Sistemas Operacionais

Unidade 2: Gerência de Processos – Comunicação entre Processos

Introdução

- Os processos executando concorrentemente podem ser do tipo:
  - Independentes
    - Um processo é independente se não puder afetar ou ser afetado pelos outros processos em execução no sistema.
    - Qualquer processo que não compartilhe dados com qualquer outro processo é independente.
  - Cooperativos
    - Um processo é cooperativo se puder afetar ou ser afetado por outros processos em execução no sistema.
    - Qualquer processo que compartilhe dados com outros processos.

Introdução

- Os processos cooperativos precisam de mecanismos de comunicação entre processos que lhe permitam a troca de dados.
- A cooperação entre processos é descrita frequentemente na literatura como **IPC** (*Inter-Process Communication*).
- Existem dois modelos fundamentais de comunicação entre processos:
  - Memória compartilhada
    - É estabelecida uma área compartilhada para que os processos possam ler ou escrever nessa área.
  - Memória distribuída
    - A comunicação é toda feita através de troca de mensagens.

IPC - Memória Compartilhada

- Problema: Condições de Corrida (*race conditions*)
  - **Condições de Corrida** são situações onde dois ou mais processos estão acessando dados compartilhados, e o resultado final do processamento depende de quem roda quando.

Exemplos de Condição de Corrida

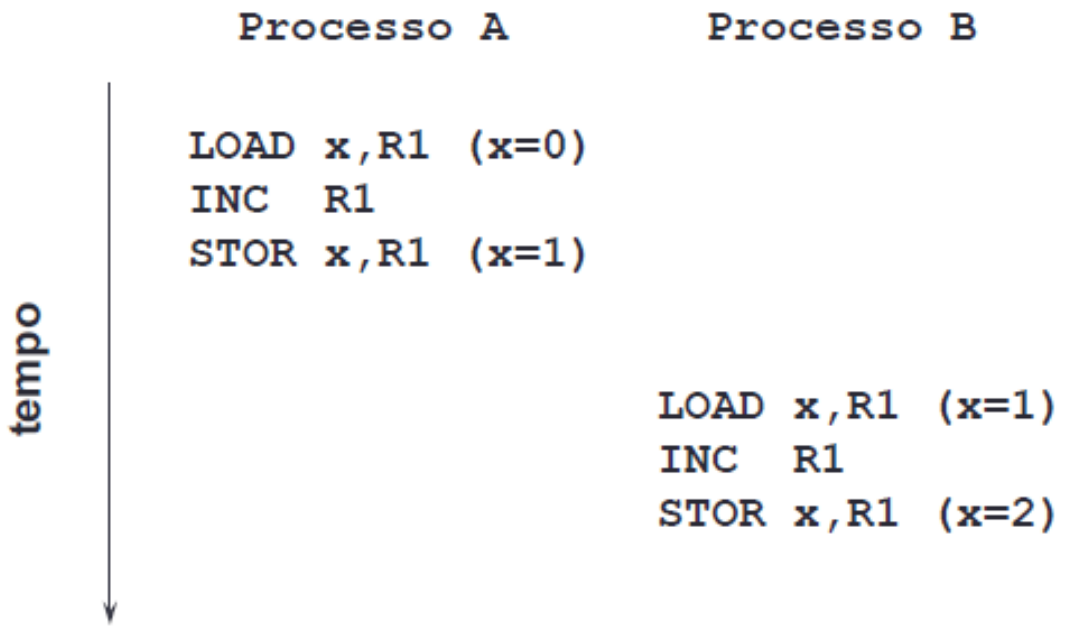
- Primeiro exemplo de condições de corrida:
  - Considere dois processos A e B com o seguinte código:

Processo A	Processo B
`x=x+1;`	`x=x+1;`

  - Considere que, inicialmente, o valor de x=0. Que valores de x podemos obter quando os processos A e B se executam simultaneamente? (Suponha um escalonador preemptivo).

Exemplos de Condição de Corrida

Caso 1: Processo A -> Processo B

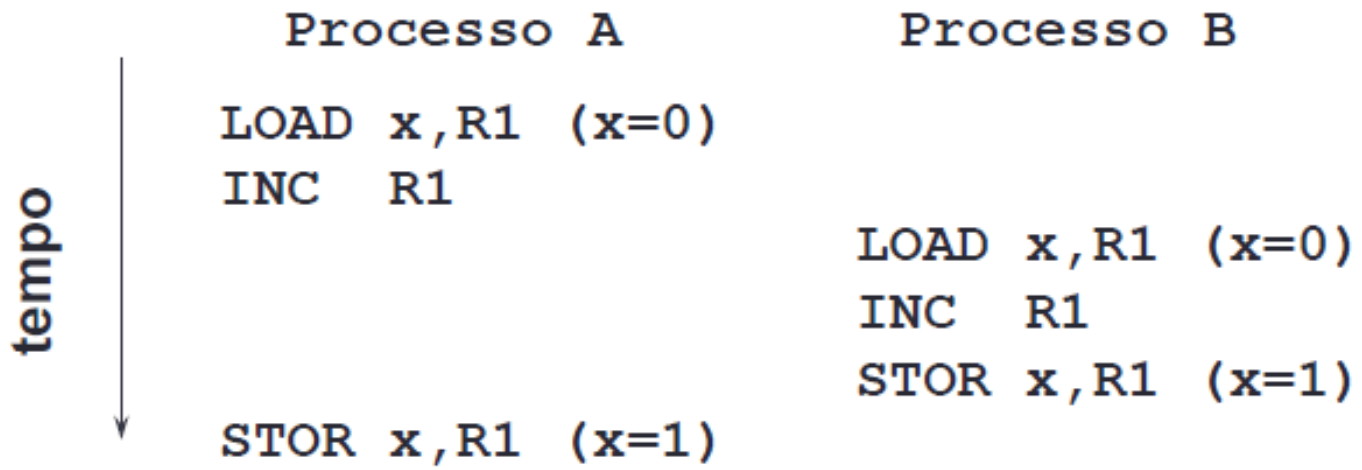


Valor final: x=2

Exemplos de Condição de Corrida

Caso 2:

Processo A -> Processo B -> Processo A

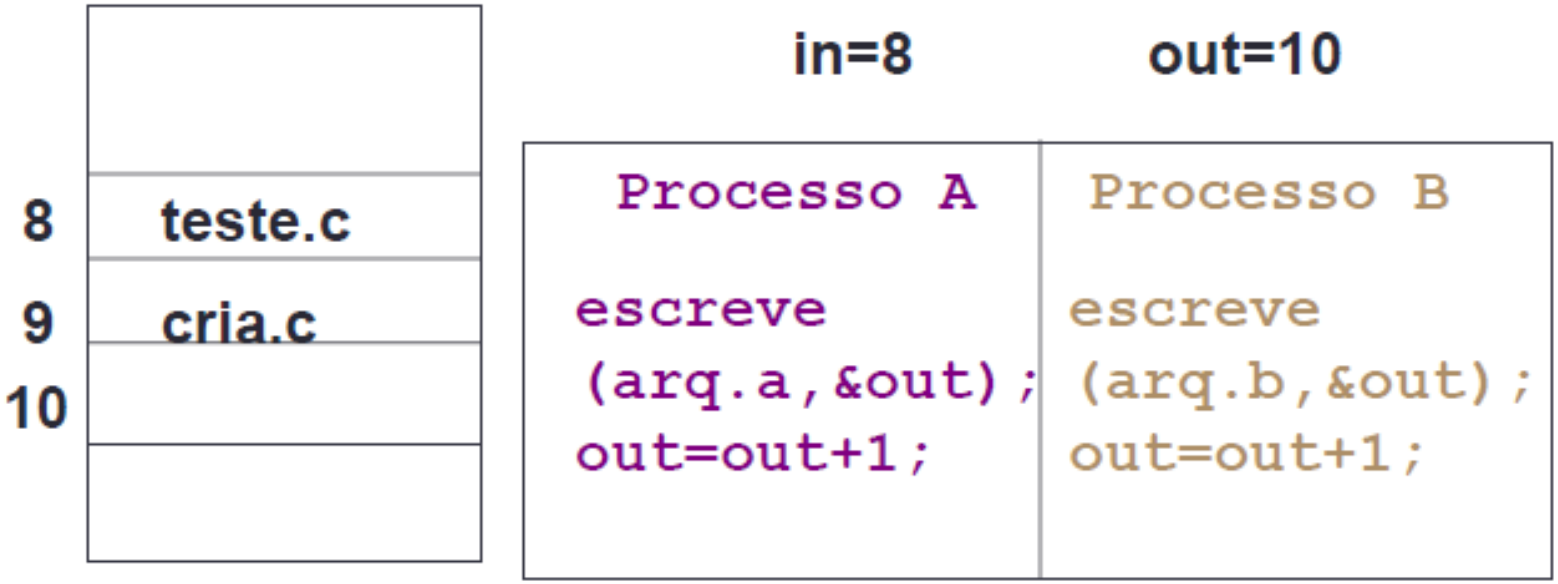


Valor final: x=1

- Esta situação pode ocorrer? Se sim, em que condições?
- Voce considera que o programador admitia, ao fazer o seu programa, que a variavel x pudesse assumir o valor 1 ao final da execução?

Exemplos de Condição de Corrida

Segundo Exemplo

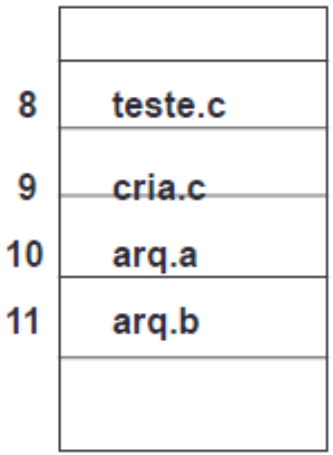


processo A: escreve arquivo
arq.a na fila de impressão
processo B: escreve arquivo
arq.b na fila de impressão

processo spool: imprime

Exemplos de Condição de Corrida

Caso 1: Processo A -> Processo B



tempo

Processo A

```
LOAD out,R1 out
(=10)
(coloca arq.a na
posição 10)
INC  R1
STOR out,R1
(out=11)
```

Processo B

```
LOAD out,R1
(out=11)
(coloca arq.b na
posição 11)
INC  R1
STOR out,R1
(out=12)
```

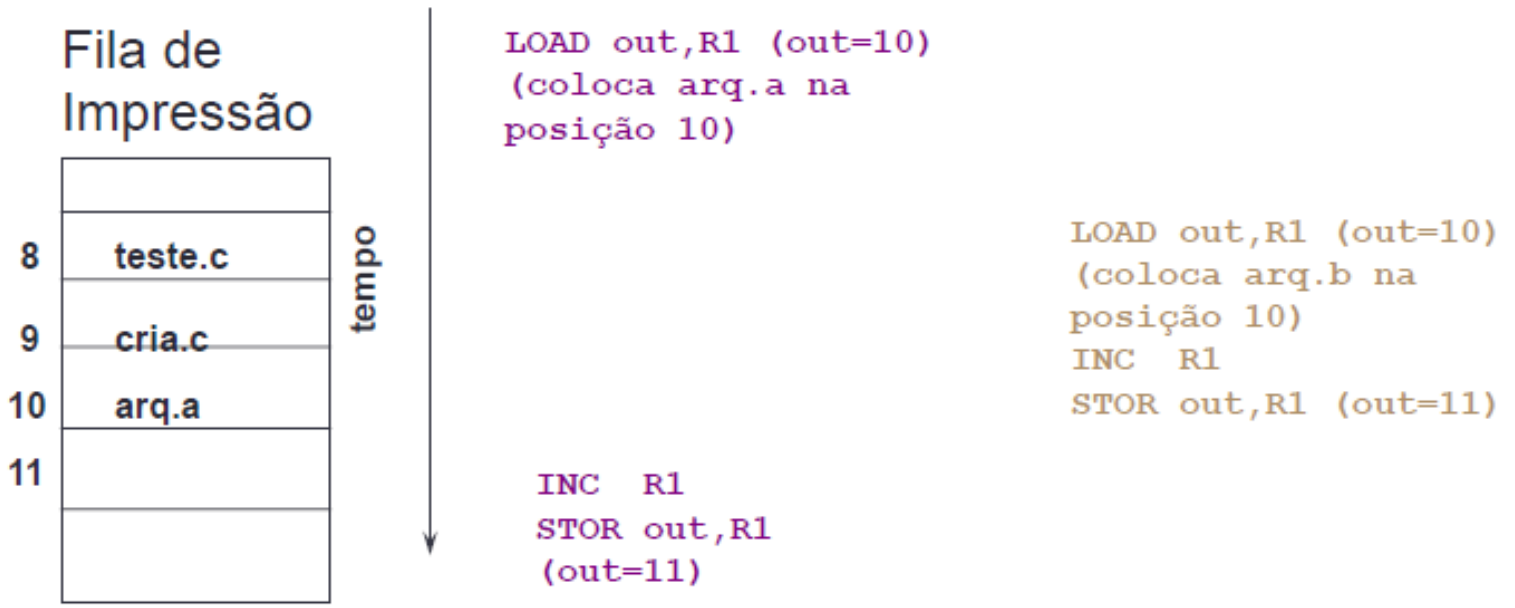
Valores Finais

in=8 out=12

Exemplos de Condição de Corrida

Caso 2:

Processo A -> Processo B -> Processo A



Valores Finais

in=8 out=11

IPC - Memória Compartilhada

- **Condições de Corrida (Solução):** encontrar alguma forma de proibir que mais de um processo acesse o dado compartilhado ao mesmo tempo, isto é, estabelecer a **exclusão mútua** de execução.
- **Exclusão Mútua:** impedir que dois ou mais processos acessem um mesmo recurso ao mesmo tempo.
- **Região Crítica:** parte do código do programa onde é feito o acesso à memória compartilhada (ou ao recurso compartilhado), ou seja, é a parte do programa cujo processamento pode levar à ocorrência de condições de corrida.

Solução para Condição de Corrida

- Uma boa solução para a condição de corrida requer que quatro condições sejam satisfeitas:
  1. Dois ou mais processos não podem estar simultaneamente dentro de suas regiões críticas correspondentes;
  2. Nenhuma consideração pode ser feita a respeito da velocidade relativa dos processos, ou a respeito do número de processadores disponíveis no sistema;
  3. Nenhum processo que esteja executando fora de sua região crítica pode bloquear a execução de outro processo;
  4. Nenhum processo pode ser obrigado a esperar indefinidamente para entrar em sua região crítica.

Propostas de Solução para Exclusão Mútua

- **Inibição das Interrupções:** consiste em inibir as interrupções de cada processo logo após o seu ingresso em uma **região crítica**, habilitando-as outra vez imediatamente antes de deixá-las.
- Desabilitando as interrupções, o processo não pode ser interrompido pelo escalonador pois este retira a CPU dos processos através de interrupções.

Propostas de Solução para Exclusão Mútua

- **Inibição das Interrupções:**

```
desabilita_interrupções();
/*executa a seção critica */
x=x+1;
habilita_interrupções();
```

Perigo, não é uma boa prática atribuir aos processos de usuários o poder de desabilitar interrupções, interferindo diretamente no *kernel* do sistema operacional.

Propostas de Solução para Exclusão Mútua

- **Variáveis de Travamento:** cria-se uma variável única compartilhada (variável de travamento), cujo valor pode assumir 0 ou 1. O valor em 0 significa que não há nenhum processo executando a sua região crítica, e o valor em 1 significa que algum processo está executando sua região crítica.

Problema: dois processos (A e B) tentam entrar em suas regiões críticas simultaneamente, então A, por exemplo, pega o valor da VT em 0, porém A pode não conseguir atualizar o valor de VT antes que B o acesse, o que vai gerar condição de corrida.

Propostas de Solução para Exclusão Mútua

- **Estrita Alternância:** mais uma forma implementada por software, que utiliza a variável inteira vez (*turn*).
  - A variável inteira "*vez*" estabelece de quem é a vez de entrar na região crítica.

```
while (TRUE)
{
  while (vez != 0) { }
  regiao_critica();
  vez = 1;
  regiao_não_critica();
}
```

P1

```
while (TRUE)
{
  while (vez != 1) { }
  regiao_critica();
  vez = 0;
  regiao_não_critica();
}
```

P2

Propostas de Solução para Exclusão Mútua

- **Estrita Alternância:** não é uma boa idéia quando um dos processos é muito mais lento que o outro. Esta solução requer a entrada estritamente alternada de dois processos em suas regiões críticas;
  - **Solução não implementada na prática!!!**

O teste contínuo do valor de uma variável, aguardando que ela assuma determinado valor é denominado **espera ocupada**.

Propostas de Solução para Exclusão Mútua

- **Algoritmo de Dekker:** além da variável vez, serve-se de um vetor que indica a intenção de um processo de entrar na região crítica.
- Os processos são Pi e Pj, onde j=1-i.

Propostas de Solução para Exclusão Mútua

Algoritmo de Dekker

```
while (TRUE)
{
  flag[i] = TRUE;
  while (flag[j] != FALSE)
  {
    if (vez == j)
    {
       flag[i] = FALSE;
       while (vez == j){ }
       flag[i] = TRUE;
    }
  }
  secao_critica();
  vez  = j;
  flag[i] = FALSE;
}
```

Pi = processo
j = 1 - i

Variáveis globais:flag, vez

Variáveis locais: i,j

Inicialização:
vez=0;
flag[0..1] = FALSE

Propostas de Solução para Exclusão Mútua

- **Algoritmo de Dekker:**
  - *P0* deseja entrar na sua R.C.

    $\Rightarrow$ *flag* = *true*

    $\Rightarrow$ *P0* verifica o *flag* de *P1*
  - *P1* se *flag* = *false*, então *P0* pode entrar imediatamente na sua R.C.
  - Caso contrário, *P0* consulta a variável *vez*
  - Se *vez* = *0*, *P0* conhece que é a sua vez de insistir e permanece em *espera ocupada*, testando o estado de *P1*.

Propostas de Solução para Exclusão Mútua

- **Algoritmo de Dekker:**
  - *P1* em determinado momento declina da sua vez
  - Este fato permite ao processo *P0* prosseguir
  - Após *P0* usar a sua R.C.

    $\Rightarrow$ *flag* = *false* para liberá-la

    $\Rightarrow$ *vez* = *1* para transferir o direito para *P1*

Propostas de Solução para Exclusão Mútua

- **Semáforos:** também baseados em um tipo de variável (semáforo) que pode sofrer duas operações básicas: **DOWN (ou P)** e **UP (ou V)**.
  - O semáforo fica associado a um recurso compartilhado, indicando quando o recurso está sendo acessado por um dos processos concorrentes;
  - Se o valor da variável semáforo for diferente de zero, nenhum processo está utilizando o recurso; caso contrário, o processo fica impedido do acesso;
  - Sempre que deseja entrar em sua região crítica, um processos executa uma instrução **DOWN (sem)**;

Propostas de Solução para Exclusão Mútua

- **Semáforos:**
  - Se o semáforo for maior que 0, este é decrementado de 1, e o processo que solicitou a operação pode executar sua região crítica;
  - Entretanto, se uma instrução DOWN é executada em um semáforo cujo valor seja igual a 0, o processo que solicitou a operação ficará no estado de bloqueio;
  - Além disso, o processo que está acessando o recurso, ao sair de sua região crítica, executa uma instrução **UP (sem)**, incrementando o semáforo de 1 e liberando o acesso ao recurso;

Propostas de Solução para Exclusão Mútua

- **Semáforos:**

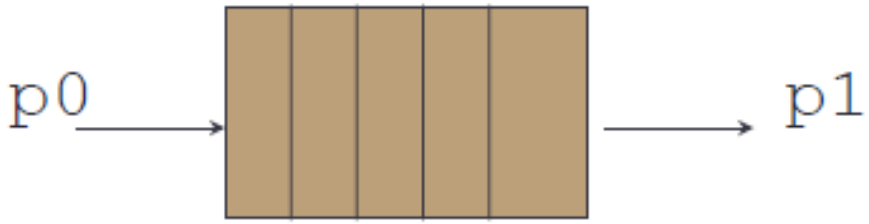
```
P(sem): if (sem > 0)
           sem = sem -1;
        else
           bloqueia(sem);


V(sem): if (processos_espera())
          acorda_processo();
        sem = sem +1;


P(sem);
/* seção crítica */
V(sem);
```

Propostas de Solução para Exclusão Mútua

- **Semáforos:**
  - A verificação do valor do semáforo, a modificação do seu valor e, eventualmente a colocação do processo para dormir são operações atômicas;
  - **Operações atômicas** são únicas e indivisíveis;
  - Os semáforos aplicados ao problema da exclusão mútua são chamados de ***mutex*** (*mutual exclusion*) ou binários, por apenas assumirem os valores 0 e 1.

Exemplo de Utilização dos Mecanismos de Exclusão Mútua

- Problema: Produtor-Consumidor
  - Neste problema, dois processos p0 e p1 compartilham um *buffer* de tamanho fixo. O processo p0 escreve dados no *buffer* e o processo p1 retira dados do *buffer*.



Exemplo de Utilização dos Mecanismos de Exclusão Mútua

- Problema: Produtor-Consumidor
  - Como o *buffer* tem tamanho fixo, devemos ter uma variável que controle o número de mensagens no *buffer*, para que o processo produtor não escreva no *buffer* cheio e o processo consumidor não retire dados do *buffer* vazio.
  - O número de mensagens no *buffer* é uma variável compartilhada e o acesso a ela pode levar a condições de corrida.

Exemplo Produtor-consumidor com Semáforo

P (sem) =Down (sem)
V (sem) = Up (sem)

```
semaphore mutex = 1; /* controla a seção crítica */
semaphore vazio = N; /* vazias */
semaphore cheio = 0; /* ocupadas */
```

```
void produtor()
{
  int item;
  while (TRUE)
  {
    produz_item(&item);
    P(&vazio);
    P(&mutex);
    insere_item(&item);
    V(&mutex);
    V(&cheio);
  }
}
```

```
void consumidor()
{
  int item;
  while (TRUE)
  {
    P(&cheio);
    P(&mutex);
    remove_item(&item);
    V(&mutex);
    V(&vazio);
    consome_item(item);
  }
}
```

Propostas de Solução para Exclusão Mútua

- Até agora, ao programador é dado um conjunto de primitivas e ele deve garantir que estas primitivas sejam utilizadas corretamente (ex: todo *down(s)* deve ter um *up(s)* associado).
- A programação da concorrência é dita, neste caso, programação de baixo nível.

Propostas de Solução para Exclusão Mútua

- **Monitores:** é um conjunto de procedimentos, variáveis e estruturas de dados, todas agrupadas em um módulo especial.
  - Monitores são mecanismos de sincronização de alto nível, que tornam mais fácil o desenvolvimento de programas concorrentes;
  - Sua característica mais importante é a implementação automática da exclusão mútua entre seus procedimentos, ou seja, somente um processo pode estar ativo dentro do monitor em um dado instante de tempo;

Propostas de Solução para Exclusão Mútua

- **Monitores:**
  - Toda vez que algum processo chama um desses procedimentos, o monitor verifica se já existe outro processo executando algum procedimento do monitor;
  - Caso exista, o processo ficará aguardando sua vez até que tenha permissão para executar;
  - Toda a implementação da exclusão mútua nos monitores é realizada pelo compilador e não mais pelo programador, como no caso do uso dos semáforos;
  - **Java** é um exemplo de linguagem que permite o uso de monitor.

Propostas de Solução para Exclusão Mútua

- **Monitores:**
  - Quem se preocupa em garantir a exclusão mútua é o compilador e não o programador.
  - A única tarefa do programador é identificar as seções críticas e colocá-las dentro do monitor.
  - Para implementar a exclusão mútua dentro de um monitor, o compilador usa geralmente estruturas de baixo nível, como por exemplo, semáforos.
  - **Java** é um exemplo de linguagem que permite o uso de monitor.

Propostas de Solução para Exclusão Mútua

- **Monitores:**
  - Adicionando-se a palavra-chave ***synchronized*** à declaração de um método, Java garante que, uma vez iniciado qualquer *thread* executando esse método, a nenhuma outra *thread* será permitida executar qualquer outro método ***synchronized*** naquela classe.

Propostas de Solução para Exclusão Mútua

Variáveis Condicionais (ou de condição):

- Variáveis de condição são utilizadas para ordenar a execução de diferentes processos. Elas permitem que um processo se bloqueie esperando uma ação de outro processo.
- Existem duas operações possíveis sobre as variáveis de condição:
  - wait(var): bloqueia o processo em var
  - signal(var): acorda o(s) processo(s) bloqueados em var.

Propostas de Solução para Exclusão Mútua

- **Variáveis Condicionais (ou de condição):**
  - Monitores implementam exclusão mútua e sincronização entre *threads*;
  - Uma *thread* que esteja correntemente dentro de um monitor poderá ter de esperar fora dele até que um outra *thread* execute uma ação no monitor;
  - Por exemplo, no problema produtor/consumidor, o produtor, ao verificar que o consumidor ainda não consumiu nenhum item, deve esperar fora do monitor, para que o consumidor possa pegar o item;
  - Assim, a *thread* que está dentro do monitor utiliza a **variável condicional** para esperar por uma condição fora do monitor.

Propostas de Solução para Exclusão Mútua

Variáveis Condicionais:

- O monitor associa a variável condicional a cada situação distinta que poderá obrigar a *thread* a esperar;
- Assim, as operações **wait** (VariavelCondicional) e **signal** (VariavelCondicional), são usadas para a sincronização entre as *threads;*
- **Variáveis condicionais** são diferentes de *variáveis convencionais* porque elas têm uma fila associada;
- Uma *thread* que chamar **wait** em uma variável condicional será colocada na fila associada com aquela variável.

Propostas de Solução para Exclusão Mútua

Variáveis Condicionais:

- Enquanto a *thread* estiver na fila, ela estará fora do monitor, de modo que uma outra *thread* poderá eventualmente entrar no monitor e chamar ***signal***.
- Uma *thread* que chame ***signal*** em uma variável condicional particular fará que uma *thread*, que esteja esperando naquela variável condicional, seja retirada da fila associada àquela variável, e entre novamente no monitor.

Exemplo Produtor-consumidor com Monitor

```
monitor produtor-consumidor
condicao cheio, vazio;
int cont;
cont = 0;
procedure insere
{
    if (cont == N) wait(cheio);
    insere_item();
    cont++;
    if (cont == 1) signal(vazio);
}
procedure remove
{
     if (cont == 0) wait(vazio);
     remove_item();
     cont--;
     if (cont == N-1) signal(cheio);
}
end monitor;
```

```
produtor()
{
    while(TRUE)
    {
      produz_item(&item);
      produtor-consumidor.insere();
    }
}
consumidor()
{
    while(TRUE)
    {
      produtor-consumidor.remove();
      consome_item(item);
    }
}
```

Comunicação por Troca de Mensagens

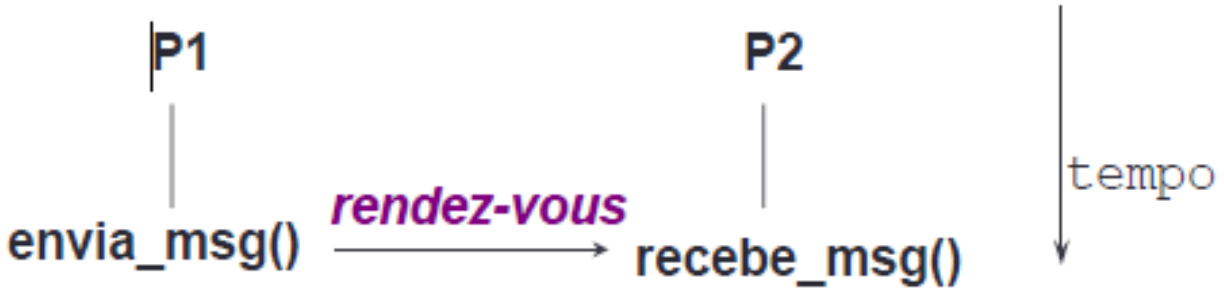
- Para a comunicação/sincronização entre os processos que não compartilham memória física, os conceitos de semáforos e monitores perdem suas funcionalidades, pois as variáveis de controle não poderiam ser compartilhadas.
- Assim, para a comunicação entre os processos que usam memória distribuída, faz-se necessário a Troca de Mensagens.
- A troca de mensagens pode ser utilizada tanto em sistemas com memória compartilhada quanto em sistema com memória distribuída.

Comunicação por Troca de Mensagens

- A troca de mensagens é explícita. Um processo envia uma mensagem e outro processo a recebe. Só neste momento, a comunicação é estabelecida.
- As primitivas básicas para a troca de mensagem são:
  - send(destination, &message)
  - receive(source, &message)
- Quando projetamos a comunicação por troca de mensagens devemos nos preocupar com os seguintes aspectos no projeto das primitivas:
  - tipo das primitivas: bloqueadas/não-bloqueadas
  - tipo da comunicação: 1 para 1, n para 1, n para n

Comunicação por Troca de Mensagens

- A comunicação por troca de mensagem é composta por duas partes: uma que quer enviar mensagens e outra que deseja recebê-las. O momento onde a troca de mensagens realmente ocorre é denominado ***rendez-vous***.

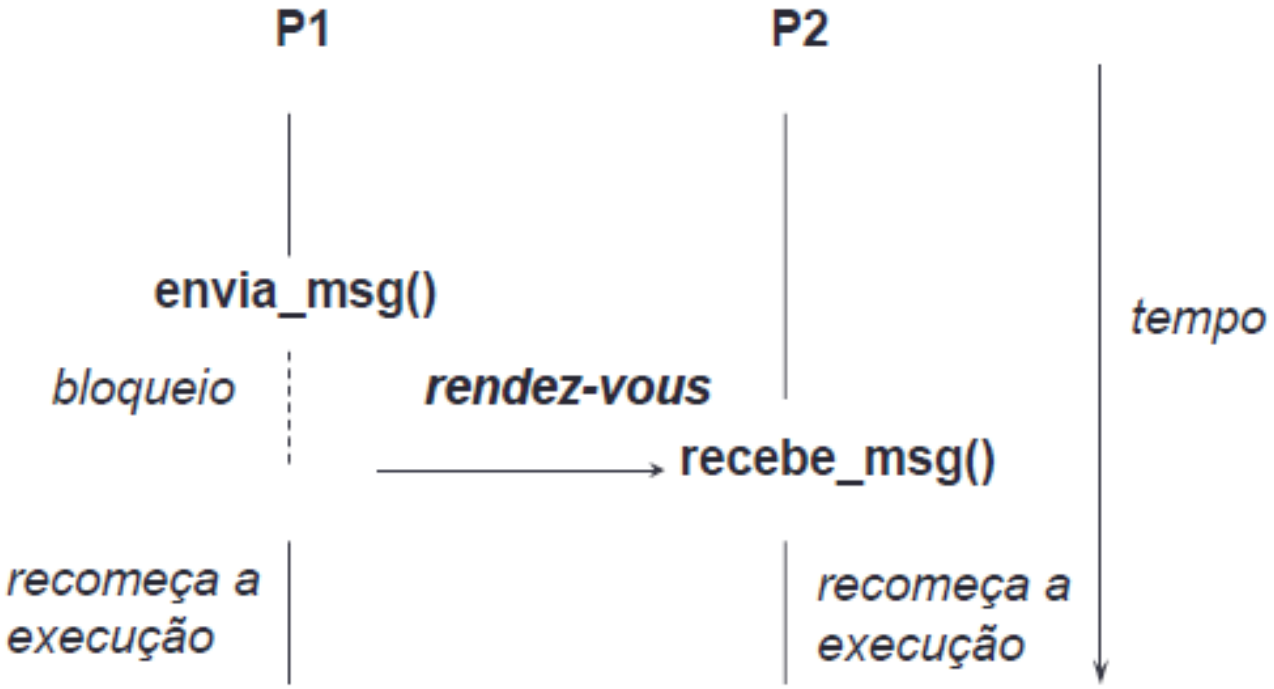


Troca de Mensagens

- No projeto das primitivas de comunicação, deve-se determinar o comportamento no caso da outra parte não estar em ponto de rendez-vous quando executamos a nossa primitiva.
- Basicamente, há dois tipos de comunicação:
  - Comunicação Síncrona
  - Comunicação Assíncrona.
- Na Comunicação Síncrona, quando um processo envia uma mensagem (SEND) ele fica esperando até que o processo receptor leia a mensagem e vice-versa (primitivas bloqueantes).
- Na Comunicação Assíncrona, nem o receptor permanece aguardando o envio de uma mensagem, nem o transmissor o seu recebimento (primitivas não-bloqueantes).

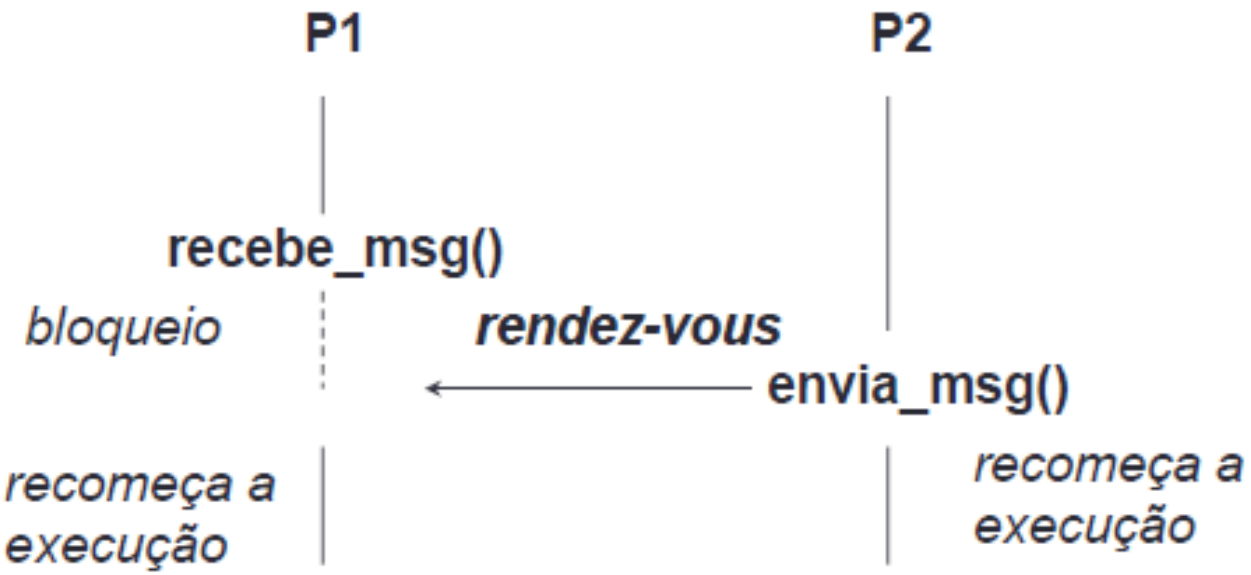
Troca de Mensagens

- Primitivas Bloqueantes:



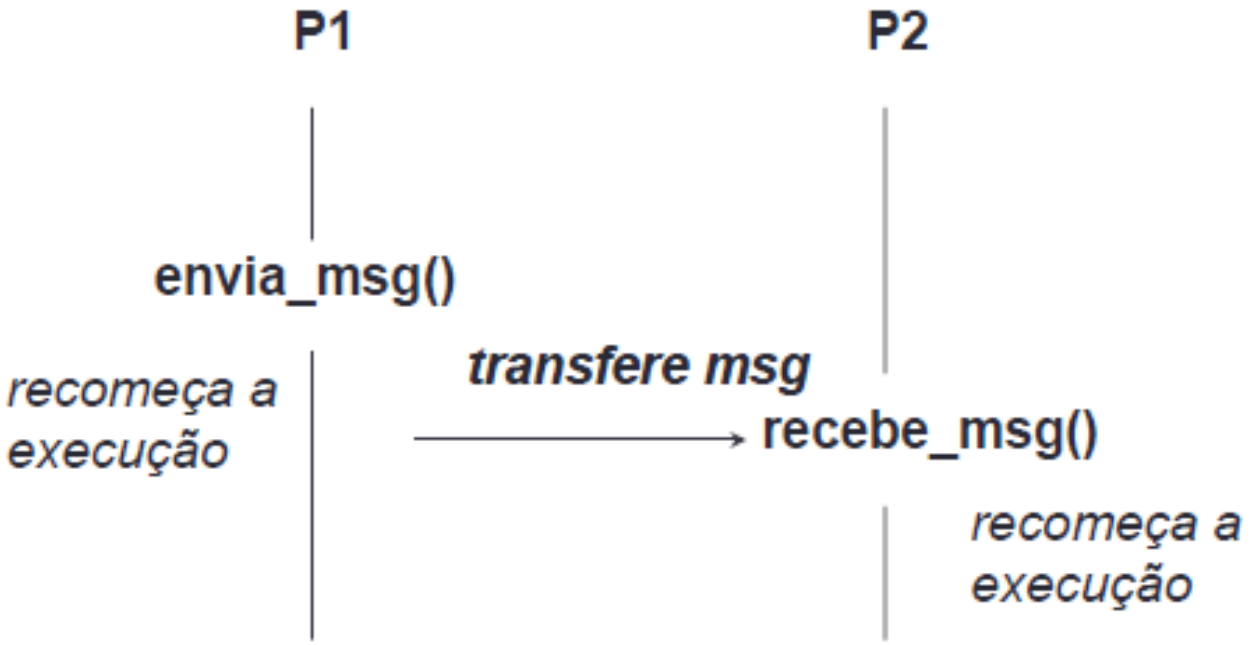
Troca de Mensagens

- Primitivas Bloqueantes:



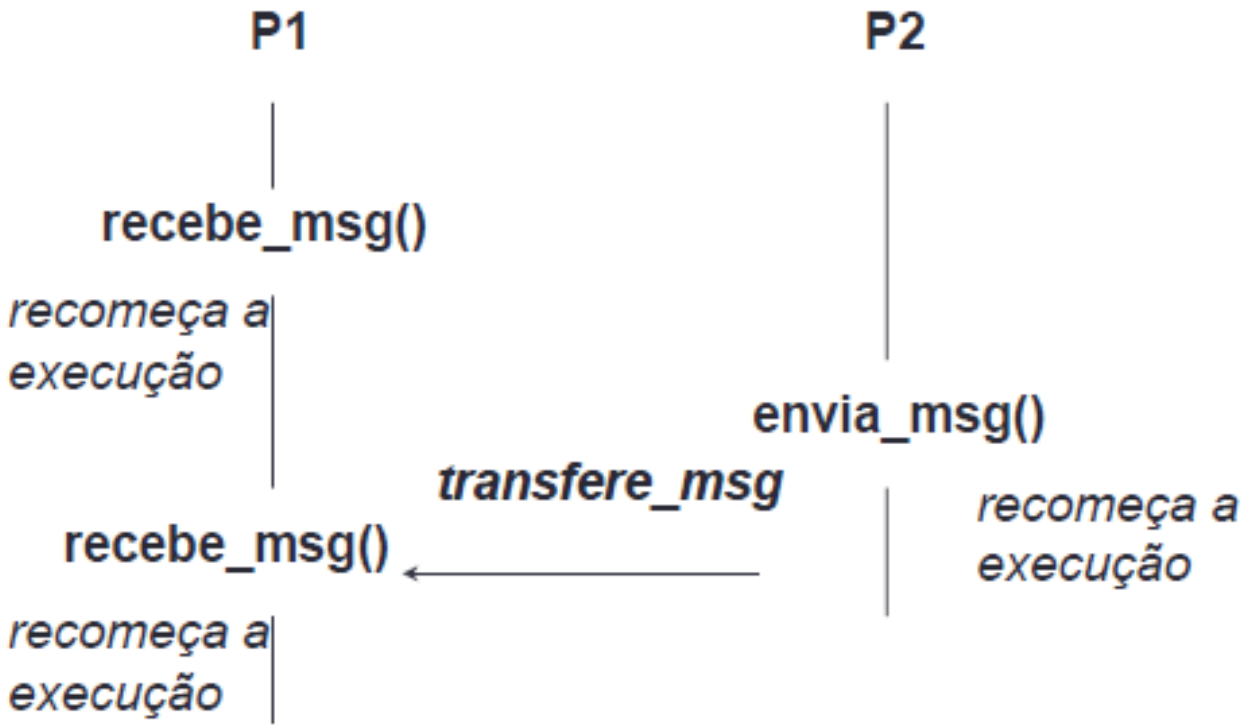
Troca de Mensagens

- Primitivas Não-bloqueantes:



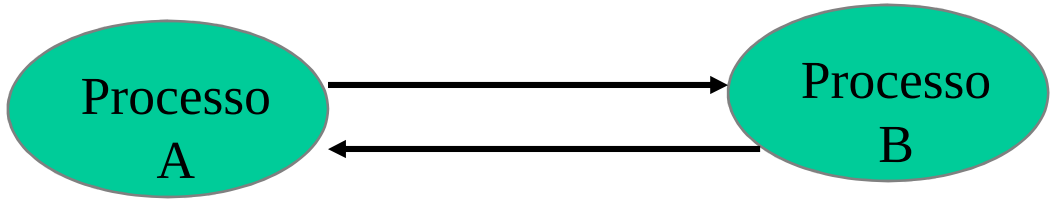
Troca de Mensagens

- Primitivas Não-bloqueantes:



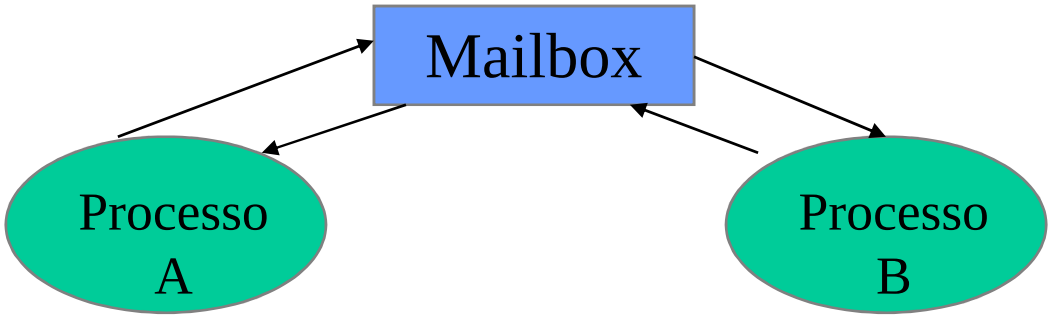
Troca de Mensagens

- A comunicação entre processos pode ser feita através de endereçamento direto ou indireto.
- No **endereçamento direto** o processo que deseja enviar ou receber uma mensagem deve endereçar explicitamente o nome do processo receptor ou transmissor.



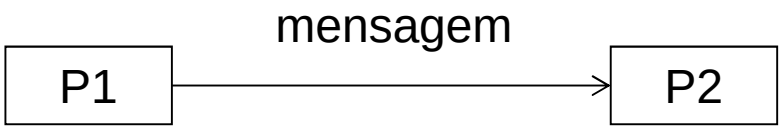
Troca de Mensagens

- O **endereçamento indireto** utiliza uma área compartilhada, na qual as mensagens podem ser colocadas pelo processo transmissor e retiradas pelo receptor.
- Esse tipo de *buffer* é conhecido como ***mailbox.***
- No endereçamento indireto, vários processos podem estar associados ao *mailbox*, e os parâmetros dos procedimentos SEND e RECEIVE passam a ser nomes de *mailboxes* e não mais nomes de processos.



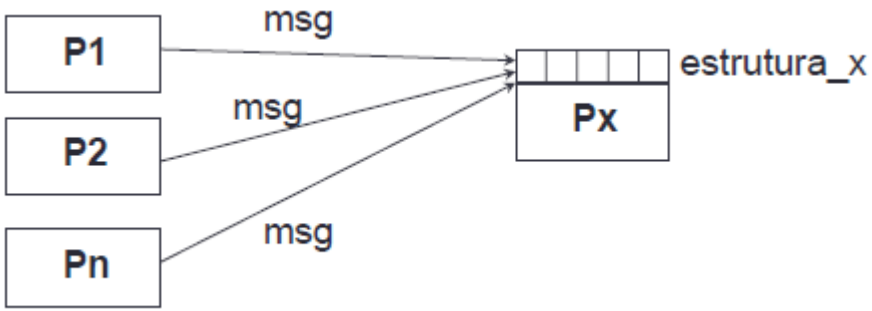
Troca de Mensagens

- Processos envolvidos na comunicação:
  - **1 para 1**: O processo *x* envia uma mensagem ao processo *y*. O processo *y* recebe somente a mensagem do processo *x*.



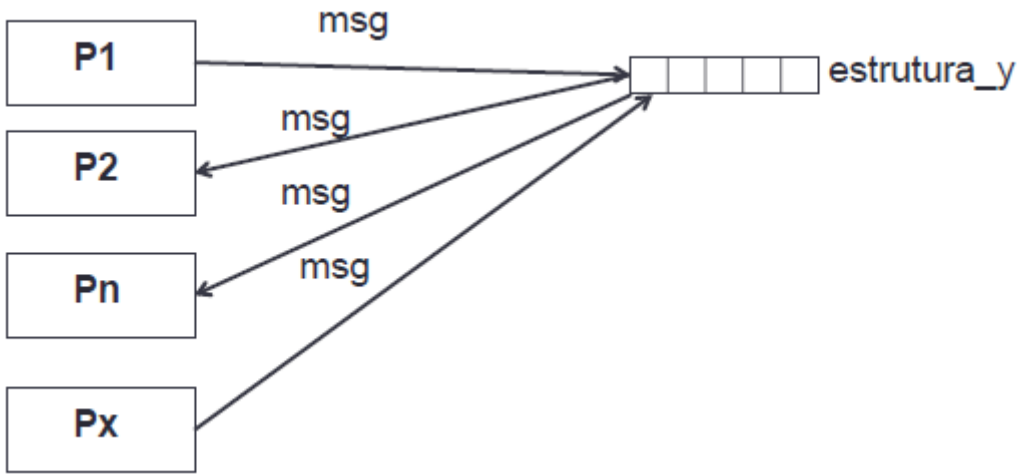
Troca de Mensagens

- Processos envolvidos na comunicação:
  - **n para 1**: O processo *x* envia uma mensagem ao processo *y*. O processo *y* recebe mensagem de qualquer processo.
  - Mecanismos que implementam a comunicação n para 1: portas, caixas-postais restritas.



Troca de Mensagens

- Processos envolvidos na comunicação:
  - **n para n**: O processo *x* envia uma mensagem a qualquer processo. O processo *y* recebe mensagem de qualquer processo.
  - Mecanismos que implementam a comunicação n para n: caixas-postais genéricas.



Exemplo Produtor-consumidor com Troca de Mensagem

- Neste exemplo, vamos utilizar primitivas síncronas 1 a 1.

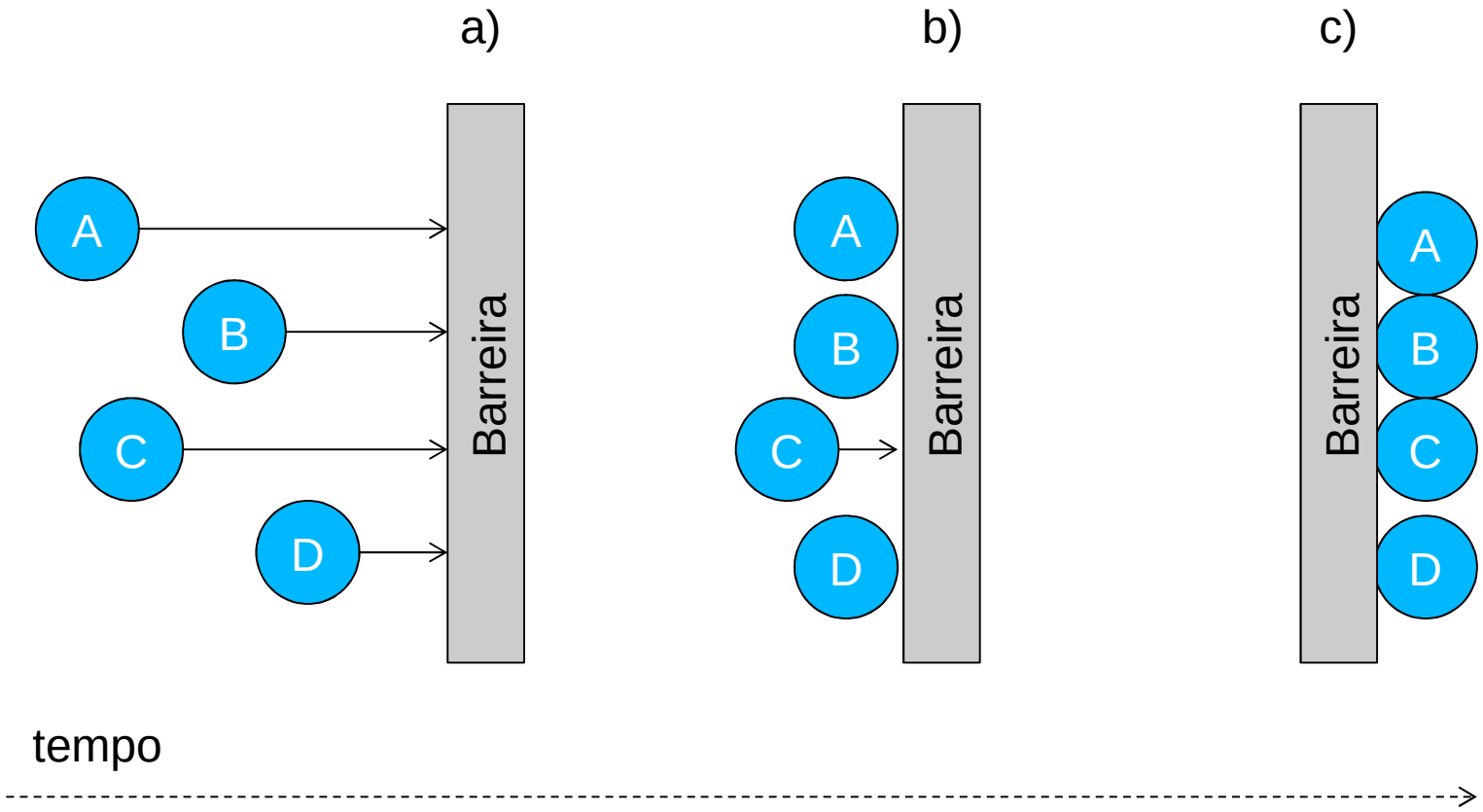
```
produtor()
{
  while(TRUE)
  {
     produz_item(&item);
     send(p2, &item);
  }
}
```

```
consumidor()
{
  while(TRUE)
  {
     receive(p1, &item);
     consome_item(item);
  }
}
```

Barreiras

- Este mecanismo de sincronização é dirigido aos grupos de processos, em vez de situações que envolvem apenas dois processos do tipo produtor-consumidor;
- Isso é importante porque algumas aplicações são divididas em fases, e têm como regra que nenhum processo pode avançar para a próxima fase até que todos os processos estejam prontos a fazê-lo;
- Isso pode ser conseguido por meio da alocação de uma **barreira** no final de cada fase, a qual é implementada por meio de uma primitiva *barrier()*;
- Quando alcança a barreira, um processo fica bloqueado até que todos os processos alcancem a barreira.

Barreiras



Referências Bibliográficas

[1] Tanenbaum, A. S., Woodhull, A. S. **Sistemas Operacionais: projeto e implementação**. 2ª ed. Porto Alegre: Bookman, 2000.

[2] Stallings, William. **Operating Systems: internals and Design Principles**. 4ª ed. New Jersey: Prentice Hall. 2001.

[3] Silberschatz, A., Galvin, P. B., Gagne, G. **Operating System Concepts**. 6ª ed. Editora John Wiley & Sons, Inc. 2002.

[4] SHAY, William A., Sistemas Operacionais. Makron Books. São Paulo. 1996.

[5] Deitel, Deitel, Choffnes – Sistemas Operacionais . Prentice Hall – São Paulo 3a Edição, 2005.