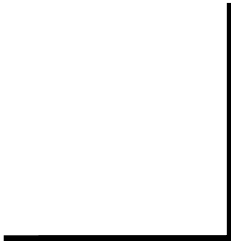


**Universidade Federal do Rio Grande do Norte  
Unidade Acadêmica Especializada em Ciências Agrárias  
Escola Agrícola de Jundiaí  
Curso de Análise e Desenvolvimento de Sistemas  
TAD0006 - Sistemas Operacionais - Turma 01**

# Processos - Comunicação

Antonino Feitosa  
antonino.feitosa@ufrn.br

Macaíba, maio de 2025



# Roteiro

---

- Condições de Corrida
- Regiões Críticas
- Exclusão Mútua com Espera Ocupada
- Dormir e Acordar
- Semáforos

# Introdução



# Introdução

- Processo precisam interagir, comunicar-se
- Comunicação entre processos (Interprocess Communication - IPC)
- Principais questões:
  - Como passar informações?
  - Como garantir o acesso à informação atualizada?
  - Como coordenar as ações dos processos?

# Introdução

- Todos os conceitos podem ser expandidos para threads.
- A passagem de informação é direta, pois as threads compartilham o espaço de endereçamento do processo.

# Condições de Corrida

---

# Condições de Corrida

- A comunicação pode ser implementada por uma área da memória compartilhada.
  - Estrutura de dados no núcleo.
  - Arquivo compartilhado.
- A comunicação ocorre por meio da leitura e escrita de informações nessas regiões.

# Condições de Corrida

---

- Exemplo: spool de impressão
  - Serviço (processo) que recebe as tarefas de impressão, as armazena temporariamente e as envia para a impressora na ordem correta.
    - Evita que o processo fique suspenso enquanto aguarda a impressão.
- Implementação por um diretório de impressão.
  - Diretório com posições fixas: cada posição armazena um arquivo a ser impresso.
  - Arquivo de controle com duas informações:
    - in: próxima posição livre para uma nova impressão
    - out: posição do próximo arquivo a ser impresso

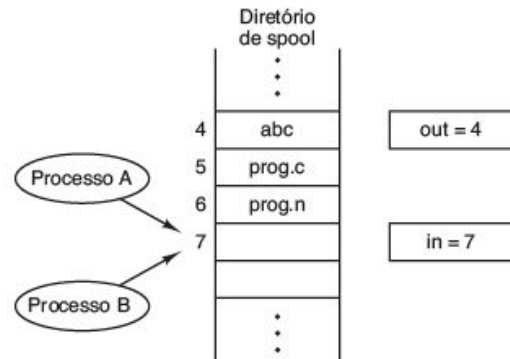


# Condições de Corrida

## 1. Considere a seguinte situação:

- A lê **in** e armazena o valor, 7, em uma variável local.
- Troca de contexto, executando o processo B.
- B lê **in** e armazena o valor, 7, em uma variável local.
- B atualiza o valor de **in** para 8.
- B escreve o nome do arquivo na posição 7.
- Troca de contexto, executando o processo A.
- A escreve o nome do arquivo na posição 7.
- A atualiza o valor de **in** para 8.

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



# Condições de Corrida

- O estado do spool é consistente.
- Ocorreu uma sobrescrita do valor armazenado por B.
  - A atrapalhou a execução de B.
  - O arquivo de B nunca será impresso.
- O resultado depende de quem executa precisamente e quando.
  - Comportamento imprevisível!

# Condições de Corrida

- **Condições de corrida:** situações em que dois ou mais processos estão lendo ou escrevendo alguns dados **compartilhados** e o resultado final depende de quem executa precisamente e quando.

# Regiões Críticas

---

# Regiões Críticas

---

- Como evitar as condições de corrida?
  - Controle da memória compartilhada!
- Precisamos garantir uma **exclusão mútua**: somente um processo está acessando a área compartilhada (variável ou arquivo).
- Sistema operacional: deve oferecer operações primitivas apropriadas para alcançar a exclusão mútua.

# Regiões Críticas

- **Região crítica** ou seção crítica: parte do programa onde a memória compartilhada é acessada.
- Condições de corrida só podem ocorrer se dois processos estiverem na região crítica ao mesmo.
  - Condição necessária, mas não suficiente

# Regiões Críticas

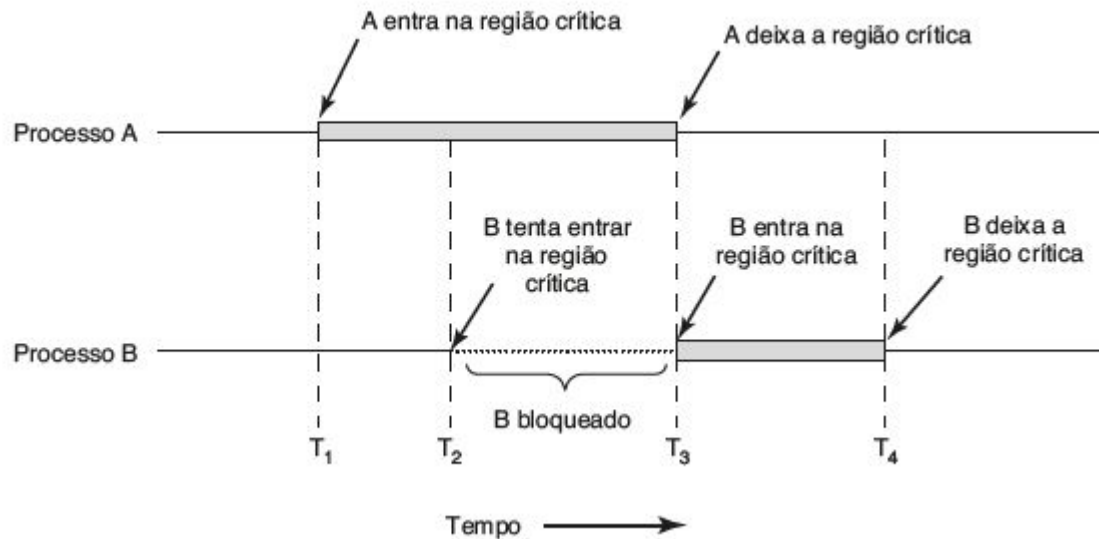
---

Solução para as condições de corrida de modo correto e eficiente precisam satisfazer os seguintes critérios:

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.

# Regiões Críticas

**FIGURA 2.22** Exclusão mútua usando regiões críticas.





# Regiões Críticas

1. O processo A entra na sua região crítica no tempo T1.
2. O processo B tenta entrar em sua região crítica no tempo T2, mas não consegue porque outro processo já está em sua região crítica e só permitimos um de cada vez.
3. B é temporariamente suspenso até o tempo T3, quando A deixa sua região crítica, permitindo que B entre de imediato.
4. B sai (em T4) e estamos de volta à situação original sem nenhum processo em suas regiões críticas.

# Exclusão Mútua com Espera Ocupada

---

# Exclusão Mútua com Espera Ocupada

- Existem vários mecanismos para garantir que somente um processo acessa a região crítica garantindo exclusão mútua.
  - Desabilitação de interrupções
  - Alternância explícita
  - Instrução TSL

# Desabilitação de Interrupções

---

# Desabilitação de Interrupções

- Considere um sistema que possui um único processador.
- Se as interrupções forem desabilitadas, não há troca de contexto.
  - As instruções são desabilitadas antes de entrar na região crítica e reativadas após a saída.
  - Processo precisam de controle sobre a ativação das interrupções!
    - Função do sistema operacional
    - Se o processo decidir que não deve reativar as interrupções?
    - E se tivermos múltiplos processadores?
      - Cenário cada vez mais comum.

# Alternância Explícita

---

# Alternância Explícita

- Consiste em usar o estado de uma variável para determinar o acesso à região crítica.
- **Espera ocupada:** testar continuamente uma variável até que algum valor apareça é chamado.
- **Trava giratória (spin lock):** Uma trava que usa a espera ocupada é chamada de .

**FIGURA 2.23** Uma solução proposta para o problema da região crítica. (a) Processo 0. (b) Processo 1. Em ambos os casos, certifique-se de observar os pontos e vírgulas concluindo os comandos while.

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

(a)

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

(b)

# Alternância Explícita

- O que ocorre se um processo for muito mais rápido que outro?
  - Por exemplo, se o processo 0 for mais rápido que o 1.
-



# Alternância Explícita

- Observe que ele viola a condição 3, pois no cenário que um processo é mais rápido que outro, o processo mais lento bloqueará o acesso à região crítica executando fora dela.
- Evita as condições de corrida, mas não é eficiente.

# Instrução TSL

---

# Instrução TSL

- Solução em hardware.
- Instrução TSL (test and set lock - teste e configure trava).
- Instrução que efetua a leitura e o armazenamento de valor ao mesmo tempo.
  - CPU bloqueia o acesso ao barramento de memória.
    - Impede que todos os processadores acessem a memória.
    - Diferente da desabilitação das interrupções que atuam em um único processador.

# Instrução TSL

---

- Implementação por duas funções:
- Entrar na região crítica:
  - Executa o TSL habilitando o lock
  - Se a leitura indicar que a região está livre, entra na região crítica
  - Caso contrário, efetua espera ocupada até TSL estar desabilitado
- Sair da região crítica:
  - Executa o TSL desabilitando o lock

# Instrução TSL

- O processo devem executar as duas funções de modo correto.
- Depende da cooperação dos processos.

# Observações

---

# Observações

---

- Essas soluções necessitam de espera ocupada.
  - Esperam para entrar na região crítica se ela não estiver liberada.
  - Desperdiça tempo do CPU.
  - Sofre do problema da inversão de prioridade.
    - Processo de alta prioridade permanecerá em espera ocupada enquanto um processo de baixa prioridade bloqueado na região crítica.
- Necessitamos de primitivas que bloqueiam os processos quando eles não são autorizados a entrar na região crítica.

# Dormir e Acordar

---



# Dormir e Acordar

- Primitivas sleep (dormir) e wakeup (acordar).
- Dormir: chamada do sistema que bloqueia o processo até que outro processo o desperte.
- Acordar: chamada do sistema que desperta outro processo.
  - O processo alvo deve ser passado como parâmetro.
  - O processo alvo acorda se estiver dormindo.
    - Nada acontece se ele estiver acordado.

# Dormir e Acordar: Produtor-Consumidor

- Dois processos compartilham de uma região de memória de tamanho fixo comum.
- Um deles, o produtor, insere informações na memória, e o outro, o consumidor, as retira dele.

# Dormir e Acordar: Produtor-Consumidor

```
#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);  
    }
```

```
}
```

```
/* numero de lugares no buffer */  
/* numero de itens no buffer */
```

```
/* repita para sempre */  
/* gera o proximo item */  
/* se o buffer estiver cheio, va dormir */  
/* ponha um item no buffer */  
/* incremente o contador de itens no buffer */  
/* o buffer estava vazio? */
```

# Dormir e Acordar: Produtor-Consumidor

```
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);
    }
}
```

*/\* repita para sempre \*/*  
*/\* se o buffer estiver cheio, va dormir \*/*  
*/\* retire o item do buffer \*/*  
*/\* decresca de um contador de itens no buffer \*/*  
*/\* o buffer estava cheio? \*/*  
*/\* imprima o item \*/*

# Dormir e Acordar: Produtor-Consumidor

- Quando a memória está cheia e o produtor quer inserir um item, ele deve dormir até que exista espaço
  - O espaço é liberado pelo consumo de um item.
- Quando a memória está vazia e o consumidor quer remover um item, ele deve dormir até exista um item.
  - O item é inserido pelo produtor.
- A variável count indica se a memória está cheia ou vazia.

# Dormir e Acordar: Produtor-Consumidor

---

- A solução apresentada apresenta uma condição de corrida ao acessar a variável count.
- Exemplo:
  - Considere a memória vazia (count == 0).
  - O consumidor efetua a leitura de count com o valor 0.
  - Troca de contexto para o produtor.
  - O produtor insere um valor e efetua a chamada de acordar o consumidor.
    - Porém, o consumidor ainda não efetuou a chamada de sleep, está acordado e chamada de acordar do produtor é desperdiçada.
  - O consumidor, com o valor antigo de count em 0, irá dormir.
    - Quando a memória encher, o produtor irá dormir também.

# Semáforos



# Semáforos

---

- Controlam a quantidade de sinais de acordar por uma variável inteira.
- As operações são renomeadas para down (dormir) e up (acordar).
- As verificações e alterações na variável do semáforo são efetuadas de modo atômico.
  - Evita condições de corrida.



# Semáforos

- A operação down verificar o estado do semáforo:
  - Valor  $> 0$ : decrementa o valor (consome um sinal de acordar) e continua,
  - Valor  $== 0$ : o processo é colocado para dormir sem completar o down para o momento.
- A operação de up incrementa o valor do semáforo.
  - Se um processo estiver dormindo, ele é acordado para completar a operação de down.

# Semáforos: Produtor-Consumidor

---

**FIGURA 2.28** O problema do produtor-consumidor usando semáforos.

```
#define N 100
typedef int semaphore;
semaphore mutex = 1;
semaphore empty = N;
semaphore full = 0;

void producer(void)
{
    int item;

    while (TRUE) {
        item = produce_item();
        down(&empty);
        down(&mutex);
        insert_item(item);
        up(&mutex);
        up(&full);
    }
}
```

```
/* numero de lugares no buffer */
/* semaforos sao um tipo especial de int */
/* controla o acesso a regioao critica */
/* conta os lugares vazios no buffer */
/* conta os lugares preenchidos no buffer */

/* TRUE e a constante 1 */
/* gera algo para por no buffer */
/* decresce o contador empty */
/* entra na regioao critica */
/* poe novo item no buffer */
/* sai da regioao critica */
/* incrementa o contador de lugares preenchidos */
```

# Semáforos: Produtor-Consumidor

```
void consumer(void)
{
    int item;

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

*/\* laço infinito \*/*  
*/\* decresce o contador full \*/*  
*/\* entra na regioao critica \*/*  
*/\* pega item do buffer \*/*  
*/\* sai da regioao critica \*/*  
*/\* incrementa o contador de lugares vazios \*/*  
*/\* faz algo com o item \*/*

# Semáforos: Produtor-Consumidor

---

- Observações:
  - O semáforo mutex controla o acesso à região crítica.
    - Semáforos binários são chamados de mutex.
    - Servem somente para implementação de exclusão mútua.
  - Os semáforos full e empty são responsáveis pela sincronização.
    - Determinam quando determinadas sequências de instruções ocorrem ou não.

# Primitivas de Sincronização

- Semáforos são primitivas de sincronização que evitam condições de corrida, desde que utilizadas corretamente.
  - A alteração da ordem dos ups e downs pode gerar estados inconsistentes em que todos os processos dormem.
- Primitivas de alto nível:
  - Monitores
  - Troca de Mensagens
  - Barreiras

# Resumo

---

# Resumo

---

- Condições de Corrida
- Regiões Críticas
- Exclusão Mútua com Espera Ocupada
- Dormir e Acordar
- Semáforos

# Dúvidas?

