SISTEMAS OPERACIONAIS

Comunicação e Sincronização de Processos

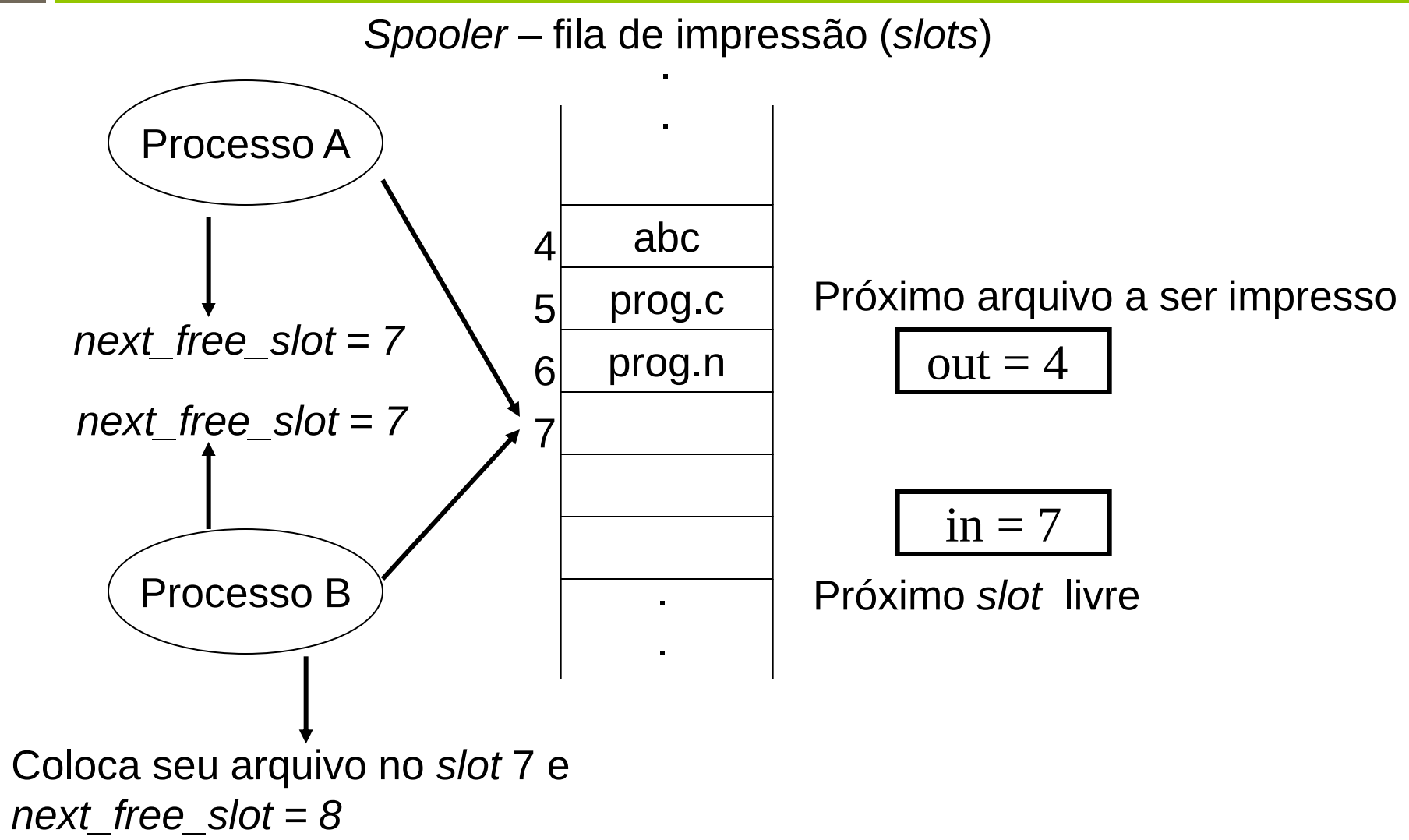
Comunicação de Processos

- 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;

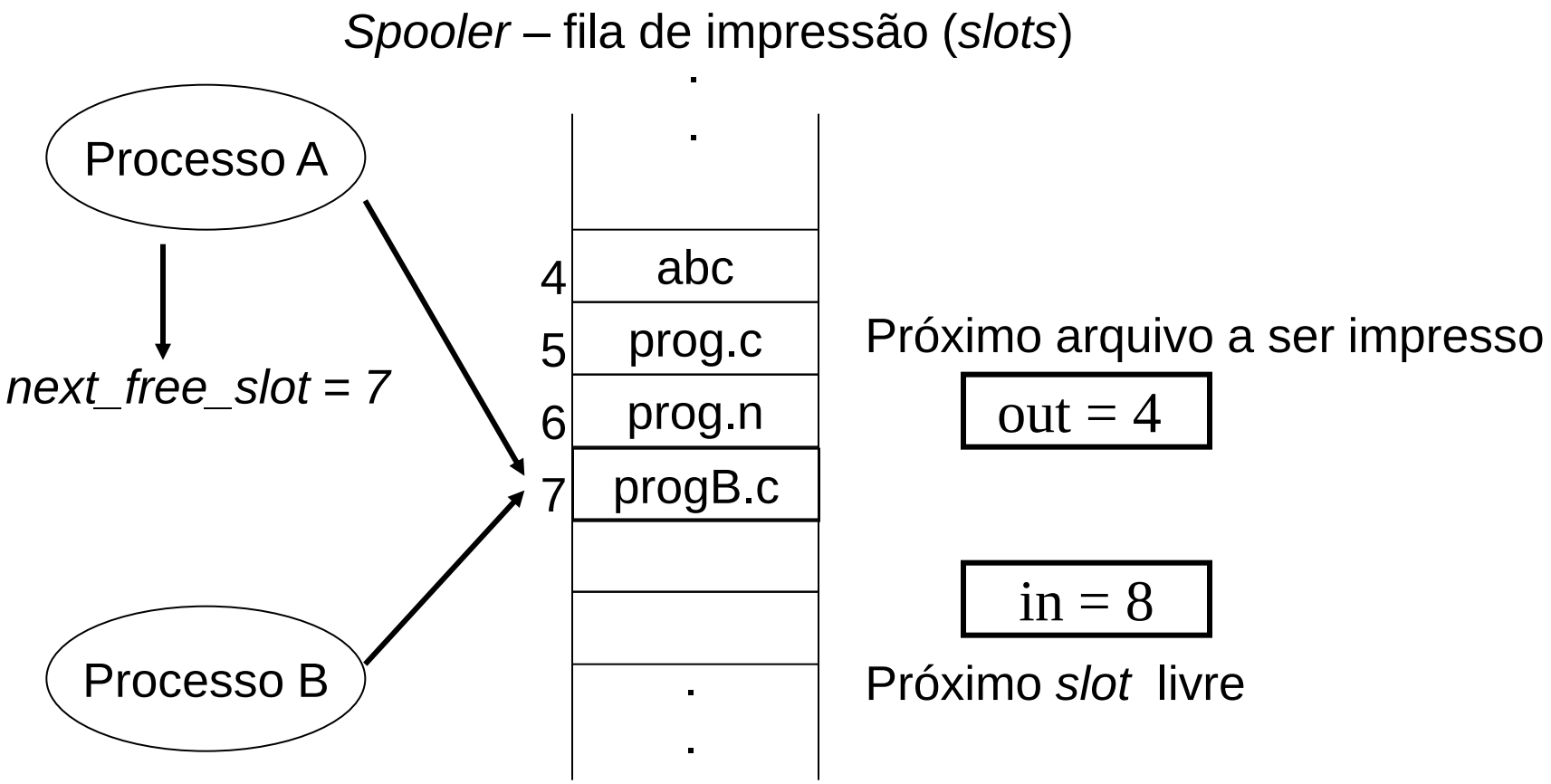
Comunicação de Processos – *Race Conditions*

- *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...

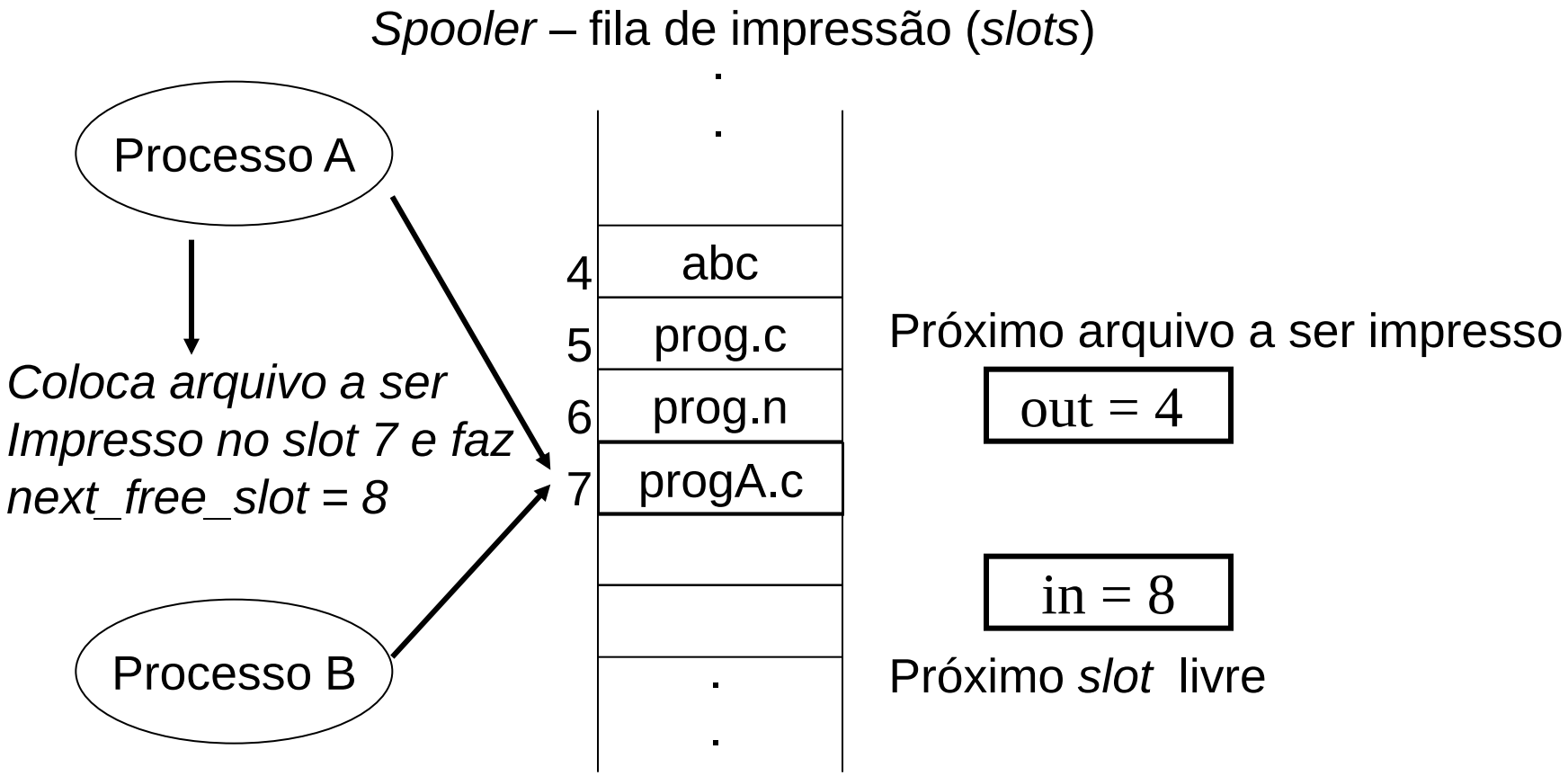
Comunicação de Processos - *Race Conditions*



Comunicação de Processos - *Race Conditions*



Comunicação de Processos - *Race Conditions*



Processo B nunca receberá sua impressão!!!!!

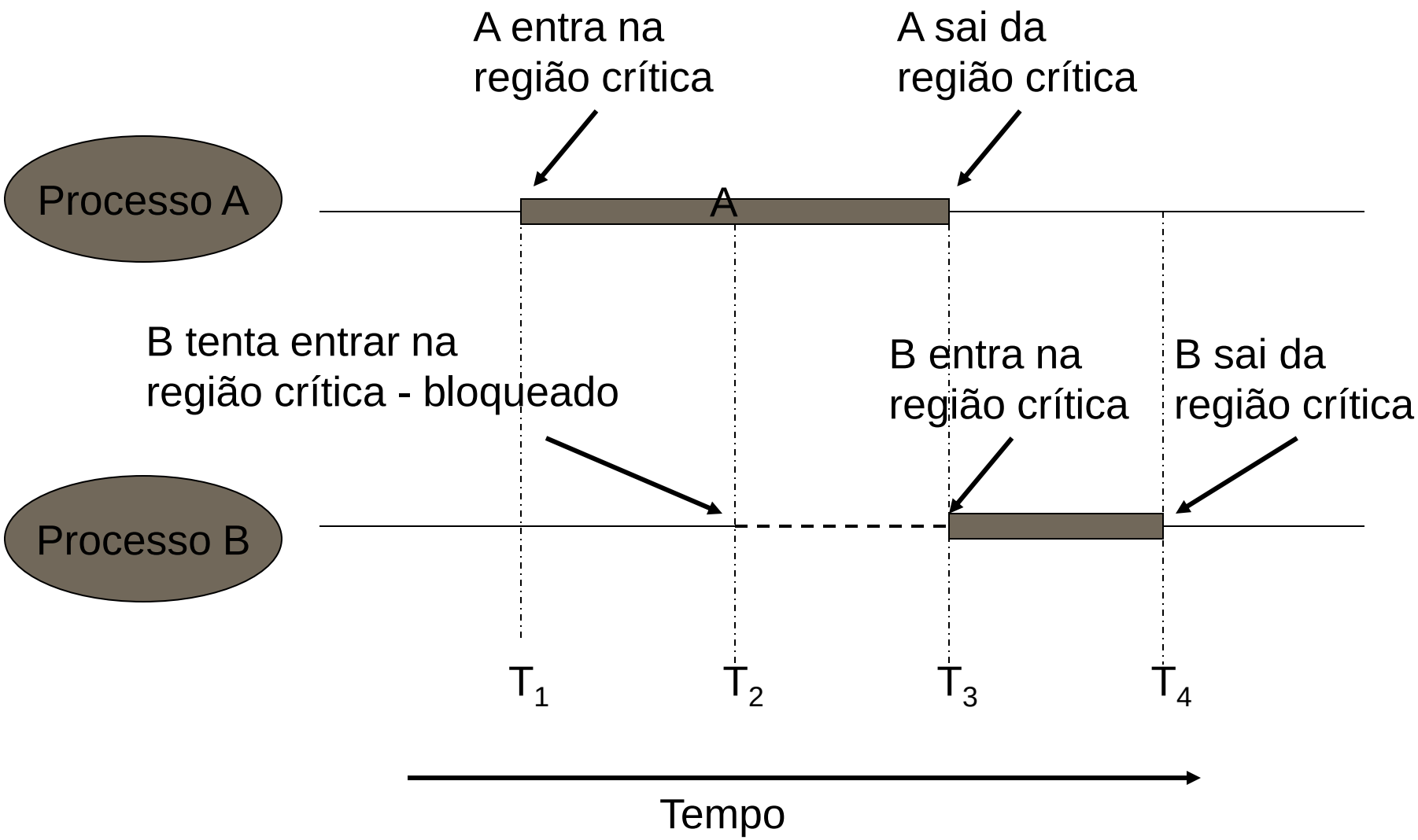
Comunicação de Processos – Regiões Críticas

- 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:
  - Dois processos não podem estar simultaneamente em regiões críticas;
  - Nenhuma restrição deve ser feita com relação à CPU;
  - Processos que não estão em regiões críticas não podem bloquear outros processos que desejam utilizar regiões críticas;
  - 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);
  - Problemas com 2 CPUs (interrupções de apenas uma são desabilitadas)
  - Desativar interrupções é interessante dentro do S.O.
    - e.g. atualizar lista de processos prontos

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:
  - Inicialmente o processo A inspeciona turn, descobre que é 0 e entra na sua região critica;
  - O processo B também descobre que ela é 0 e, portanto entra no laço fechado
    - Espera ativa ou busy wating (evitar, desperdício de CPU);
  - Quando processo A sai da região crítica configura turn como 1, permitindo que processo B entre na sua região crítica;
  - Suponha que o processo B termine sua região crítica rapidamente;
    - Ambos os processos estão na sua regiões não-críticas e turn = 0;
  - Processo A executa novamente, saindo da região crítica e configurando turn =1;
  - Repentinamente o processo A termina de trabalhar na sua região não crítica e volta ao início de seu laço
    - Processo A não terá permissão para entrar na sua região crítica, pois turn = 1.
    - Processo B está em região não crítica!

Comunicação de Processos – Exclusão Mútua

- Problema do *Strict Alternation:*
  - *Solução viola condição 3 estabelecida*
    - *O processo A está sendo bloqueado por um processo que não está em sua região crítica!*
    - *Se a região crítica fosse o spooler, o processo A não teria permissão para imprimir um arquivo porque o processo B está fazendo outra coisa!*

Comunicação de Processos – Exclusão Mútua

- Solução de Peterson
  - 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

```
#define FALSE 0
#define TRUE 1
#define N 2

int turn;
int interested[N];

void enter_region(int process)
{
  int other;

  other = 1 - process;
  interested[process] = TRUE;
  turn = process;
  while (turn ==  process) && interested[other] == TRUE) ;
}

void leave_region(int process)
{
  interested[process] = FALSE;
}
```

Comunicação de Processos – Exclusão Mútua

- Solução de Peterson
  - A solução consiste em fazer com que um processo, antes de entrar na região crítica, execute a rotina enter_region() com seu próprio número de processo. Ao terminar a utilização da região crítica o processo executa leave_region().
  - Quando 2 processos chamam enter_region() ao mesmo tempo ambos indicarão seu interesse colocando a respectiva entrada no vetor interested (TRUE).
  - Um dos processos deixa a vez (pois altera esta variável por último) e, portanto habilita o outro processo a continuar, aguardando até que o outro processo indique que não está mais interessado executando leave_region().
  - O exclusão mútua se dá pelo looping ‘infinito’, o que faz com que um dos processos não termine a chamada enter_region() até que o outro mude seu estado.
  - O problema dessa solução é que os próprios processos devem chamar enter_region() e leave_region(). Se um deles for mal implementado ou trapaceiro pode monopolizar a utilização da região crítica.

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);
    - A CPU que executa TSL bloqueia o barramento de memória, proibindo outras CPUs de acessar a memória até que ela tenha terminado.

Comunicação de Processos – Exclusão Mútua

- Instrução TSL:
  - ***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 não era 0, LOCK estava config.;
                                | portanto, entra no laço,
    RET                         | Retorna para quem fez a chamada;
                                | entra na região crítica

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:
    - Processos H (alta prioridade) e L (baixa prioridade)
    - L entra na RC
    - H começa a executar e quer entrar na RC
      - Como H é de alta prioridade L nunca entra no escalonamento (problema da inversão de prioridade)

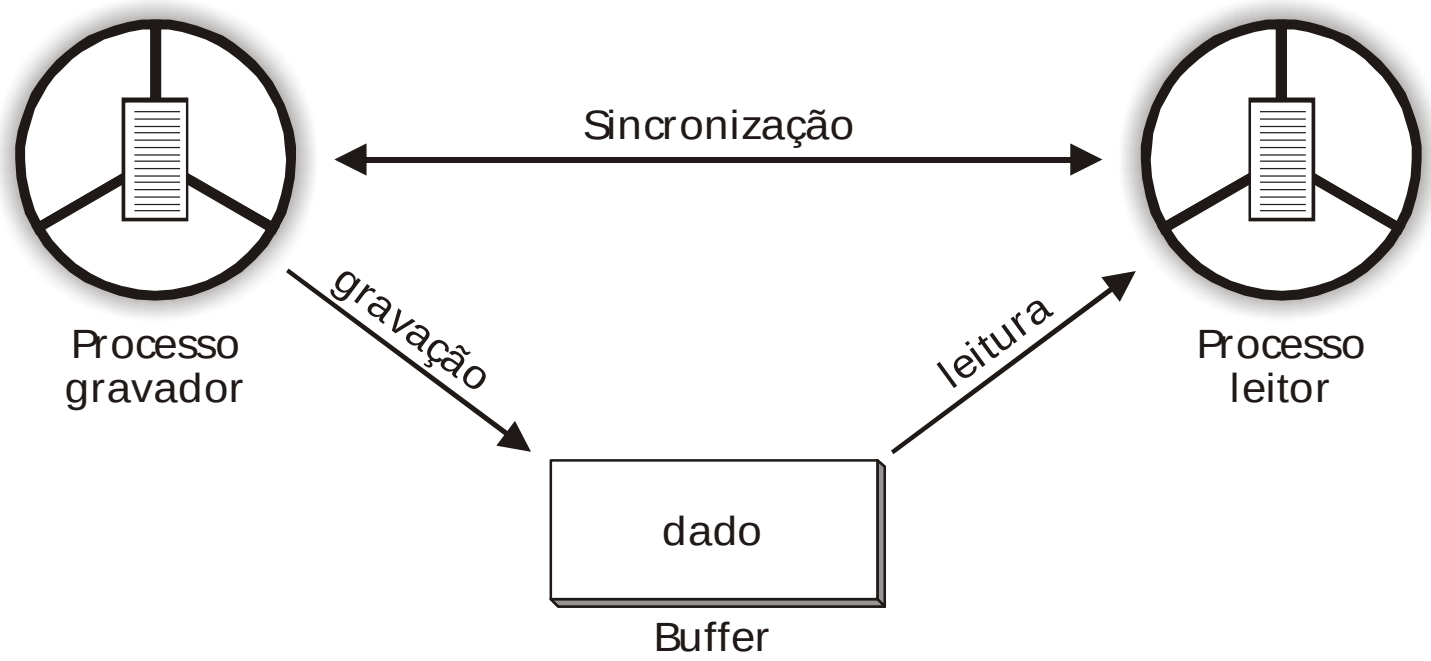
Comunicação de Processos – Primitivas *Sleep*/*Wakeup*

- 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 – Produtor Consumidor



Comunicação de Processos – Primitivas *Sleep*/*Wakeup*

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

Se `count` = valor máximo (buffer cheio)

Então processo produtor é colocado para dormir

Senão produtor coloca dados no *buffer* e incrementa `count`

Comunicação de Processos – Primitivas *Sleep*/*Wakeup*

- Consumidor:

Se `count` = 0 (buffer vazio)

Então processo vai “dormir”

Senão retira os dados do *buffer* e decrementa `count`

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*

```
# 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);
   }
}
```

Para aqui

Problema: se *wakeup* chega antes do consumidor dormir

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!!!!