Universidade Federal
de Mato Grosso
Campus Rondonópolis

# Sistemas Operacionais

## Aula 6 – Comunicação interprocessos (*Interprocess Comunication)*

Prof. Msc. Cleyton Slaviero

cslaviero@gmail.com

# Exemplo

- Comprando uma vaga em um avião
  - Operador OP1 (no Brasil) lê Cadeira1 vaga;
  - Operador OP2 (no Japão) lê Cadeira1 vaga;
  - Operador OP1 compra Cadeira1;
  - Operador OP2 compra Cadeira1;



Universidade Federal
de Mato Grosso
Campus Rondonópolis

# Exemplo

- Solução
  - Apenas um operador pode estar vendendo em um determinado momento;
    - Isso gera uma fila de clientes nos computadores;
    - Problema: ineficiência!

# Exemplo (2)

- Dois processos: A e B
  - A produz dados
  - B imprime dados
- Problemas
  - E se não há memória para mais dados?
  - E se a memória está vazia?



Universidade Federal
de Mato Grosso
Campus Rondonópolis

# 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: sequência adequada

Ex: **a = c+c; x=a+y;**



Universidade Federal de Mato Grosso
Campus Rondonópolis

# 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

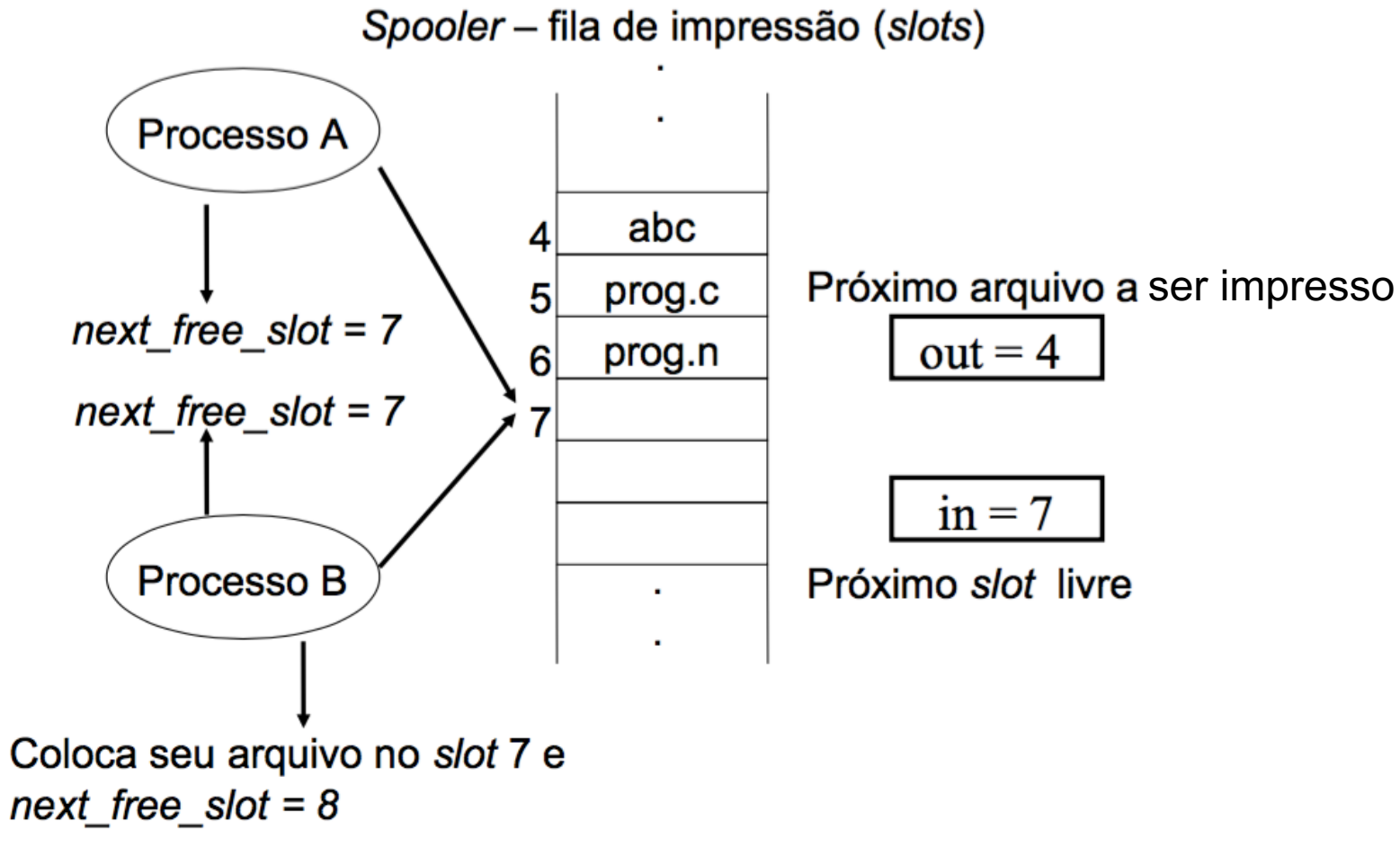
# Comunicação de processos

- Em alguns Sistemas Operacionais
  - Comunicação por meio de uma área comum aos processos/threads
    - Espaço na memória
    - Arquivo compartilhado
- Para processos, espaço comum pode estar no kernel

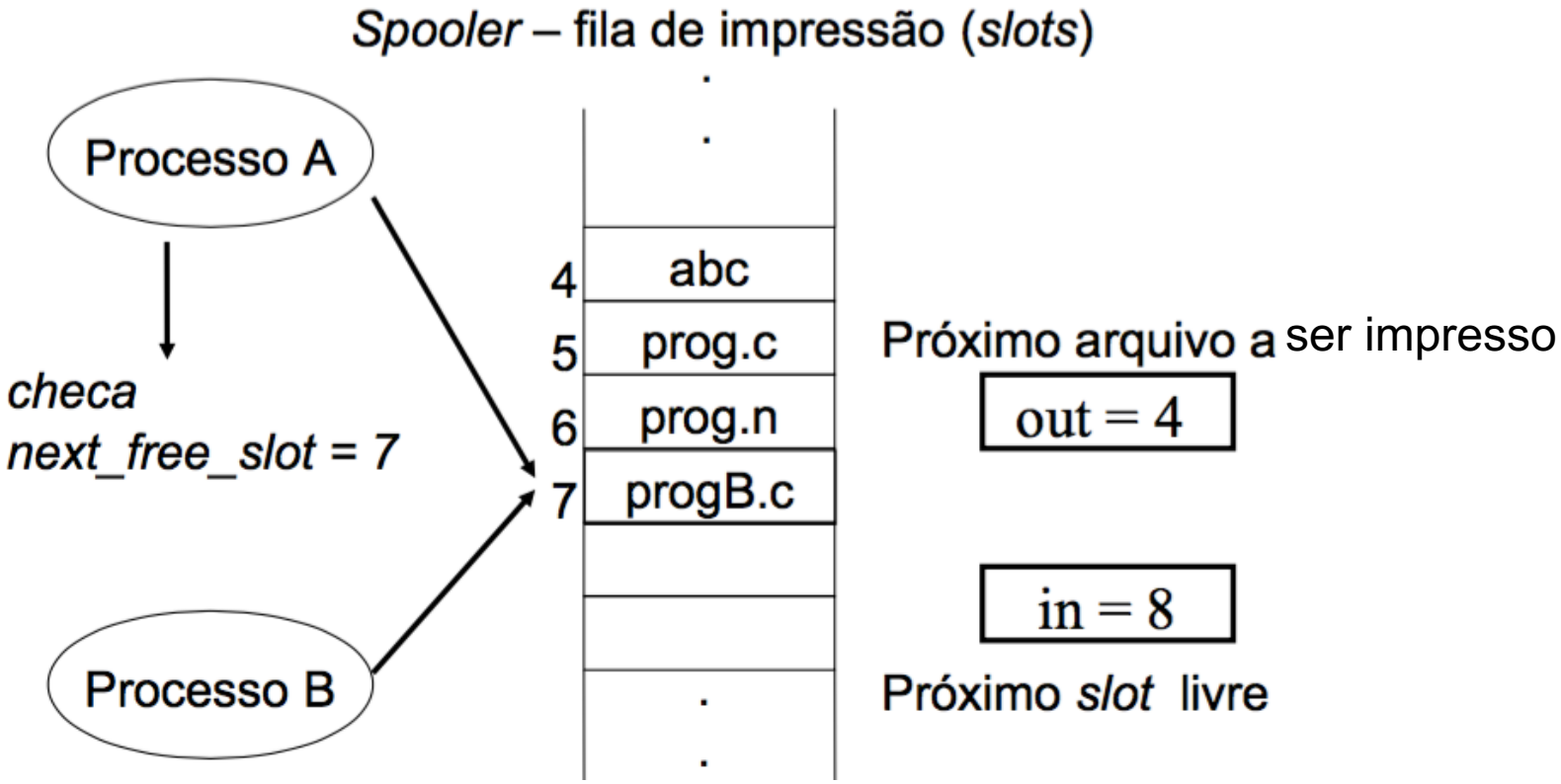
# Comunicação de processos

- Um exemplo: spooler de impressão
- Quando um processo deseja imprimir um arquivo, ele coloca o nome do arquivo em uma lista de impressão (diretório de spool).
- Um processo chamado “daemon de impressão”, 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.

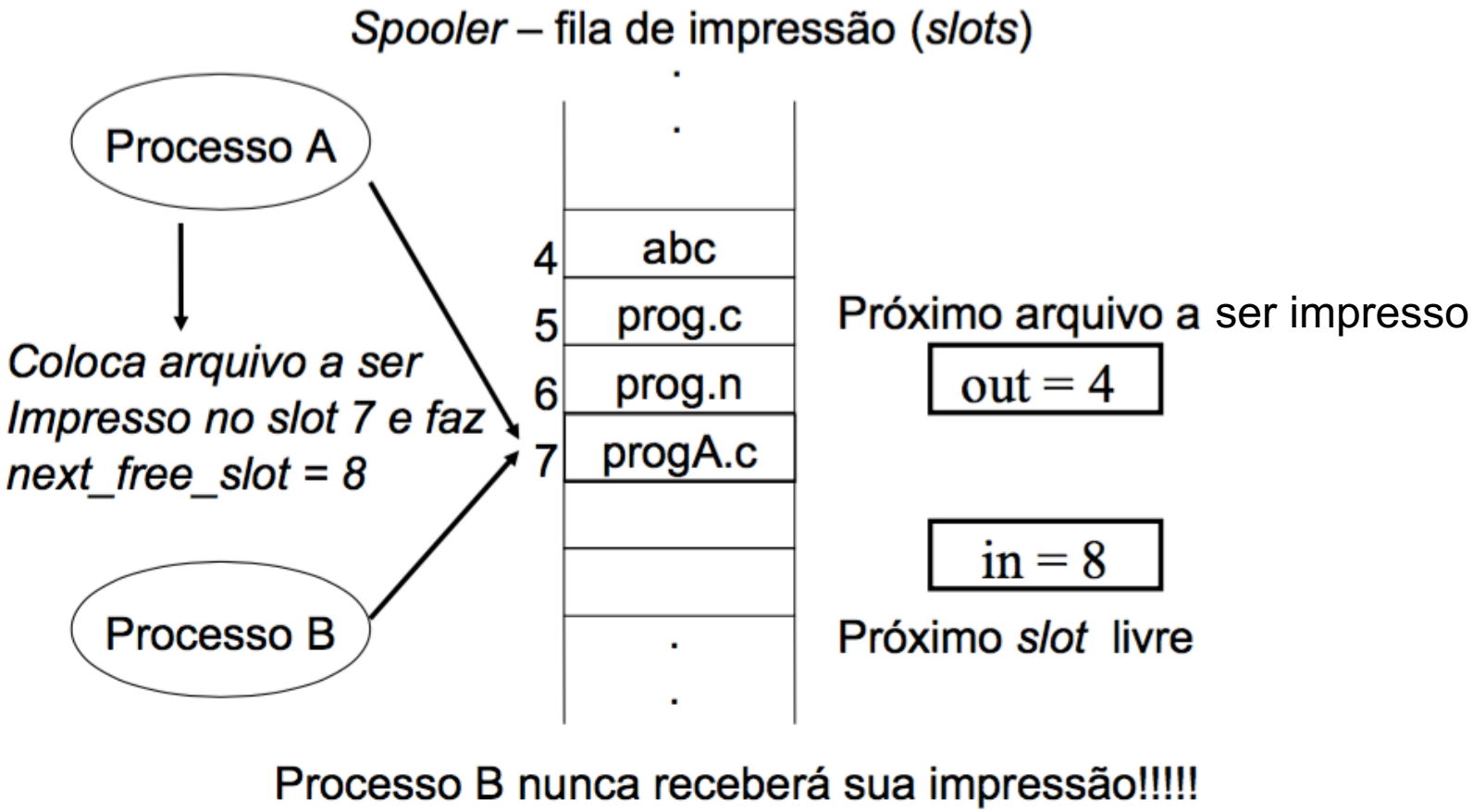
# Condições de corrida



# Condições de corrida



# Condições de corrida




Processo B nunca receberá sua impressão!!!!!

# Condições de corrida

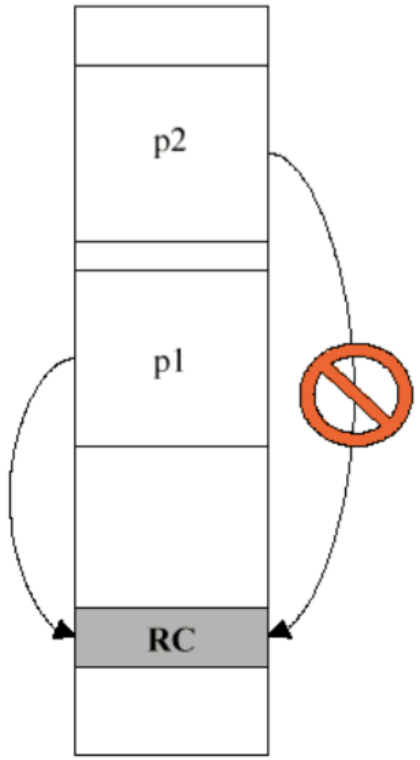
- *Condições de corrida (race conditions)*: processos acessam recursos compartilhados concorrentemente;
- – Recursos: memória, arquivos, impressoras, discos, variáveis;

# Condições de corrida

- 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 ou regiões críticas (R.C.)**.

# Condições de corrida

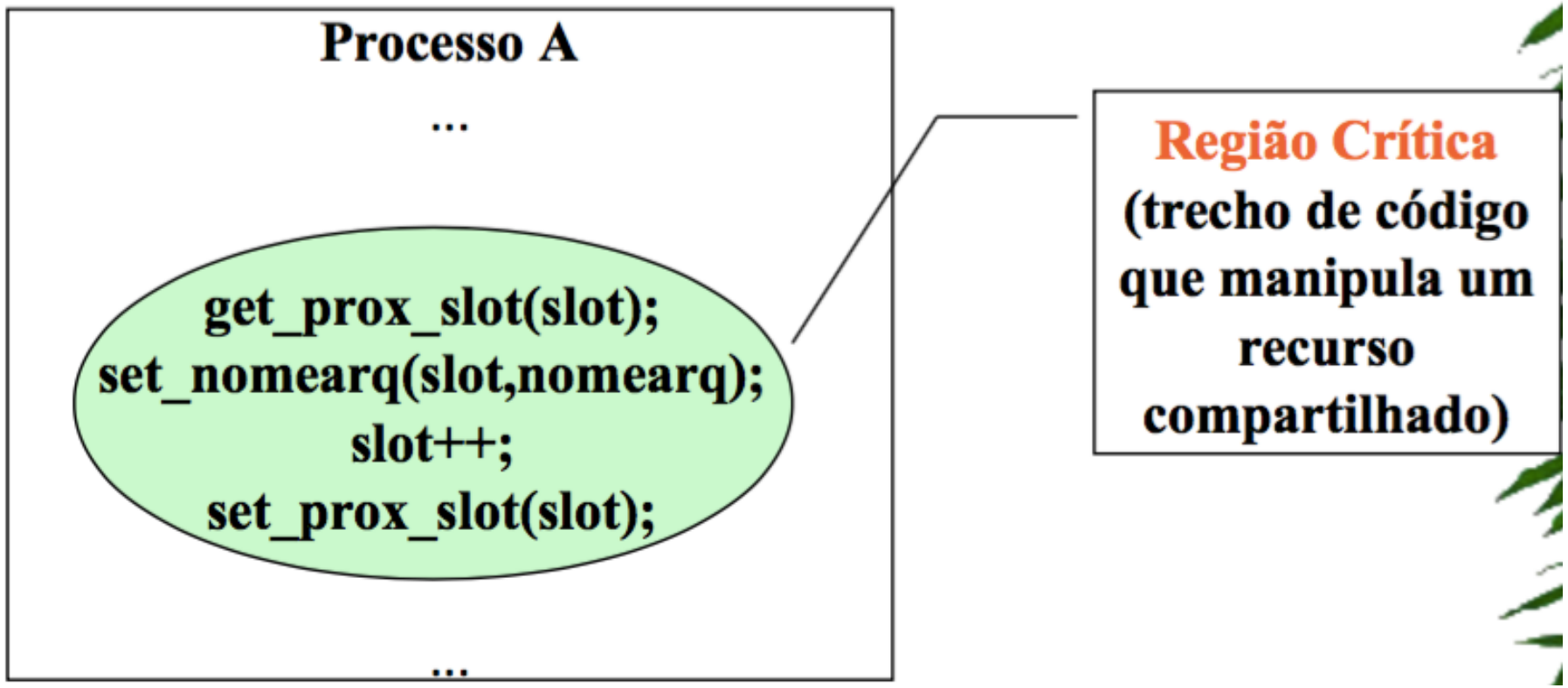
- Enquanto P1 acessa uma região crítica, P2 não pode acessar!






Universidade Federal de Mato Grosso
Campus Rondonópolis

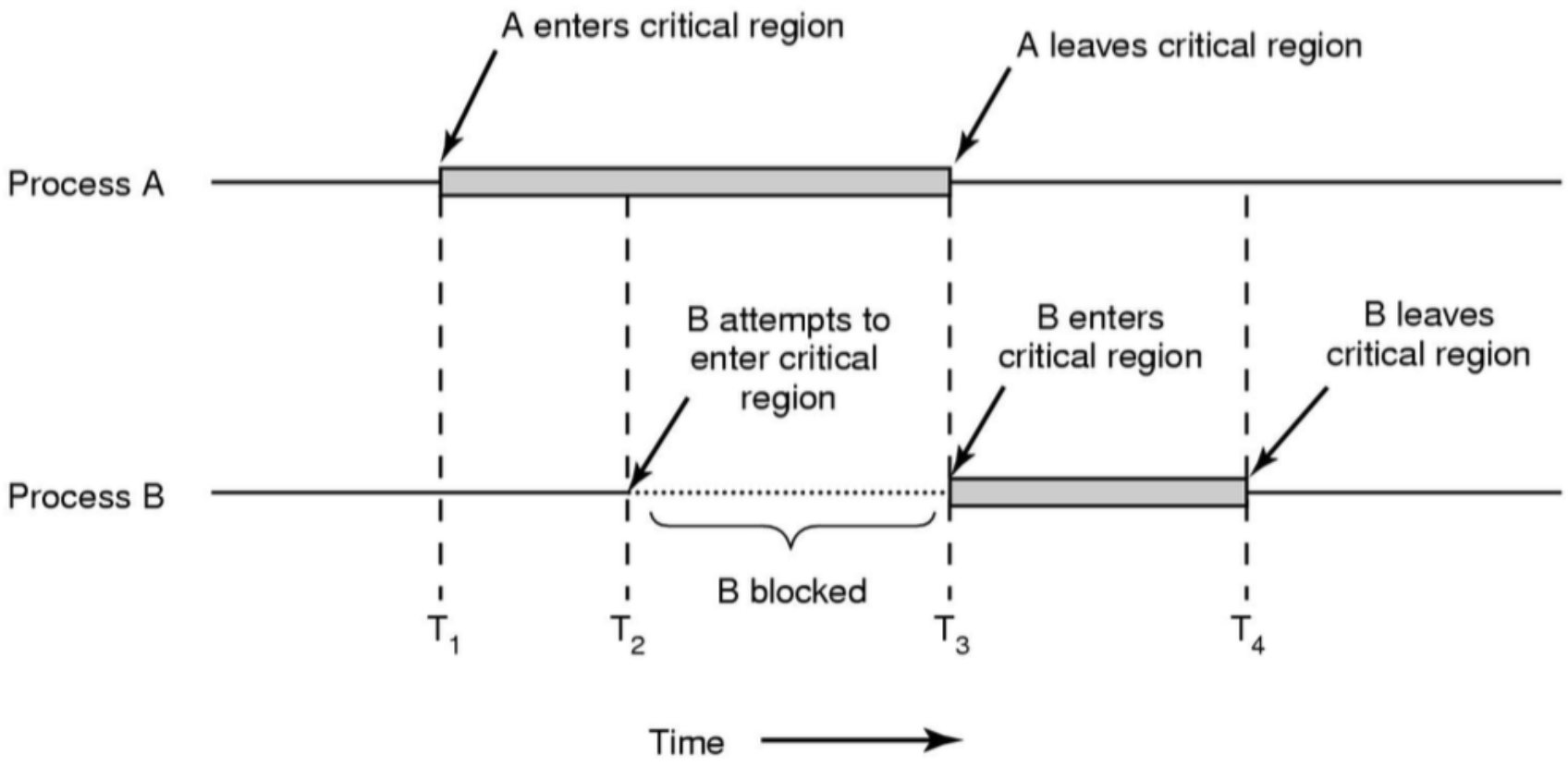
# Regiões críticas



# 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

# Região crítica



# Regiões críticas e exclusão mútua

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

# Comunicação de 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;

# 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

- Condições necessárias para evitar condição de corrida
  1. Dois processos nunca podem estar simultaneamente em suas regiões críticas
  2. Nada pode ser afirmado sobre a velocidade ou sobre o número de CPU's
  3. Nenhum processo executando fora de sua região crítica pode bloquear outros processos
  4. Nenhum processo deve esperar eternamente para entrar em sua região crítica

# Soluções para exclusão mútua

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

# Comunicações 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)*
  - Chaveamento obrigatório  (*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 CPU 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*:
  - Solução de software
  - 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==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 (chaveamento obrigatório)
  - Fragmentos de programa controlam o acesso as regiões críticas
  - Variável **`turn`**, inicialmente em 0, estabelece qual processo pode entrar na região crítica
  - Outro processo, caso encontre turn=0, segue tentando até conseguir
    - Espera ociosa
    - Variável turn = spin lock (trava do 'giro')

# Comunicação de processos – exclusão mútua

- Strict alternation (chaveamento obrigatório)
  - Fragmentos de programa controlam o acesso as 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:*
  - Suponha que o Processo **B** é mais rápido e sai da região crítica;
  - Ambos os processos estão fora da região crítica e turn com valor 0;
  - 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;
  - Ao sair da região crítica, o Processo **A** atribui o valor 1 a variável turn e entra na sua região não 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:*
  - Novamente ambos os processos estão na região não crítica e a variável turn está com valor 1;
  - Quando o processo A tenta novamente entrar na região crítica, não consegue, pois turn ainda está com valor 1;

```
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

**O processo A fica bloqueado pelo processo B que NÃO está na sua região crítica, violando a condição 3;**

# Comunicação de processos – exclusão mútua

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

# Comunicação de processos – exclusão mútua

- Solução de Peterson
  - enter_region – esoera até que seja seguro para processo entrar
  - leave_region – indica o termino do processo e permite que outro processo entre se assim desejar
  - Se dois processos entram praticamente simultaneamente, o que armazenou o seu número na variável turn é o que conta – o primeiro é sobreposto e é perdido

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* number of processes */

int turn;                               /* whose turn is it? */
int interested[N];                      /* all values initially 0 (FALSE) */

void enter_region(int process);         /* process is 0 or 1 */
{
     int other;                         /* number of the other process */

     other = 1 – process;               /* the opposite of process */
     interested[process] = TRUE;        /* show that you are interested */
     turn = process;                    /* set flag */
     while (turn == process && interested[other] == TRUE) /* null statement */ ;
}

void leave_region(int process)          /* process: who is leaving */
{
     interested[process] = FALSE;       /* indicate departure from critical region */
}
```

# 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
```

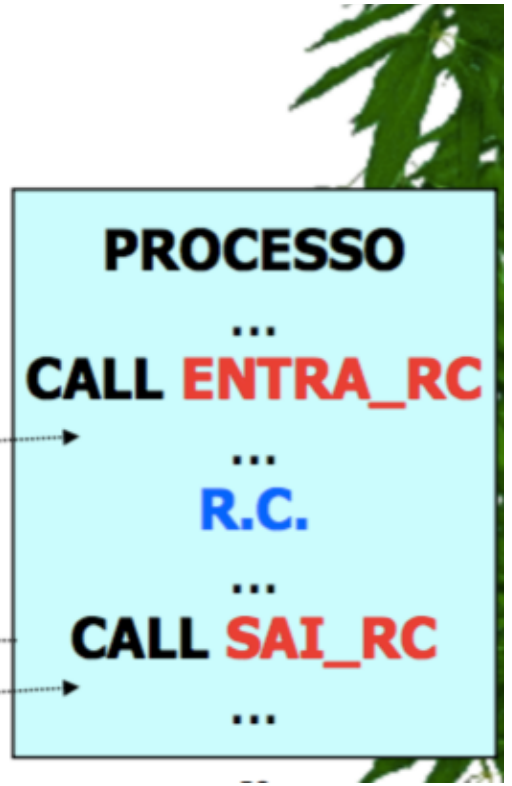
# Comunicação de processos – exclusão mútua

- **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 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 CPUs de acessar a memória até ela terminar.

# Instrução TSL - 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
```



# 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 CPU;
  - Pode provocar “bloqueio perpétuo”(*deadlock*)em sistemas com prioridades.

# Dormir e acordar

- Soluções  TSL (ou XCGH em proc. Intel x86) dependem de espera ociosa
  - Gasta tempo de CPU
  - Inversão de prioridade
    - Imagine dois processos, H e L
    - H sempre é executando quando no estado pronto
    - L entra em região crítica, H fica pronto pra executar
    - H tenta executar, mas região crítica → espera ociosa
    - H nunca irá executar, pois L nunca será escalonado (H tem prioridade)
- Solução: bloquear
  - Wake e sleep
  -

# Dormir e acordar

- Exemplo: Produtor e consumidor (ou buffer limitado – *bounded buffer*
  - Um produtor insere itens num buffer, outro remove
  - Problema para o produtor
    - Buffer cheio
  - Problema para o consumidor
    - Buffer vazio

# Dormir e acordar

- Uma solução
  - Coloca-se o produtor/consumidor para dormir
  - O outro o acorda quando a condição for satisfeita

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

void producer(void)
{
     int item;

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


void consumer(void)
{
     int item;

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

# Dormir e acordar

- Desvantagens:
  - Mesmos problemas de antes
  - Condição de corrida: Variável count tem acesso irrestrito
    - Escalonador pode trocar de processo na hora em que count é verificado pelo consumidor
    - Sinal de acordar é perdido
  - E se usarmos um bit de espera – guarda sinais de acordar
    - Ótimo pra dois processos, péssimo para n

# Semáforos

- E se usarmos uma variável para contar o número de sinais salvos para uso futuro
  - Semáforo
- Proposto por Dijkstra (1965)
  - Duas operações (*up* e *down*)
    - Generalizações do sleep e wake, respectivamente
  - **Down** verifica se o valor é maior que 0; caso positivo, decrementa; se for negativo, põe o processo para dormir
  - **Ação atômica**
    - Fundamental para que semáforos funcionem
  - **Up** incrementa o valor de um semáforo
    - Se um processo estiver dormindo, sistema escolhe (e.g. aleatório) um processo para acordar

# Semáforos

- Semáforos resolvem o problema de perda do sinal de acordar
  - Precisam estar implementados de maneira indivisível
- Up em down geralmente são programados como chamadas de sistema
  - Se várias CPUs estiverem sendo usadas, cada semáforo deverá ser protegido por uma variável de trava, com o uso da instrução TSL

# Semáforos

- Só para lembrar
  - TSL/XCGH é diferente de espera ocupada do produtor/consumidor
  - Operação de semáforo dura apenas alguns microssegundos
  - Produtor/consumidor pode demorar um tempo arbitrariamente longo

# Semáforos

- Uma solução para o produtor/consumidor
  - Três semáforos
    - Full – número de lugares preenchidos
    - Empty – conta número de lugares vazios
    - Mutex – garante a exclusão mútua
      - Semáforo binário

```
#define N 100                                   /* number of slots in the buffer */
typedef int semaphore;                          /* semaphores are a special kind of int */
semaphore mutex = 1;                            /* controls access to critical region */
semaphore empty = N;                            /* counts empty buffer slots */
semaphore full = 0;                             /* counts full buffer slots */

void producer(void)
{
     int item;

     while (TRUE) {                             /* TRUE is the constant 1 */
          item = produce_item( );                /* generate something to put in buffer */
          down(&empty);                         /* decrement empty count */
          down(&mutex);                         /* enter critical region */
          insert_item(item);                    /* put new item in buffer */
          up(&mutex);                           /* leave critical region */
          up(&full);                            /* increment count of full slots */
     }
}


void consumer(void)
{
     int item;

     while (TRUE) {                             /* infinite loop */
          down(&full);                          /* decrement full count */
          down(&mutex);                         /* enter critical region */
          item = remove_item( );                /* take item from buffer */
          up(&mutex);                           /* leave critical region */
          up(&empty);                           /* increment count of empty slots */
          consume_item(item);                   /* do something with the item */
     }
}
```