SISTEMAS OPERACIONAIS

Sincronização entre Processos

Andreza Leite
andreza.leite@univasf.edu.br

Sincronização

- Freqüentemente, os processos precisam se comunicar com outros processos.
  - ✓ Isto ocorre quando os processos **compartilham** ou **trocam** dados entre si.
- Há a necessidade dessa comunicação ocorrer, de preferência, de uma forma bem estruturada e sem interrupções.
  - ✓ As interrupções limitam o desempenho e aumentam a complexidade.

Sincronização

- Três tópicos serão abordados:
  - Como um processo passa informação para um outro processo.
  - Como garantir que dois ou mais processos não invadam uns aos outros.
  - Como garantir uma seqüência adequada quando existe uma dependência entre processos.

Sincronização

- Abordaremos diversos mecanismos para tratar essas questões:
  - ✓ Condições de disputa
  - ✓ Regiões críticas
  - ✓ Exclusão mútua
  - ✓ Dormir e acordar
  - ✓ Semáforos

Sincronização

- Em alguns SOs, processos podem compartilhar um armazenamento comum, onde cada um é capaz de ler e escrever.
  - ✓ Este armazenamento comum pode, por exemplo, estar na memória principal, em um arquivo compartilhado entre outros.

Comunicação e Sincronização

- Exemplo de comunicação interprocessos: ***spool* de impressão.**
  - ✓ Quando um processo quer imprimir um arquivo, este processo entra com o nome do arquivo em um diretório de *spool* especial.
  - ✓ Um outro processo, verifica periodicamente se existe algum arquivo para ser impresso, se houver, os imprime e remove seus nomes do diretório.

Comunicação e Sincronização

- Suponha que o diretório de *spool* tenha um grande número de vagas numeradas 0, 1, 2,...
- Cada vaga é capaz de conter um nome de arquivo.
- Suponha também que existem duas variáveis compartilhadas.
  - ✓ **Out** → aponta para o próximo arquivo a ser impresso.
  - ✓ **In** → Aponta para a próxima vaga livre do diretório.

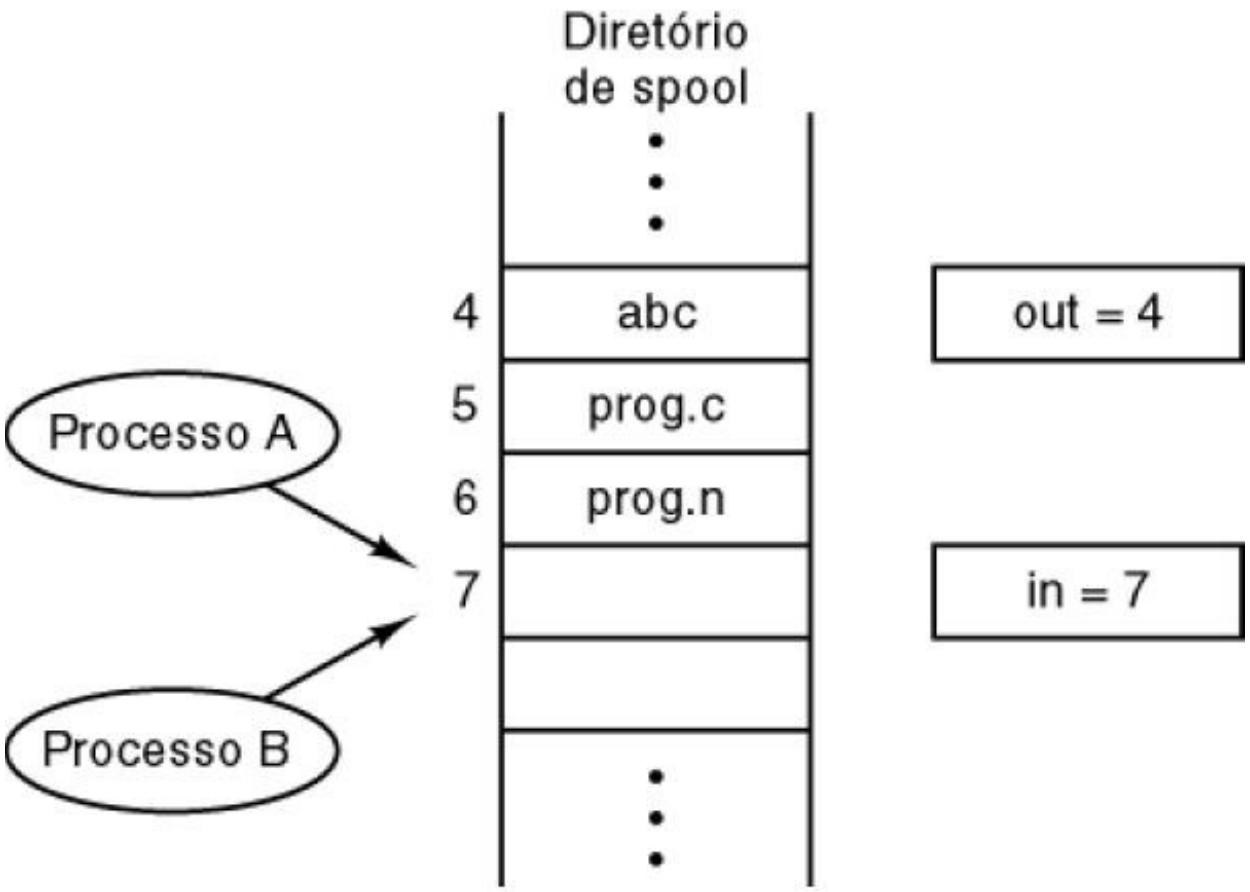
Sincronização

- Em um dado instante, as vagas 4 a 6 estão preenchidas (com os nomes dos arquivos na fila de impressão), e

  In = 7.

- **Problema** → Quase simultaneamente, os processos A e B decidem colocar um arquivo na fila de impressão.

Condições de Disputa



Dois processos que querem ter acesso simultâneo a memória compartilhada.

Condições de Disputa

- Pode ocorrer o seguinte:
  - ✓ O processo A lê **In** e armazena o valor 7 na sua variável local chamada *proxima_vaga_livre*.
  - ✓ Logo em seguida, ocorre uma interrupção do relógio e a CPU decide que o processo A já executou o suficiente e então alterna para o processo B.
  - ✓ O processo B também lê In e obtém igualmente o valor 7. Da mesma forma, B armazena o 7 na sua variável local próxima_vaga_livre.
  - ✓ Neste momento, ambos os processos tem a informação de que a vaga livre é a 7.
  - ✓ B prossegue sua execução, armazenando o nome do seu arquivo na vaga 7 e atualiza In para 8.

Condições de Disputa

- ✓ Em seguida, o processo A executa novamente de onde parou. Verifica a variável local proxima_vaga_livre, que é igual a 7, e então escreve o nome do seu arquivo na vaga 7, apagando o nome que B acabou de colocar lá. O processo A atualiza o valor de In para 8.

- ✓ Resultado → O processo B nunca terá seu arquivo impresso.

- ✓ Situações como esta são chamadas de condições de disputa.

Regiões Críticas

- A parte do programa que gera uma condição de disputa é chamada de **região crítica** ou **seção crítica.**

- Programas ou partes de programas que não geram condições de disputa são chamados de **códigos reentrante** ou **código público.**

Condições de Disputa

- **Como evitar as condições de disputa?**
  - Prover algum meio de assegurar que outros processos sejam impedidos de usar uma variável ou um arquivo compartilhado que já estiver em uso por outro processo

    → **Exclusão mútua.**

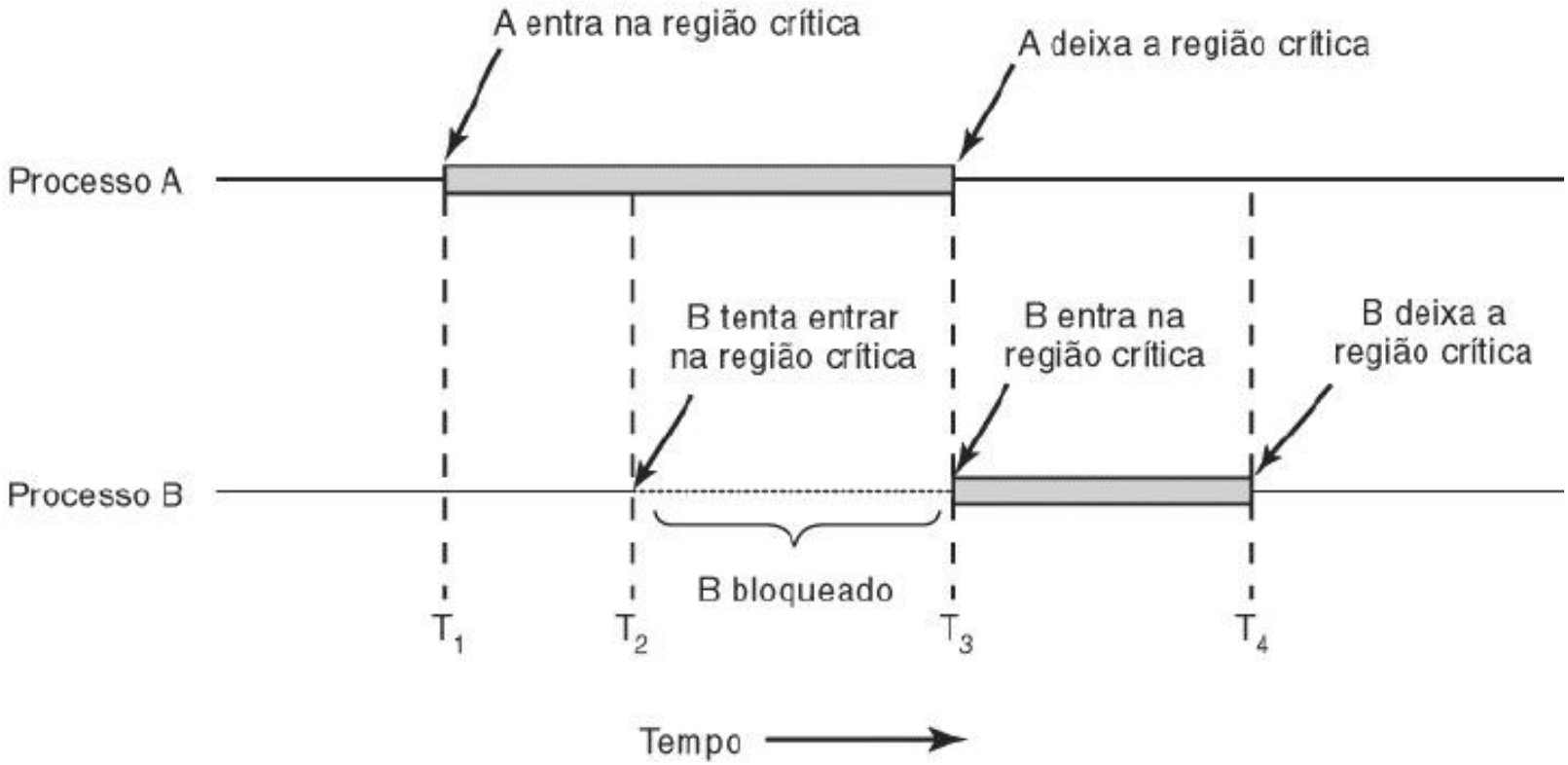
Regiões Críticas

- Quatro condições necessárias para garantir a **exclusão mútua:**

1. Nunca dois processos podem estar simultaneamente em suas regiões críticas.
2. Nada pode ser afirmado sobre a velocidade ou sobre o número de CPUs.
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.

Regiões Críticas

- Comportamento desejado.



Exclusão mútua usando regiões críticas.

Exclusão Mútua

- Alternativas para realizar exclusão mútua com espera ociosa.

  - ✓ Desabilitar interrupções.
  - ✓ Variáveis de impedimento/trava (*lock variables*).
  - ✓ Alternância obrigatória.
  - ✓ Solução/Algoritmo de Peterson.

    (Permite a dois ou mais processos ou subprocessos compartilharem um recurso sem conflitos, utilizando apenas memória compartilhada para a comunicação)

Dormir e Acordar

- A solução de Peterson é correta mas apresenta o defeito de precisar da espera ociosa.
  - Quando quer entrar na região crítica um processo verifica se sua entrada é permitida. Se não for, ele ficará em um laço até que possa entrar.
    - Gasta tempo de CPU
- Observemos as primitivas de comunicação entre processos que bloqueiam em vez de gastar tempo de CPU quando não podem entrar em sua região crítica
  - Uma das mais simples é o par *sleep* e *wakeup*

Dormir e Acordar

- *Sleep* é uma chamada de sistema que faz com que o processo que a chama durma/ fique suspenso até que outro processo o desperte.
- A chamada *wakeup* tem como parâmetro o processo a ser despertado
- Estas chamadas podem ainda podem ter outro parâmetro:
  - um endereço de memória para equiparar os *wakeups* a seus respectivos *sleeps*.

Problema do produtor- consumidor

```
#define N 100                                   /* número de lugares no buffer */
int count = 0;                                  /* número de itens no buffer */

void producer(void)
{
    int item;

    while (TRUE) {                              /* repita para sempre */
        item = produce_item( );                 /* gera o próximo item */
        if (count == N) sleep( );               /* se o buffer estiver cheio, vá dormir */
        insert_item(item);                      /* ponha um item no buffer */
        count = count + 1;                      /* incremente o contador de itens no buffer */
        if (count == 1) wakeup(consumer);       /* o buffer estava vazio? */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                              /* repita para sempre */
        if (count == 0) sleep( );               /* se o buffer estiver cheio, vá dormir */
        item = remove_item( );                  /* retire o item do buffer */
        count = count - 1;                      /* decresça de um o contador de itens no buffer */
        if (count == N - 1) wakeup(producer);   /* o buffer estava cheio? */
        consume_item(item);                     /* imprima o item */
    }
}
```

Figura 2.22 O problema produtor–consumidor com uma condição de disputa fatal.

Semáforos

- Uma variável inteira para contar o número de sinais de acordar salvos para uso futuro.
- Este pode conter valor 0 (nenhum sinal de acordar salvo) ou valor positivo (um ou mais sinais de acordar salvos)
- Duas operações (generalizações de *sleep* e *wakeup*)
  - *Down* – decrementa (gasta um sinal de acordar), se o valor do semáforo é maior que 0. Senão será posto pra dormir.
  - *Up* – incrementa o valor do semáforo e um processo que esteja dormindo pode ser escolhido para terminar o seu *down*.
- As operações sobre os semáforos são atômicas e indivisível.

Problema do produtor- consumidor

```
#define N 100                                   /* número de lugares no buffer */
typedef int semaphore;                          /* semáforos são um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso à região crítica */
semaphore empty = N;                            /* conta os lugares vazios no buffer */
semaphore full = 0;                             /* conta os lugares preenchidos no buffer */

void producer(void)
{
    int item;

    while (TRUE) {                              /* TRUE é a constante 1 */
        item = produce_item( );                 /* gera algo para pôr no buffer */
        down(&empty);                           /* decresce o contador empty */
        down(&mutex);                           /* entra na região crítica */
        insert_item(item);                      /* põe novo item no buffer */
        up(&mutex);                             /* sai da região crítica */
        up(&full);                              /* incrementa o contador de lugares preenchidos */
    }
}


void consumer(void)
{
    int item;

    while (TRUE) {                              /* laço infinito */
        down(&full);                            /* decresce o contador full */
        down(&mutex);                           /* entra na região crítica */
        item = remove_item( );                  /* pega item do buffer */
        up(&mutex);                             /* sai da região crítica */
        up(&empty);                             /* incrementa o contador de lugares vazios */
        consume_item(item);                     /* faz algo com o item */
    }
}
```

Figura 2.23 O problema produtor–consumidor usando semáforos.