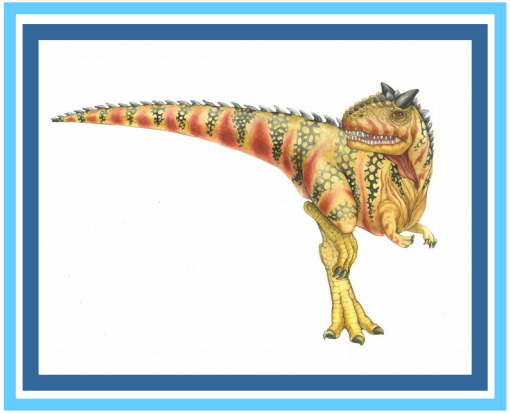
Capítulo 5: Sincronização entre Processos



Introdução

- Processos que executam concorrentemente podem ser interrompidos a qualquer momento.
  - Dados compartilhados podem se tornar inconsistentes.
- Para que não haja inconsistência dos dados, mecanismos de sincronização devem ser implementados.
- Exemplo: **printx.c**

Produtor e Consumidor

Produtor

```
while (true) {
/* produce an item*/
  while (counter==BUFFER_SIZE)
    ; /* do nothing */
  buffer[in] = next_produced;
  in = (in + 1)%BUFFER_SIZE;
  counter++;
}
```

Consumidor

```
while (true) {
    while (counter == 0)
        ; /* do nothing */
    next_consumed = buffer[out];
    out = (out + 1) % BUFFER_SIZE;
        counter--;
    /* consume the item*/
}
```

in → próxima posição vazia
out → posição do elemento que será consumido.
counter → número de elementos no buffer.

Condição de corrida

- **`counter++`** could be implemented as

```
register1 = counter
register1 = register1 + 1
counter = register1
```

- 0 **`counter--`** pode ser implementado da seguinte maneira:

```
register2 = counter
register2 = register2 - 1
counter = register2
```

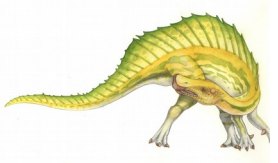
- Considere a seguinte execução com “count = 5” :

S0: produtor executa `register1 = counter` {register1 = 5}
S1: produtor executa `register1 = register1 + 1` {register1 = 6}
S2: consumidor executa `register2 = counter` {register2 = 5}
S3: consumidor executa `register2 = register2 – 1` {register2 = 4}
S4: produtor executa `counter = register1` {counter = 6 }
S5: consumidor executa `counter = register2` {counter = 4}

Seção crítica e condição de corrida

```
do {
        entry section
                critical section
        exit section
                remainder section
} while (true);
```

- Somente um processo por vez pode estar na seção crítica.
- Cada processo deve pedir permissão para entrar na seção crítica (**entry section**) e deve informar o término (na seção **exit section**).
- **Condição de corrida (race condition):** é uma situação onde vários processo acessam e manipulam o mesmo dado concorrentemente e o resultado da execução depende da ordem na qual os acessos acontecem.



Soluções

Condições necessárias para evitar condições de corridas:

- Dois processos não podem estar simultaneamente dentro de suas regiões críticas.
- Nada pode ser afirmado sobre a velocidade ou sobre o número de CPUs.
- Nenhum processo sendo executado fora de sua região crítica pode bloquear outros processos.
- Nenhum processo deve esperar eternamente para entrar em sua região crítica.

Tanenbaum p.71

Exclusão mútua

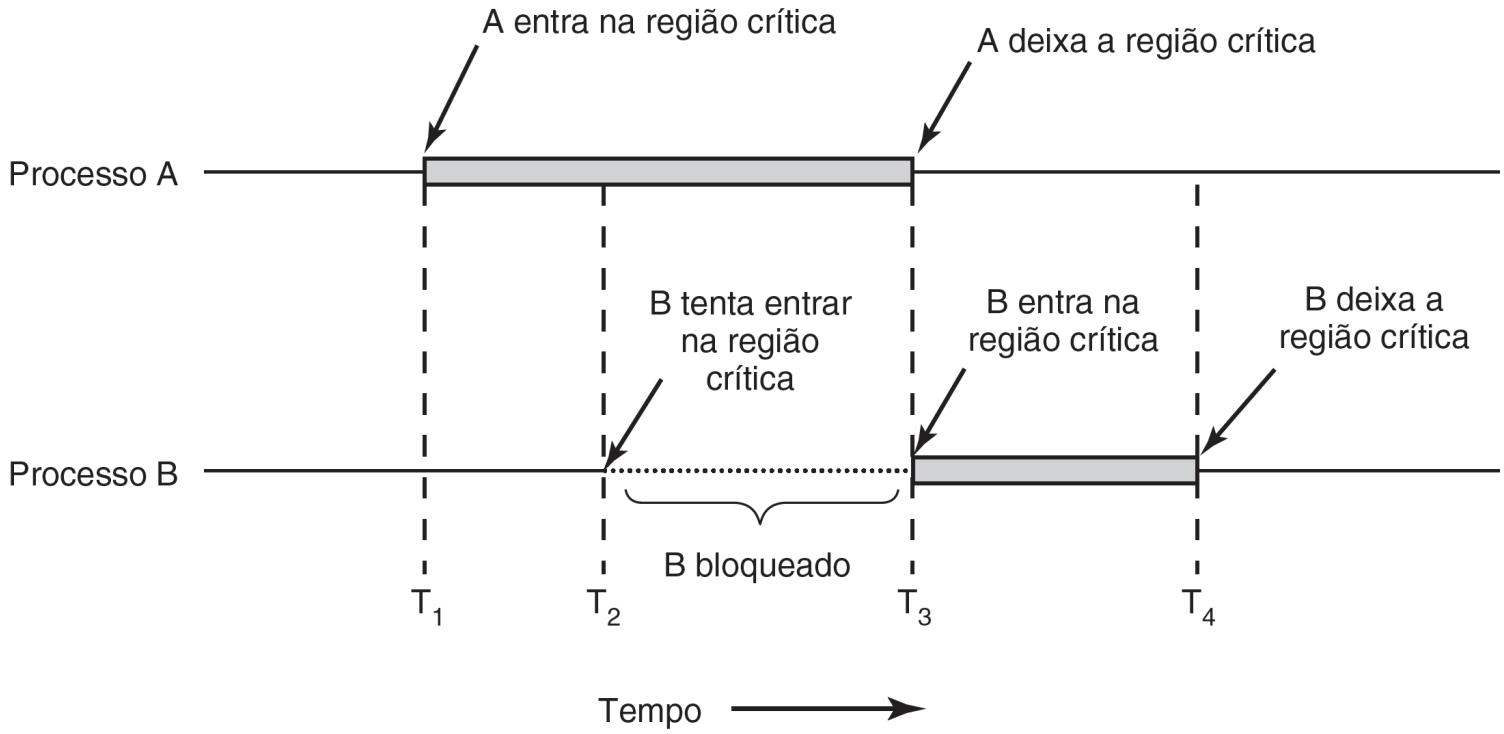


Figura 2.17 Exclusão mútua usando regiões críticas.

Tanenbaum p.72

Seção crítica no OS

O que o problema da seção crítica tem haver com o conteúdo de Sistemas Operacionais?

Seção crítica no OS

O que o problema da seção crítica tem haver com o conteúdo de Sistemas Operacionais?

- **Kernel preemptivo** – permite que processos sejam interrompidos quando estão executando no modo kernel
  - Deve ser cuidadosamente projetado para garantir que dados compartilhados do Kernel estejam livres de condições de corrida.
- **Kernel não-preemptivo** – não permitem que processos sejam interrompidos quando estão executando no modo kernel.
  - Livres de condições de corrida.

Seção crítica no OS

O que o problema da seção crítica tem haver com o conteúdo de Sistemas Operacionais?

- **Kernel não-preemptivo**
  - Suponha que:
    - um processo A controla a execução de um áudio e precisa entrar no modo kernel a cada 2 ms;
    - um outro processo B gasta 5ms para executar uma chamada de sistema.
    - Então o áudio terá que ser interrompido pois não há preempção. Isso ocorre mesmo que eles não possuam dados compartilhados.

- O que ocorrerá se o Kernel for preemptivo?

- O que poderá ocorrer se o Kernel for preemptivo e os processos acessarem dados compartilhados?

Seção crítica no OS

O que o problema da seção crítica tem haver com o conteúdo de Sistemas Operacionais?

- Em um Kernel **não-preemptivo** com processos acessando dados compartilhados, um processo de mais baixa prioridade poderá bloquear outro processo de mais alta prioridade?

- Em um Kernel **preemptivo** com processos acessando dados compartilhados, um processo de mais baixa prioridade poderá bloquear outro processo de mais alta prioridade?

Solução para evitar condições de corrida

Uma solução simples para evitar condições de corrida é **desabilitar as interrupções.**

- Inibe as trocas de contextos.
- **Não existe preempção quando está na seção crítica.**
- Se a tarefa entrar em um laço infinito, todo sistema trava.
- Com as interrupções desabilitas, não é possível realizar entrada/saída.
  - » Se o buffer de uma placa de rede estiver cheio, novos dados de entrada podem ser perdidos caso não sejam tratados pelo Kernel.

- Devido a esses problemas, a inibição de interrupção nunca é realizada pelos processos de usuário.

Espera ocupada: solução “óbvia”

Uma primeira classe de soluções para o problema da exclusão mútua no acesso a seções críticas consiste em testar continuamente uma condição que indica se a seção desejada está livre ou ocupada.

- Solução óbvia:

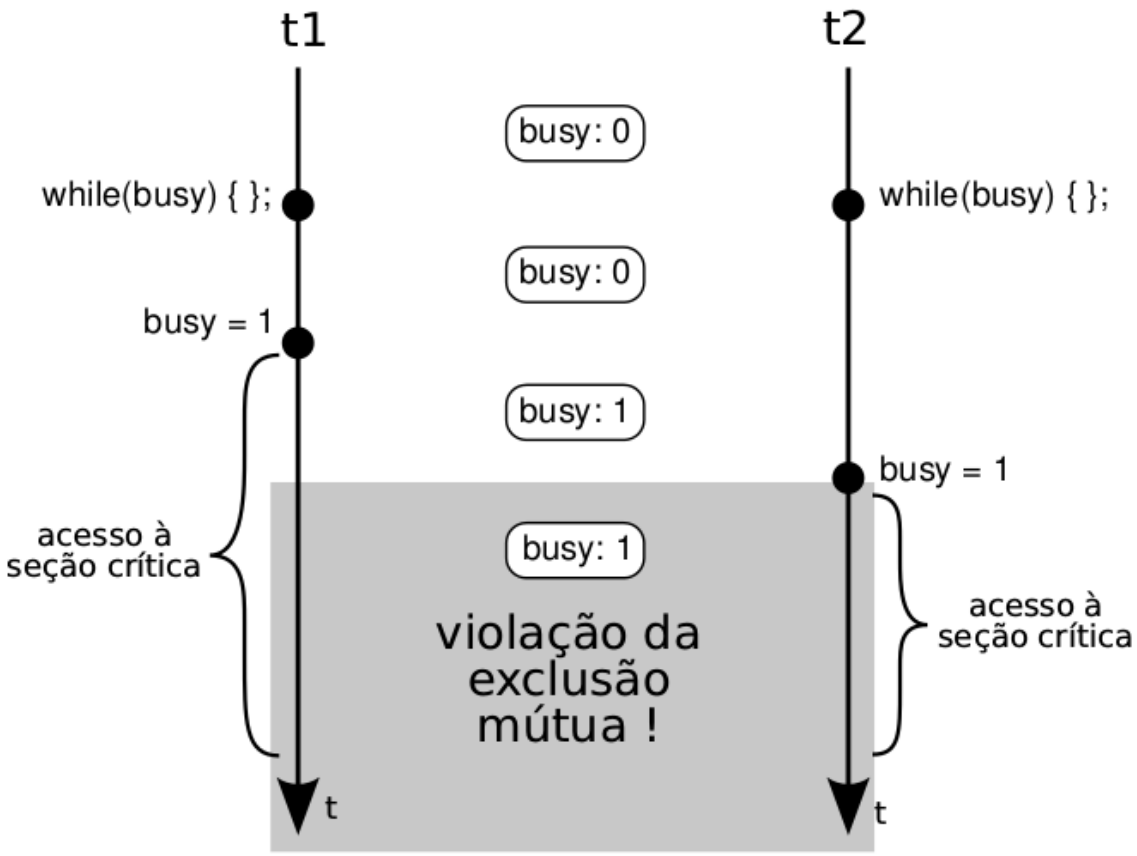
```
int busy = 0 ;          // a seção está inicialmente livre

void enter (int task)
{
   while (busy) ;       // espera enquanto a seção estiver ocupada
   busy = 1 ;           // marca a seção como ocupada
}

void leave (int task)
{
   busy = 0 ;           // libera a seção (marca como livre)
}
```

Esta solução garante que apenas uma tarefa executará a seção crítica?

Espera ocupada: solução “óbvia”



Referência: Capítulo 4 do livro do Mazieiro.

Espera ocupada: alternância de uso

```
int turn = 0 ;
int num_tasks ;

void enter (int task)        // task vale 0, 1, ..., num_tasks-1
{
   while (turn != task) ;    // a tarefa espera seu turno
}

void leave (int task)
{
   if (turn < num_tasks-1)   // o turno é da próxima tarefa
      turn ++ ;
   else
      turn = 0 ;             // volta à primeira tarefa
}
```

Esta solução garante a exclusão mútua entre as tarefas?

Espera ocupada: alternância de uso

```
int turn = 0 ;
int num_tasks ;

void enter (int task)        // task vale 0, 1, ..., num_tasks-1
{
   while (turn != task) ;    // a tarefa espera seu turno
}

void leave (int task)
{
   if (turn < num_tasks-1)   // o turno é da próxima tarefa
      turn ++ ;
   else
      turn = 0 ;             // volta à primeira tarefa
}
```

Uma tarefa que não está acessando a seção crítica pode bloquear outras tarefas que desejam o acesso?

Espera ocupada: solução de Perterson

```
do {
    flag[1] = true;
    turn = 2;
    while (flag[2] && turn = = 2);
    critical section
    flag[1] = false;
    remainder section
} while (true);
```

```
do {
    flag[2] = true;
    turn = 1;
    while (flag[1] && turn = = 1);
    critical section
    flag[2] = false;
    remainder section
} while (true);
```

- flag[i] → indica que o processo Pi está pronto para entrar na seção crítica.
- turn → indica de quem é a vez de entrar na seção crítica.
  - Se ambos processos tentam entrar ao mesmo tempo (turn =1 and turn =2), apenas uma das atribuições permanecerá.
- Restrito a 2 processos que alternam a execução de suas seções críticas.

Espera ocupada: solução de Perterson

```
do {
   flag[1] = true;
   turn = 2;
   while (flag[2] && turn = = 2);
   critical section
   flag[1] = false;
   remainder section
} while (true);
```

```
do {
   flag[2] = true;
   turn = 1;
   while (flag[1] && turn = = 1);
   critical section
   flag[2] = false;
   remainder section
} while (true);
```

- O acesso a seção crítica ocorrerá de maneira alternada?
- Um processo P1 que não está acessando a seção crítica pode bloquear um outro processo P2 que deseja o acesso?

Espera ocupada: instrução test_and_set

```
int busy = 0 ;

void enter (int task)
{
   while (busy) ;
   busy = 1 ;
}

void leave (int task)
{
   busy = 0 ;
}
```

O uso de uma variável *busy* para controlar a entrada em uma seção crítica é uma ideia interessante. Infelizmente esta solução não funciona porque **o teste da variável *busy* e seu ajuste são feitos em momentos distintos do código**, permitindo condições de corrida.

Espera ocupada: instrução test_and_set

```
int busy = 0 ;

void enter (int task)
{
   while (busy) ;
   busy = 1 ;
}

void leave (int task)
{
   busy = 0 ;
}
```

```
int lock = 0 ;

void enter (int *lock)
{
   while ( TSL (*lock) ) ;
}

void leave (int *lock)
{
  (*lock) = 0 ;
}
```

O uso de uma variável *busy* para controlar a entrada em uma seção crítica é uma ideia interessante. Infelizmente esta solução não funciona porque **o teste da variável *busy* e seu ajuste são feitos em momentos distintos do código**, permitindo condições de corrida.

Espera ocupada: instrução test_and_set

- test_and_set é executado atomicamente. - variável *lock* inicializada como FALSO

Quando o recurso está liberado, a execução do TSL retorna “false” e já aloca o recurso (*target=true).

```
boolean TSL (boolean *target){
    boolean rv = *target;
      *target = TRUE;
      return rv:
}
```

Sempre bloqueia o recurso

```
do {
     while (TSL(&lock))
      ; /* do nothing */
      /* critical section */
      lock = false;
      /* remainder section */
} while (true);
```

Sai do “loop” quando algum processo atribui “false” ao “lock”

Espera ocupada: instrução compare_and_swap

Se “lock==0” (recurso liberado) então faz lock=1 (recurso alocado)

```
int compare_and_swap(int *value, int expected, int new_value) {
    int temp = *value;
    if (*value == expected)
        *value = new_value;
    return temp;
}
```

```
do {
    while (compare_and_swap(&lock, 0, 1) != 0)
       ; /* do nothing */
   /* critical section */
   lock = 0;
   /* remainder section */
} while (true);
```

Espera ocupada: instrução compare_and_swap

As instruções *TSL* e *compare_and_swap* garatem a exclusão mútua?

Com as instruções *TSL* e *compare_and_swap*, haverá alternância entre os processos que acessam a região crítica?

Suponha que quando um processo A entra em sua seção crítica, ele gasta 1 ms para sair. Ao utilizar a instrução *TSL* para garantir exclusão mútua, pode acontecer deste processo bloquear eternamente um outro processo B de entrar na seção crítica?

Analise a seguinte solução para exclusão mútua com espera ocupada.

```
do {
 waiting[i] = true; //Indica que o processo i está esperando
 key = true; //Indica se o recurso está ocupado ou não
 while (waiting[i] && key)
   key = test_and_set(&lock); //Sai do “while” quando recurso livre
 waiting[i] = false;
 /* Executa a seção crítica */
 /*Procura na lista pelo o próximo processo que está esperando*/
 j = (i + 1) % n; //Inicia a busca no próximo
 while ((j != i) && !waiting[j]) //Enquanto não encontrar
    j = (j + 1) % n; //Passa para o próximo
 if (j == i) //Se não há processos esperando
   lock = false; //Libera o recurso
 else
   waiting[j] = false; //Libera o processo “j” do “while” na
linha 4.
  /*executa o restante da seção */
} while (true);
```

Instruções TSL e compare_and_swap

A solução anterior resolve qual problema?

Espera ocupada

Apesar das soluções para o problema de acesso à seção crítica usando espera ocupada garantirem a exclusão mútua, elas podem apresentar alguns problemas:

- Ineficiência : as tarefas que aguardam o acesso a uma seção crítica ficam testando continuamente uma condição, consumindo tempo de processador sem necessidade. O procedimento adequado seria suspender essas tarefas até que a seção crítica solicitada seja liberada.

- Injustiça : não há garantia de ordem no acesso à seção crítica; dependendo da duração de *quantum* e da política de escalonamento, uma tarefa pode entrar e sair da seção crítica várias vezes, antes que outras tarefas consigam acessá-la.

Espera ocupada

Analise:

Em um sistema com 2 processadores, 2 processos estão acessando o mesmo recurso e utilizando espera ocupada para evitar condição de corrida. A espera ocupada pode ser uma boa estratégia para implementar a exclusão mútua caso o recurso seja utililizado pelos processos por um intervalo curto de tempo.

Na situação descrita acima, o uso de espera ocupada seria uma boa solução caso houvesse apenas 1 processador?

Espera ocupada

Em Sistema Operacionais as implementações de espera ocupada são chamadas de *spinlocks*.

As soluções abaixo podem melhorar o desempenho do sistema computacional?

- Utilizar s*pinlocks* apenas quando o recurso desejado está bloqueado por uma *thread* em execução, caso contrário, o processo solicitante é colocado em espera.

- Após uma solicitação de um recurso que está bloqueado, a *thread* solicitante pode ser colocada na fila de prontos novamente (ou seja, ela não ficará nem em *spinlock* e nem em espera). Dessa forma, pode acontecer do recurso ser liberado em uma próxima tentativa de acesso.

Continuação na próxima aula