
UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE RUSSAS

# *Rus0013 - Sistemas Operacionais*

## Aula 05 e 06: Comunicação Interprocessos

***Professor Pablo Soares***

***2022.2***

# Sumário

- Comunicação Interprocessos
- Condições de Disputa
- Regiões Críticas
- Exclusão Mútua com espera ociosa
  - Desabilitando Interrupções
  - Variáveis de Impedimento
  - Alternância Obrigatória
  - Solução de Peterson
  - Instrução TSL
- Dormir e Acordar
  - O problema produtor-consumidor
- Semáforos
- Monitores
- Troca de Mensagens
- Barreiras

# Comunicação Interprocessos

- Frequentemente processos precisam se comunicar com outro processos
  - Ex: Em um pipeline do interpretador de comandos, a saída do 1º processo deve ser passada para o 2º processo
  - Há uma necessidade de comunicação entre processos que deve ocorrer, de preferência, de uma maneira bem estruturada e sem interrupções

# Comunicação Interprocessos

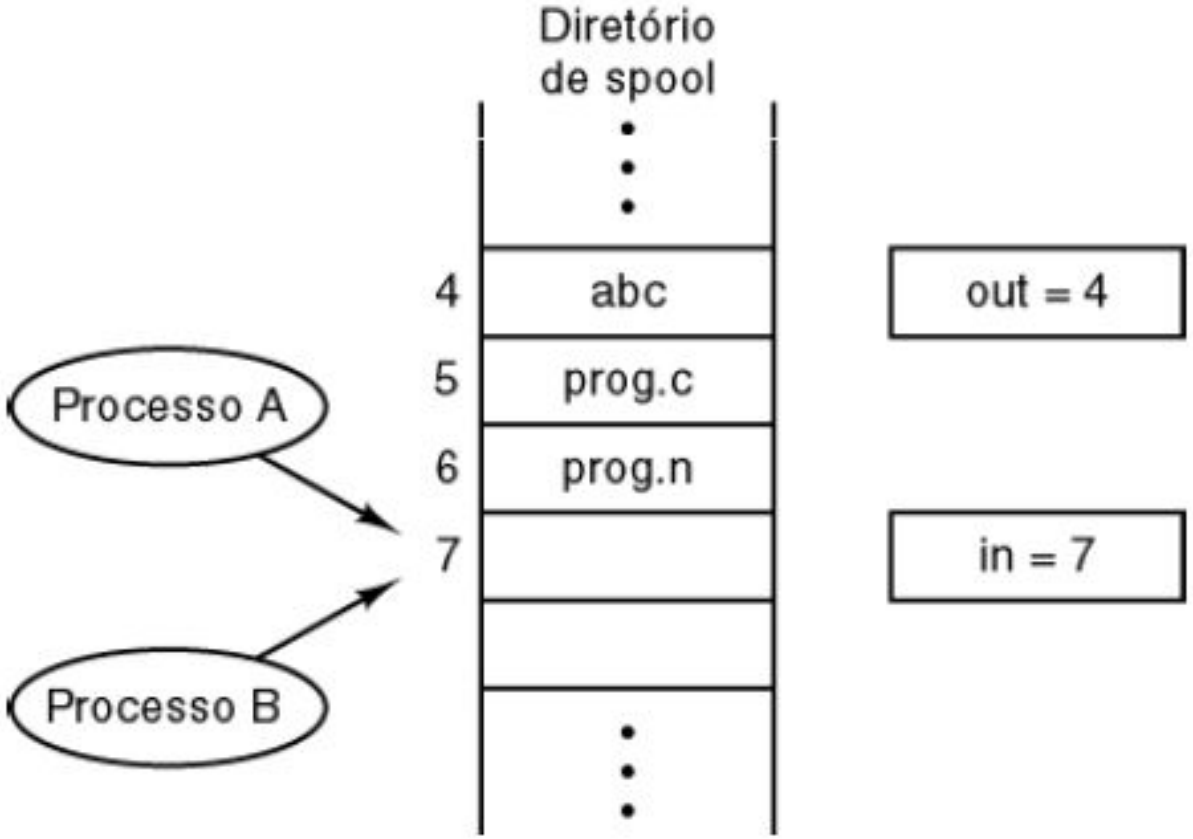
- Há 3 tópicos relacionados
  - Um processo passa informação para outro
  - Garantir que 2 ou mais processos não invadam uns aos outros
  - Sequência adequada quando existirem dependências
- Os mesmos problemas e soluções envolvendo comunicação entre processos também se aplica a threads

# Condições de disputa

- Processos que trabalham juntos podem compartilhar algum armazenamento comum
  - Memória principal ou arquivo
  - Ler e Escrever
- Ex: Quando um processo quer imprimir ele coloca o nome do arquivo em um diretório de spool e, outro processo, o daemon de impressão, verifica periodicamente se há algum arquivo para ser impresso

# Condições de disputa

- 2 processos querem ter acesso simultaneamente à memória compartilhada



# Condições de disputa

- **Condição de disputa**: 2 ou mais processos estão lendo ou escrevendo algum dado compartilhado e cujo resultado final depende das informações de quem e quando executa precisamente.

# Condições de disputa

**O que fazer para evitar condições de disputa?**

# Regiões Críticas

- Impedir que dois processos ou mais leiam ou escrevam ao mesmo tempo na memória compartilhada.
  - **Exclusão mútua**

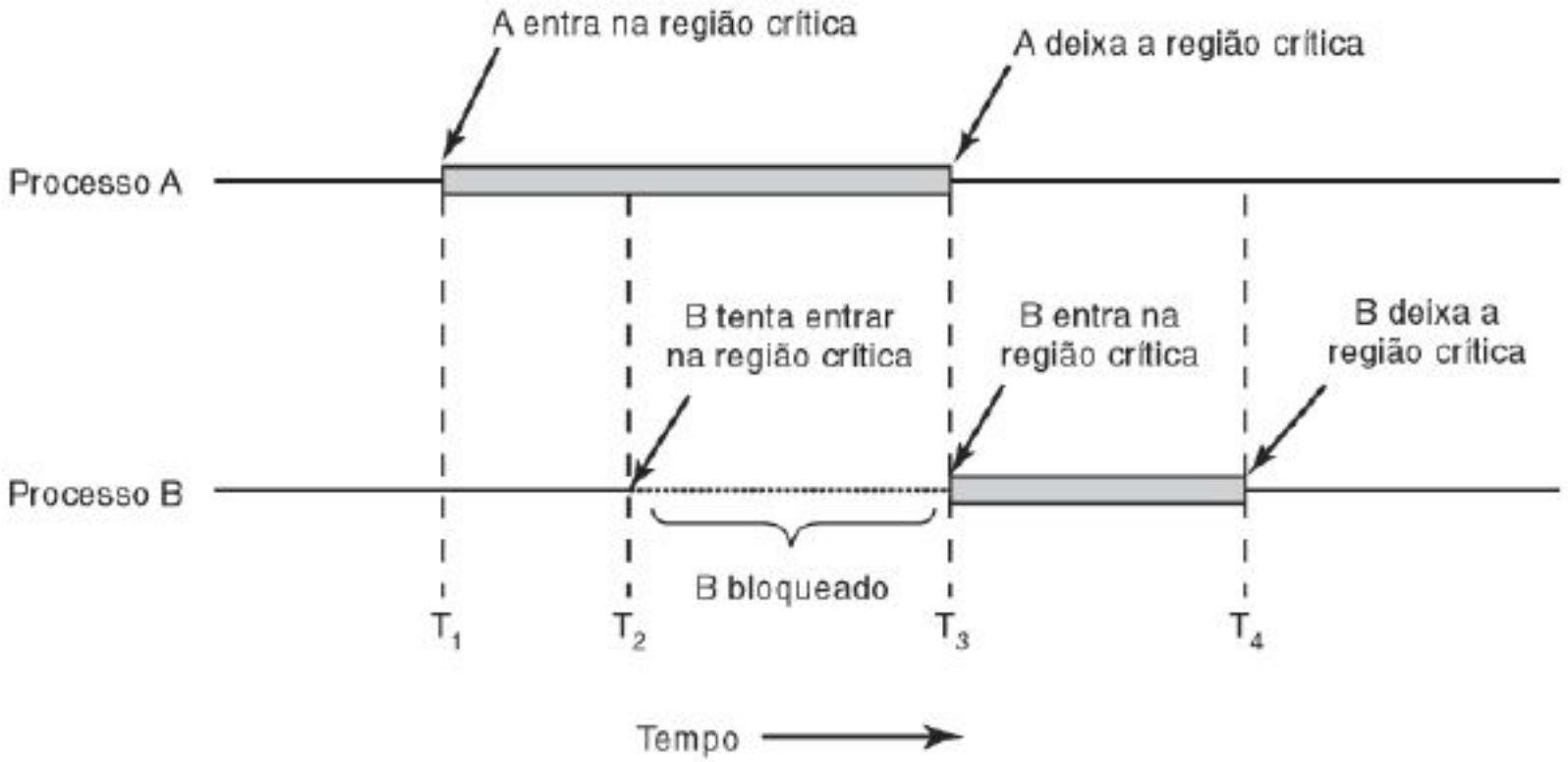
- O problema anterior aconteceu porque "B" começou a usar uma variável compartilhada antes que "A" terminasse de usá-la

# Regiões Críticas

- Regiões do código em que há acesso a um recurso compartilhado são chamadas de **regiões críticas** ou **seção crítica**
- Satisfazer 4 condições para uma boa solução
  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 outro processo
  4. Nenhum processo deve esperar eternamente para entrar em sua região crítica

# Regiões Críticas

- Exclusão Mútua usando regiões críticas



# Exclusão Mútua com Espera Ociosa

- Enquanto um processo estiver ocupado atualizando a memória compartilhada em sua região crítica, nenhum outro processo cause problema invadindo-a.
  1. Desabilitando interrupções
  2. Variáveis de impedimento
  3. Alternância obrigatória
  4. Solução de Peterson
  5. Instrução TSL

# Exclusão Mútua com Espera Ociosa

1. Desabilitando interrupções
   - A solução mais simples
     - Entrou na região crítica (Desabilita interrupções)
     - Saiu da região crítica (Habilita interrupções)
   - A CPU não será alternada para outro processo
   - Não é prudente dar aos processos o poder de desligar interrupções
   - Em multiprocessadores, afetará apenas a CPU que executou a instrução *disable*, as outras poderão ter acesso a memória compartilhada

# Exclusão Mútua com Espera Ociosa

2. Variáveis de impedimento (*lock variable*)
   - Uma única variável compartilhada (*lock*) inicialmente com o valor 0
   - Para entrar na região crítica, um processo testa antes se há impedimento, verificando o valor de *lock*, quando tem acesso modifica para 1
   - Apresenta exatamente a mesma falha do diretório de *spool*

# Exclusão Mútua com Espera Ociosa

- 3 Alternância obrigatória
  - Utiliza uma variável compartilhada *turn* que indica vez de qual processo pode entrar na **região crítica**, a qual deve ser modificada para o próximo processo antes de sair da região crítica
  - Não é uma boa idéia quando um dos processos for muito mais lento que o outro
  - Essa situação viola a condição 3: um processo bloqueia outro que não está em sua região crítica

# Exclusão Mútua com Espera Ociosa

- Alternância obrigatória

```
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)

# Exclusão Mútua com Espera Ociosa

4 Solução de Peterson

- Consiste em 2 processos em ANSI C
- Antes de entrar na região crítica cada processo

- chama *enter_region* com seu número de processo (0 ou 1). Esta chamada fará com que ele fique esperando até que seja seguro entrar. Depois de terminar o uso da região crítica, o processo chama *leave_region* para permitir a outro processo entrar

# Exclusão Mútua com Espera Ociosa

- 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 inicialmente em 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 */
}
```

# Exclusão Mútua com Espera Ociosa

5. Instrução TSL
   - Instrução test and set lock

   - TSL RX, LOCK copia o valor de RX para LOCK. Um processo pode entrar em sua região crítica apenas no caso de LOCK ser 0. A leitura do valor de LOCK e sua restauração para 0 podem ser feitas por instruções ordinárias

# Exclusão Mútua com Espera Ociosa

- Instrução TSL

```
enter_region:
        TSL REGISTER,LOCK          | copia lock para o registrador e põe lock em 1
        CMP REGISTER,#0            | lock valia zero?
        JNE enter_region           | se fosse diferente de zero, lock estaria ligado,
                                     portanto continue no laço de repetição
        RET | retorna a quem chamou; entrou na região crítica

leave_region:
        MOVE LOCK,#0               | coloque 0 em lock
        RET | retorna a quem chamou
```

# Dormir e Acordar

- As soluções baseadas em espera ociosa tem a desvantagem de provocar gasto de CPU.

- Outra abordagem é que o SO disponibilize duas chamadas de sistema (*sleep* e *wakeup*). *Sleep* faz com que um processo "durma" até ser acordado por outro processo através de um *wakeup*

# Dormir e Acordar

- O problema produtor-consumidor
  - 2 processos (produtor e consumidor) compartilham um buffer de tamanho fixo
  - O problema se origina quando o produtor quer colocar um novo item no buffer mas ele está cheio ou o consumidor remover quando está vazio
  - A solução é colocar o processo, impedido pela capacidade do buffer, para dormir até que o outro modifique o buffer e acorde o anterior

# Dormir e Acordar

- O problema 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) {                             /* número de itens no buffer */
          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 vazio, 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 */
     }
}
```

# Dormir e Acordar

- Para manter o controle do número de itens no buffer, precisamos de uma variável count
- Problema: o buffer está vazio, o consumidor verifica *count* em seguida o escalonador começa a executar o produtor que insere itens no buffer e envia um sinal de acordar para o consumidor
  - Infelizmente, o consumidor ainda não está logicamente adormecido e o sinal para acordar é perdido

# Semáforos

- Usa uma variável inteira para contar o número de sinais de acordar salvos para uso futuro
  - Down: operação que decrementa a variável do semáforo se maior que zero, caso contrário é posto para dormir
  - Up: incrementa o valor do semáforo e um dos processos bloqueados será liberado
- Verificar o valor, alterá-lo e possivelmente ir dormir são tarefas executadas todas como uma única **ação atômica**

# Semáforos

```
#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 o item do buffer */
          up(&mutex);                           /* deixa a região crítica */
          up(&empty);                           /* incrementa o contador de lugares vazios */
          consume_item(item);                   /* faz algo com o item */
     }
}
```

# Mutexes

- Basicamente são semáforos simplificados onde não há necessidade de "contar"
- Um mutex é uma variável de 1 bit que pode estar impedida ou desimpedida
- Quando um processo(thread)
  - Tenta entrar **mutex_lock()**
  - Sai chama **mutex_unlock()**
- Podem ser implementados no espaço de usuário
  - **Pacotes de Threads**

# Mutexes

```
mutex_lock:
        TSL REGISTER,MUTEX          | copia mutex para o registrador e o põe em 1
        CMP REGISTER,#0             | o mutex era zero?
        JZE ok                      | se era zero, o mutex estava desimpedido, portanto retorne
        CALL thread_yield           | o mutex está ocupado; escalone um outro thread
        JMP mutex_lock              | tente novamente mais tarde
ok:     RET | retorna a quem chamou; entrou na região crítica


mutex_unlock:
        MOVE MUTEX,#0               | põe 0 em mutex
        RET | retorna a quem chamou
```

# O problema com Semáforos

- Suponha que os dois **downs** no produtor estivessem invertidos;
- Buffer completamente cheio;
- Ocorreria um **deadlock**
  - Capítulo 6

# Monitores

- Uma unidade básica de sincronização de alto nível
- É uma coleção (módulo ou pacote)
  - Procedimentos
  - Variáveis e estrutura de dados.
- Os processo podem chamar os procedimentos
  - Mas não podem acessar as estruturas internas diretamente
- Garantia de exclusão mútua
  - Somente um processo pode estar ativo em um **monitor** em um dado momento

# Monitores

```
monitor example
        integer i;
        condition c;

        procedure producer ( );

        .
        .
        .
        end;

        procedure consumer ( );

        .       .       .

        end;
end monitor;
```

# Monitores

- Quando um processo chama um procedimento de um monitor, é verificado se outro processo está ativo.
  - Se estiver, o processo que chamou é suspenso até que o outro deixe o monitor,
  - Senão o processo que chamou poderá entrar
- Cabe ao compilador implementar a exclusão mútua nas entradas do monitor
- É preciso também um modo de bloquear processos quando não puderem continuar

# Monitores

- A solução são variáveis condicionais, com duas operações: **wait** e **signal**
  - Quando um procedimento do monitor descobre que não pode prosseguir emite um **wait** sobre uma variável condicional (resultando no bloqueio do processo), permitindo que outro processo proibido de entrar no monitor agora entre
  - Esse outro processo pode acordar seu parceiro adormecido a partir da emissão de um **signal** para a variável condicional

# Monitores

- Problema produtor-consumidor nível

```
monitor ProducerConsumer
        condition full, empty;
        integer count;
        procedure insert(item: integer);
        begin
                if count = N then wait(full);
                insert_item(item);
                count := count + 1;
                if count = 1 then signal(empty)
        end;
        function remove: integer;
        begin
                if count = 0 then wait(empty);
                remove = remove_item;
                count := count - 1;
                if count = N - 1 then signal(full)
        end;
        count := 0;
end monitor;
```

```
procedure producer;
begin
        while true do
        begin
                item = produce_item;
                ProducerConsumer.insert(item)
        end
end;
procedure consumer;
begin
        while true do
        begin
                item = ProducerConsumer.remove;
                consume_item(item)
        end
end;
```

# Monitores

- Algumas linguagens suportam monitores.
  - java
- Java suporta threads de usuário e também permite que métodos sejam agrupados em classes

# Troca de Mensagens

- Permite a troca de informações entre processos usando 2 primitivas:
  - **send**(destination, &message)
- Envia uma mensagem para um dado destino
  - **receive**(source, &message)
- Recebe uma mensagem de uma dada origem ou uma origem qualquer
- Se nenhuma mensagem estiver disponível o receptor poderá ficar bloqueado, até chegue alguma

# Troca de Mensagens

- Possui problemas específicos como perda da mensagem pela rede
  - Para prevenir pode-se usar confirmação de recebimento (acknowledgement)
  - Mas se o receptor receber a mensagem e não a confirmação?
- As mensagens podem ser numeradas
  - Chamadas send e receive não podem ser ambíguas
- Autenticação deve ser garantida
  - Desempenho deve ser garantido em troca de mensagens de processos na mesma máquina
- Copiar mensagens é mais lento que realizar operações sobre monitores ou semáforos

# Troca de Mensagens

- Problema produtor-consumidor
  - Pode ser resolvido com troca de mensagens sem memória compartilhada
  - Considerando que todas as mensagens são do mesmo tamanho e que aquelas enviadas e ainda não recebidas são armazenadas pelo SO
  - O consumidor envia N mensagens vazias para o produtor, que se tiver algum item para fornecer pegará uma mensagem vazia e retornará uma cheia. O no. total de mensagens permanece cte.

# Troca de Mensagens

```
#define N 100                                   /* número de lugares no buffer */

void producer(void)
{
     int item;
     message m;                                 /* buffer de mensagens */

     while (TRUE) {
          item = produce_item( );               /* gera alguma coisa para colocar no buffer */
          receive(consumer, &m);                /* espera que uma mensagem vazia chegue */
          build_message(&m, item);              /* monta uma mensagem para enviar */
          send(consumer, &m);                   /* envia item para consumidor */
     }
}

void consumer(void)
{
     int item, i;
     message m;

     for (i = 0; i < N; i++) send(producer, &m);  /* envia N mensagens vazias */
     while (TRUE) {
          receive(producer, &m);                /* pega mensagem contendo item */
          item = extract_item(&m);              /* extrai o item da mensagem */
          send(producer, &m);                   /* envia a mensagem vazia como resposta */
          consume_item(item);                   /* faz alguma coisa com o item */
     }
}
```

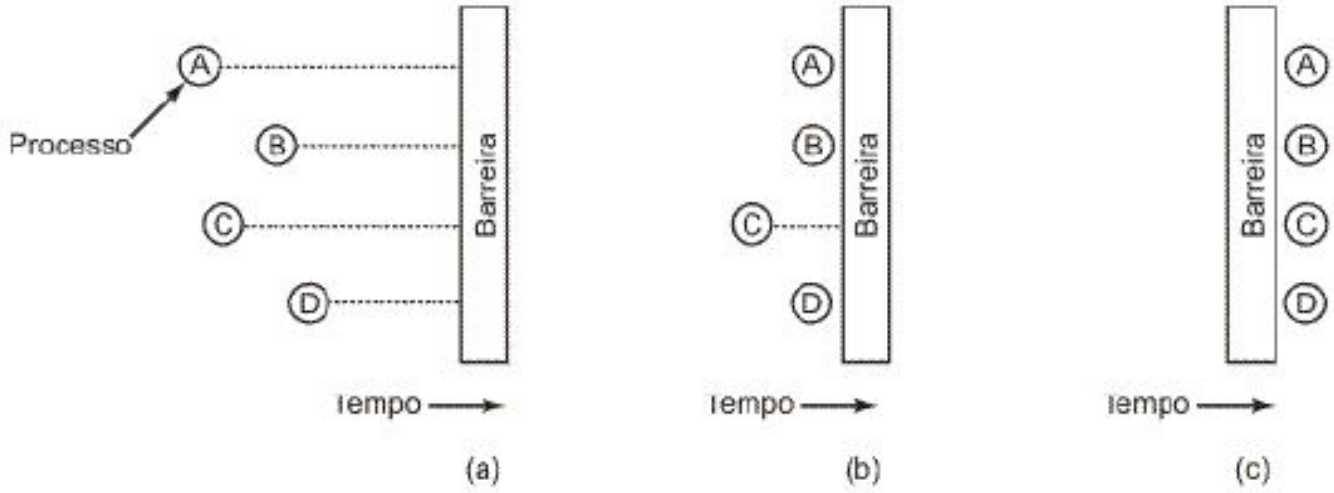
# Troca de Mensagens

- Se o produtor trabalhar mais rápido que o consumidor?
- Se o consumidor trabalhar mais rápido que o produtor?
- Troca de mensagens é bastante usada em programação paralela. Ex: MPI (messagepassing interface – interface de troca de mensagem)

# Barreiras

- Mecanismo de sincronização dirigido a grupos de processos
- Algumas aplicações são divididas em fase e têm como regra que nenhum processo pode avançar para a próxima fase até que todos os processos estejam prontos para fazê-lo
- Isso pode ser conseguido por meio de uma barreira no final de cada fase
- Quando uma barreira é alcançada o processo é bloqueado até que todos a alcancem

# Barreiras



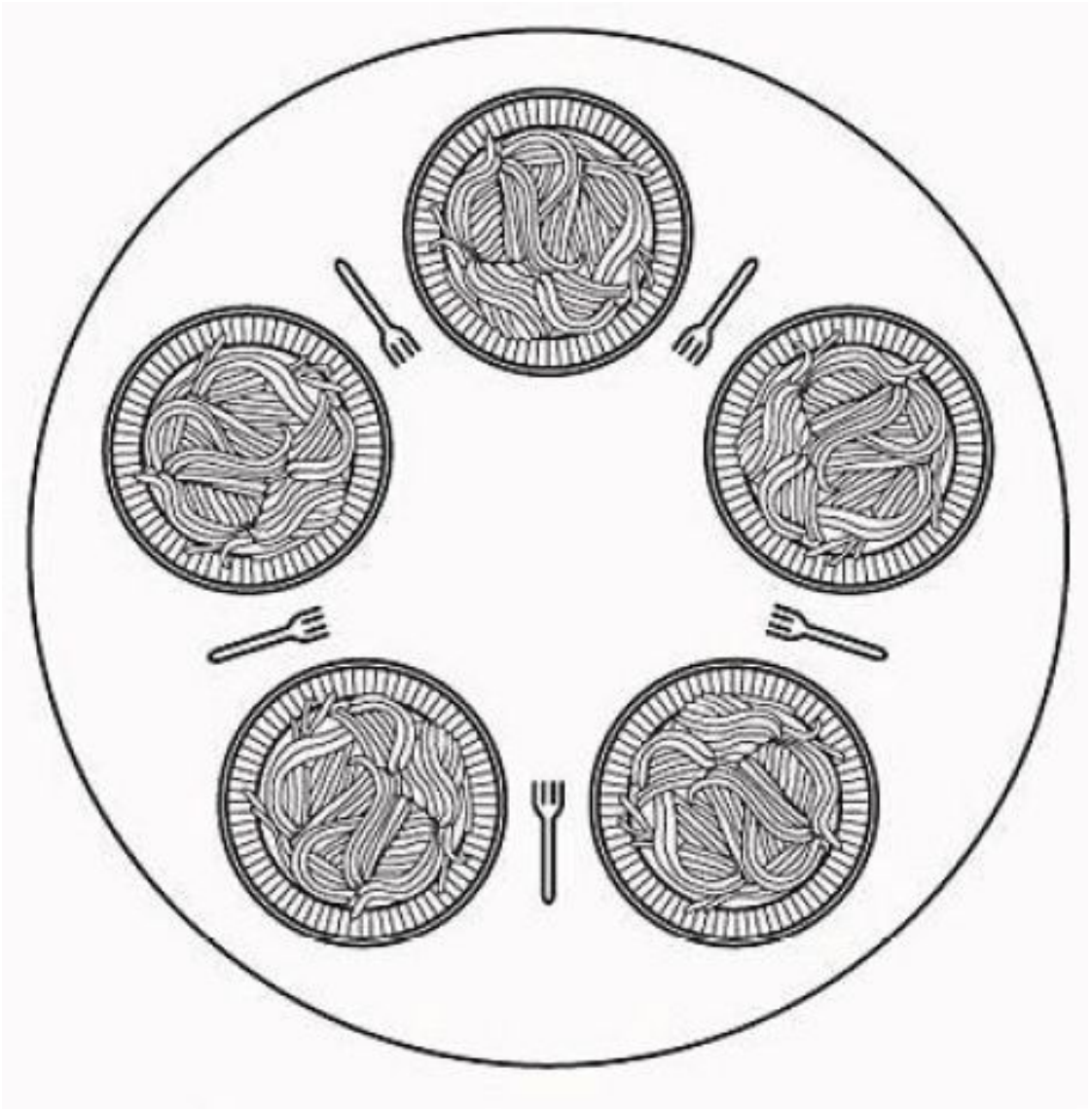

- Uso de uma barreira

  a) processos se aproximando de uma barreira

  b) todos os processos, exceto um, bloqueados pela barreira

  c) último processo chega, todos passam

# O problema do Jantar dos Filósofos



# Referências


- Andrew S. Tanenbaum. “Sistemas Operacionais Modernos”. 3ª Edição, Prentice Hall, 2010.


UNIVERSIDADE FEDERAL DO CEARÁ

CAMPUS DE RUSSAS

# *Rus0013 - Sistemas Operacionais*

## Aula 05: Comunicação Interprocessos

***Professor Pablo Soares***

***2022.2***