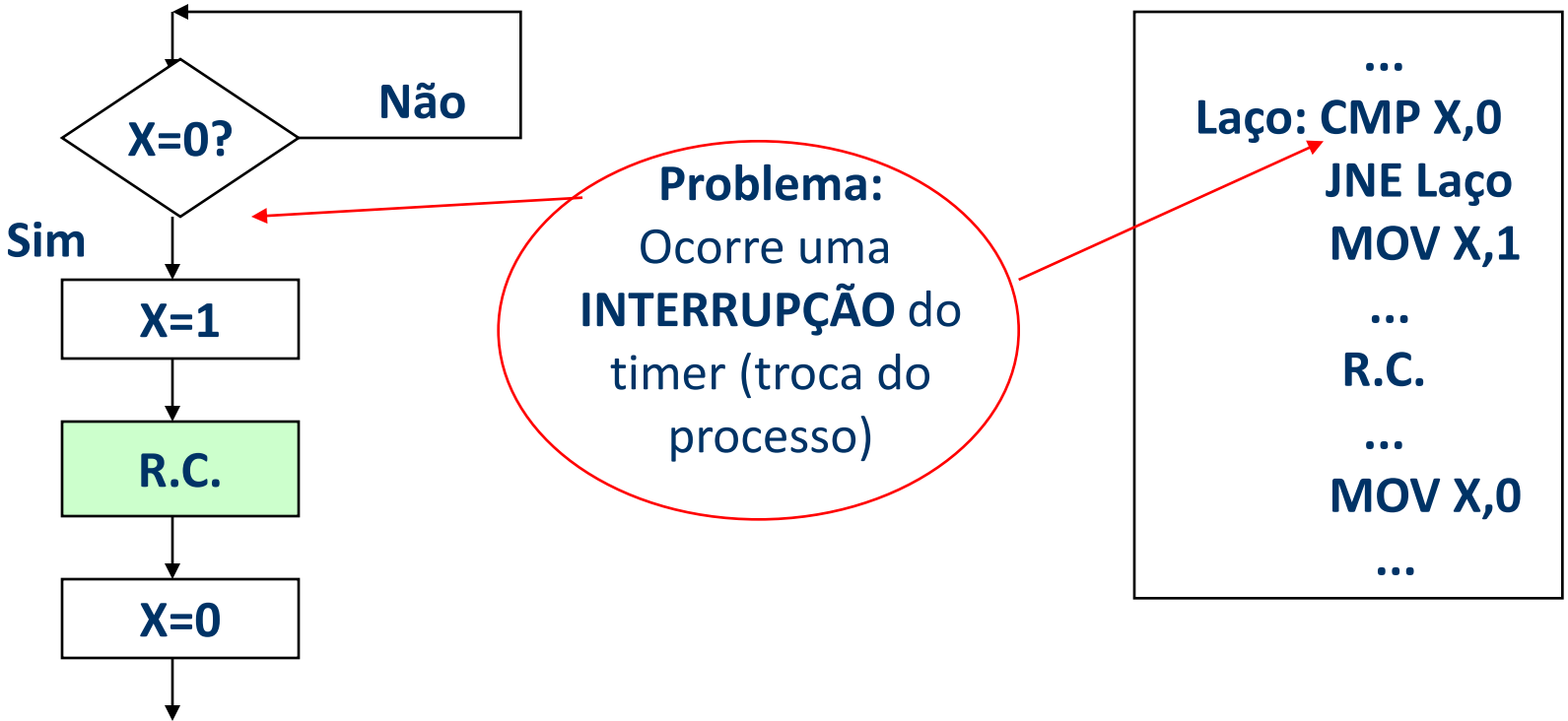
Desabilitando as Interrupções

- 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 entre Processos
Exclusão Mútua com espera ociosa

Variáveis de Trava

- Consiste no uso de uma **variável, compartilhada, de trava**. Se a variável está em zero, significa que nenhum processo está na R.C., e "1" significa que existe algum processo na R.C.



Comunicação entre Processos
Exclusão Mútua com espera ociosa

- **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 entre Processos
Exclusão Mútua com espera ociosa

- **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 entre Processos
Exclusão Mútua com espera ociosa

- 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 entre Processos
Exclusão Mútua com espera ociosa

- ***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)

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

(Processo B)

Comunicação entre Processos
Exclusão Mútua com espera ociosa

- **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 entre Processos
Exclusão Mútua com espera ociosa

- **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 <u>NÃO</u> está na sua região crítica, violando a condição 3**;

Departamento de Computação

Comunicação entre Processos
Exclusão Mútua com espera ociosa

- **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 entre Processos
Exclusão Mútua com espera ociosa

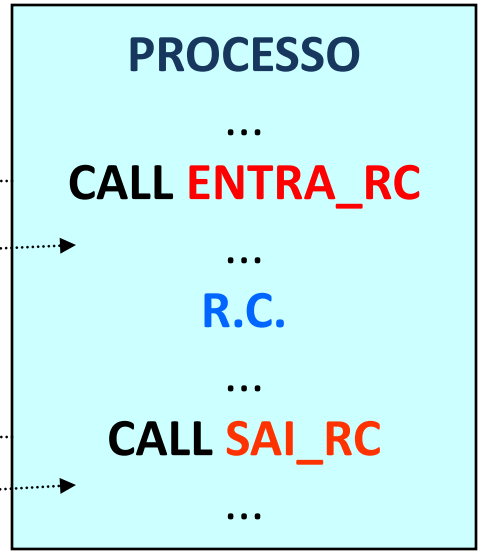
Instrução TSL (Test and Set Lock)

- 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 entre Processos
Exclusão Mútua com espera ociosa

Instrução TSL (Test and Set Lock) - Exemplo

```
ENTRA_RC:
    TSL reg, LOCK;   //copia lock para reg
                     //e coloca 1 em lock
    CMP reg,0    ;   //lock era zero?
    JNZ ENTRA_RC;  // se a trava fosse diferente de zero
                   // estaria ligado, e
                   // voltari  ao laço
    RET            //retorna pra quem chamou, entrou na
                   //regiáo crítica

SAI_RC:
    MOV lock,0  ;  //coloca lock em 0
    RET            //retorna pra quem chamou
```



Comunicação entre Processos
Exclusão Mútua com espera ociosa

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;

Departamento de Computação

Comunicação entre Processos
Primitivas *Sleep/Wakeup*

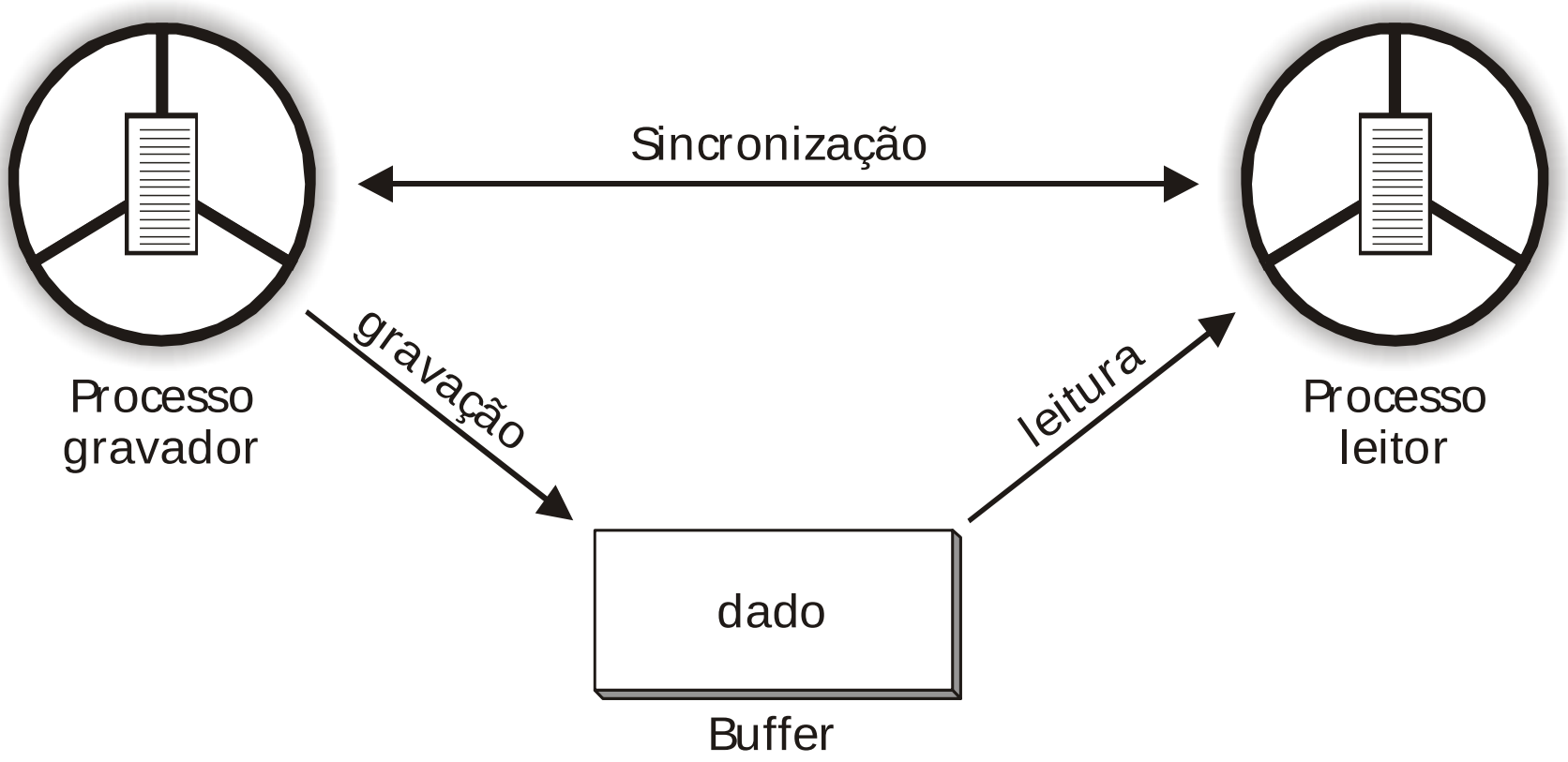
- Para solucionar esse problema de espera, um par de primitivas ***Sleep*** e ***Wakeup*** é utilizado → BLOQUEIO E DESBLOQUEIO de processos.
- A primitiva ***Sleep*** é uma chamada de sistema que bloqueia o processo que a chamou, ou seja, suspende a execução de tal processo até que outro processo o "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***;

Departamento de Computação

Comunicação entre 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 entre Processos
Sincronização Produtor-Consumidor



Comunicação entre 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);
   }
}
```

interrupção

Comunicação entre 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 entre 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...

Departamento de Computação

Comunicação entre Processos
Primitivas Sleep/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.

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 entre Processos
Semáforos

- Idealizado por E. W. Dijkstra (1965) como um mecanismo de sincronização que permitia implementar, de forma simples, a exclusão mútua e sincronização condicional entre processos. ;
- Variável inteira que armazena o número de sinais *wakeups* enviados;
- Um semáforo é uma variável inteira, não-negativa, que só pode ser manipulada por duas primitivas de chamadas de sistema:
  - `down` ***(sleep);***
  - `up` ***(wake).***
- Estas instruções são indivisíveis, isto é, não pode ser interrompida.

Comunicação entre Processos
Semáforos

- Semáforos são classificados em dois tipos:
  - **Semáforos binários**: chamados de mutexes (mutual exclusion semaphores), só podem assumir valores 0 e 1.

  Permitem a implementação da exclusão mútua e são utilizados nos casos onde a sincronização condicional é exigida.
  - **Semáforos contadores**: podem assumir qualquer valor inteiro positivo, além do 0.

  Semáforos contadores são úteis quando aplicados em problemas de sincronziação condicional onde existem processo concorrentes alocando recursos do mesmo tipo.

Departamento de Computação

Comunicação entre Processos
Semáforos

- Variável utilizada para controlar o acesso a recursos compartilhados
  - 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 entre Processos
Semáforos

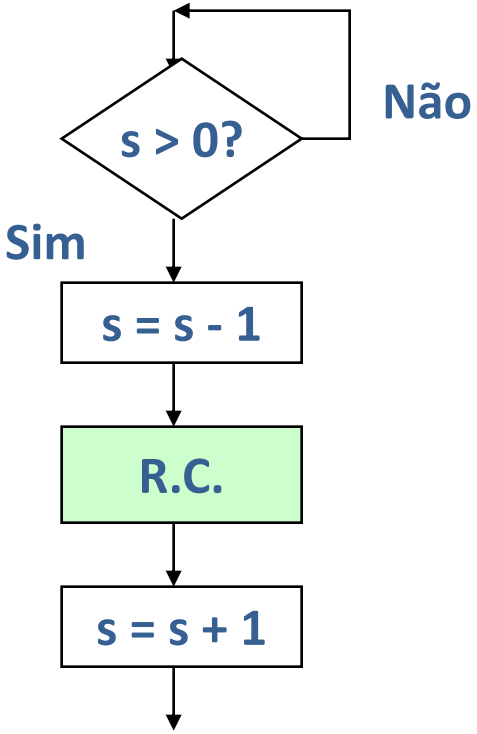
- down(semáforo)
  - Verifica se o valor do semáforo é maior que 0
  - Se for, semáforo=semáforo – 1
  - Se não for, o processo que executou o down bloqueia
- up(semáforo)
  - semáforo=semáforo + 1
  - Se há processos bloqueados nesse semáforo, escolhe um deles e o desbloqueia
    - Nesse caso, o valor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

Operações sobre semáforos são atômicas.

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 entre Processos
Semáforos

SEMÁFOROS

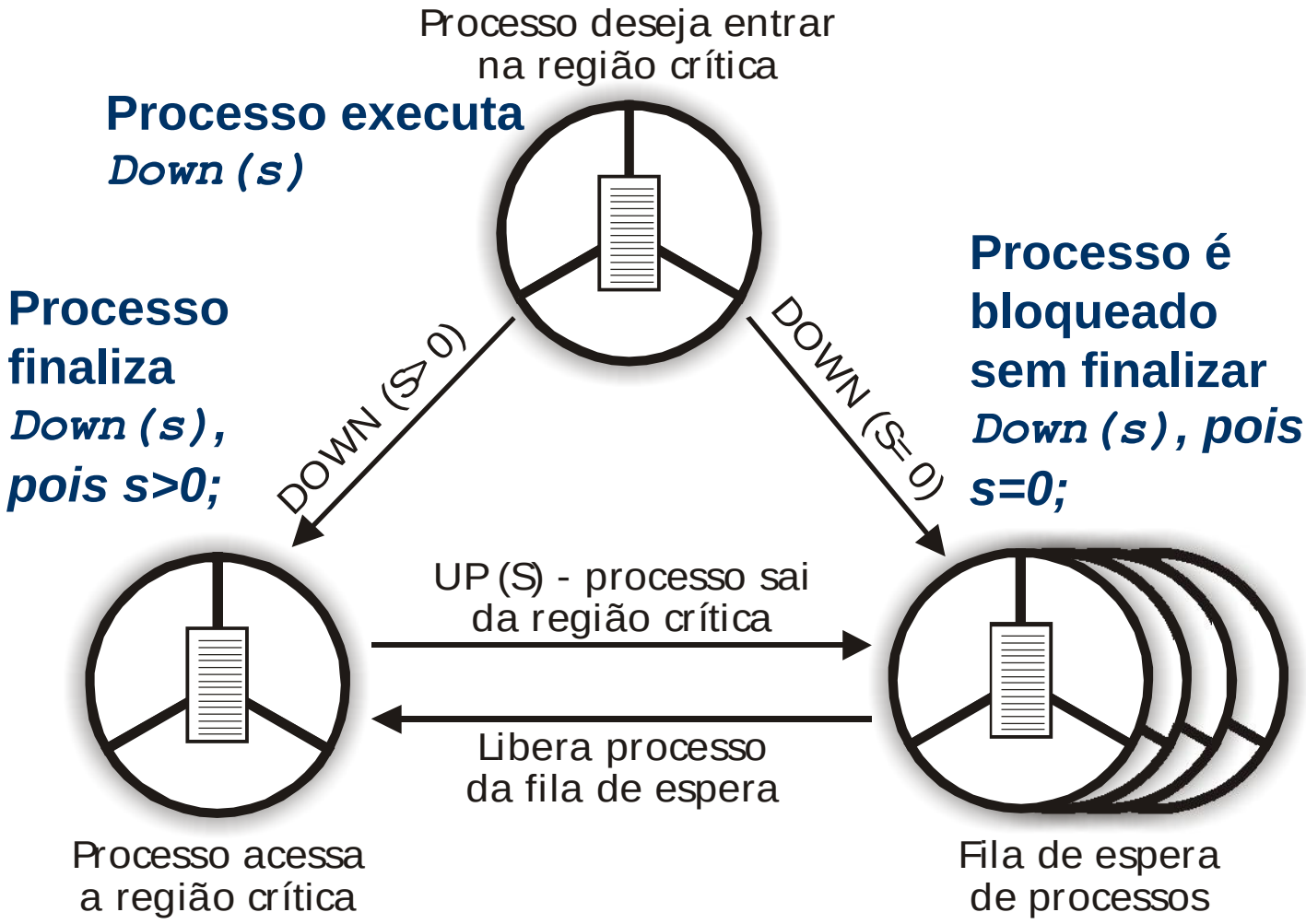
2ª. Implementação – Associando uma fila Qi a cada semáforo si

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)
senão bloqueia o processo, colocando-o na fila Qi;

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

Comunicação entre Processos
Semáforo Binário



Comunicação entre 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 entre 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.

Soluções

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

Departamento de Computação

Comunicação entre 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;
- O monitor é formado por procedimentos e variáveis encapsulados dentro de um módulo, implementando de forma automática a exclusão mútua entre os procedimentos declarado.

Comunicação entre Processos
Monitores

```
monitor example
   int i;
   condition c;

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

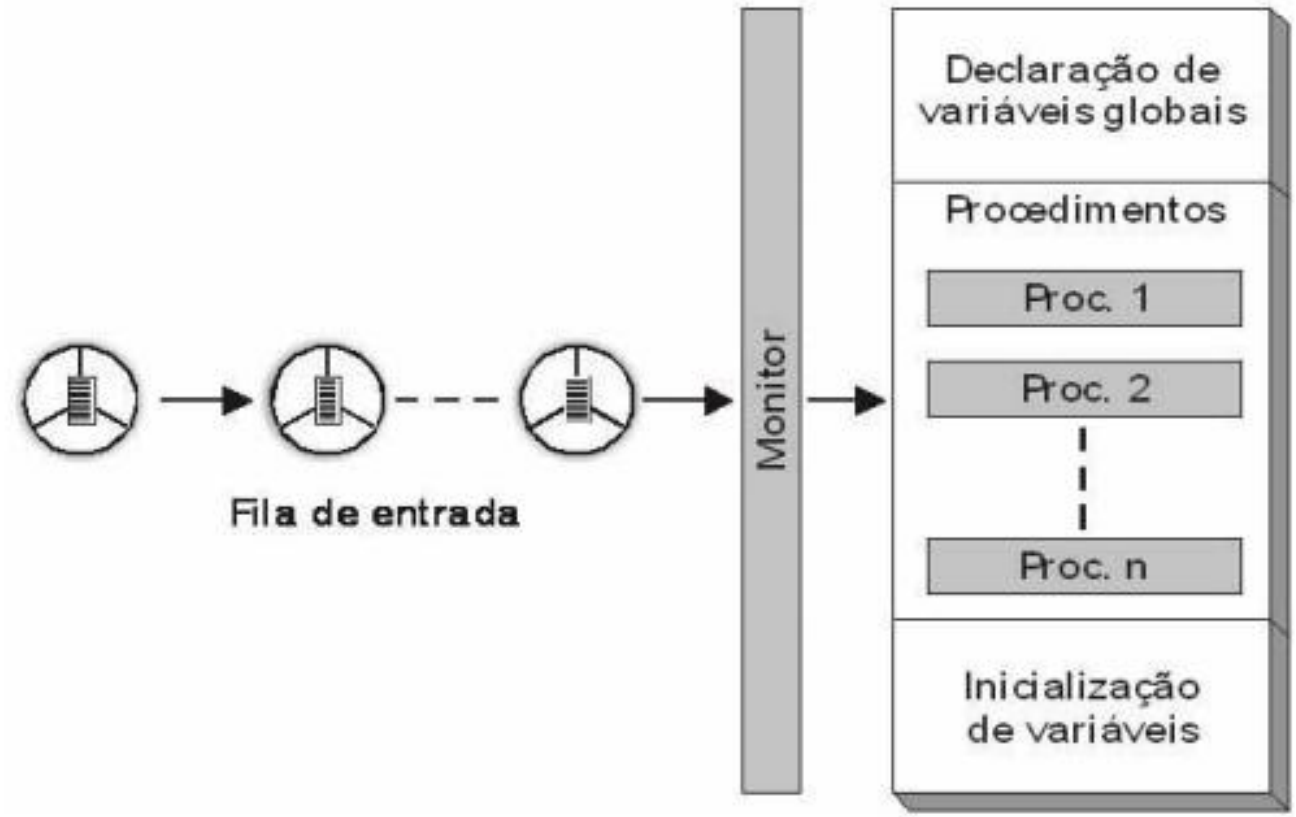
Estrutura básica de um Monitor

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 entre Processos
Monitores



Comunicação entre 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 entre Processos
Monitores

- *Condition Variables* (`condition`): variáveis que indicam uma condição; e
- Operações Básicas: `WAIT e SIGNAL`

  `wait (condition)` → bloqueia o processo;

  `signal (condition)` → "acorda" o processo que executou um `wait` na variável `condition` e foi bloqueado;

Comunicação entre Processos
Monitores

- Variáveis condicionais não são contadores, portanto, não acumulam sinais;
- Se um sinal é enviado sem ninguém (processo) estar esperando, o sinal é perdido;
- Assim, um comando `WAIT` deve vir antes de um comando `SIGNAL`.

Comunicação entre Processos
Monitores

- Como evitar dois processos ativos no monitor ao mesmo tempo?
- (1) Hoare → colocar o processo mais recente para rodar, suspendendo o outro!!! (*sinalizar e esperar*)
- (2) B. Hansen → um processo que executa um `SIGNAL` deve deixar o monitor imediatamente;
  - O comando `SIGNAL` deve ser o último de um procedimento do monitor;

A condição (2) é mais simples e mais fácil de se implementar.

Departamento de Computação

Comunicação entre 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;

Problemas Clássicos de Comunicação entre processos

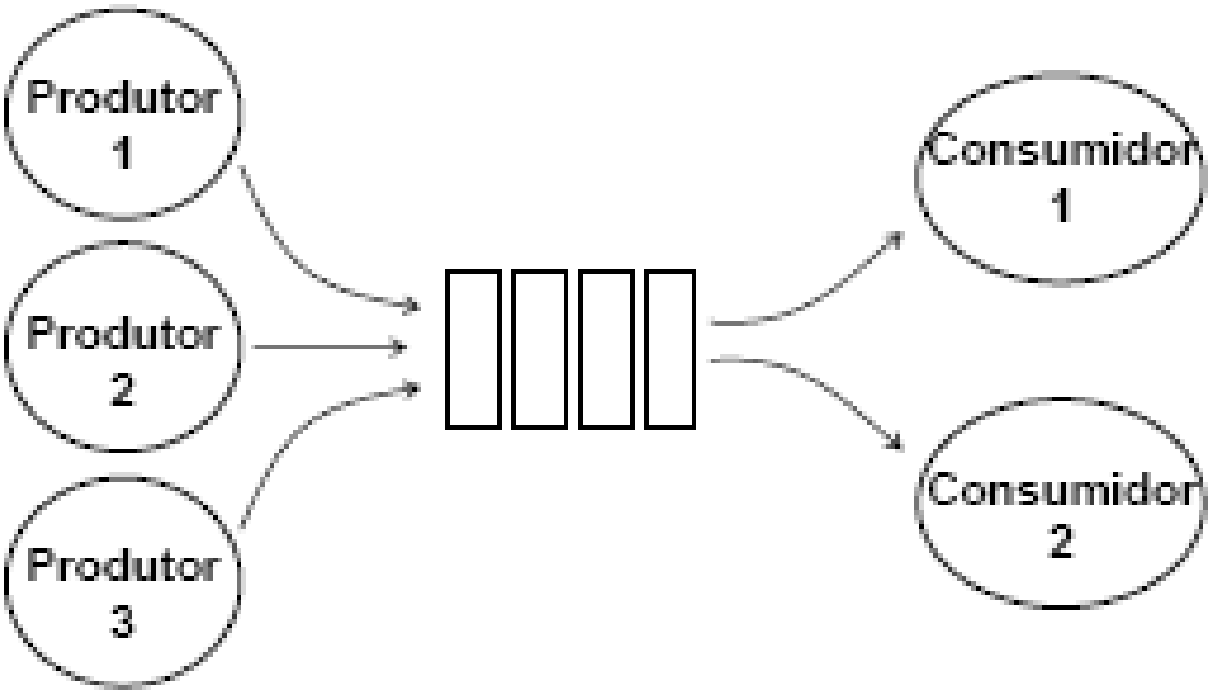
Departamento de Computação

Produtor - Consumidor

- Um sistema é composto por entidades produtoras e entidades consumidoras.
- Entidades produtoras
  - Responsáveis pela produção de itens que são armazenados em um buffer (ou em uma fila)
  - Itens produzidos podem ser consumidos por qualquer consumidor
- Entidades consumidoras
  - Consomem os itens armazenados no buffer (ou na fila)
  - Itens consumidos podem ser de qualquer produtor

Departamento de Computação

Produtor - Consumidor



Departamento de Computação

Leitores - Escritores

- Um sistema com uma base de dados é acessado simultaneamente por diversas entidades. Estas entidades realizam dois tipos de operações:
  - Leitura
  - Escrita
- Neste sistema é aceitável a existência de diversas entidades lendo a base de dados.
- Porém, se um processo necessita escrever na base, nenhuma outra entidade pode estar realizando acesso à base.

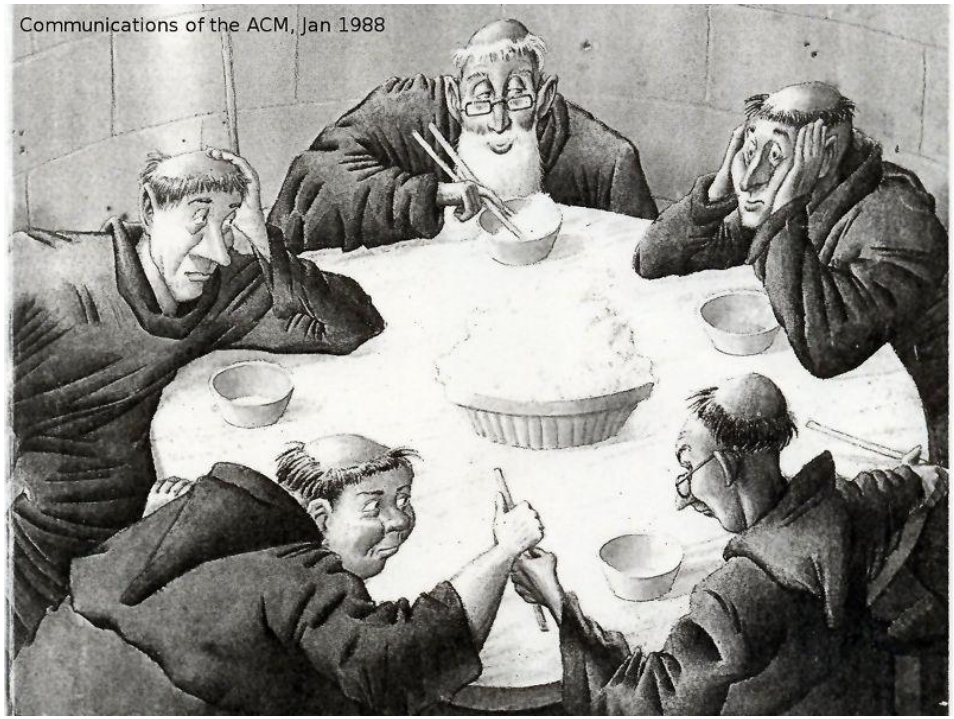
O Problema dos Leitores e Escritores

```
typedef int semaphore;          /* use sua imaginação */
semaphore mutex = 1;            /* controla o acesso a 'rc' */
semaphore db = 1;               /* controla o acesso a base de dados */
int rc = 0;                     /* número de processos lendo ou querendo ler */

void reader(void)
{
    while (TRUE) {              /* repete para sempre */
        down(&mutex);           /* obtém acesso exclusivo a 'rc' */
        rc = rc + 1;            /* um leitor a mais agora */
        if (rc == 1) down(&db); /* se este for o primeiro leitor ... */
        up(&mutex);             /* libera o acesso exclusivo a 'rc' */
        read_data_base( );      /* acesso aos dados */
        down(&mutex);           /* obtém acesso exclusivo a 'rc' */
        rc = rc - 1;            /* um leitor a menos agora */
        if (rc == 0) up(&db);   /* se este for o último leitor ... */
        up(&mutex);             /* libera o acesso exclusivo a 'rc' */
        use_data_read( );       /* região não crítica */
    }
}


void writer(void)
{
    while (TRUE) {              /* repete para sempre */
        think_up_data( );       /* região não crítica */
        down(&db);              /* obtém acesso exclusivo */
        write_data_base( );     /* atualiza os dados */
        up(&db);                /* libera o acesso exclusivo */
    }
}
```

Uma solução para o problema dos leitores e escritores

Problemas clássicos de comunicação entre processos

- **Problema do Jantar dos Filósofos**
  - Cinco filósofos desejam comer espaguete;
  - No entanto, para poder comer, cada filósofo precisa utilizar dois garfo e não apenas um. Portanto, os filósofos precisam compartilhar o uso do garfo de forma sincronizada.
  - Os filósofos comem e pensam;
  - Cada um precisa de 2 garfos para comer;
  - Pega um garfo por vez.

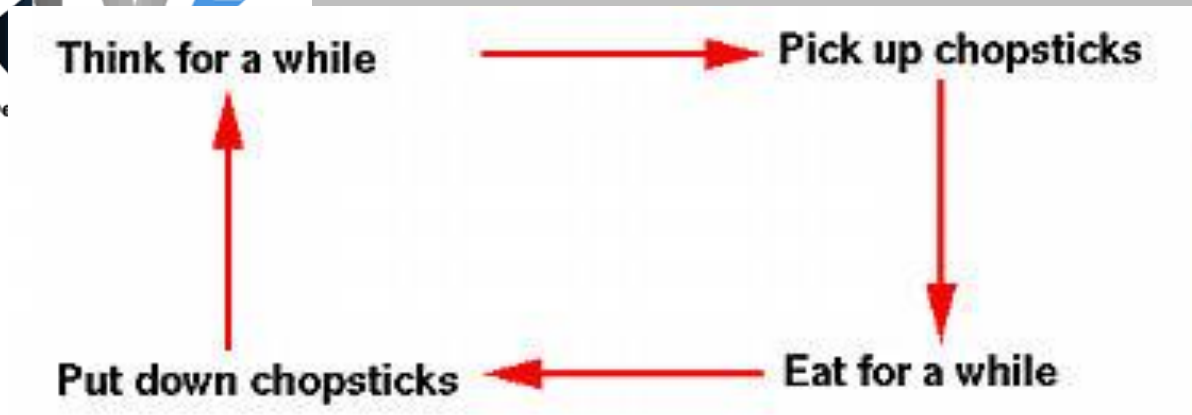


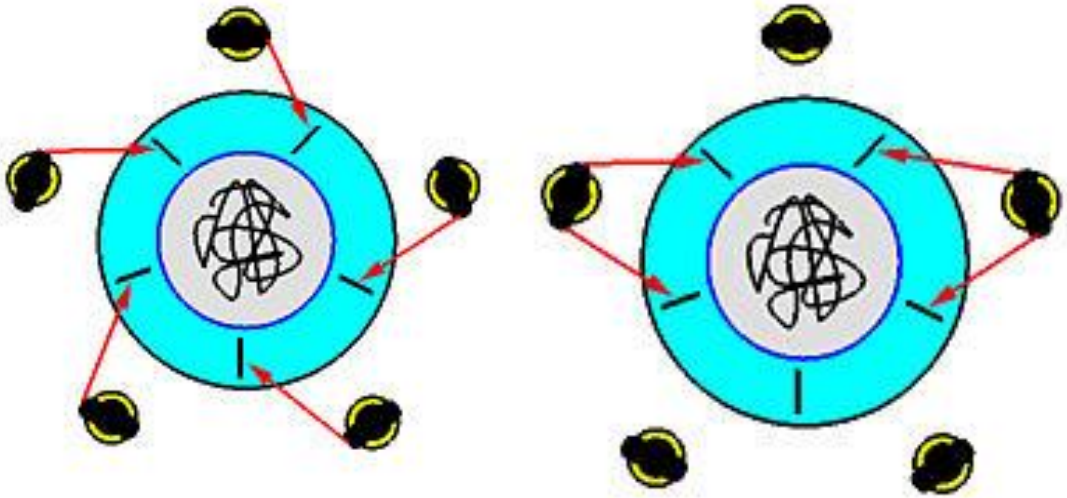
Como prevenir deadlock??

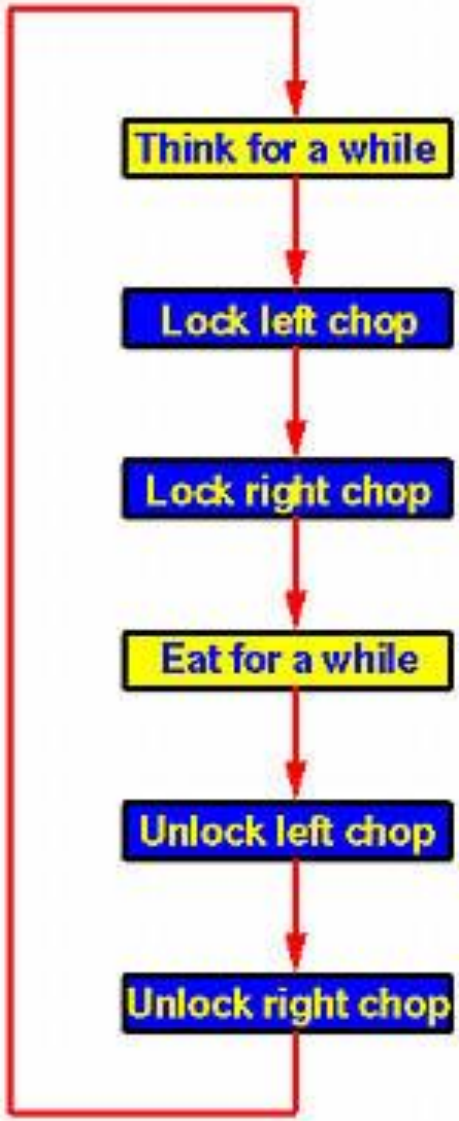
Departamento de Computação

Problemas clássicos de comunicação entre processos

- A vida do filósofo consiste na alternância de períodos de alimentação e reflexão.
  - Quando um filósofo fica com fome, ele tenta pegar os garfos a sua volta (garfos a sua esquerda e direita), em qualquer ordem, um de cada vez.
  - Se o filósofo conseguir pegar os dois garfos ele inicia seu período de alimentação. Após algum tempo ele devolve os garfos a sua posição original e retorna ao período de reflexão


Think for a while
Pick up chopsticks
Eat for a while
Put down chopsticks




Think for a while
Lock left chop
Lock right chop
Eat for a while
Unlock left chop
Unlock right chop

Problemas clássicos de comunicação entre processos

- Problemas que devem ser evitados:
  - *Deadlock* – todos os filósofos pegam **um garfo** ao mesmo tempo;
  - *Starvation* – os filósofos **fiquem indefinidamente pegando garfos simultaneamente**;

Solução 1 para Filósofos (1/2)

```
#define N 5                                     /* number of philosophers */

void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
    }
}
```

Solução 1 para Filósofos (2/2)

- Problemas da solução 1:
  - Execução do `take_fork(i)` → Se todos os filósofos pegarem o garfo da esquerda, nenhum pega o da direita → *Deadlock*;
- Se modificar a solução (mudança 1):
  - Verificar antes se o garfo da direita está disponível. Se não está, devolve o da esquerda e começa novamente → *Starvation (Inanição);*
  - Tempo fixo ou tempo aleatório;

Solução 1 para Filósofos (2/2)

- Se modificar a solucão (mudanca 2):

```
#define N 5                                     /* number of philosophers */
semaphore mutex = 1;
void philosopher(int i)                         /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                               /* philosopher is thinking */
        down(&mutex);
        take_fork(i);                           /* take left fork */
        take_fork((i+1) % N);                   /* take right fork; % is modulo operator */
        eat( );                                 /* yum-yum, spaghetti */
        put_fork(i);                            /* put left fork back on the table */
        put_fork((i+1) % N);                    /* put right fork back on the table */
        up(&mutex);
    }
}
```

Somente um filósofo come!

Departamento de Computação

Solução 2 para Filósofos usando Semáforos (1/3)

- Não apresenta:
  - *Deadlocks*;
  - *Starvation*;
- Permite o máximo de “paralelismo”;

Solução 2 para Filósofos usando Semáforos (2/3)

```
#define N              5                    /* number of philosophers */
#define LEFT           (i+N−1)%N            /* number of i's left neighbor */
#define RIGHT          (i+1)%N              /* number of i's right neighbor */
#define THINKING       0                    /* philosopher is thinking */
#define HUNGRY         1                    /* philosopher is trying to get forks */
#define EATING         2                    /* philosopher is eating */
typedef int semaphore;                      /* semaphores are a special kind of int */
int state[N];                               /* array to keep track of everyone's state */
semaphore mutex = 1;                        /* mutual exclusion for critical regions */
semaphore s[N];                             /* one semaphore per philosopher */

void philosopher(int i)                     /* i: philosopher number, from 0 to N−1 */
{
    while (TRUE) {                          /* repeat forever */
        think( );                           /* philosopher is thinking */
        take_forks(i);                      /* acquire two forks or block */
        eat( );                             /* yum-yum, spaghetti */
        put_forks(i);                       /* put both forks back on table */
    }
}
```

Solução 2 para Filósofos usando Semáforos (3/3)

```
void take_forks(int i)                  /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = HUNGRY;              /* record fact that philosopher i is hungry */
        test(i);                        /* try to acquire 2 forks */
        up(&mutex);                     /* exit critical region */
        down(&s[i]);                    /* block if forks were not acquired */
}

void put_forks(i)                       /* i: philosopher number, from 0 to N–1 */
{
        down(&mutex);                   /* enter critical region */
        state[i] = THINKING;            /* philosopher has finished eating */
        test(LEFT);                     /* see if left neighbor can now eat */
        test(RIGHT);                    /* see if right neighbor can now eat */
        up(&mutex);                     /* exit critical region */
}

void test(i)                            /* i: philosopher number, from 0 to N–1 */
{
        if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
                state[i] = EATING;
                up(&s[i]);
        }
}
```

Departamento de Computação

Barbeiro Sonolento

- Uma barbearia possui:
  - 1 barbeiro
  - 1 cadeira de barbeiro
  - N cadeira para espera de clientes
- Se, em um determinado momento, não houverem clientes para serem atendidos, o barbeiro dorme.
  - Quando um cliente chega, ele acorda e atende o cliente.
  - Quando um cliente chega e o barbeiro estiver atendendo um cliente, ele aguarda sua vez sentado na cadeira de espera.
  - Quando um cliente chega e não existem cadeiras de espera disponíveis, o cliente vai embora.

O Problema do Barbeiro Sonolento (1)

O Problema do Barbeiro Sonolento (2)

```
#define CHAIRS 5                    /* número de cadeiras para os clientes à espera */

typedef int semaphore;              /* use sua imaginação */

semaphore customers = 0;            /* número de clientes à espera de atendimento*/
semaphore barbers = 0;              /* número de barbeiros à espera de clientes */
semaphore mutex = 1;                /* para exclusão mútua */
int waiting = 0;                    /* clientes estão esperando (não estão cortando) */

void barber(void)
{
    while (TRUE) {
        down(&customers);           /* vai dormir se o número de clientes for 0 */
        down(&mutex);               /* obtém acesso a 'waiting' */
        waiting = waiting – 1;      /* decresce de um o contador de clientes à espera */
        up(&barbers);               /* um barbeiro está agora pronto para cortar cabelo */
        up(&mutex);                 /* libera 'waiting' */
        cut_hair( );                /* corta o cabelo (fora da região crítica) */
    }
}

void customer(void)
{
    down(&mutex);                   /* entra na região crítica */
    if (waiting < CHAIRS) {         /* se não houver cadeiras livres, saia */
        waiting = waiting + 1;      /* incrementa o contador de clientes à espera */
        up(&customers);             /* acorda o barbeiro se necessário */
        up(&mutex);                 /* libera o acesso a 'waiting' */
        down(&barbers);             /* vai dormir se o número de barbeiros livres for 0 */
        get_haircut( );             /* sentado e sendo servido */
    } else {
        up(&mutex);                 /* a barbearia está cheia; não espere */
    }
}
```

Solução para o problema do barbeiro sonolento

Alguns links interessantes

- http://www.doc.ic.ac.uk/~jnm/concurrency/classes/Diners/Diners.html
- http://journals.ecs.soton.ac.uk/java/tutorial/java/threads/deadlock.html
- http://users.erols.com/ziring/diningAppletDemo.html

Soluções

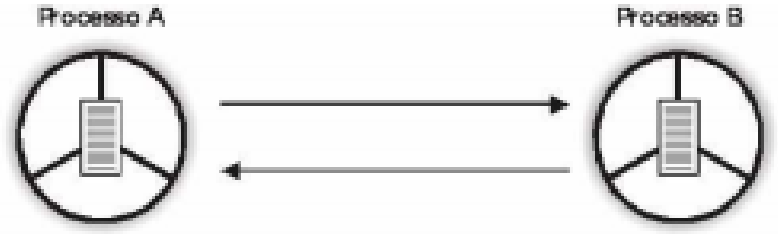
- Exclusão Mútua:
  - Espera Ocupada;
  - Primitivas *Sleep/Wakeup*;
  - Semáforos;
  - Monitores;
  - **Passagem de Mensagem**;

Comunicação entre Processos
Passagem de Mensagem

- Os processos cooperativos podem fazer uso de um buffer para trocar mensagens através de duas rotinas: **SEND** (receptor, mensagem) e **RECEIVE** (transmissor, mensagem).
- A rotina **SEND** permite o envio de uma mensagem para um processo receptor, enquanto a rotina **RECEIVE** possibilita o recebimento de mensagem enviada por um processo transmissor.

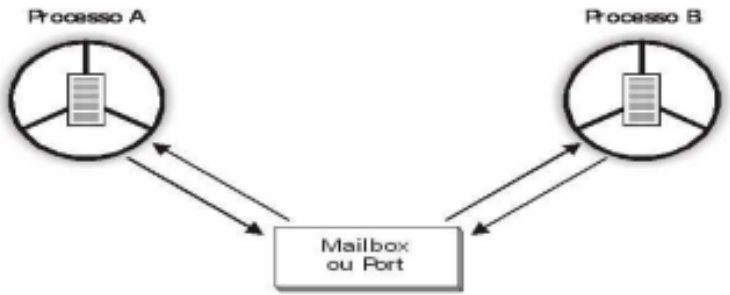
Comunicação entre Processos
Passagem de Mensagem

- O mecanismo de troca de mensagens exige que os processos envolvidos na comunicação tenham suas execuções sincronizadas.
- A troca de mensagens entre os processos pode ser implementada de duas maneiras distintas: comunicação direta e comunicação indireta.
- Comunicação direta entre dois processos exige que, ao enviar ou receber uma mensagem, o processo enderece explicitamente o nome do processo receptor ou transmissor.
- Uma característica deste tipo de comunicação é só permitir a troca de mensagem entre dois processos..



Comunicação entre Processos
Passagem de Mensagem

- A comunicação indireta entre processos utiliza uma área compartilhada, onde a mensagens podem ser colocadas pelo processo transmissor e retiradas pelo receptor.
- Esse tipo de buffer é conhecido como **mailbox** ou **port**, e suas características, como identificação e capacidade de armazenamento de mensagens, são definidas no momento
- de criação.
- Na comunicação indireta, vários processos podem estar associados a mailbox, e os parâmetros dos procedimentos SEND e RECEIVE passam a ser nomes de mailboxes e não mais nomes de processos.



Comunicação entre Processos
Outros mecanismos

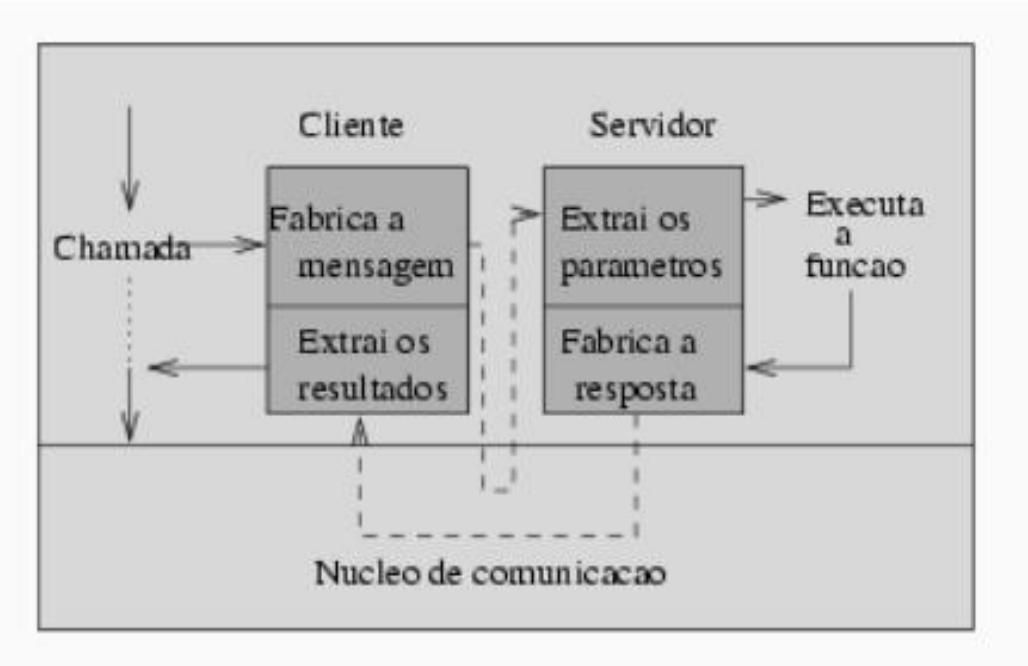
- RPC – *Remote Procedure Call*

– Permite a chamada de um procedimento em uma máquina por um programa sendo executado em outra máquina.

– Em um RPC, um programa em uma máquina P, (o cliente) chama um procedimento em uma máquina Q, (o servidor) enviando os parâmetros adequados. O cliente é suspenso e a execução do procedimento começa. Os resultados são enviados da máquina Q para a máquina P e o cliente é acordado.

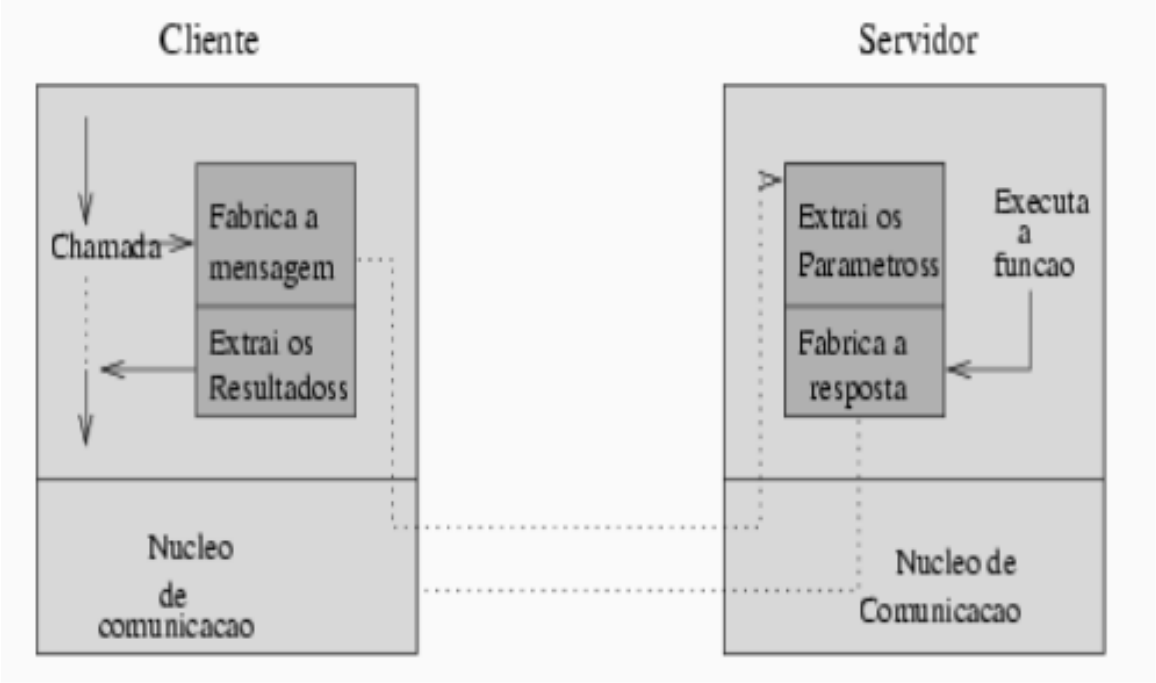
Comunicação entre Processos
Outros mecanismos

- RPC – *Remote Procedure Call*



Comunicação entre Processos
Outros mecanismos

- RPC – *Remote Procedure Call*



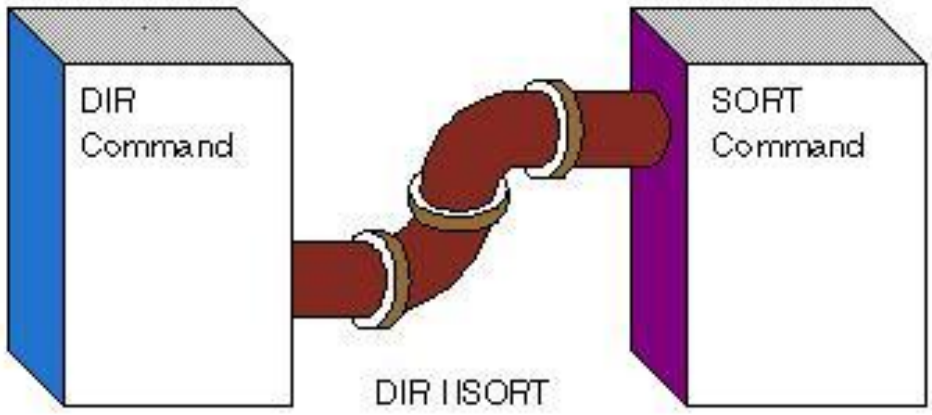
Comunicação entre Processos
Outros mecanismos

- *Pipe*:
  - Permite a criação de filas de processos;
  - **`ps -ef | grep alunos`**;
  - Saída de um processo é a entrada de outro;
  - Existe enquanto o processo existir;
- *Named pipe*:
  - Extensão de pipe;
  - Continua existindo mesmo depois que o processo terminar;
  - Criado com chamadas de sistemas;
- *Socket*:
  - Par endereço IP e porta utilizado para comunicação entre processos em máquinas diferentes;
  - Host X (192.168.1.1:1065) Server Y (192.168.1.2:80);

Comunicação entre Processos
Passagem de Mensagem

Comunicação entre Processos no UNIX: PIPES

Pipes: usados no sistema operacional UNIX para permitir a comunicação entre processos. Um pipe é um modo de conectar a saída de um processo com a entrada de outro processo, sem o uso de arquivos temporários. Um pipeline é uma conexão de dois ou mais programas ou processos através de pipes.



Departamento de Computação

REFERÊNCIAS

- MACHADO, F. B. & MAIA, L. P., Arquitetura de Sistemas Operacionais, 4 Edição, São Paulo, LTC, 2007.
- TANENBAUM, A. S. Sistemas Operacionais Modernos: 2ª edição, São Paulo, editora Prentice Hall, 2003.
- SILBERSCHATZ, A. Sistemas Operacionais – Conceitos: São Paulo, editora LTC, 2004.