Sistemas Operacionais I

Profa. Kalinka Regina Lucas Jaquie Castelo Branco
kalinka@icmc.usp.br

Universidade de São Paulo

Outubro de 2020

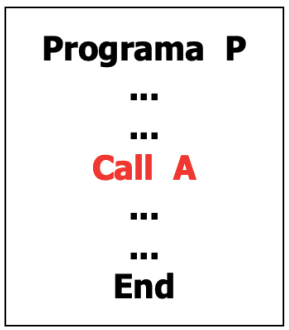
Correções com *Threads* Simultâneos

- Não determinismo:
  - O escalonador pode executar *threads* em qualquer ordem
  - O escalonador pode alternar tópicos a qualquer momento
  - Isso pode tornar o teste muito difícil
- Tópicos Independentes
  - Nenhum estado compartilhado com outros tópicos
  - Condições determinísticas e reproduzíveis
- Tópicos de cooperação
  - Estado compartilhado entre vários tópicos
- **Meta: Correção por Projeto**

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: sequência adequada.

Funcionamento de Sub-rotinas comuns



Comunicação de Processos

- Quais são os possíveis valores de x logo após a finalização de todas as *threads*?
- Inicialmente x==0 e y ==0

Thread A – x=1; Thread B – y=2;

- **Deve obrigatoriamente ser 1**. A *thread* B não interfere na *thread* A.

Comunicação de Processos

- Quais são os possíveis valores de x logo após a finalização de todas as *threads*?
- Inicialmente x==0 e y ==0

Thread A – x= y+1; Thread B – y=2;y = y*2;

- Qual o valor???.

Comunicação de Processos

- Quais são os possíveis valores de x logo após a finalização de todas as *threads*?
- Inicialmente x==0 e y ==0

Thread A – x= y+1; Thread B – y=2;y = y*2;

- 1 ou 3 ou 5 (não-determinismo).
- *Condição de Corrida/Condição de Disputa/Race Condition: Thread A corre contra Thread B*

Comunicação de Processos

Mecanismos simples de comunicação e sincronização de processos

- Em um sistema de multiprocessamento ou multiprogramação, os processos geralmente precisam se comunicar com outros processos.
- Comunicação entre processos é mais eficiente se for estruturada e não utilizar interrupções.

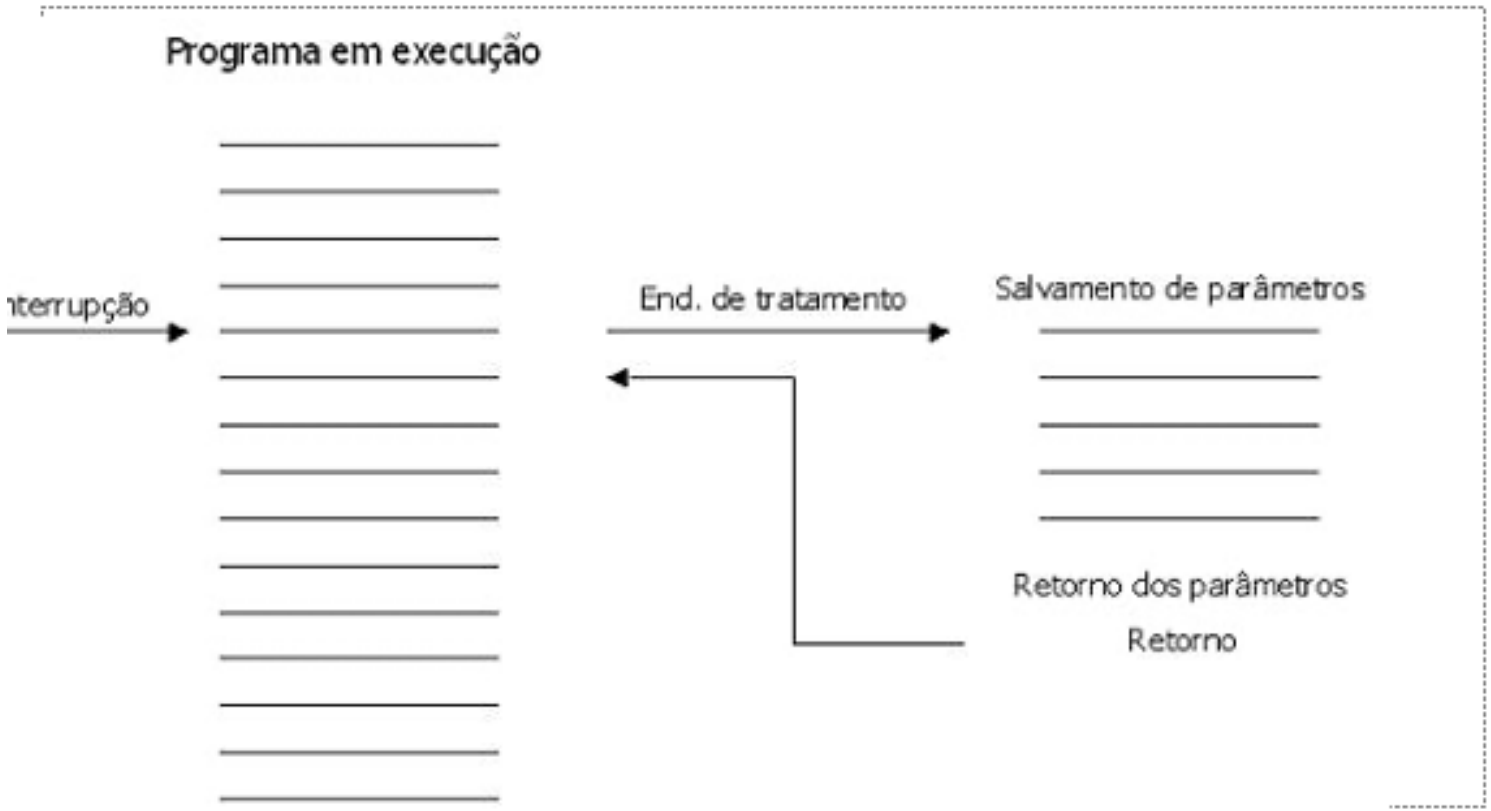
Veremos alguns mecanismos e problemas de comunicação inter-processos.

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 da interrupção o controle retorna ao programa interrompido exatamente no mesmo estado em que estava quando ocorreu a interrupção.

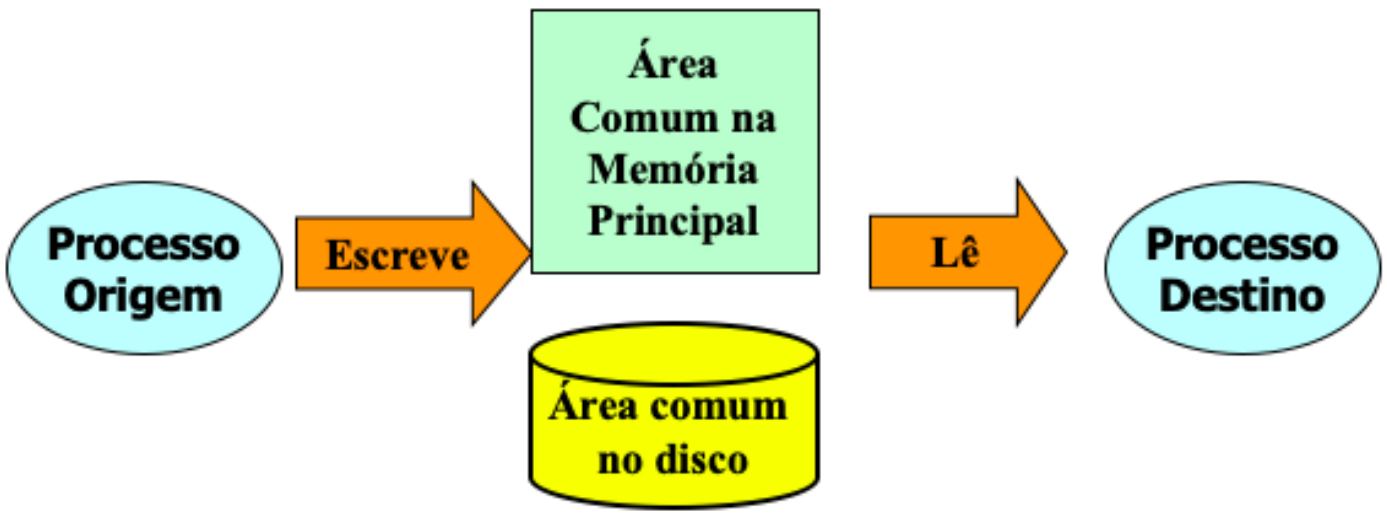
Comunicação de Processos



Comunicação de Processos

Condição de Corrida

- Em alguns sistemas operacionais: os processos se comunicam por meio de alguma área de armazenamento comum. Esta área pode estar na memória principal ou pode ser um arquivo compartilhado.



Comunicação de Processos

Definição

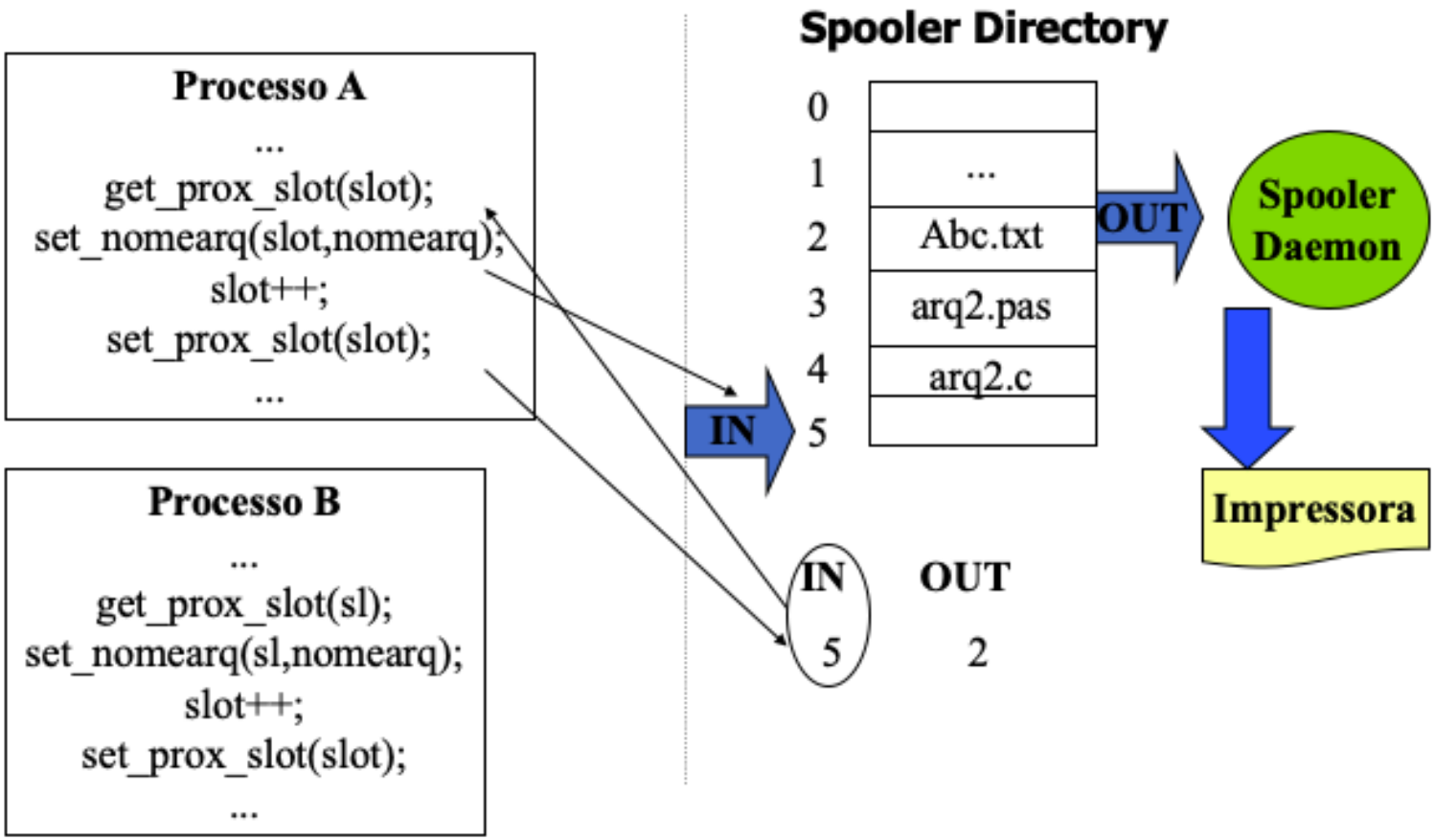
- Condição de Corrida: situações onde dois ou mais processos estão lendo ou escrevendo algum dado compartilhado e o resultado depende de quem processa no momento propício.
- Depurar programas que contém condições de corrida não é fácil, pois não é possível prever quando o processo será suspenso.

Comunicação de Processos

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.

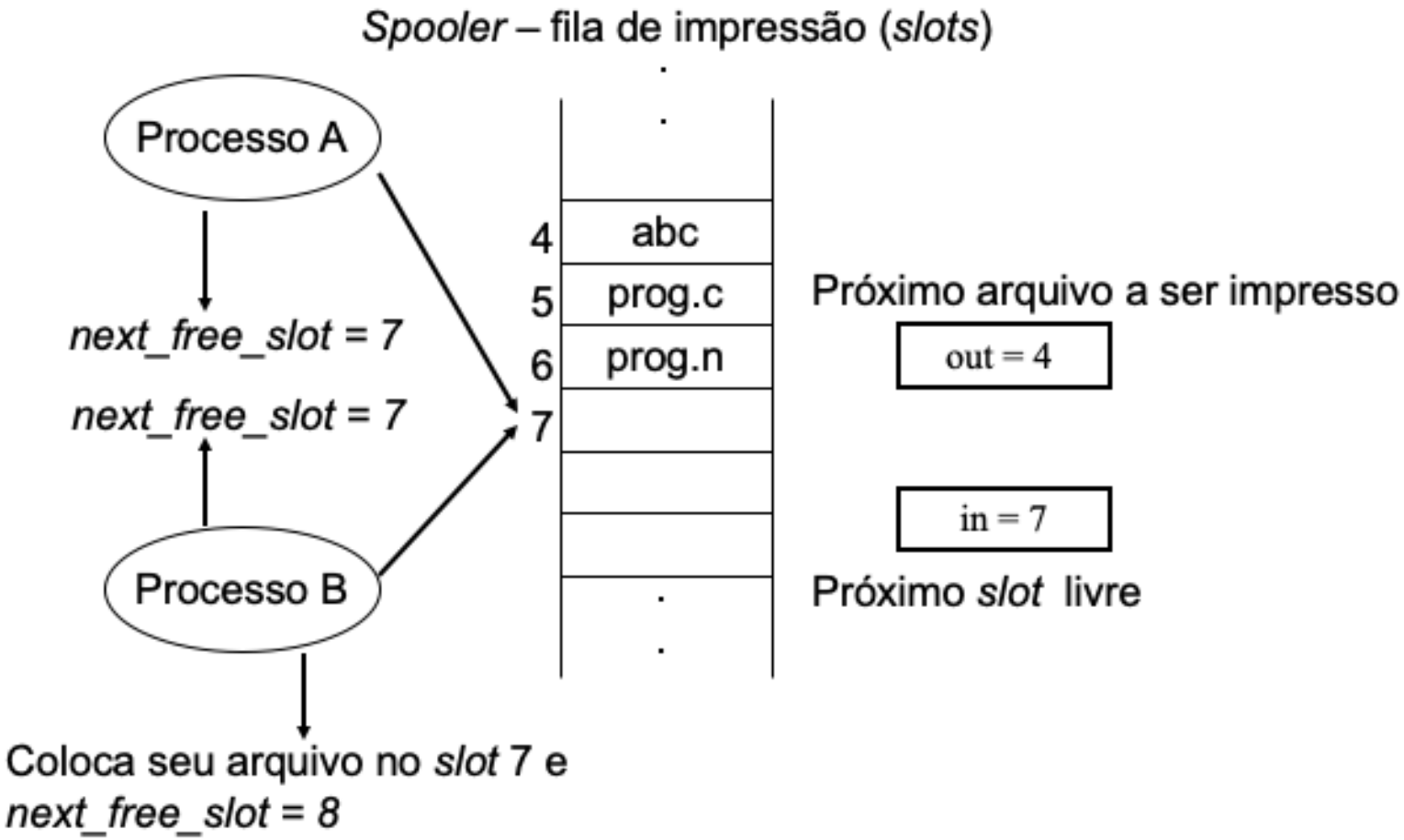
Comunicação de Processos



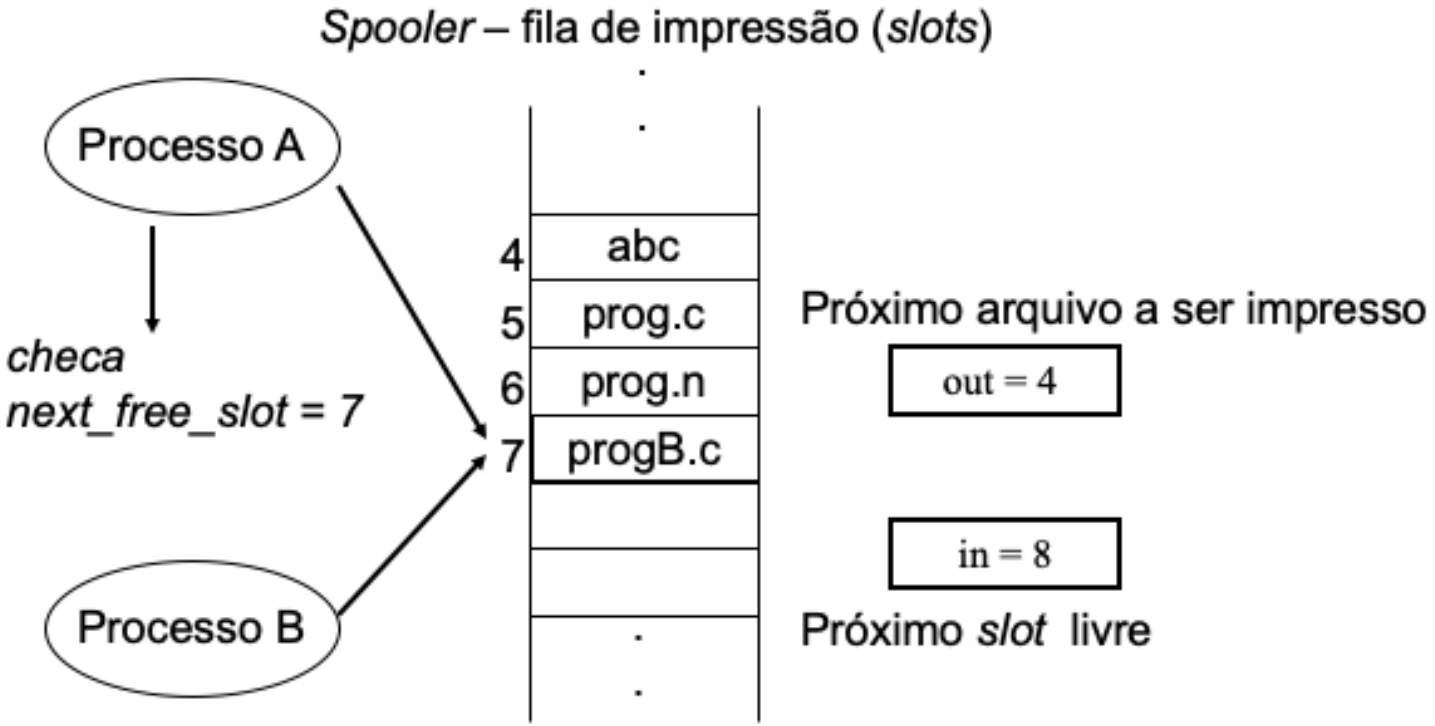
Comunicação de Processos

- *Race Condition*: 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.

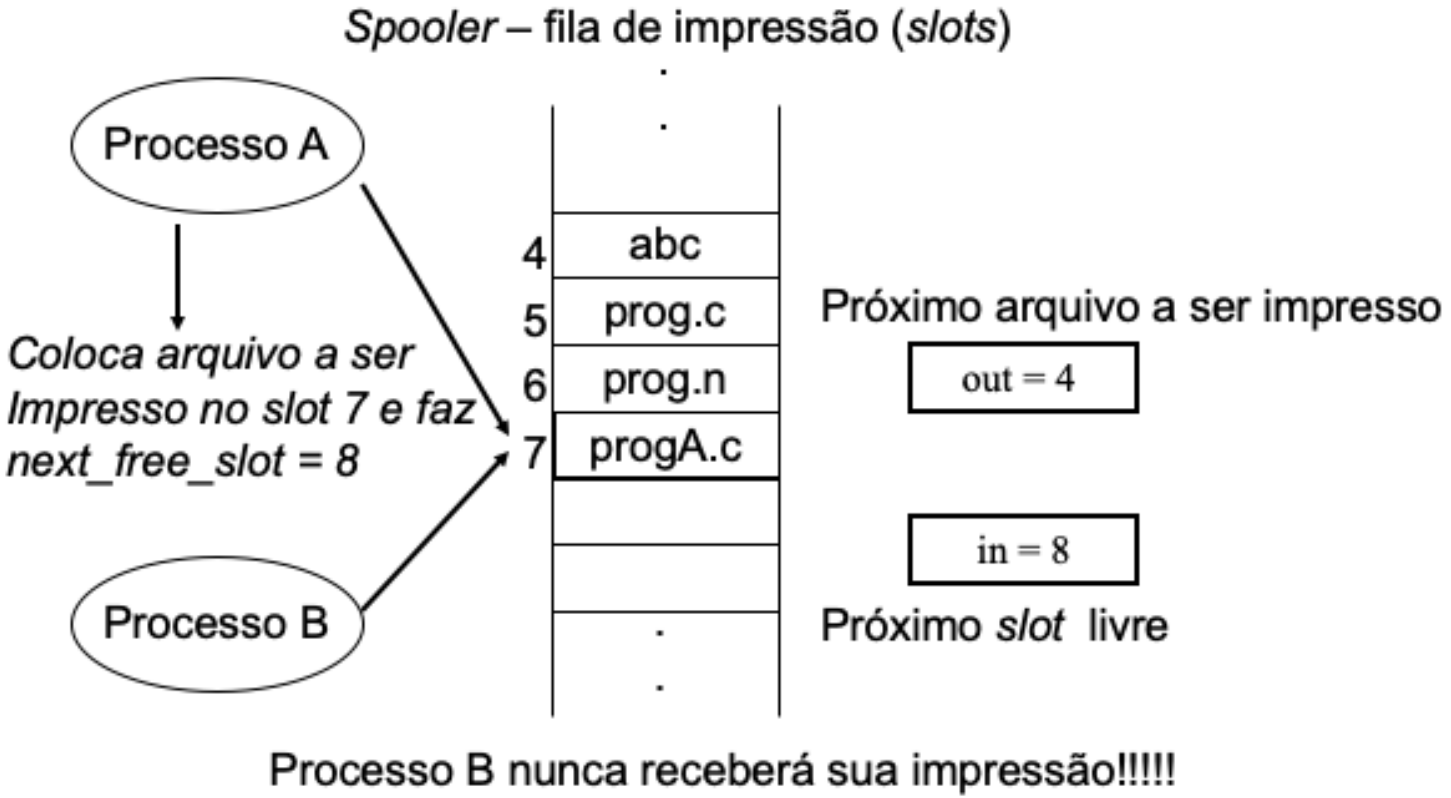
Comunicação de Processos



Comunicação de Processos



Comunicação de Processos

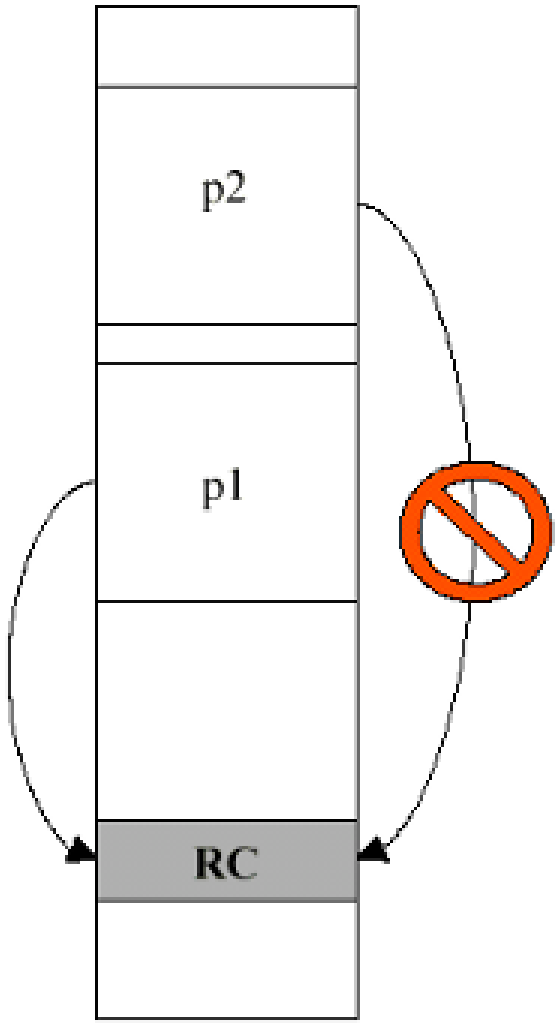


Comunicação de Processos

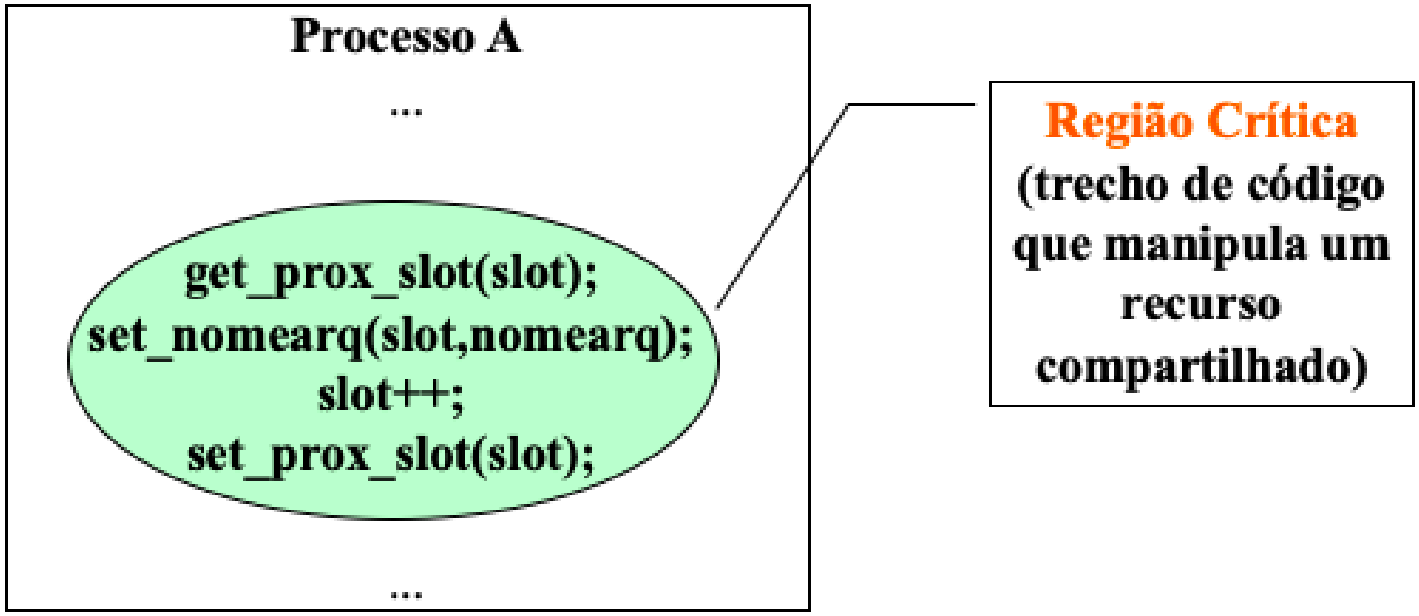
Região Crítica

- 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



Comunicação de Processos



Comunicação de Processos

Região Crítica

- Seção do programa onde são efetuados acessos (para leitura e escrita) a recursos compartilhados por dois ou mais processos
- É necessário assegurar que dois ou mais processos não se encontrem simultaneamente na região crítica.

Comunicação de Processos

- 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????

Sim?? Não??? Talvez???

Comunicação de Processos

Venda de cadeiras de um avião....

- 1. Operador OP1(Brasil) lê Cadeira1 vaga;
- 2. Operador OP2(Japão) lê Cadeira1 vaga;
- 3. Operador OP1 compra Cadeira1;
- 4. Operador OP2 compra Cadeira1;

Comunicação de Processos

Solução simples para exclusão mútua...

- Caso da 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

Como solucionar problemas de *Race Condition*??

- 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.

Como fazer isso????

Comunicação de Processos

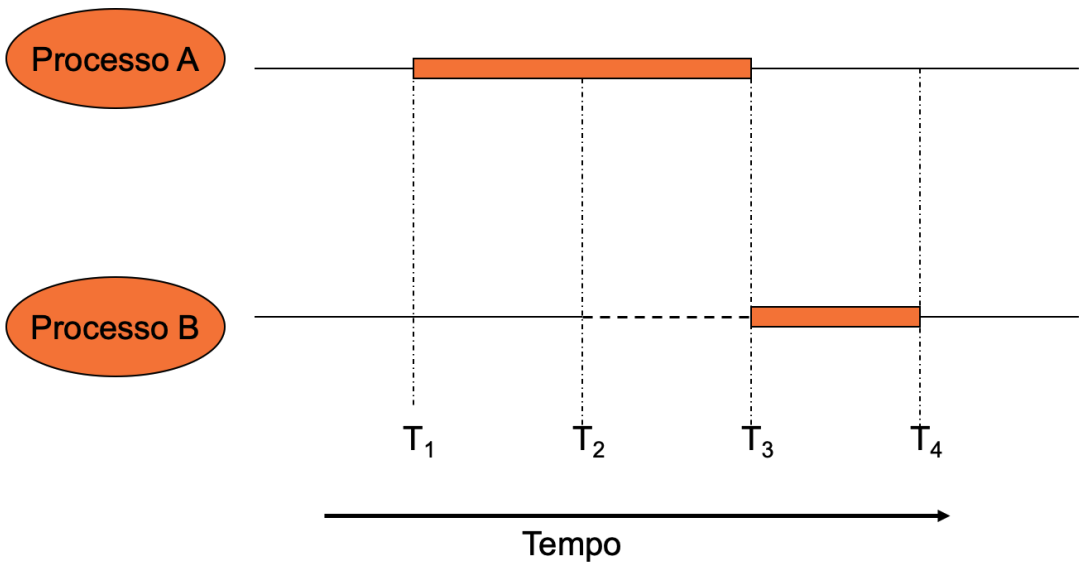
- Assegura-se a exclusão mútua recorrendo aos mecanismos de sincronização fornecidos pelo S.O.
- Estas afirmações são válidas também para as *threads* (é ainda mais crítico, pois todas as *threads* dentro do mesmo processo compartilham os mesmos recursos).

Comunicação de Processos

Regras para programação concorrente (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 de Processos



Comunicação de Processos

Soluções para Exclusão Mútua

- Espera Ocupada.
- Primitivas *Sleep/Wakeup*.
- Semáforos.
- Monitores.
- Passagem de Mensagem.

Comunicação de Processos

Espera Ocupada

- 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

Espera Ocupada - **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 a condição 2)**.
- Não é uma solução segura, pois um processo pode não habilitar novamente suas interrupções e não ser finalizado **(viola a condição 4)**.

Comunicação de Processos

Espera Ocupada - **Desabilitar 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-las.
- Desvantagens
  - Processo pode esquecer de reabilitar as interrupções;
  - Em sistemas com várias CPUs, desabilitar interrupções em uma CPU não evita que as outras acessam 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 esse método de exclusão mútua.

Comunicação de Processos

Espera Ocupada - Variável *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

Espera Ocupada - Variável *lock* - Problema

- Suponha que um processo A leia a variável *lock* com valor 0.
- Antes que o processo A possa 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 a condição 1)**

Espera Ocupada - Variável *lock* - Problema

lock==0;

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

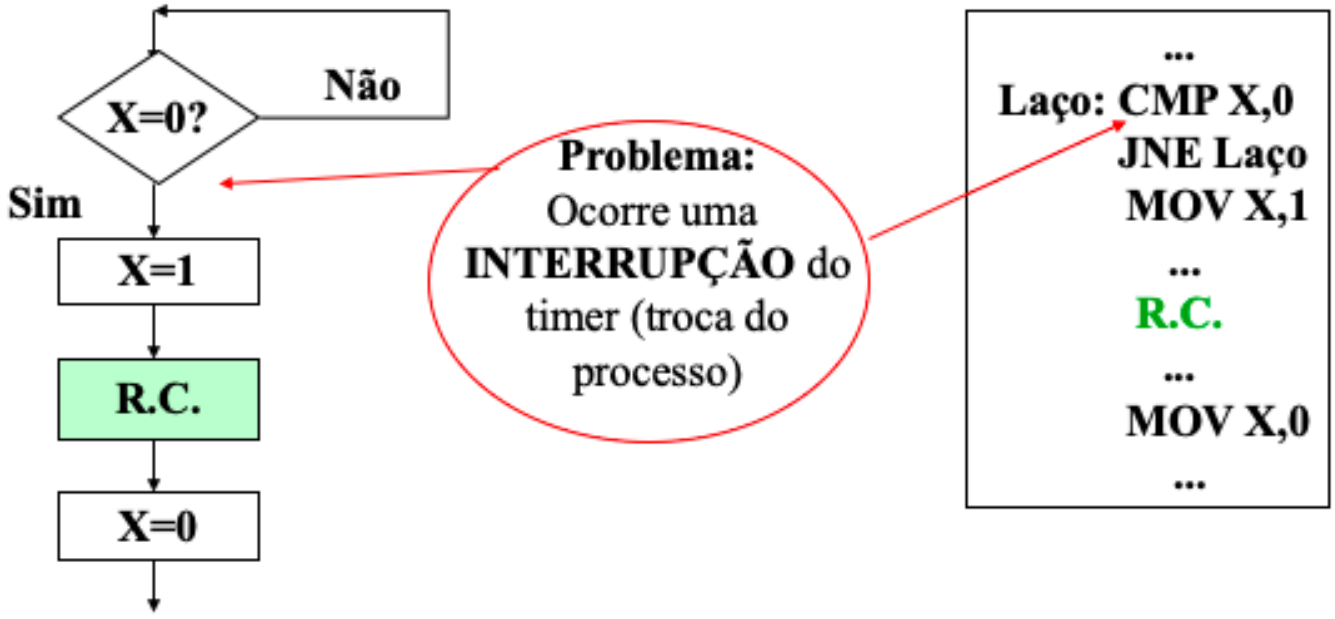
Processo A

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

Processo B

Espera Ocupada - Variável *lock* - Problema

- 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.



Espera Ocupada - *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)

Espera Ocupada - *Strict Alternation* - Problema

- Suponha que o Processo B é mais rápido e sai da região crítica.
- Ambos os processos estão fora da região crítica e o *turn* com valor 0.
- O Processo A termina antes de executar sua região não crítica e retorna ao início do *loop*. Com o *turn* está em 0, 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 à variável *turn* e entra em sua região não crítica.
- 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.
- **Assim, o Processo A fica bloqueado pelo Processo B que NÃO está na sua região crítica - violando a condição 3.**

Comunicação de Processos

Espera Ocupada - 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.

Espera Ocupada - Solução de Peterson e Instrução TSL (*Test and Set Lock*)

- Instruções TSL: utilizam registradores de hardware.
  - TSL RX, LOCK; (lê o conteúdo de *lock* em RX, e armazena um valor diferente de 0 em *lock* - operação indivisível).
  - *LOCK* é compartilhada
    - Se *LOCK*==0, então região crítica "liberada".
    - Se *LOCK*not equal 0, então região crítica "ocupada".

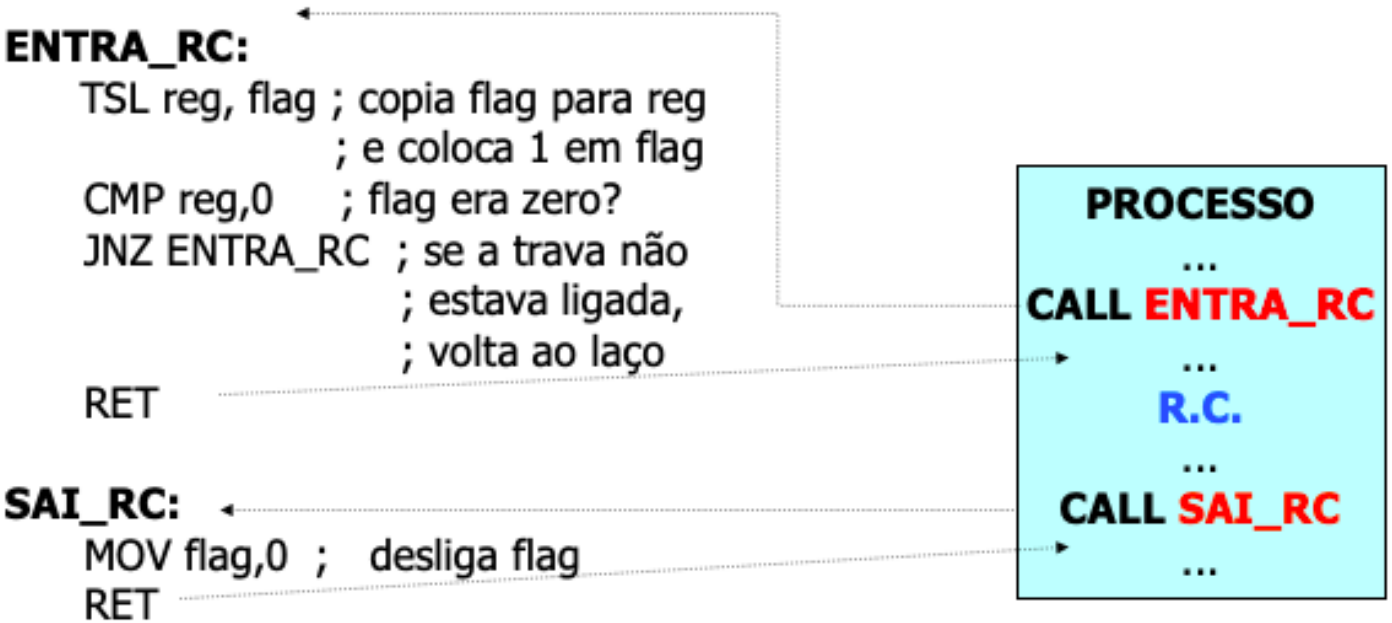
```
enter_region:
    TSL REGISTER, LOCK          | Copia lock para reg. e lock=1
    CMP REGISTER, #0            | lock valia zero?
    JNE enter_region            | Se sim, entra na região crítica,
                                | Se não, continua no laço
    RET                         | Retorna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

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

Comunicação de Processos

Espera Ocupada - Solução de Peterson e Instrução TSL (*Test and Set Lock*)

- Solução implementada **com uso de hardware**.
- Muitos computadores possuem uma instrução especial, chamada TSL (*test and set lock*, que funciona assim: ela lê o conteúdo de uma palavras 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.

Espera Ocupada - Solução de Peterson e Instrução TSL (*Test and Set Lock*)



Comunicação de Processos

Exclusão Mútua com Espera Ocupada

- **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.

Comunicação de Processos

Soluções para Exclusão Mútua

- Espera Ocupada.
- **Primitivas Sleep/Wakeup.**
- Semáforos.
- Monitores.
- Passagem de Mensagem.

Comunicação de Processos

Primitivas *Sleep/Wakeup*

- Todas as soluções apresentadas até agora utilizam a espera ocupara: processos ficam em estado de espera (*looping*) até que possam utilizar a região crítica.
  - desperdiça tempo de CPU.
  - pode provocar bloqueio perpétuo (*deadlock* em sistemas com prioridades (situações inesperadas).

Comunicação de Processos

Primitivas *Sleep/Wakeup*

- Para solucionar esse problema de espera, cria-se um par de primitivas *Sleep* e *Wakeup* - 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

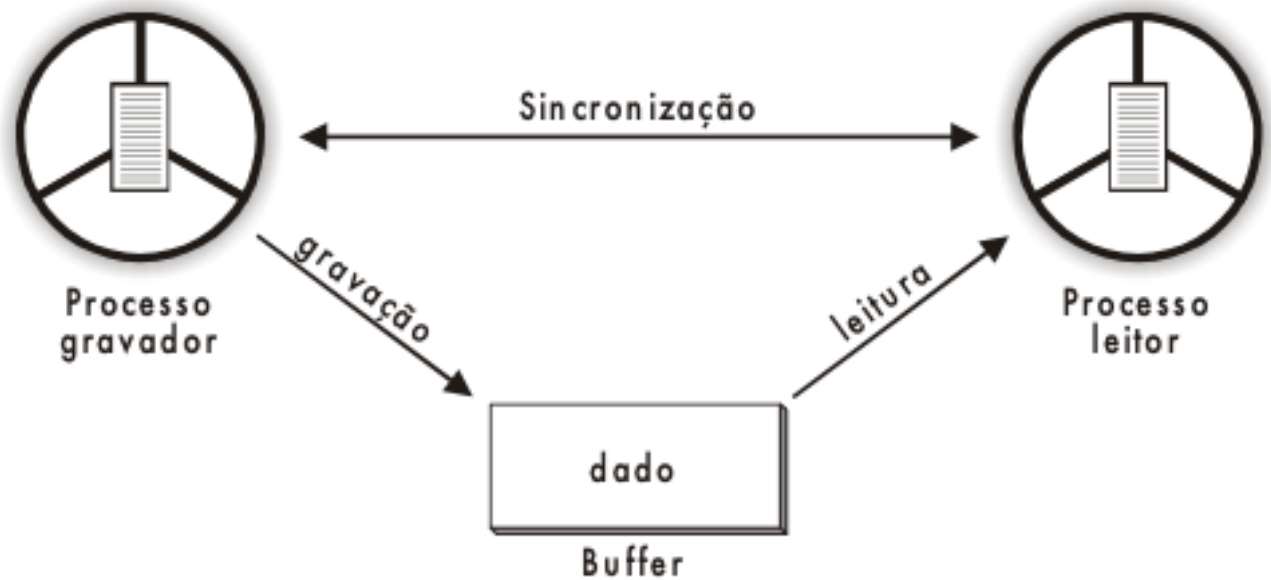
- Problema do Produtor/Consumidor (*bounded buffer*): dois processos compartilham um *buffer* de tamanho fixo. O processo produtor coloca dados no *buffer* e o processo consumidor retira dados do *buffer*.
- Problemas
  - Produtor deseja colocar dados quando o *buffer* ainda está cheio.
  - Consumidor deseja retirar dados quando o *buffer* está vazio.
  - **Solução:** colocar os processos para "dormir", até que eles possam ser executados.

Comunicação de Processos

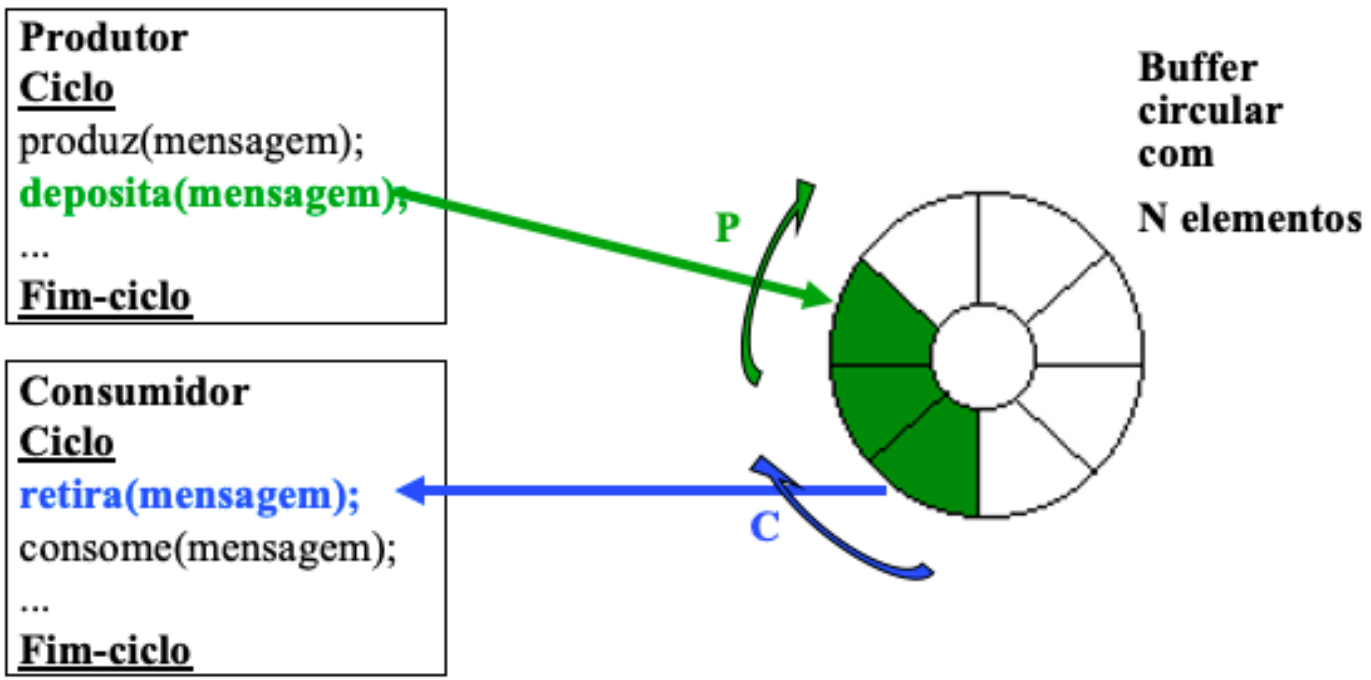
Primitivas *Sleep/Wakeup* - Produtor/Consumidor

- **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.
- **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.

Primitivas *Sleep/Wakeup* - Sincronização Produtor/Consumidor



Primitivas *Sleep/Wakeup* - Sincronização Produtor/Consumidor (*Buffer* Circular)



Comunicação de Processos

Primitivas *Sleep/Wakeup* - Sincronização
Produtor/Consumidor (*Buffer* Circular)

- Restrições do Problema:
  - O produtor não deve exceder a capacidade finita do *buffer*.
  - O consumidor não poderá consumir mensagens mais rapidamente do que forem produzidas.
  - As mensagens devem ser retiradas do *buffer* na mesma ordem em que foram inseridas.
  - Restrição de exclusão mútua no acesso ao *buffer* circular.

Comunicação de Processos

Primitivas *Sleep/Wakeup* - Sincronização
Produtor/Consumidor - Exemplo

- Para 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*.

Primitivas *Sleep/Wakeup* - Sincronização Produtor/Consumidor - Exemplo

```
# 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);
   }
}
```

Primitivas *Sleep/Wakeup* - Sincronização Produtor/Consumidor - Exemplo

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

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

```
consumidor()
{
  while(TRUE)
   {
     if (contador==0)      Sleep();
     retira_item();
     contador - = 1;
     if (contador==N-1)
                Wakeup(produtor);
    consome_item();
   }
}
```

interrupção

Primitivas *Sleep/Wakeup* - Problema

- 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*.
- 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...

Comunicação de Processos

Primitivas *Sleep/Wakeup* - Problema

- 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*

- 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.

Comunicação de Processos

Soluções para 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
  - semáforo=0 – recurso está sendo utilizado.
  - semáforo maior 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 - podem ser implementadas como chamadas ao sistema, desabilitando todas as interrupções por um breve momento.

Comunicação de Processos

Semáforos

- Semáforos usados para implementar exclusão mútua são chamados de **mutex**(*mutual exclusion semaphor*) ou binários, por apenas assumirem os valore 0 e 1 - O recurso é a própria região crítica.
- Vamos resolver o problema do Produtor/Consumidor usando semáforos??
  - *mutex* - exclusão mútua
  - *full* e *empty* - sincronização

Comunicação de Processos

Semáforos (idealizado por E.W. Dijkstra (1965)

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

Comunicação de Processos

Semáforos

- **Down**: verifica se o valor do semáforo é maior do que 0; se for, o semáforo é decrementado; Se o valor for 0, o processo é colocado para dormir sem completar sua operação de *down*.
- Todas essas ações são chamadas de **ações atômicas**.
  - **Ações atômicas** garantem que quando uma operação no semáforo está sendo executada, nenhum processo pode acessar o semáforo até que a operação seja finalizada ou bloqueada.

Comunicação de Processos

Semáforos

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

Comunicação de Processos

Semáforos

- Mecanismo criado para solucionar o problema de armazenar múltiplos *WAKEUPS* (E.W.Dijkstra).
- O mecanismo envolve a utilização de uma variável compartilhada chamada semáforo, e de duas operações primitivas indivisíveis que atuam sobre ela.
- A variável compartilhada pelos processos, poderá assumir valores inteiros não negativos e sua manipulação será restrita às operações **P** e **V** (ou **Down** e **Up**, ou **Wait** e **Signal**, respectivamente).

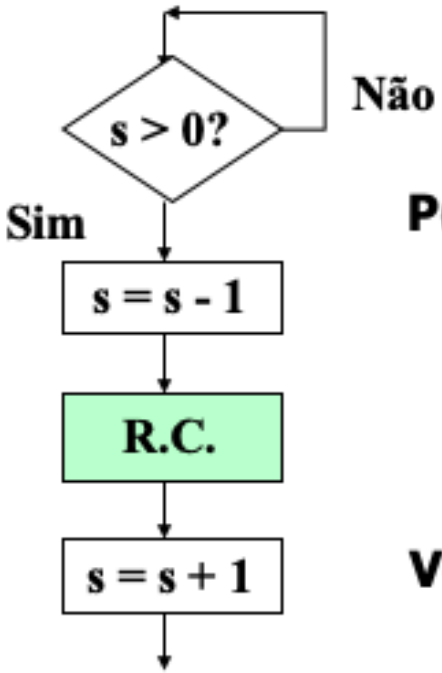
Comunicação de Processos

Semáforos

- As operações que atuam em um semáforo denominado s, incluindo seus efeitos são definidos a seguir:
  - **P(s):** Espera até que s maior 0 e então decrementa s.
  - **V(s):** Incrementa s.

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;

Semáforos (2a. Implementação - Associando uma fila Q_i a cada semáforo s_i)

- 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.

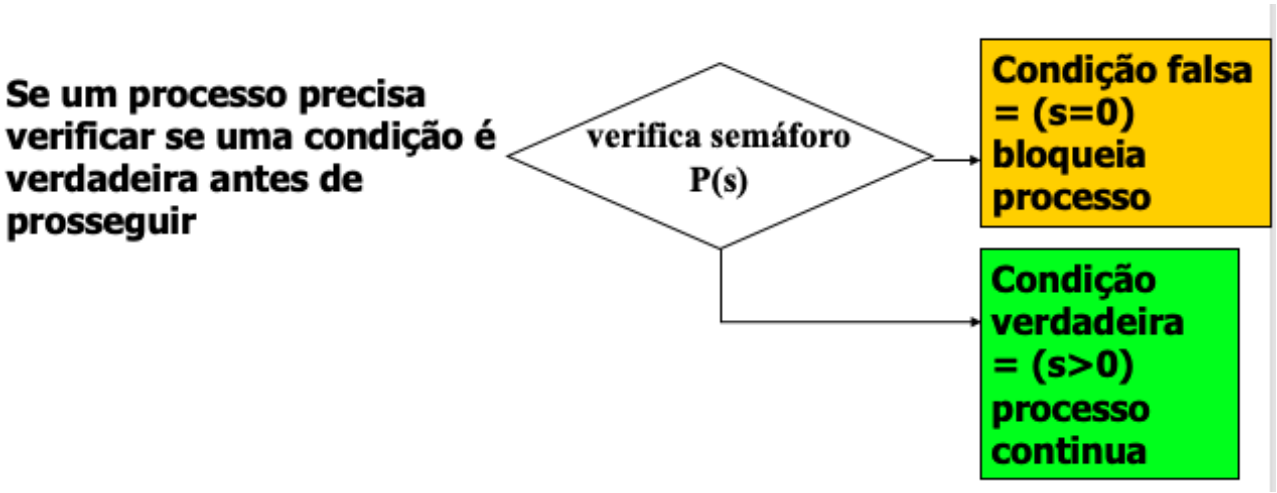
Comunicação de Processos

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 valore 0 e 1.

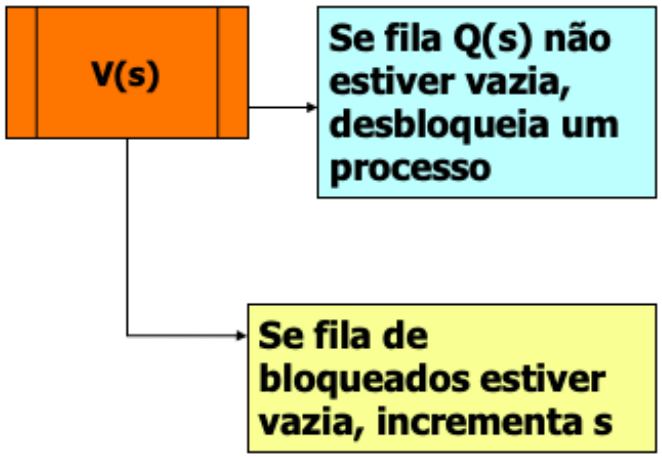
Semáforos

- Sincronização de processos é baseada em condições - Cada condição é representada por um semáforo.

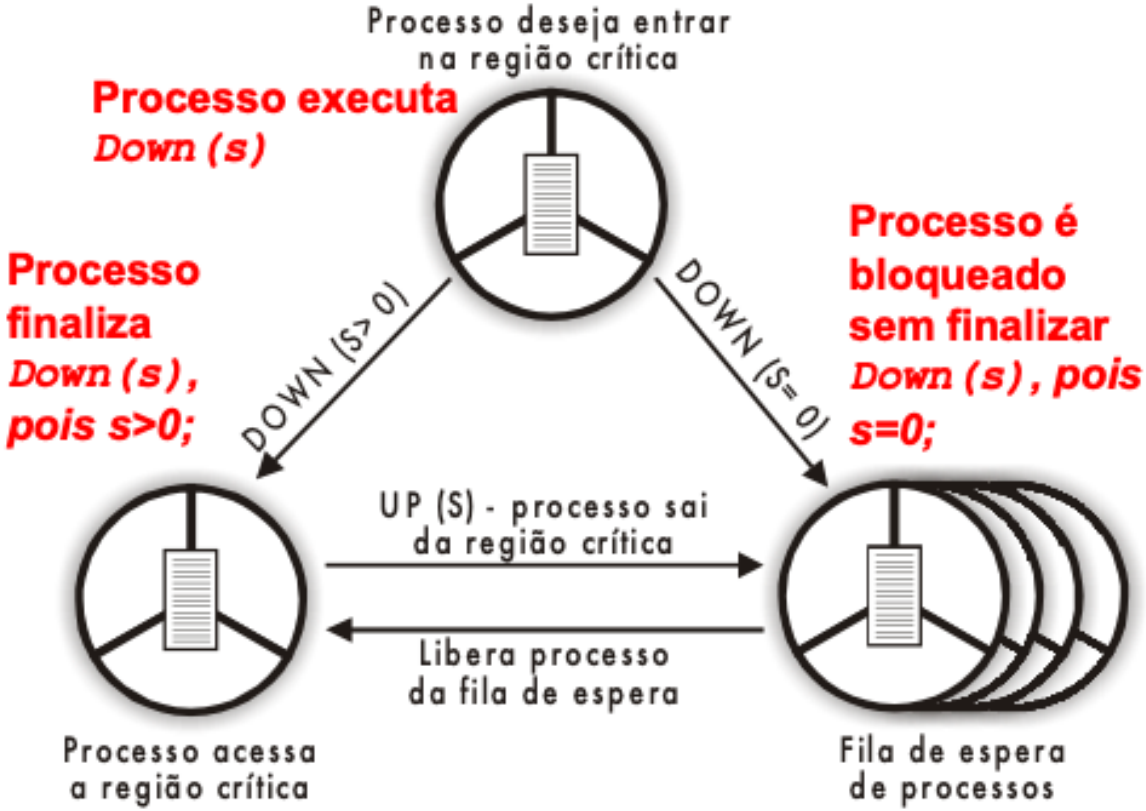


Semáforos

Se um processo torna uma condição verdadeira, deve sinalizar isto através do semáforo, usando V(s)



Semáforos Binários



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**.

Semáforos

```
# include "prototypes.h"
# define N 100

typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer (void){
  int item;
  while (TRUE){
    produce_item(&item);
    down(&empty);
    down(&mutex);
    enter_item(item);
    up(&mutex);
    up(&full);
  }
}
```

```
void consumer (void){
  int item;
  while (TRUE){
    down(&full);
    down(&mutex);
    remove_item(item);
    up(&mutex);
    up(&empty);
    consume_item(item);
  }
}
```

Comunicação de Processos

Semáforos - Problema (Erro de programação pode gerar *deadlocks*)

- 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.

Comunicação de Processos

Soluções para 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.

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**;

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 - Execução

- *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 de 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 de 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.

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;
```

Monitores

- A exclusão mútua automática dos procedimentos do monitor garante que, por exemplo, se o produtor dentro de um procedimento do monitor descobrir que o *buffer* está cheio, esse produtor será capaz de terminar a operação de *WAIT* sem se preocupar, pois o consumidor não estará ativo dentro do monitor até que *WAIT* tenha terminado e o produtor tenha sido marcado como não mais executável.

Comunicação de Processos

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.

Continuemos com **Comunicação entre Processos** ....