Capítulo

5

Execução assíncrona concorrente

Sumário

PEARSON Prentice Hall

Objetivos

- **Este capítulo apresenta:**
  - Os desafios da sincronização de processos e threads concorrentes.
  - Seções críticas e a necessidade de exclusão mútua.
  - Como implementar primitivas de exclusão mútua em software.
  - Primitivas de exclusão mútua em hardware.
  - Utilização e implementação de semáforos.

5.1 Introdução

Execução concorrente

- Pode haver mais de um thread no sistema simultaneamente.
- O thread pode ser executado independentemente ou cooperativamente.
- Execução assíncrona
  - Os threads em geral são independentes.
  - Podem se comunicar ou sincronizar ocasionalmente.
  - O gerenciamento dessas interações é complexo e difícil.

5.1.1 - Sincronização

Frequentemente, 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.

Três tópicos serão abordados:

Como um processo passa informação para um outro processo.

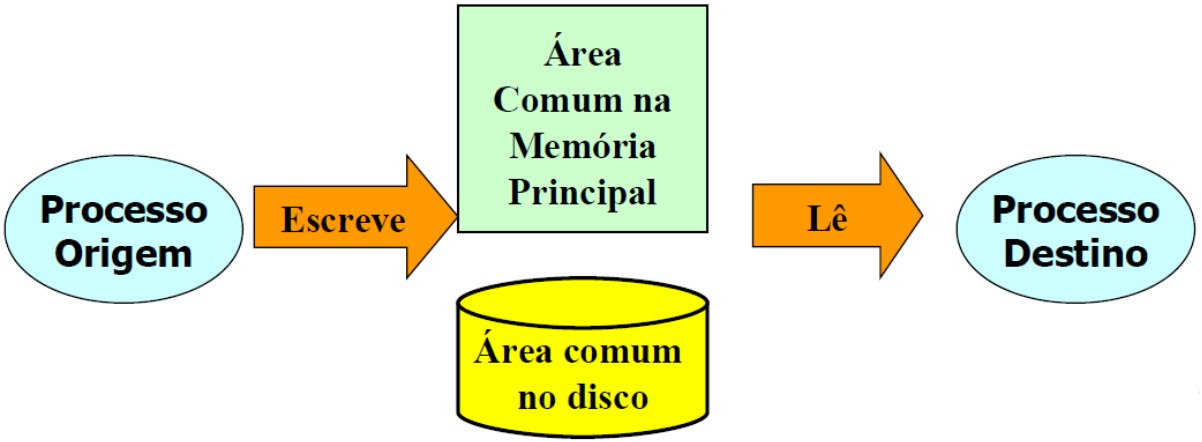
Como garantir que dois ou mais processos não invadam uns aos outros.

Como garantir uma sequência adequada quando existe uma dependência entre processos.

PEARSON Prentice Hall

5.1.2 - Condições de corrida

Em alguns Sistemas Operacionais: os processos se comunicam através de alguma área de armazenamento comum. Esta área pode estar na memória principal ou pode ser um arquivo compartilhado.



Exemplo: a = b + c; x = a + y;

5.1.2 - Condições de corrida

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.

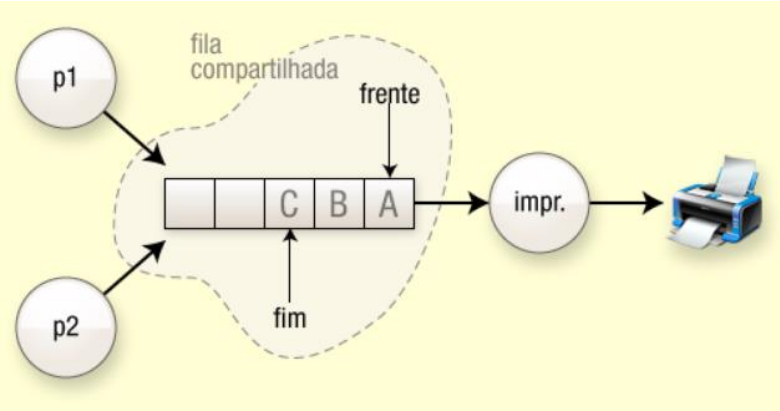
Race Conditions: processos acessam recursos compartilhados concorrentemente: memória, arquivos, impressoras, discos, variáveis;

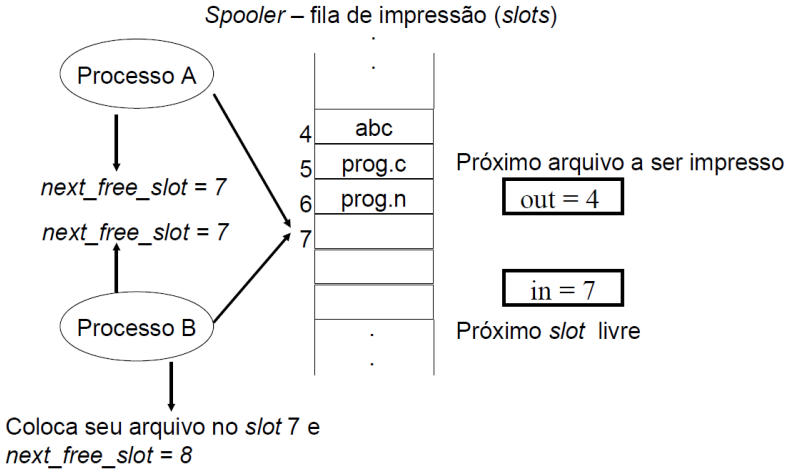
Impressão: quando um processo deseja imprimir um arquivo, ele coloca o arquivo em um local especial chamado ***spooler*** (tabela).

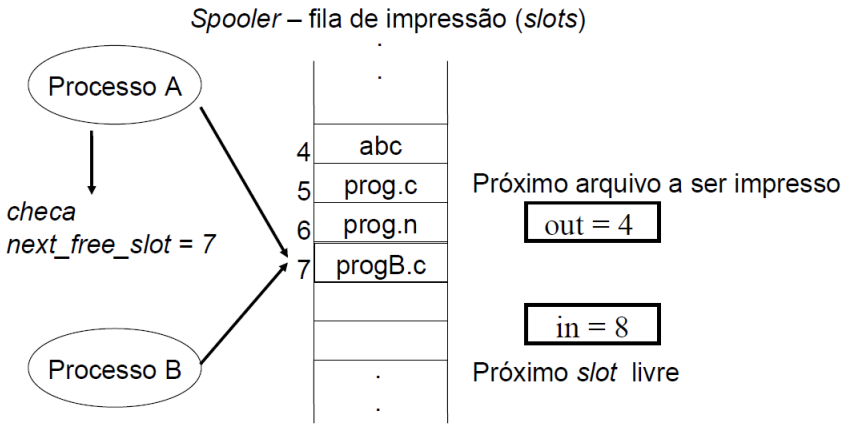
- Um outro processo, chamado ***printer spooler***, checa se existe algum arquivo a ser impresso. Se existe, esse arquivo é impresso e retirado do ***spooler***. Imagine dois processos que desejam ao mesmo tempo imprimir um arquivo...

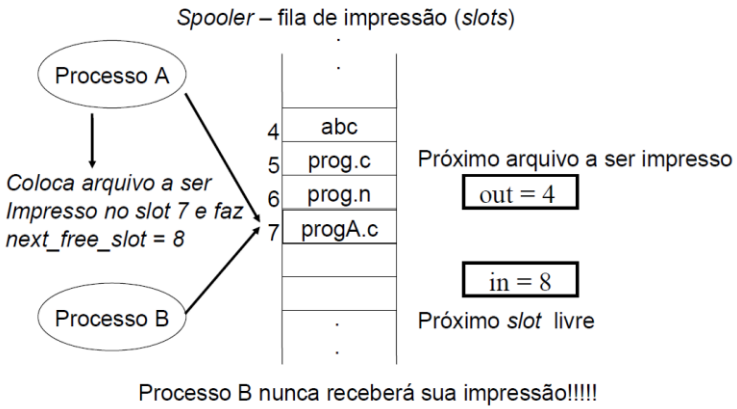
PEARSON Prentice Hall

5.1.2 - Condições de corrida









5.1.2 - Condições de corrida

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.
- 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.

PEARSON
Prentice Hall

5.2 Regiões Críticas e Exclusão Mútua

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 críticas ou regiões críticas (**R.C.).**

Reservar uma cadeira em avião

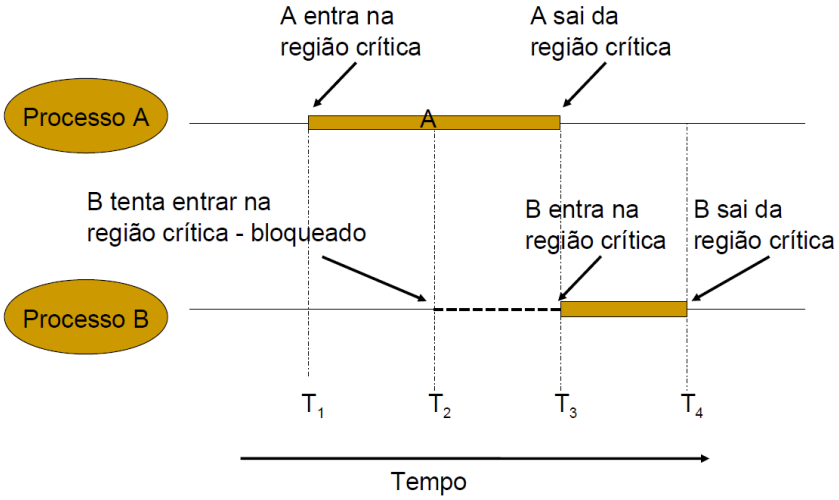
1. Operador OP1 (no Brasil) lê Cadeira1 vaga;
2. Operador OP2 (no Japão) lê Cadeira1 vaga;
3. Operador OP1 compra Cadeira1;
4. **Operador OP2 compra Cadeira1;**

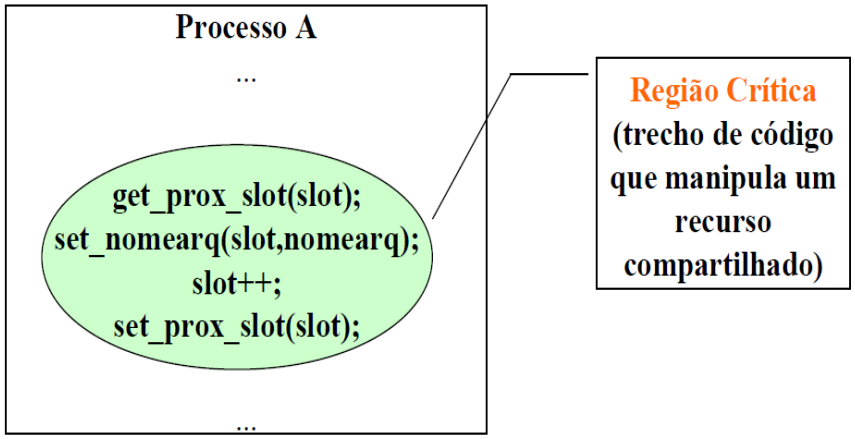
PEARSON Prentice Hall

5.2.1 Regras para programação concorrente

Precisamos que quatro condições se mantenham para chegar a uma boa solução:

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(**Inanição**).





5.3 Soluções exclusão mútua

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

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)

PEARSON Prentice Hall

Desabilitando 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 funciona em ambiente multicore, pois um núcleo fica bloqueado enquanto que outros não.
- Não é uma solução segura, pois um processo pode não habilitar novamente suas interrupções e não ser finalizado;
  - **Viola condição 4;**

Variáveis de impedimento/trava (lock variables) - 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;
- Apresenta o mesmo problema do exemplo do spooler de impressão;
- **Viola condição 1;**

```
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

5.3.3 Exclusão mútua com espera ociosa

Strict Alternation:

- Fragmentos de programa controlam o acesso às regiões críticas;
- Variável ***turn***, inicialmente em ***0***, estabelece qual processo pode entrar na região crítica;
- Turn não são uma boa ideia quando um dos processos é muito mais lento do que o outro.
- **Viola condição 3;**

```
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

5.3.4 Exclusão mútua com espera ociosa

Solução/Algoritmo de Peterson.

- Antes de entrar em uma região crítica o processo deve executar a função ***entra_rc(id)*** (id é o identificador do processo). Ao sair, deve executar ***sai_rc(id)***. Supondo que há apenas dois processos:

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 */
}
```

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.

sleep(): bloqueia o processo (para de do estado "rodando" para "bloqueado") que a chamou até que outro processo o "acorde".

wakeup(pid): acorda o processo cujo identificador é **pid**.

5.4.1 O Problema do Produtor/Consumidor com Fila (buffer) de tamanho limitado

Os dois processos compartilham um buffer de tamanho fixo.

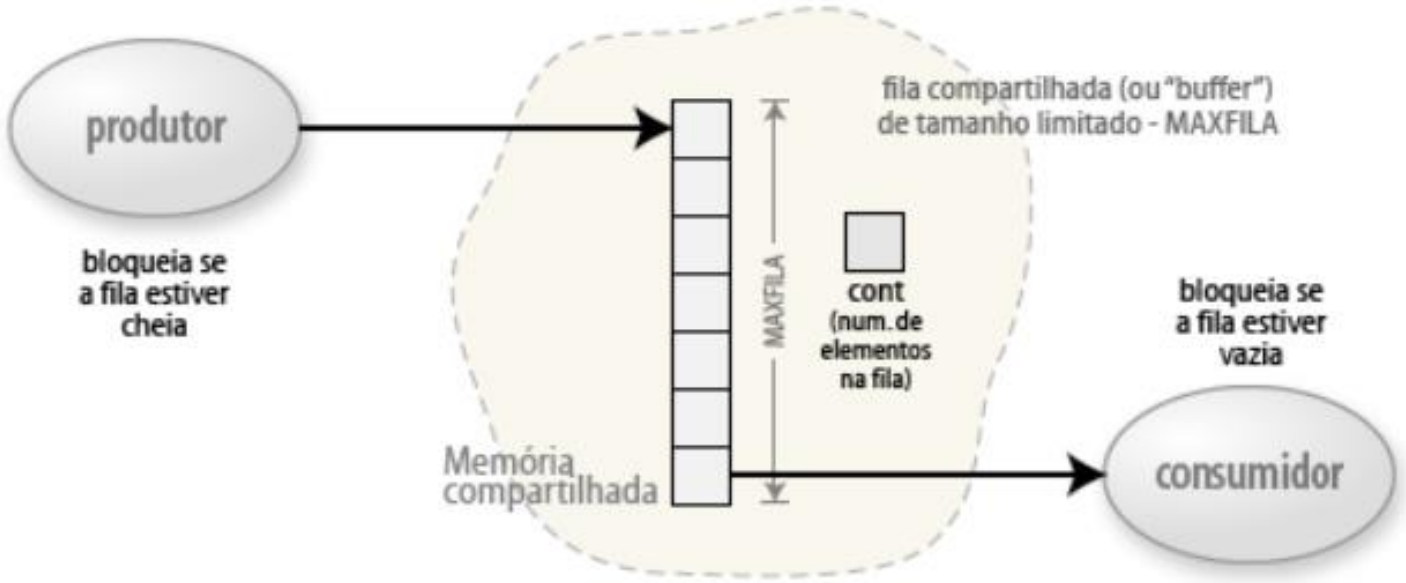
O produtor insere itens na fila (buffer) e o consumidor os retira da fila.

Problemas:

Para o produtor: deseja colocar itens na fila, mas ela está cheia.

Para o consumidor: deseja remover itens da fila, mas ela está vazia.

Solução: bloquear o produtor (sleep) quando a fila estiver cheia e bloquear o consumidor se a fila estiver vazia.



5.4.1 O Problema do Produtor/Consumidor com Fila (buffer) de tamanho limitado

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

void producer(void)
{
     int item;

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

Problema: perda do wakeup.

Solução: quando chegar um wakeup e não estiver dormindo, incrementa um contador e, da próxima vez que quiser dormir, verifica antes se o contador é maior que zero. Se for o caso, decrementa o contador sem dormir.

PEARSON Prentice Hall

5.5 Semáforos

Um semáforo é um contador de "wakeups".

down (sem)	Generalização do sleep.	`if (sem == 0) sleep();` `sem--;`

up (sem)	Generalização do wakeup.	`sem++;`
		`if (sem == 1)`
		`  wakeup(processo_dormindo_em_sem);`

Três semáforos (três situações para bloquear processo)

cheio: conta número de posições da fila (buffer) já preenchidos;

vazio: conta número de posições da fila ainda vazias;

mutex: para assegurar que mais de um processo não acesse o buffer ao mesmo tempo.

Os dois primeiros semáforos são necessários para a sincronização entre processos, e o terceiro para implementar exclusão mútua de execução.

5.5 Semáforos

```
#define N 100                                   /* numero de lugares no buffer */
typedef int semaphore;                          /* semaforos sao um tipo especial de int */
semaphore mutex = 1;                            /* controla o acesso a regiao critica */
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 e a constante 1 */
          item = produce_item( );                /* gera algo para por no buffer */
          down(&empty);                          /* decresce o contador empty */
          down(&mutex);                          /* entra na regiao critica */
          insert_item(item);                     /* poe novo item no buffer */
          up(&mutex);                            /* sai da regiao critica */
          up(&full);                             /* incrementa o contador de lugares preenchidos */
     }
}


void consumer(void)
{
     int item;

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


Exercício

Suponha que uma universidade queira mostrar o quão politicamente correta ela é, aplicando a doutrina "Separado mas igual é inerentemente desigual" da Suprema Corte dos EUA para o gênero, assim como a raça, terminando sua prática de longa data de banheiros segregados por gênero no campus.

No entanto, como uma concessão para a tradição, ela decreta que se uma mulher está em um banheiro, outras mulheres podem entrar, mas nenhum homem, e vice-versa. Um sinal com uma placa móvel na porta de cada banheiro indica em quais dos três estados possíveis ele se encontra atualmente:

- Vazio.
- Mulheres presentes.
- Homens presentes.

Em alguma linguagem de programação de que você goste, escreva as seguintes rotinas: ***woman_wants_to_enter***, ***man_wants_to_enter***, ***woman_leaves***, ***man_leaves***. Você pode usar as técnicas de sincronização e contadores que quiser.

PEARSON Prentice Hall