Universidade Tecnológica Federal do Paraná – UTFPR
Coordenação de Ciência da Computação - COCIC
Bacharelado em Ciência da Computação

BCC34G – Sistemas Operacionais

Prof. Rogério A. Gonçalves

rogerioag@utfpr.edu.br

Aula 014

- Sincronismo entre Processos: 1ª. parte

Sincronismo entre Processos

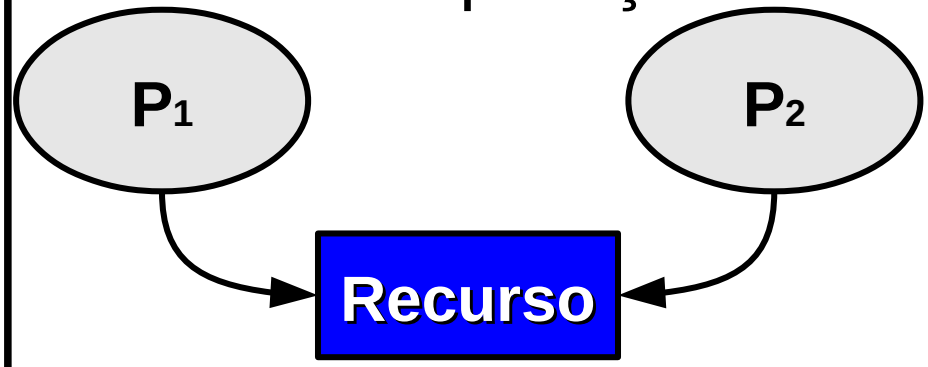
- Conceitos
- O problema da seção/região crítica
- Solução de Dekker
- Solução de Peterson
- Hardware de sincronismo
- Semáforos
- Monitores

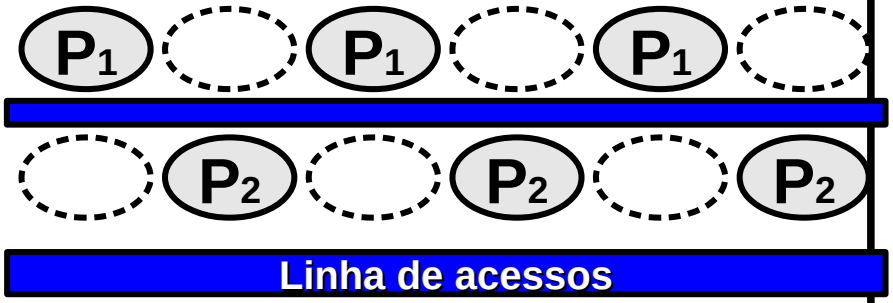
Comunicação de processos

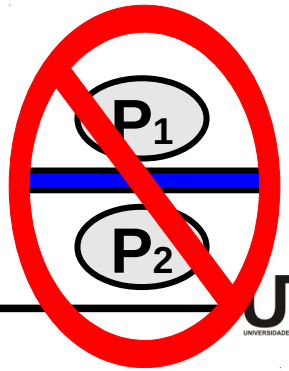
- Processos precisam se comunicar;
- Processos competem por recursos;
- Três aspectos importantes:
  - Como um processo passa informação para outro processo?
  - Como garantir que processos não invadam espaços uns dos outros?
  - Dependência entre processos: qual é a sequência adequada?

Conceitos

- Acesso concorrente aos dados compartilhados
  - pode resultar em inconsistência de dados
- Manutenção da consistência
  - requer mecanismos para garantir a execução ordenada dos processos em cooperação



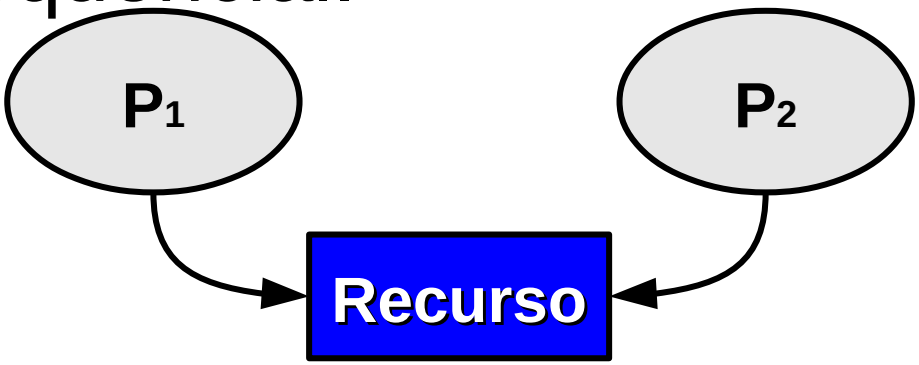


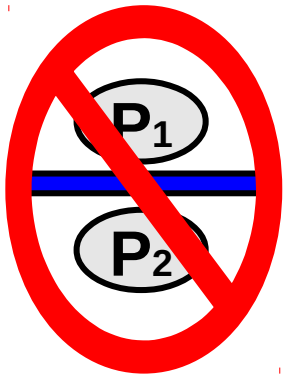


Condição de corrida

Situação em que vários processos acessam e manipulam os mesmos dados concorrentemente e o resultado da execução depende da ordem específica em que ocorre o acesso.

O acesso ao recurso/dado deve ser sequencial.





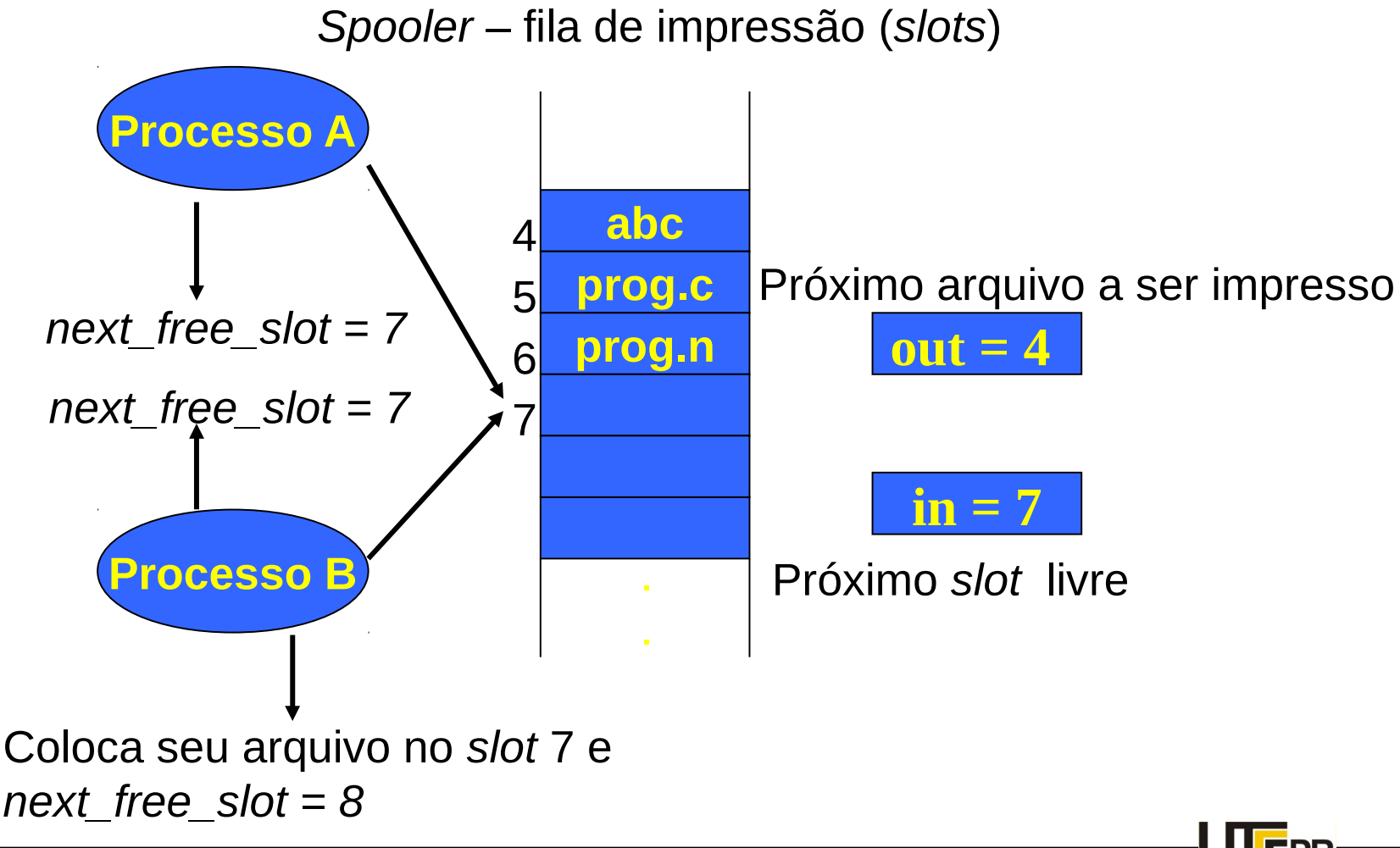
Condição de corrida

- Processos acessam recursos compartilhados concorrentemente
    - Recursos: memória, arquivos, impressoras, discos, variáveis;
    - Exemplo: Impressão de um arquivo por dois processos “printer spooler”.

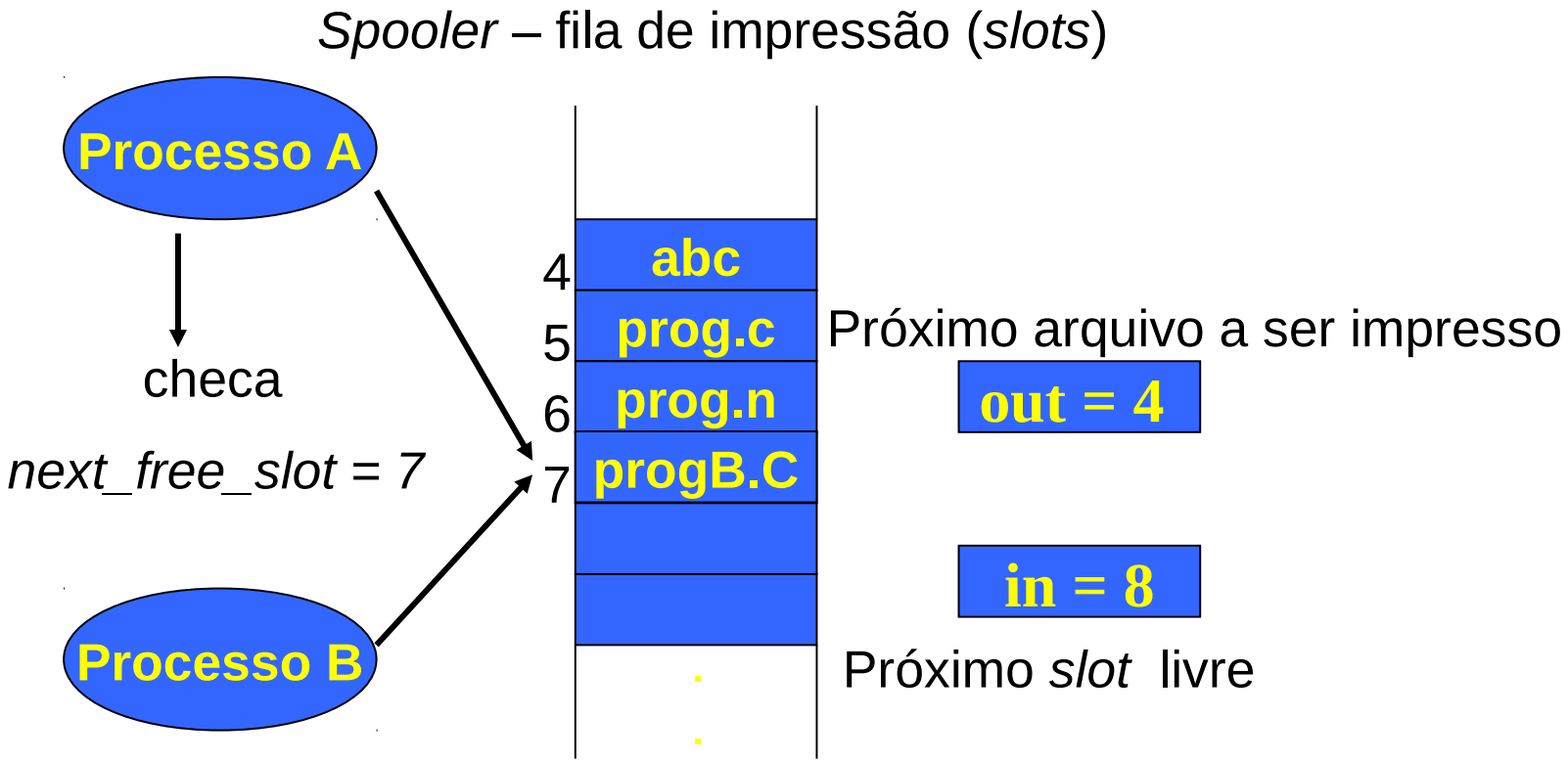
Problema

- Problema: dois processos/*threads* acessando dados simultaneamente
- Os dados podem ficar inconsistentes.
- O chaveamento de contexto pode ocorrer a qualquer momento. Por exemplo, antes de o *thread* terminar de mudar um valor.

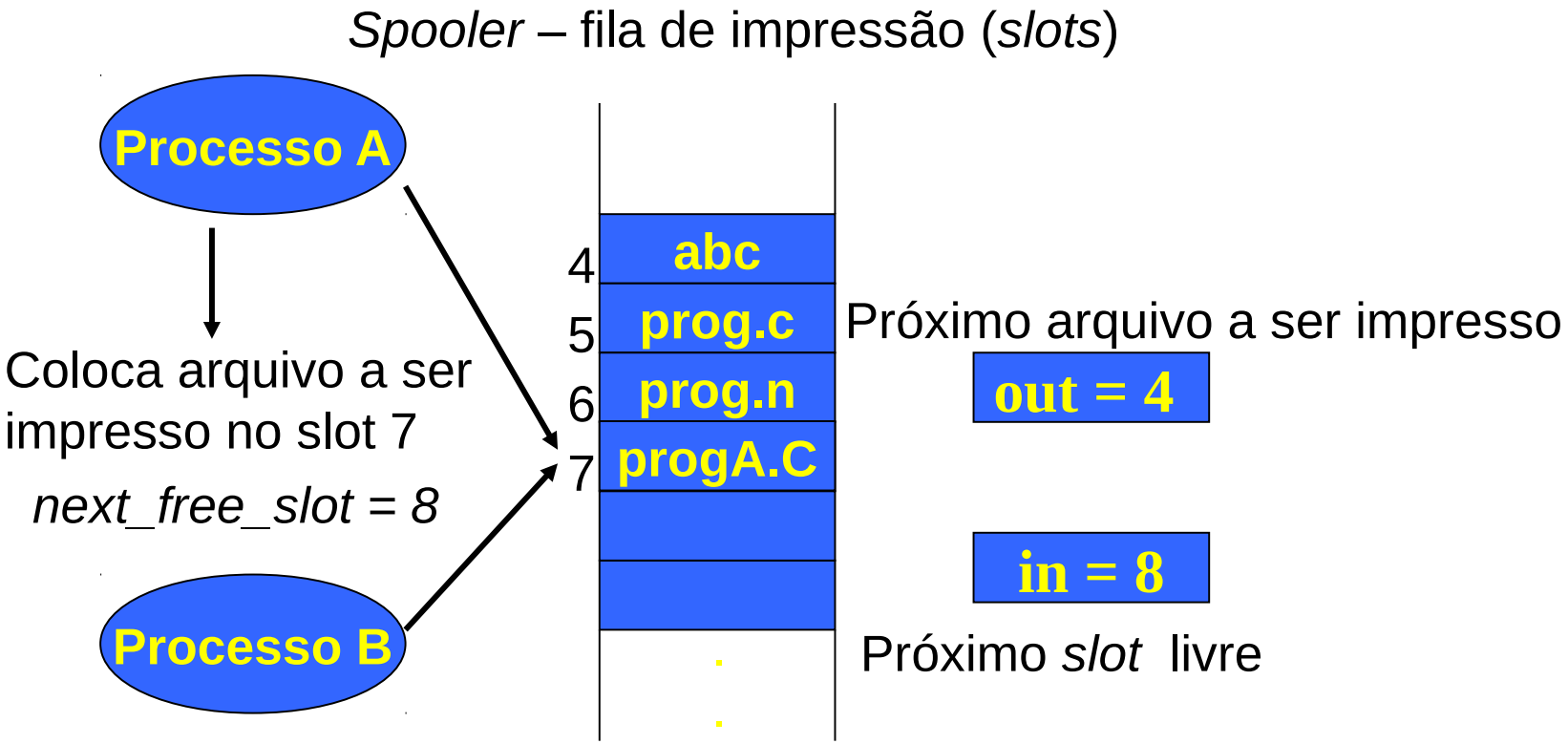
Condição de corrida



Condição de corrida



Condição de corrida



Processo B nunca receberá sua impressão...

Condição de corrida

- count++ poderia ser implementado como:
  reg_1 = count
  reg_1 = reg_1 + 1
  count = reg_1
- count-- poderia ser implementado como:
  reg_2 = count
  reg_2 = reg_2 - 1
  count = reg_2
- Considere esta execução intercalada:

  Inicialmente: “count = 5”

  t0: processoA executa reg_1 = count {reg_1 = 5}
  t1: processoA executa reg_1 = reg_1 + 1 {reg_1 = 6}
  t2: processoB executa reg_2 = count {reg_2 = 5}
  t3: processoB executa reg_2 = reg_2 - 1 {reg_2 = 4}
  t4: processoA executa count = reg_1 {count = 6 }
  t5: processoB executa count = reg_2 {count = 4}

Regiões críticas

- Como solucionar problemas de condição de corrida?
  - Proibir que mais de um processo leia ou escreva em recursos compartilhados concorrentemente (ao “mesmo tempo”)

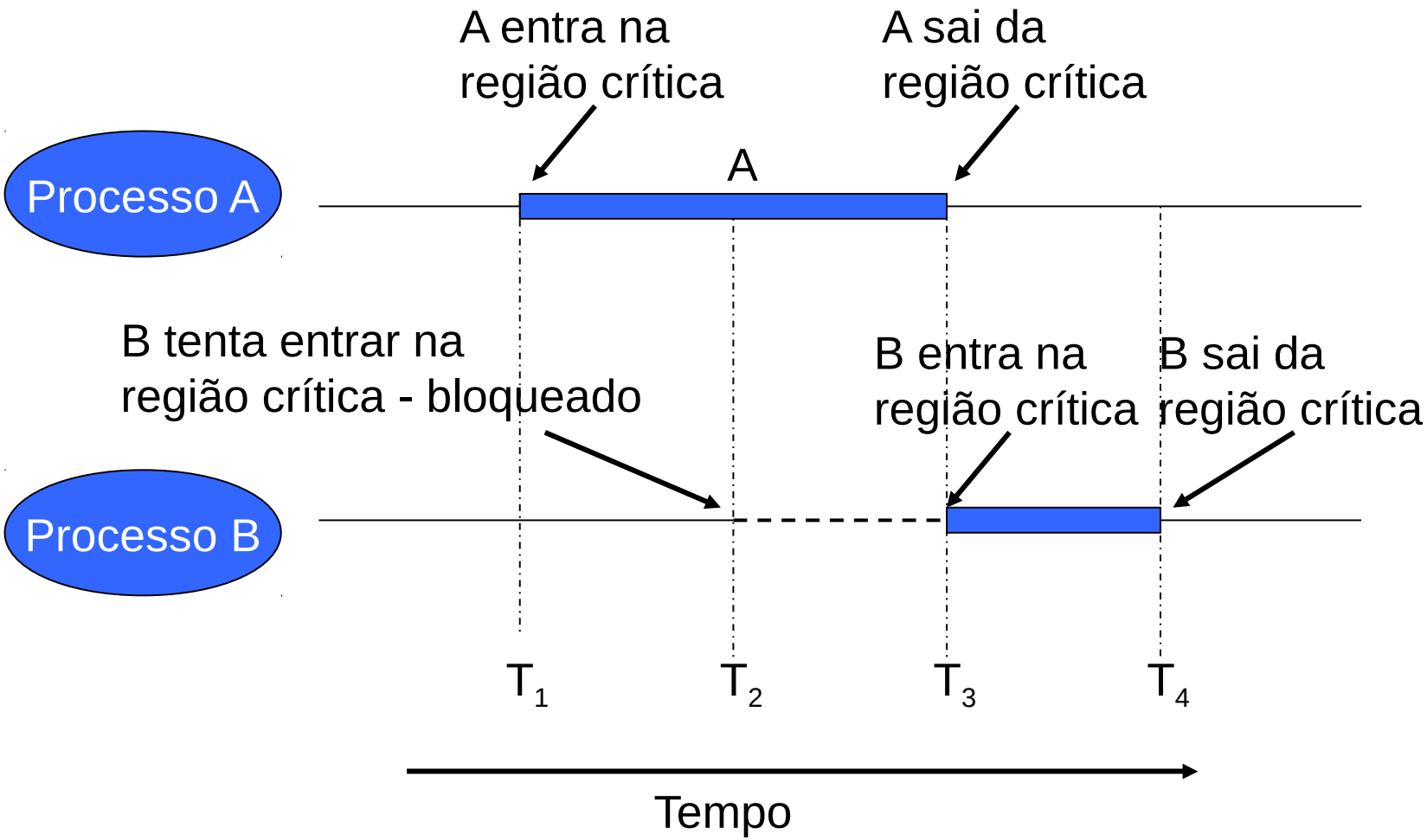
    Recursos compartilhados → regiões críticas

  - **Exclusão mútua**: garantir que um processo não terá acesso à uma região crítica quando outro processo está utilizando essa região;

Regiões críticas

- Quatro condições para uma boa solução:
  1) Dois processos não podem estar simultaneamente em regiões críticas;
  2) Nenhuma restrição deve ser feita com relação à CPU;
  3) Processos que não estão em regiões críticas não podem bloquear outros processos que desejam utilizar regiões críticas;
  4) Processos não podem esperar para sempre para acessarem regiões críticas;

Exclusão mútua



Solução

- Esses dados precisam ser acessados de uma maneira mutuamente exclusiva.
- Deve-se permitir o acesso a apenas um *thread* por vez.
- Outros *threads* devem esperar até que o recurso seja desbloqueado.
- O acesso é serializado.
- Esse processo precisa ser gerenciado de modo que o tempo de espera não seja exagerado.

Problema de seção crítica

- ***Condição de corrida*** – Quando há acesso simultâneo aos dados compartilhados e o resultado final depende da ordem de execução.
- ***Seção crítica*** – Seção do código onde os dados compartilhados são acessados.
- ***Seção de entrada*** - Código que solicita permissão para entrar em sua seção crítica.
- ***Seção de saída*** – Código executado após a saída da seção crítica.

Estrutura de um processo típico

```
while (true) {
```

Seção de Entrada

Seção Crítica

Seção de Saída

Seção Restante

```
}
```

Primitivas de Exclusão Mútua

- Indicam quando dados críticos estão para ser acessados.
  - Esses mecanismos em geral são oferecidos por linguagens de programação ou bibliotecas.
- Delimitação do início e fim de uma seção crítica
  - enterMutualExclusion
  - exitMutualExclusion

Exclusão mútua

- Soluções:
  1) **Espera Ocupada/Espera Ativa**
  2) Primitivas *Sleep* / *Wakeup*
  3) Semáforos
  4) Monitores
  5) Passagem de Mensagem

Exclusão mútua – espera ocupada

- Soluções de exclusão mútua utilizando espera ocupada:
  - Desabilitar interrupções;
  - Variáveis de travamento (*Locks)*;
  - Alternância Estrita
    - Solução de Dekker, Solução de Peterson e Instrução TSL;

Exclusão mútua – espera ocupada

- ***Desabilitar 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;
    - Viola condição 2;
  - Não é uma solução segura, pois um processo pode não habilitar novamente suas interrupções e não ser finalizado;
    - Viola condição 4;

Exclusão mútua – espera ocupada

- ***Variáveis Lock:***
  - O processo que deseja utilizar uma região crítica atribui um valor a uma variável chamada *lock*;
  - Se a variável está com valor 0 (zero) significa que nenhum processo está na região crítica; Se a variável está com valor 1 (um) significa que existe um processo na região crítica;
  - Apresenta o mesmo problema do exemplo do *spooler de impressão*;

Exclusão mútua – espera ocupada

- ***Variáveis Lock - 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;
    - Viola condição 1;

Exclusão mútua – espera ocupada

```
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

Seção crítica usando locks

```
while (true) {
   acquire lock
      Seção crítica
   release lock
      Seção restante
}
```

Obtém o *lock*

Libera o *lock*

Exclusão mútua – espera ocupada

- Solução de Peterson e Instrução TSL (*Test and Set Lock*):
    - Uma variável (ou programa) é utilizada para bloquear a entrada de um processo na região crítica quando um outro processo está na região;
    - Essa variável é compartilhada pelos processos que concorrem pelo uso da região crítica;
    - Ambas as soluções possuem fragmentos de programas que controlam a entrada e a saída da região crítica;

Solução de Dekker

- Combina a ideia de alternância com variáveis de travas e de alerta.
- Primeira solução de software para o problema de exclusão mútua que não exigisse uma alternância estrita.

Algoritmo de Dekker

- Solução apropriada (4 versões).
- Usa a noção de *threads* favorecidos para determinar a entrada em seções críticas.
- Resolve o conflito sobre o *thread* que deveria ser executado em primeiro lugar.
- Todo *thread* desconfigura temporariamente o *flag* de solicitação de seção crítica.
- O status favorecido alterna entre threads.
- Garante exclusão mútua.
- Evita problemas anteriores de deadlock e adiamento indefinido.

Algoritmo de Dekker

```
1       Sistema:
2
3       int favoredThread = 1;
4       boolean t1WantsToEnter = false;
5       boolean t2WantsToEnter = false;
6
7       startThreads( ); // inicializa e lança ambos os threads
8
9       Thread T1:
10
11      void main( )
12      {
13          while ( !done )
14          {
15              t1WantsToEnter = true;
16
```

Algoritmo de Dekker

```
17          while ( t2WantsToEnter )
18          {
19              if ( favoredThread == 2 )
20              {
21                  t1WantsToEnter = false;
22                  while ( favoredThread == 2 ); // espera ociosa
23                  t1WantsToEnter = verdadeiro;
24              } // termine if
25
26          } // termine while
27
28          // código da seção crítica
29
30          favoredThread = 2;
31          t1WantsToEnter = falso;
32
33          // código fora da seção crítica
34
35      } // termine o while mais externo
36
37  } // termine Thread T1
```

Algoritmo de Dekker

```
38
39      Thread T2:
40
41      void main( )
42      {
43          while ( !done )
44          {
45              t2WantsToEnter = true;
46
47              while ( t1WantsToEnter )
48              {
49                  if ( favoredThread == 1 )
50                  {
51                      t2WantsToEnter = false;
52                      while ( favoredThread == 1 ); // espera ociosa
53                      t2WantsToEnter = true;
54                  } // termine if
55
56              } // termine while
```

Algoritmo de Dekker

```
57
58              // código da seção crítica
59
60              favoredThread = 1;
61              t2WantsToEnter = false;
62
63              // código fora da seção crítica
64
65          } // termine o while mais externo
66
67      } // termine Thread T2
```

Solução de Peterson

- Peterson descobriu um modo mais simples de se obter exclusão mútua, tornando obsoleta a solução de Dekker.
- Rotinas que devem ser chamadas na entrada e na saída das regiões críticas.

Solução de Peterson

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

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

void enter_region(int process);         /* processo é 0 ou 1 */
{
    int other;                          /* número de outro processo */

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

void leave_region(int process)          /* processo: quem está saindo */
{
    interested[process] = FALSE;        /* indica a saída da região crítica */
}
```

Figura 2.19 A solução de Peterson para implementar a exclusão mútua.

Exclusão mútua de n threads: o algoritmo da padaria de Lamport

- Aplicável a qualquer quantidade de threads.
  - Cria uma fila de threads em espera distribuindo “fichas” numeradas.
  - O thread é executado quando, dentre todos os outros threads, o número de sua ficha é o menor.
  - Diferentemente do algoritmo de Dekker e do algoritmo de Peterson, o algoritmo da padaria funciona em sistemas multiprocessadores e para n threads.
  - É relativamente simples de entender por ser uma condição análoga à do mundo real.

Algoritmo da Padaria de Lamport

(1 de 3)

```
1   Sistema:
2
3   // vetor que registra quais threads estão pegando uma ficha
4   boolean choosing[n];
5
6   // valor da ficha para cada thread inicializado em 0
7   int ticket[n];
8
9   startThreads( ); // inicialize e lance todos os threads.
10
11  Thread Tx:
12
13  void main( )
14  {
15      x = threadNumber( ); // armazene o número corrente do thread
16
17      while ( !done )
18      {
19          // pegue uma ficha
20          choosing[x] = true; // inicie processo de seleção de ficha
21          ticket[x] = maxValue( ticket ) + 1;
22          choosing[x] = false; // encerre processo de seleção de ficha
23
```

Algoritmo da Padaria de Lamport

```
24          // espere o número ser chamado comparando o corrente
25          // valor da ficha com o valor da ficha de outro thread
26          for ( int i = 0; i < n; i++)
27          {
28              if ( i == x )
29              {
30                  continue; // não é preciso verificar a própria ficha
31              } // termine if
32
33              // espere ociosamente enquanto thread[i] está escolhendo
34              while ( choosing[i] != false );
35
36              // espere ociosamente até que o valor corrente da ficha seja o mais baixo
37              while ( ticket[i] != 0 && ticket[i] < ticket[x] );
38
39              // código de resolução de impasse favorece ficha de menor número
40              if ( ticket[i] == ticket[x] && i < x )
41
```

Algoritmo da Padaria de Lamport

```
42                      // execute laço até thread[i] sair de sua seção crítica
43                      while ( ticket[i] != 0 ); // espere ociosamente
44              } // termine for
45
46              // código da seção crítica
47
48              ticket[x] = 0; // exitMutualExclusion
49
50              // código fora da seção crítica
51
52          } // termine while
53
54      } // termine Thread TX
```

Exclusão mútua – espera ocupada

- Instrução TSL: utiliza registradores do hardware;
  - TSL RX, LOCK; (lê o conteúdo de ***lock*** em RX, e armazena um valor diferente de zero (0) em ***lock*** – operação indivisível);
  - ***Lock*** é compartilhada
    - Se ***lock***==0, então região crítica “liberada”.
    - Se ***lock***<>0, então região crítica “ocupada”.

```
enter_region:
    TSL REGISTER, LOCK // Copia lock para reg. e lock=1
    CMP REGISTER, #0     // lock valia zero?
    JNE enter_region     // Se sim, entra na região crítica,
                         // Se não, continua no laço
    RET                  // Retorna para o processo chamador

leave_region:
    MOVE LOCK, #0        // lock=0
    RET                  // Retorna para o processo chamador
```

Exclusão mútua

- Soluções:
  1) Espera Ocupada
  2) Primitivas *Sleep* / *Wakeup*
  3) Semáforos
  4) Monitores
  5) Passagem de Mensagem

Primitivas *sleep / wakeup*

- Todas as soluções apresentadas utilizam espera ocupada → processos ficam em estado de espera (*looping)* até que possam utilizar a região crítica:
    - Tempo de processamento da CPU;
    - Situações inesperadas;

Primitivas *sleep / wakeup*

- A primitiva *Sleep* é uma chamada de sistema que bloqueia o processo que a chamou, ou seja, suspende a execução de tal processo até que outro processo o “acorde”

- A primitiva *Wakeup* é uma chamada de sistema que “acorda” um determinado processo;

- Ambas as primitivas possuem dois parâmetros: o processo sendo manipulado e um endereço de memória para realizar a correspondência entre uma primitiva *Sleep* com sua correspondente *Wakeup*;

Primitivas *sleep / wakeup*

- Problema clássico: Produtor-Consumidor:
  - Dois processos compartilham um *buffer* de tamanho fixo. O processo produtor coloca dados no *buffer* e o processo consumidor retira dados do *buffer*;
  - Problemas:
    - **Produtor** deseja colocar dados quando o *buffer* ainda está cheio;
    - **Consumidor** deseja retirar dados quando o *buffer* está vazio;
  - **Solução**: colocar os processos para "dormir", até que eles possam ser executados;

Primitivas *sleep / wakeup*

- ***Buffer***: Uma variável `count` controla a quantidade de dados presente no *buffer*.
- **Produtor**: Antes de colocar dados no *buffer*, o processo produtor checa o valor dessa variável. Se a variável está com valor máximo, o processo produtor é colocado para dormir. Caso contrário, o produtor coloca dados no *buffer* e o incrementa.
- **Consumidor**: Antes de retirar dados no *buffer*, o processo consumidor checa o valor da variável `count` para saber se ela está com 0 (zero). Se está, o processo vai “dormir”, senão ele retira os dados do *buffer* e decrementa a variável;

Primitivas *sleep / wakeup*

```
# 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)
    }
}
```

```
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);
    }
}
```

Primitivas *sleep* / *wakeup*

- Problema → Acesso irrestrito em *count*

  Pode acontecer um momento em que os dois processos durmam para sempre.

  Solução: *bit* de controle recebe um valor `true` quando um sinal é enviado para um processo que não está dormindo. No entanto, no caso de vários pares de processos, vários *bits* devem ser criados sobrecarregando o sistema!

Próxima Aula

- ***Sincronismo de Processos: 2ª. parte***