

Sistemas Operacionais

Comunicação entre processos.

Exclusão Mútua

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



Exclusão Mútua

- **Espera Ocupada (*Busy Waiting*):**
 - Enquanto um processo estiver ocupado atualizando a memória compartilhada em sua região crítica, nenhum outro processo cause problemas invadindo-a.
 - Algumas soluções:
 - Desabilitar interrupções;
 - Variáveis de Impedimento (*Lock Variables*);
 - Estrita Alternância (*Strict Alternation*);
 - Solução de Peterson e Instrução TSL.

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).

Exclusão Mútua

Desabilitar interrupções:

- 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).

Exclusão Mútua

- **Variáveis de Impedimento:**
 - 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 diretório de *spool*.

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

Exclusão Mútua

- **Estrita Alternância:**
 - Fragmentos de programa controlam o acesso às regiões críticas;
 - Variável *turn*, estabelece qual processo pode entrar na região crítica.

Exclusão Mútua

- **Estrita Alternância:**
 - Um processo A é identificado por 0;
 - E um processo B por 1;
 - O A é executado e entra em sua Região Crítica. Quando sair de sua Região Crítica a variável de estado é setada em 1, indicando que é a vez do processo B.

Exclusão Mútua

- Problema da Estrita Alternância:
 1. Supondo que B não entre na sua Região Crítica, A também não pode entrar na sua. Isto ocorre porque B **não** executou sua Região Crítica e portanto **não** mudou a variável de estado para 0 novamente;
 2. Outro ponto negativo, é que um processo que deseja entrar na sua Região Crítica fica constantemente lendo a variável de estado.


Exclusão Mútua

- Problema da Estrita Alternância:

3. Assim, o processo A fica bloqueado pelo processo B que NÃO está na sua região crítica, violando a condição 3.

Exclusão Mútua

- Solução de Peterson:**

 **O Problema de Espaço na Geladeira**

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

Exclusão Mútua

- Solução de Peterson:**

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

Exclusão Mútua

- **Solução de Peterson:**
- Consiste em dois procedimentos escritos em ANSI C;
- Antes de entrar em sua região crítica, cada processo chama *enter_region()* com seu próprio número de processo, 0 ou 1, como parâmetro;
- Ao sair de sua região crítica, o processo chama *leave_region ()*;
- **Problema:** Caso ambos os processos chamem quase que simultaneamente, o processo que armazenou por último é o que conta, o primeiro é sobreposto e perdido.

Exclusão Mútua

- Instrução TSL (*Test and Set Lock*): Utiliza registradores do *hardware*;
- Instrução TSL é uma chamada de sistema que bloqueia o acesso à memória até o término da execução da instrução
- TSL RX, LOCK; (lê o conteúdo de *lock* em RX, e armazena um valor diferente de zero (0) em *lock* – operação indivisível);
- *Lock* é compartilhada
 - Se $lock == 0$, então região crítica “liberada”.
 - Se $lock \neq 0$, então região crítica “ocupada”.

Exclusão Mútua

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

Exclusão Mútua

- Primitivas *Sleep/Wakeup*
- **Solução:**
 - Para solucionar esse problema de espera, um par de primitivas ***Sleep e Wakeup é utilizado.***
 - 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”.

Exclusão Mútua

- Primitivas *Sleep/Wakeup*
- Solução:
 - 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.

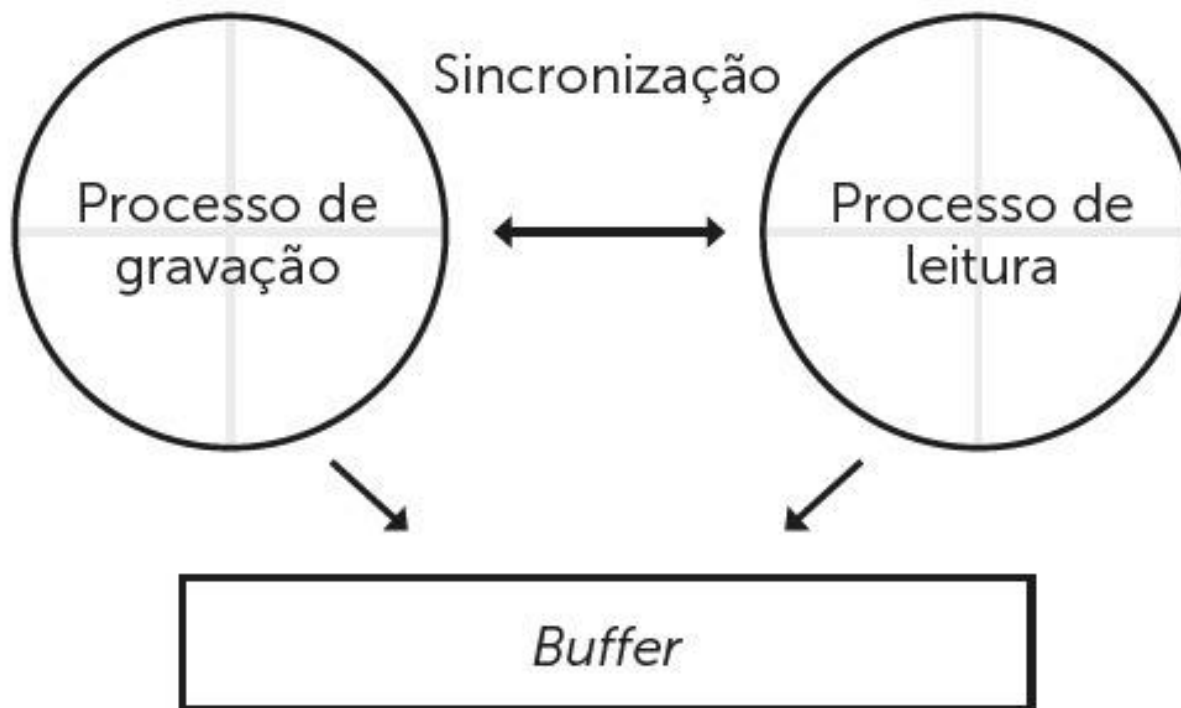
Exclusão Mútua: Primitivas *Sleep/Wakeup*

- Problemas que podem ser solucionados com o uso dessas primitivas:
 - **Problema do Produtor/Consumidor:** dois processos compartilham um *buffer de tamanho fixo*. O processo produtor coloca dados no buffer e o processo consumidor retira dados do buffer.

Exclusão Mútua: Primitivas *Sleep/Wakeup*

- 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.

Exclusão Mútua: Primitivas *Sleep/Wakeup*



Exclusão Mútua: Primitivas *Sleep/Wakeup*

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

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

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

interrupção

Exclusão Mútua: Primitivas *Sleep/Wakeup*

- *Buffer: uma variável **count** controla a quantidade de dados presente no buffer.*
- *Produtor: Antes de colocar dados no *buffer*, o processo produtor checa o valor dessa variável. Se a variável está com valor máximo, o processo produtor é colocado para dormir. Caso contrário, o produtor coloca dados no *buffer* e o incrementa.*

Exclusão Mútua: Primitivas *Sleep/Wakeup*

- Consumidor: Antes de retirar dados no *buffer*, o processo consumidor checa o valor da variável **count** para saber se ela está com 0 (zero). Se está, o processo vai “dormir”, senão ele retira os dados do *buffer* e decrementa a variável.

Exclusão Mútua: Primitivas *Sleep/Wakeup*

- **Problemas desta solução: Acesso à variável *count* é irrestrita:**
- Buffer está vazio. Consumidor testa o valor de *count*, que é zero, mas não tem tempo de executar *sleep*, pois o escalonador selecionou agora produtor.
- Este produz um item, insere-o no buffer e incrementa *count*. Como *count* = 1, produtor chama *wakeup* para acordar consumidor

Exclusão Mútua: Primitivas *Sleep/Wakeup*

- O sinal não tem efeito (é perdido), pois o consumidor ainda não está logicamente adormecido.
- Consumidor ganha a CPU, executa *sleep* e vai dormir.
- Produtor ganha a CPU e, cedo ou tarde, encherá o *buffer*, indo também dormir. Ambos dormirão eternamente.

Exclusão Mútua: Primitivas *Sleep/Wakeup*

- ***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!!!!***