Sistemas Operacionais

Prof. Dr. Helder Oliveira

Plano de Aula

- Comunicação entre processos.
- Sincronização entre processos.

Processos

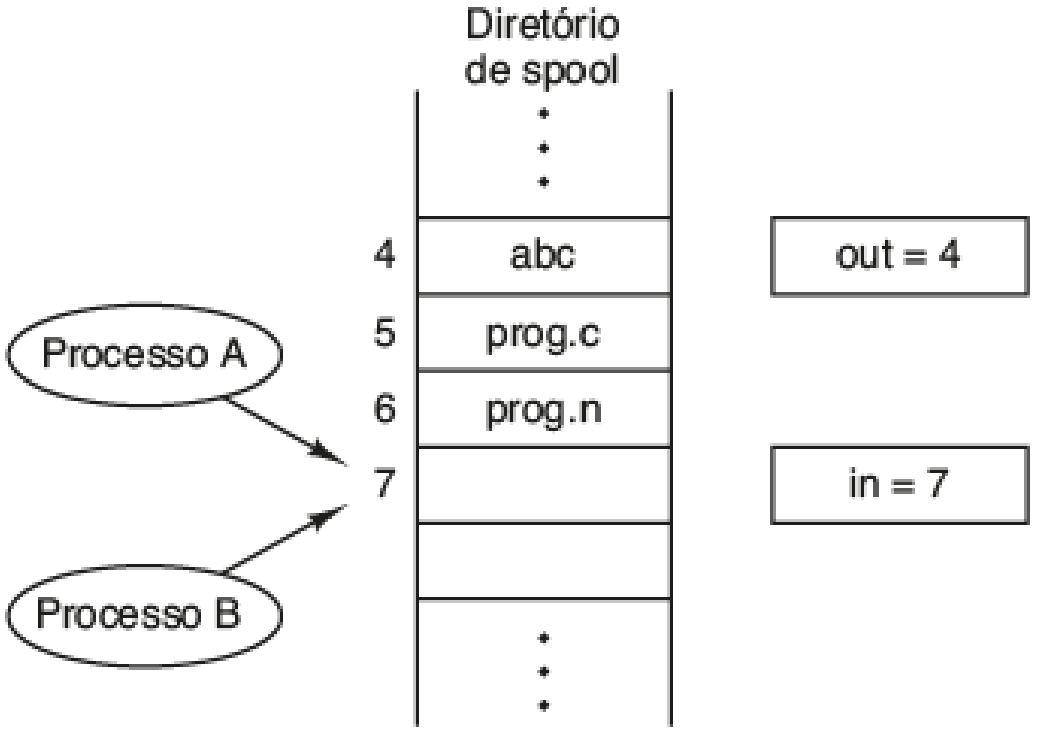
- Processos quase sempre precisam comunicar-se com outros processos.
- Comunicação entre processos (interprocess communication — IPC).
- Três questões:
  1. Como um processo pode passar informações para outro?
  2. Como certificar-se de que dois ou mais processos não se atrapalhem?
  3. Como garantir sequenciamento adequado quando dependências estão presentes?
  - A questão 2 e 3 se aplica a threads.

Condições de corrida

- Processos podem compartilhar memória
- Podem compartilhar:
  - Memoria principal
  - Arquivo compartilhado.
- Não muda a natureza da comunicação ou os problemas que surgem.

Exemplo de condição de corrida

FIGURA 2.21 Dois processos querem acessar a memória compartilhada ao mesmo tempo.



Condição de corrida

- Situação onde dois processos ou mais processo acessam recursos compartilhados concorrentemente.
- Há uma corrida pelo recurso.
- Ex: Leitura ou escrita compartilhada
  - Resultado depende de quem processa no momento.
- a = d+c
- x = a+y
- A variável **a** é a região critica.

Regiões críticas

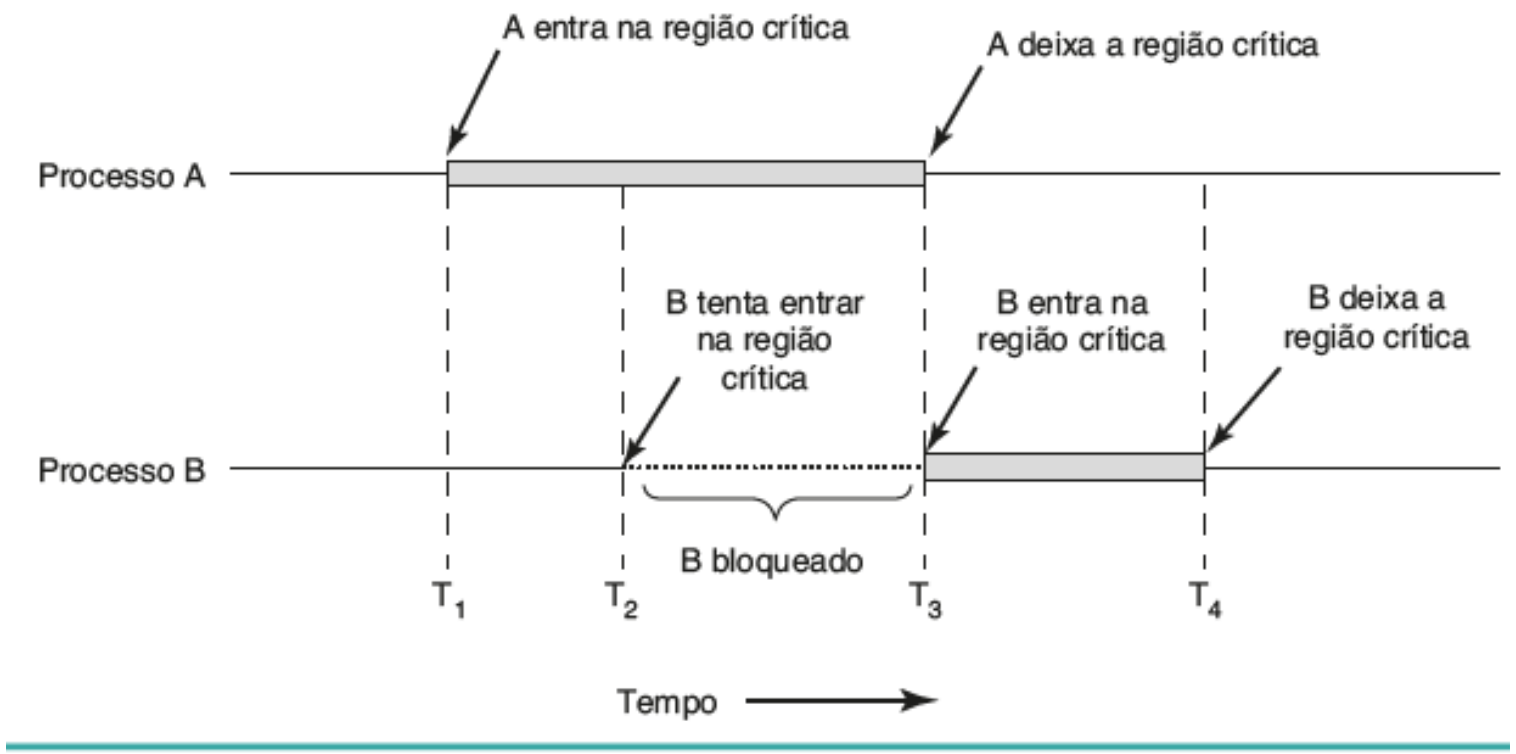
- Como evitar as condições de corrida?
  - Proibir mais de um processo de ler e escrever os dados compartilhados ao mesmo tempo.
  - **Exclusão mútua**
    - Se certificar de que se um processo está usando um arquivo ou variável compartilhados, os outros serão impedidos de realizar a mesma coisa.
  - Região crítica ou seção crítica
    - parte do programa onde a memória compartilhada é acessada e pode levar a corridas

Regiões críticas

- Resolver regiões críticas não garante que processos em paralelo cooperem de modo correto e eficiente usando dados compartilhados.
- Quatro condições:
  1. Dois processos jamais podem estar simultaneamente dentro de suas regiões críticas.
  2. Nenhuma suposição pode ser feita a respeito de velocidades ou do número de CPUs.
  3. Nenhum processo executando fora de sua região crítica pode bloquear qualquer processo.
  4. Nenhum processo deve ser obrigado a esperar eternamente para entrar em sua região crítica.

Exclusão mutua e região crítica

FIGURA 2.22 Exclusão mútua usando regiões críticas.



Soluções de exclusão mútua

- Espera ocupada (Busy waiting)
- Primitivas Dormir/Acordar (Sleep/Wakeup)
- Semáforos
- Monitores
- Troca de Mensagem

Exclusão mútua com espera ocupada

- Quando um processo está ocupado atualizando a memória compartilhada em sua região crítica, nenhum outro entrará na sua região crítica para causar problemas.
- Algumas soluções para exclusão mútua com espera ocupada:
  - Desabilitar interrupções.
  - Variáveis de travamento(lock)
  - Alternância explícita
  - Solução de Peterson
  - Instrução TSL

Exclusão mútua com espera ocupada

- **Desabilitando interrupções**
  - Processo desabilita todas as interrupções.
  - Processo reabilita um momento antes de partir.
  - Abordagem é pouco atraente.
  - Núcleo desabilitar interrupções para atualizar variáveis.

  - A conclusão é: desabilitar interrupções é muitas vezes uma técnica útil dentro do próprio sistema operacional, mas não é apropriada como um mecanismo de exclusão mútua geral para processos de usuário.

Exclusão mútua com espera ocupada

- **Variáveis do tipo trava**
  - Processo que deseja utilizar a região crítica atribui um valor a variável chamada lock.
  - Se a variável está com 0, nenhum processo está na região crítica; 1 significa que existe um processo na região crítica.
  - Consumo de CPU sem computação útil.
  - Problema: Mesma falha fatal que vimos no diretório de spool.

Exclusão mútua com espera ocupada

- **Alternância explícita**
  - A variável do tipo inteiro turn, controla de quem é a vez de entrar na região crítica.
  - Espera ocupada
  - Trava giratória (spin lock).
  - Processo A e processo B alternam a utilização da região crítica.
  - Problema: Quando um processo quer utilizar 2 vezes consecutivas.

FIGURA 2.23 Uma solução proposta para o problema da região crítica. (a) Processo 0. (b) Processo 1.

```
while (TRUE) {
        while (turn !=0)
        critical_region( );
        turn = 1;
        noncritical_region( );
}
```

(a)

```
while (TRUE) {
        while (turn !=1)
        critical_region( );
        turn = 0;
        noncritical_region( )
}
```

(b)

Exclusão mútua com espera ocupada

- **Solução de Peterson**
  - Uma maneira muito mais simples de realizar a exclusão mútua.
  - Processo chama enter_region.
  - Espera até que seja seguro entrar.
  - Processo chama leave_region para indicar que ele terminou e para permitir que outros processos entrem.

Exclusão mútua com espera ocupada

Solução de Peterson

FIGURA 2.24 A solução de Peterson para realizar a exclusão mútua.

```
#define FALSE  0
#define TRUE   1
#define N      2                        /* numero de processos */

int turn;                               /* de quem e a vez? */
int interested[N];                      /* todos os valores 0 (FALSE) */

void enter_region(int process);         /* processo e 0 ou 1 */
{
     int other;                         /* numero do outro processo */

     other = 1 – process;               /* o oposto do processo */
     interested[process] = TRUE;        /* mostra que voce esta interessado */
     turn = process;                    /* altera o valor de turn */
     while (turn == process && interested[other] == TRUE) /* comando nulo */ ;
}

void leave_region(int process)          /* processo: quem esta saindo */
{
     interested[process] = FALSE;       /* indica a saida da regiao critica */
}
```

Exclusão mútua com espera ocupada

- **Instrução TSL**
- Exige um pouco de ajuda do hardware.

  TSL RX,LOCK

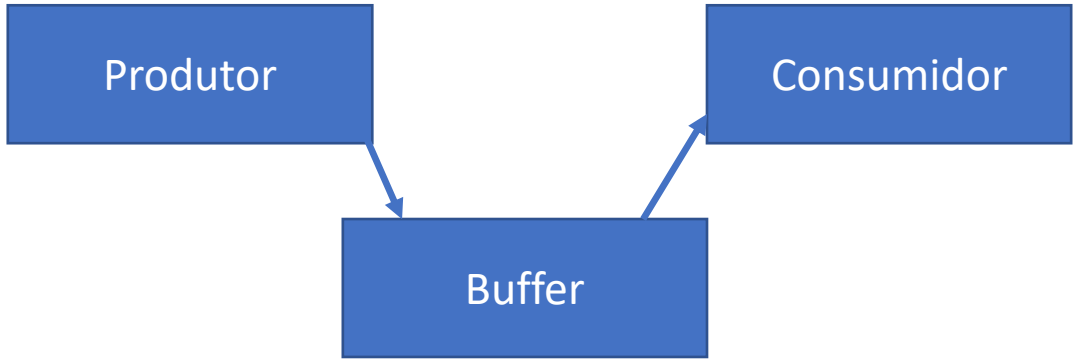
- A CPU executando a instrução TSL impede o acesso ao barramento de memória para proibir que outras CPUs acessem a memória até ela terminar.

Exclusão mútua com primitivas Dormir/Acordar

- Bloqueiam em vez de desperdiçar tempo da CPU quando eles não são autorizados a entrar nas suas regiões críticas.
- Sleep - chamada de sistema que faz com que o processo que a chamou bloqueie, isto é, seja suspenso até que outro processo o desperte.
- Wakeup tem um parâmetro, o processo a ser desperto.
- Alternativamente, tanto sleep quanto wakeup cada um tem um parâmetro, um endereço de memória usado para parear sleeps com wakeups.

Exclusão mútua com primitivas Dormir/Acordar

- Problema do produtor consumidor
  - Dois processos compartilham um buffer de tamanho fixo.
  - O processo produtor coloca dados no buffer.
  - Consumidor retira dados do buffer.
- Problema: O produtor deseja colocar dados e o buffer esta cheio: enquanto o consumidor deseja consumir dados e o buffer esta vazio.



Semáforos

- Um novo tipo de variável inteira, chamada de semáforo para controlar recursos compartilhados.
  - Contar o número de sinais de acordar.
  - Um semáforo podia ter o valor 0, indicando que nenhum sinal de despertar fora salvo, ou algum valor positivo se um ou mais sinais de acordar estivessem pendentes.
  - Duas operações nos semáforos:
    - Down e up (generalizações de sleep e wakeup).
  - Conferir o valor, modificá-lo e possivelmente dormir são feitos como uma única ação atômica indivisível.

Monitores

- Uma coleção de rotinas, variáveis e estruturas de dados que são reunidas em um tipo especial de módulo ou pacote.
- Somente um processo pode estar ativo dentro do monitor em um mesmo instante.
- Outros processos ficam bloqueados até que possam esta ativos no monitor.

FIGURA 2.33 Um monitor.

```
monitor example
    integer i;
    condition c;

    procedure producer ( );
    .
    .
    .
    end;

    procedure consumer ( );
    .       .       .
    end;
end monitor;
```

Limitações de monitores e Semáforos

- Não são boas soluções para sistemas distribuídos.
- Não proveem sincronização entre processos em maquinas diferentes.
- Monitores dependem de uma linguagem de programação.
  - Poucas linguagem suportam monitores.
- Precisamos de algo mais …. Mensagens!

Troca de mensagem

- Esse método de comunicação entre processos usa duas primitivas, **send** e **receive**, que, como semáforos e diferentemente dos monitores, são chamadas de sistema em vez de construções de linguagem.

```
send(destination, &message);
receive(source, &message);
```

- Problemas e questões de projeto
  - Confirmação de recebimento
  - Recebimento duplicado
  - Autenticação

Leitura

- SISTEMAS OPERACIONAIS MODERNO 4ª edição
  - 2.3 - Comunicação entre processos.

Dúvidas?