

# Processos Concorrentes

Programação Concorrente

Prof. Ilo Rivero

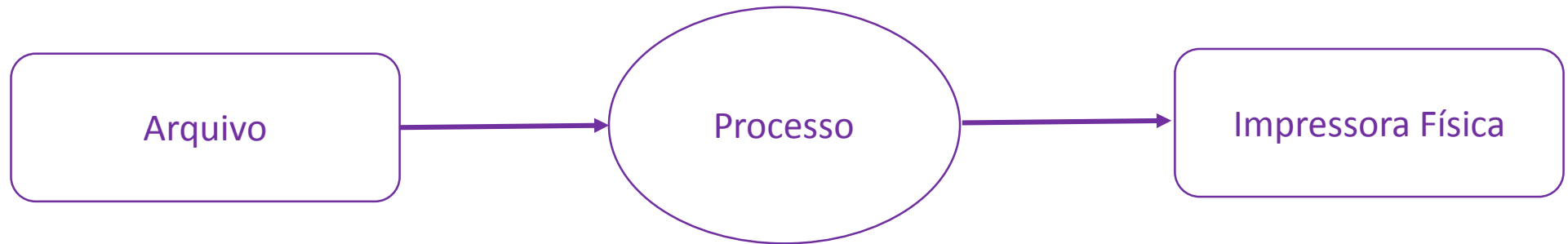
[ilorivero@live.com](mailto:ilorivero@live.com)

# Processos Concorrentes

- Um programa concorrente é executado simultaneamente por diversos processos
- Esses processos cooperam entre si, trocando informações
- Concorrente, aqui, significa “acontecendo ao mesmo tempo”
- A programação concorrente pode apresentar todos os tipos de erros da programação sequencial e ainda os erros associados com as operações entre processos

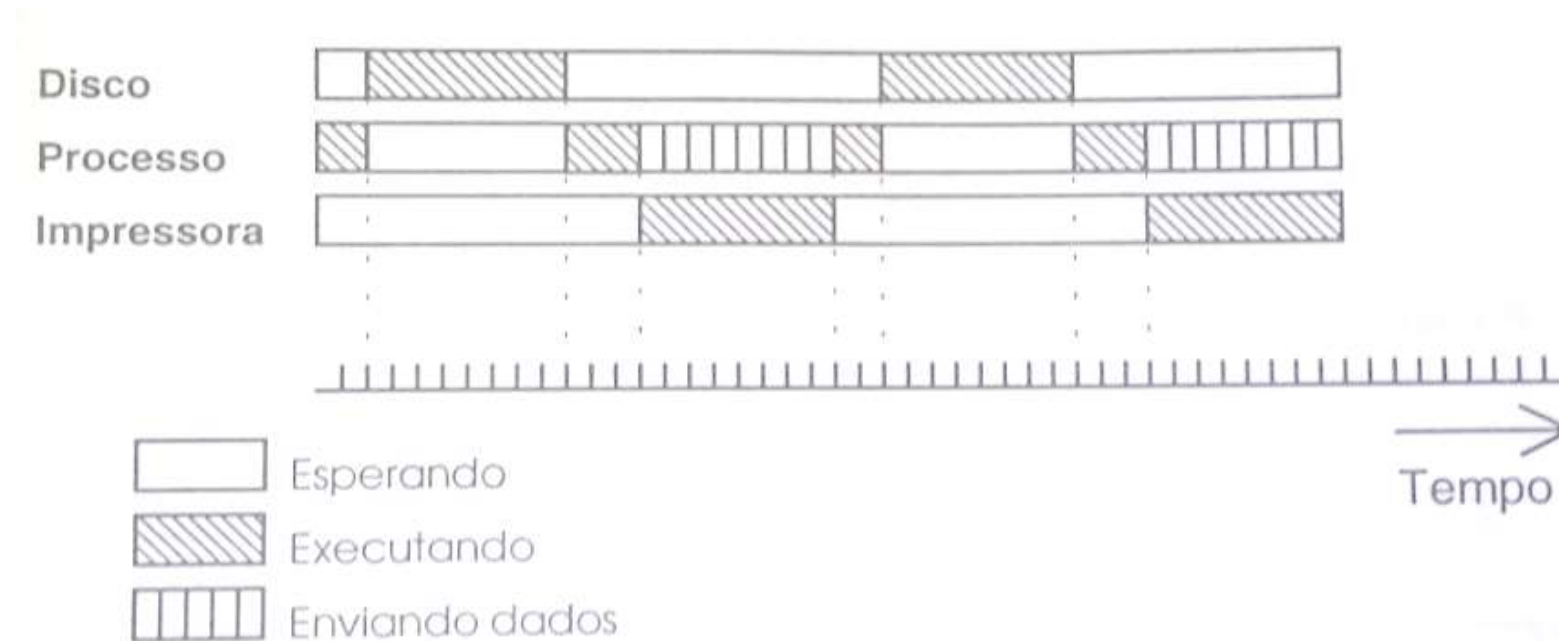
# Processos Concorrentes

- Exemplo:
  - Programa de Impressão Sequencial



# Processos Concorrentes

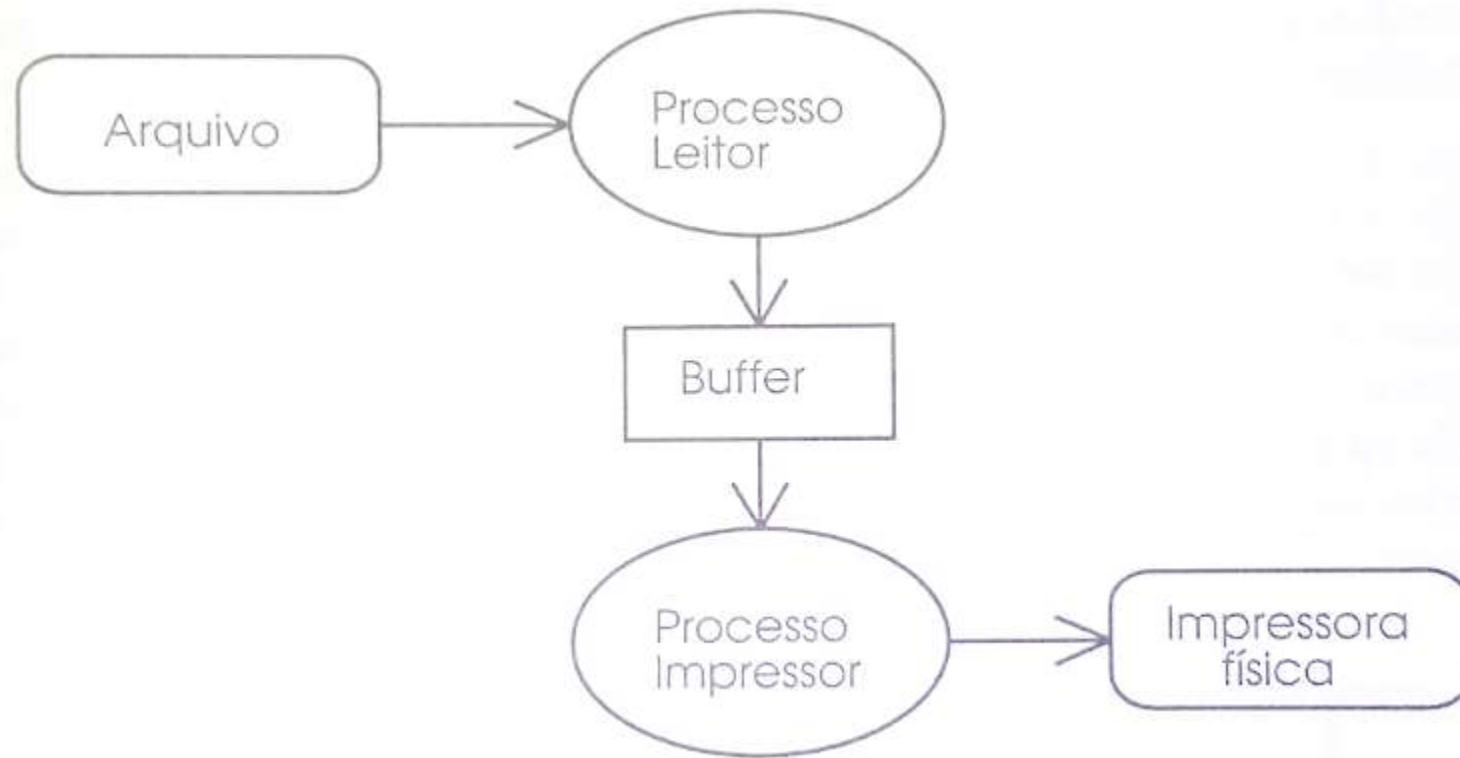
- Programa Sequencial: O processo de impressão fica preso até o término da impressão



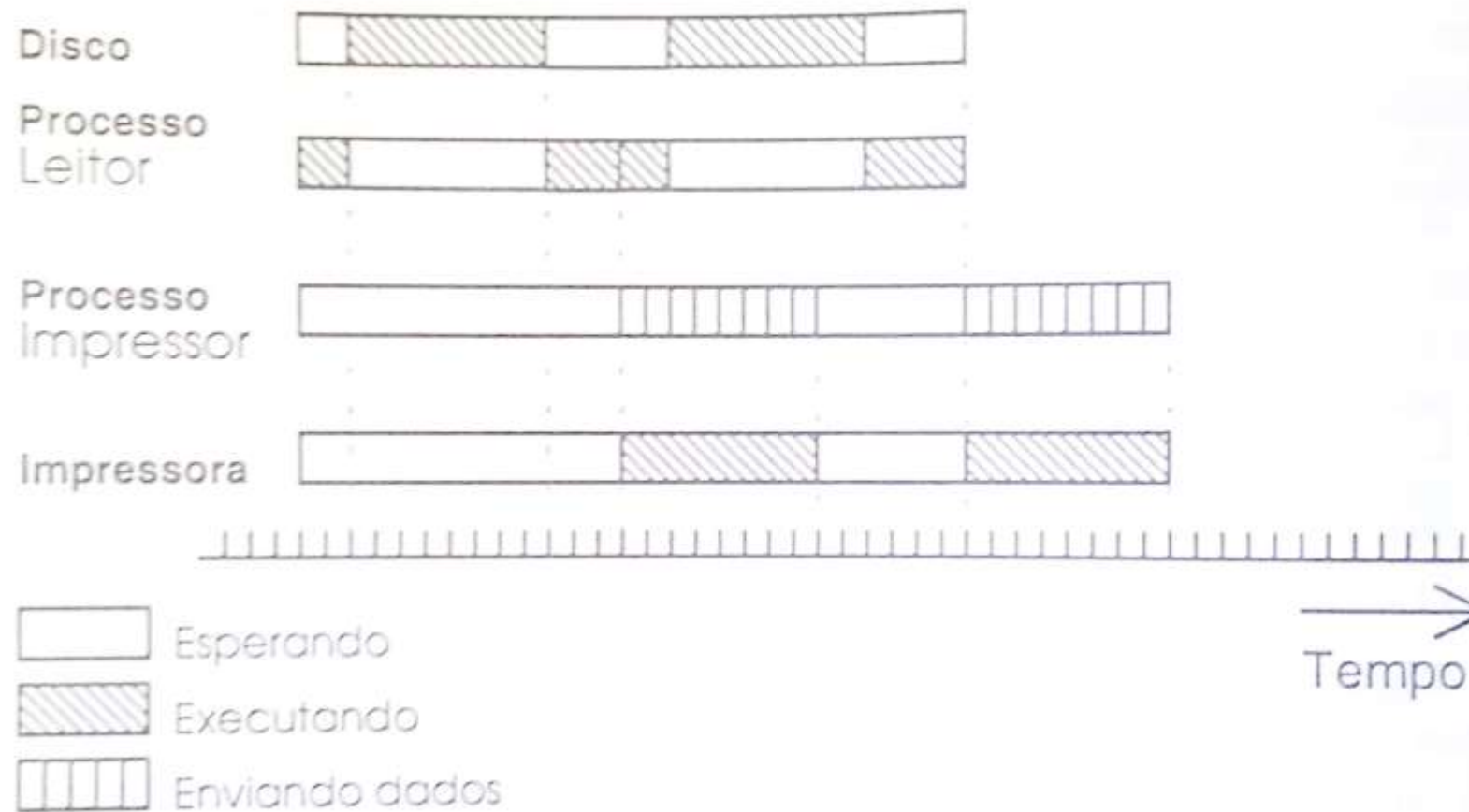
# Processos Concorrentes

- Se a aplicação de impressão for modificada para utilizar processos concorrentes, a aplicação ficará mais eficiente, pois conseguiria manter o disco e a impressora trabalhando simultaneamente
- Dois processos podem dividir o trabalho:
  - Um processo para ler arquivos em disco e colocar em um buffer
  - Outro processo retira os dados do buffer e os envia para a impressora

# Processos Concorrentes



# Processos Concorrentes

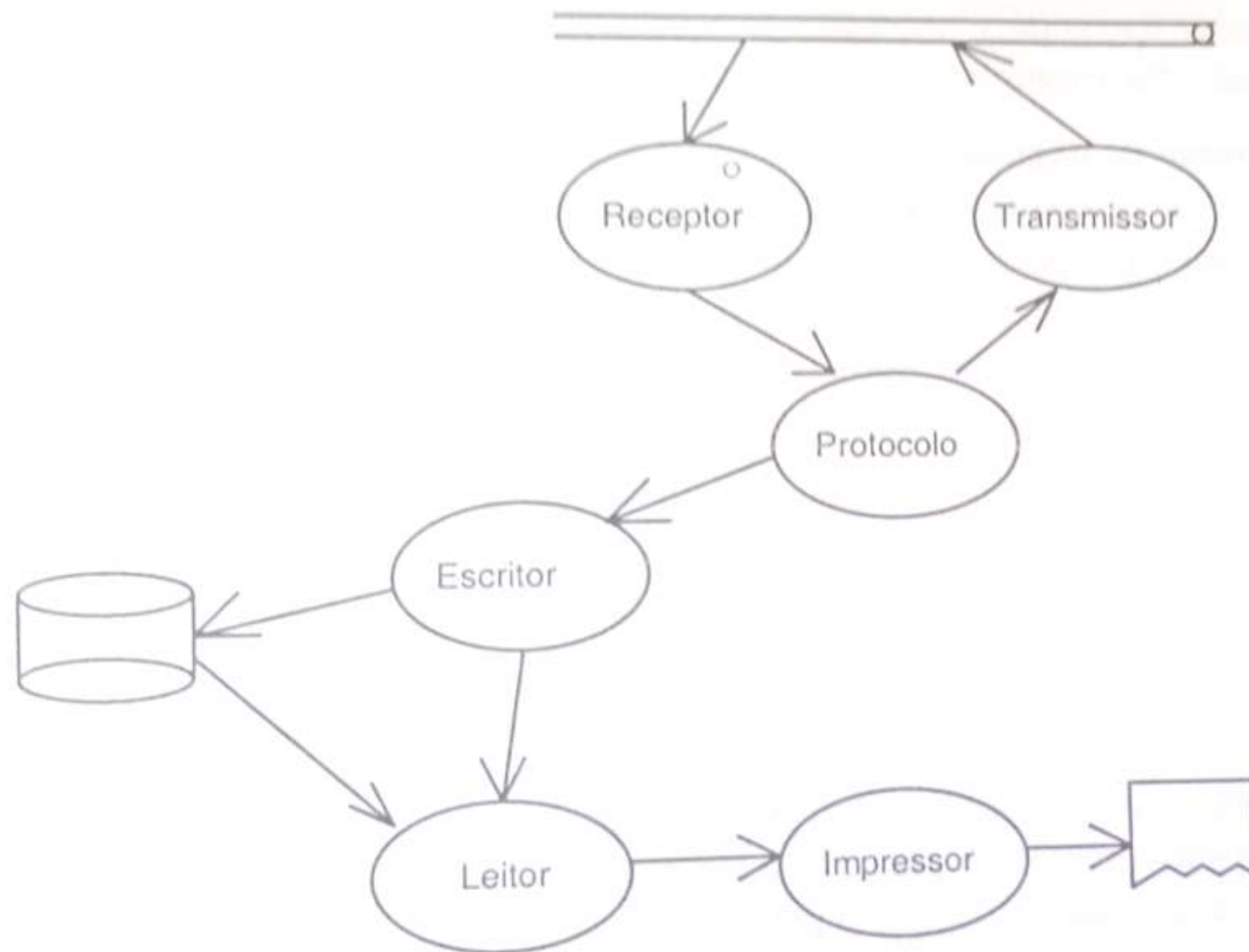


# Processos Concorrentes

- Proposta para um servidor de impressão
  - Um processo receptor recebe as mensagens da rede local e um processo transmissor as envia para a rede
  - Um processo de protocolo analisa as mensagens do receptor e as encaminha para o processo escritor ou gera mensagens de resposta e as envia ao processo transmissor
  - O processo escritor recebe a informação do processo de protocolo e as envia ao disco ou pode encaminhar diretamente para um processo leitor
  - O processo leitor captura as informações do disco ou do processo escritor e as envia para o processo impressor



# Processos Concorrentes



# Sincronismo e 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: sequência adequada;

# Sincronismo e Comunicação de Processos – *Race Conditions*

- *Race Conditions*: processos competem (disputam) por mesmos recursos;
- Memória compartilhada, arquivos compartilhados, impressora compartilhada;

# Sincronismo e Comunicação de Processos – *Race Conditions*

Considere 2 processos que compartilham as variáveis A e B:

P<sub>1</sub>

A = 1

P<sub>2</sub>

B = 2

Qual será o resultado final?

A ordem de execução dos processos tem importância?

Agora considere:

P<sub>1</sub>

A = B + 1

P<sub>2</sub>

B = 2 \* B

Qual é o resultado final?

E no seguinte caso?

P<sub>1</sub>

A = 1

P<sub>2</sub>

A = 2

Qual é o resultado final?

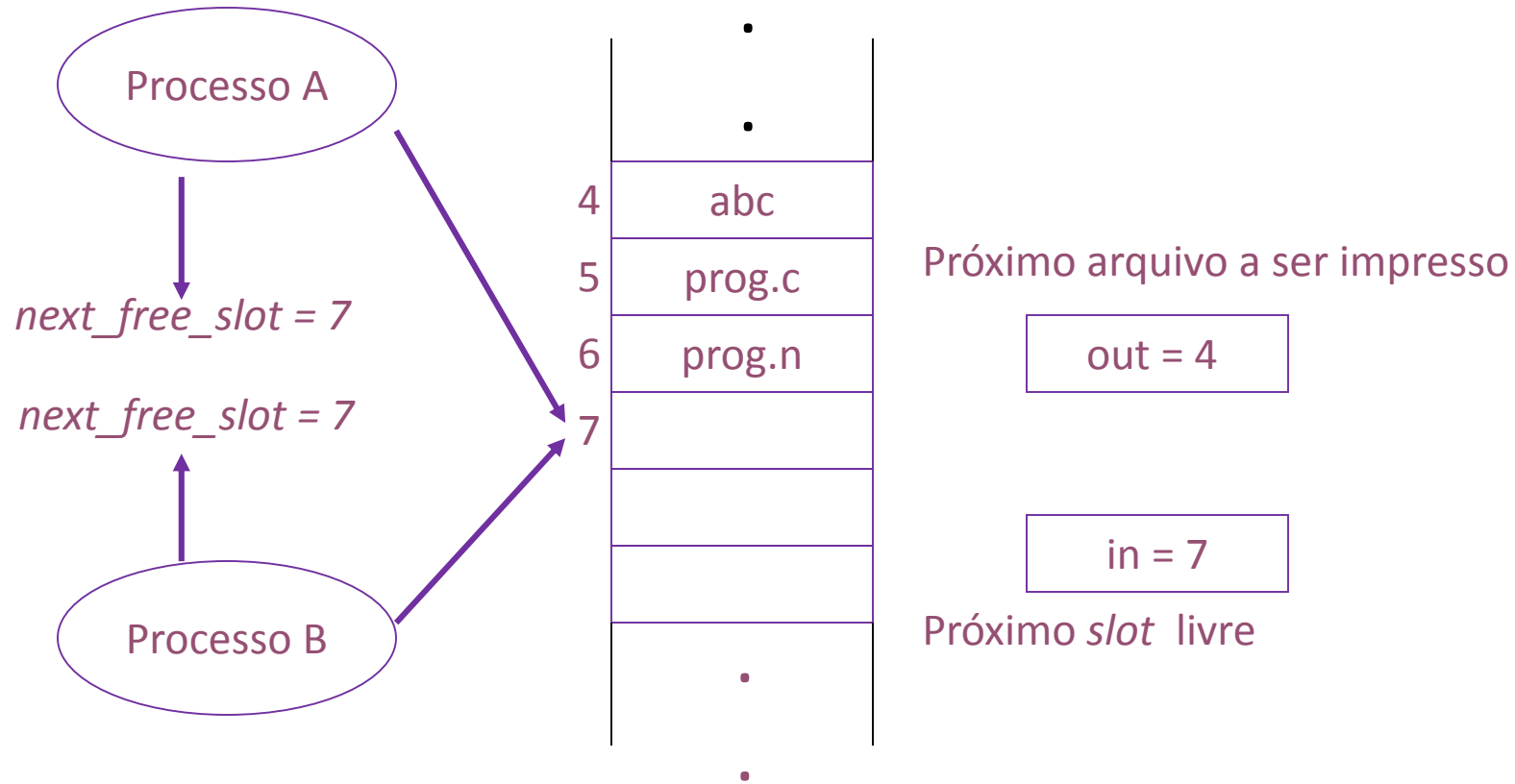
E em um computador com múltiplos processadores?

# Sincronismo e Comunicação de Processos – *Race Conditions*

- Continuando com o exemplo do servidor de impressão:
  - Quando um processo deseja imprimir um arquivo, ele coloca o arquivo em um local especial chamado **spooler** (tabela).
  - Um outro processo, chamado **printer spooler** checa se existe algum arquivo a ser impresso.
  - Se existe, esse arquivo é impresso e retirado do *spooler*. Imagine dois processos que desejam ao mesmo tempo imprimir um arquivo...

# Sincronismo e Comunicação de Processos - *Race Conditions*

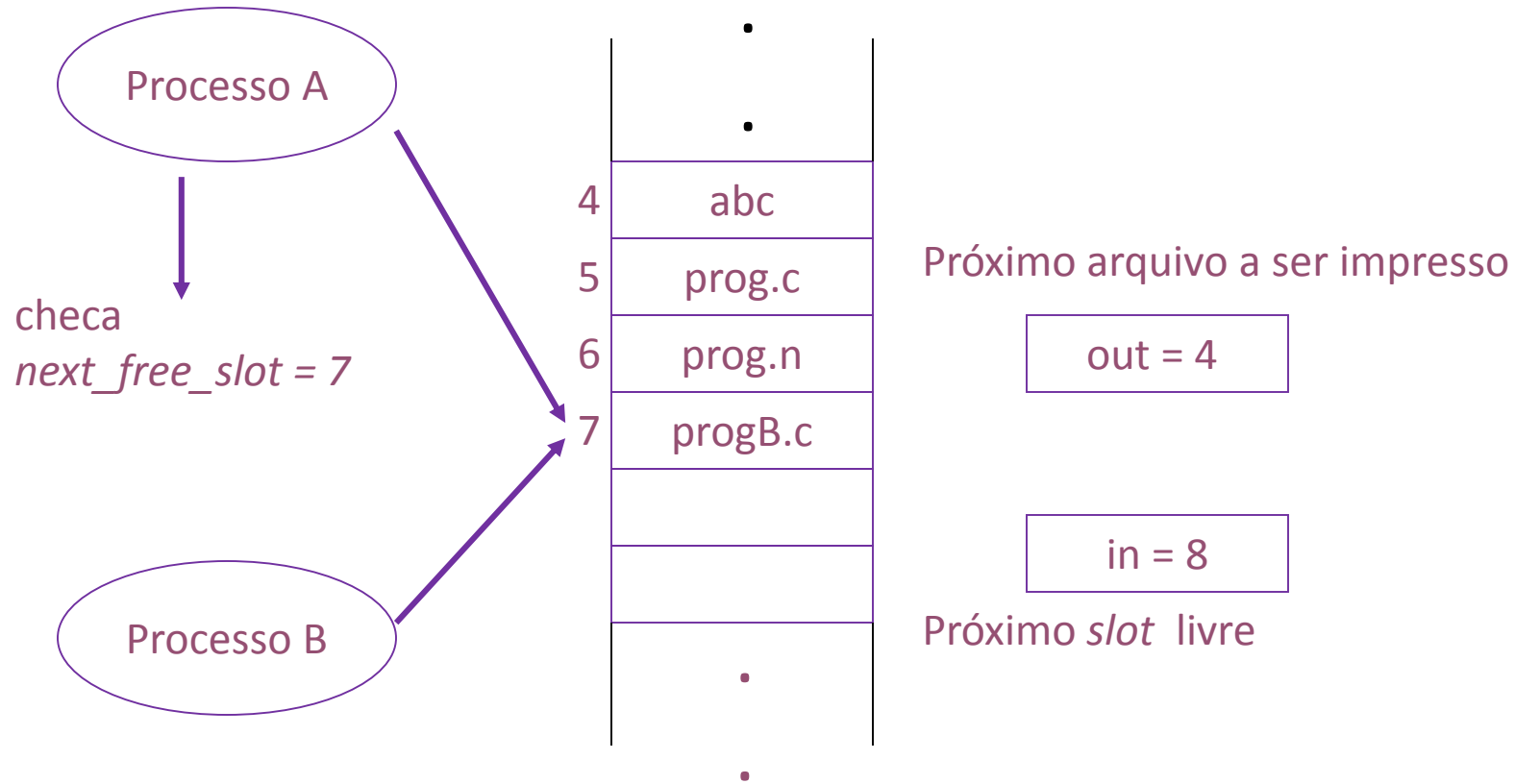
*Spooler – fila de impressão (slots)*



Coloca seu arquivo no slot 7 e  
 $next\_free\_slot = 8$

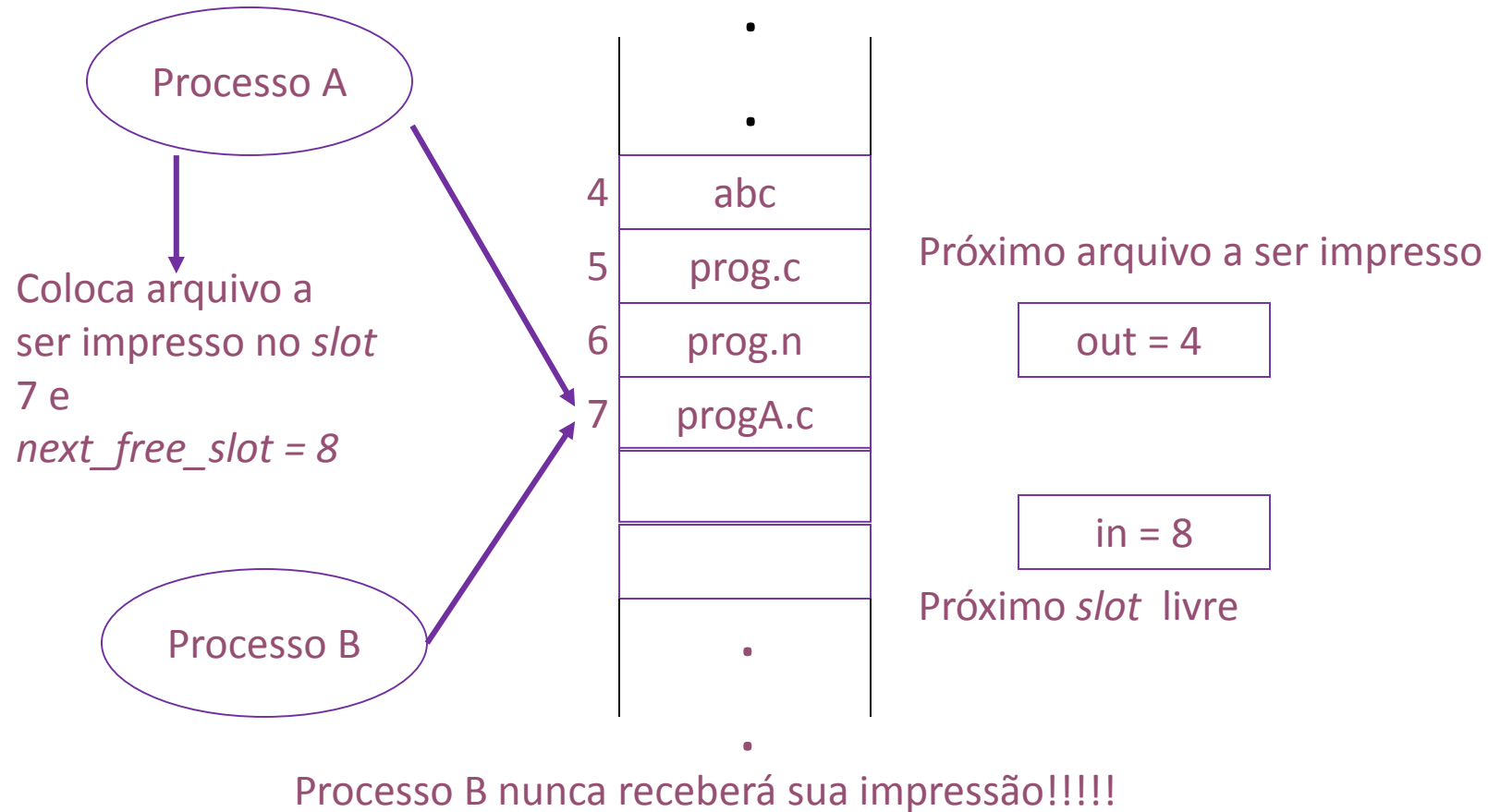
# Sincronismo e Comunicação de Processos - *Race Conditions*

*Spooler – fila de impressão (slots)*



# Sincronismo e Comunicação de Processos - *Race Conditions*

*Spooler – fila de impressão (slots)*





# Sincronismo e Comunicação de Processos – Regiões Críticas

- Como solucionar problemas de *Race Conditions*???
- Proibir que mais de um processo leia ou escreva em recursos compartilhados ao mesmo tempo – **regiões críticas**;
- Exclusão mútua: garantir que um processo não terá acesso à uma região crítica quando outro processo está utilizando essa região;

# Sincronismo e Comunicação de Processos – Regiões Críticas

## Operações Atômicas

A fim de evitar *race conditions* o conceito de **operações atômicas** é introduzido:

**Operações atômicas** são operações que não podem ser interrompidas:

- Não é possível ver as “partes” de uma operação atômica, mas apenas seu efeito final. Ou seja, não é possível ver uma “operação em progresso”

### **Atômicas**

tocar a campainha  
desligar a luz

### **Não-Atômicas**

encher um copo de água  
caminhar até a porta

# Sincronismo e Comunicação de Processos – Regiões Críticas

## Operações Atômicas

Operações atômicas são relevantes em outras áreas além dos Sistemas Operacionais:

- Elas são a base para **transações atômicas** que, por sua vez, formam uma base para uma área denominada **Processamento de Transações**.
- Esta área trata de problemas de coordenação de acessos múltiplos e concorrentes a bancos de dados:
  - Bancos eletrônicos são uma das aplicações importantes desta área

# Sincronismo e Comunicação de Processos – Regiões Críticas

## Operações Atômicas

Operações atômicas são relevantes em outras áreas além dos Sistemas Operacionais:

- Elas são a base para **transações atômicas** que, por sua vez, formam uma base para uma área denominada **Processamento de Transações**.
- Esta área trata de problemas de coordenação de acessos múltiplos e concorrentes a bancos de dados:
  - Bancos eletrônicos são uma das aplicações importantes desta área

# Sincronização

## O problema do espaço na geladeira

<b>Hora</b>	<b>Pessoa A</b>	<b>Pessoa B</b>
6:00	Olha a gel.: sem leite	...
6:05	Sai para a padaria	...
6:10	Chega na padaria	Olha a gel.: sem leite
6:15	Sai da padaria	Sai para a padaria
6:20	Em casa: guarda leite	Chega na padaria
6:25	...	Sai da padaria
6:30	...	Chega em casa: Ops!

O que houve de errado?

- Problema: falta de **comunicação**

# Sincronização

O problema anterior era causado porque uma pessoa não sabia o que a outra estava fazendo.

Uma solução para o problema envolve dois novos conceitos:

- **Exclusão mútua:** apenas um processo pode fazer alguma coisa em determinado momento.
  - Exemplo: apenas uma pessoa pode sair para comprar leite em qualquer momento
- **Seção crítica:** uma seção de código na qual apenas um processo pode executar de cada vez,
  - O objetivo é tornar **atômico** o conjunto de operações
  - Exemplo: comprar leite

# Sincronização – Exclusão Mútua

Existem várias maneiras de se obter exclusão mútua:

A maioria envolve **trancamento** (*locking*):

- Evitar que alguém faça alguma coisa em determinado momento.

Exemplo: deixar um aviso na porta da geladeira.

Três regras devem ser satisfeitas para o trancamento funcionar:

## **Regra**

1. Trancar antes de utilizar
2. Destrancar quando terminar
3. Esperar se estiver trancado

## **Exemplo da geladeira**

Deixar aviso

Retirar o aviso

Não sai para comprar se  
houver aviso

# Exclusão Mútua

Primeira tentativa de resolver o *Problema do Espaço na Geladeira*:

## **Processos A e B**

```
if (SemLeite) {  
    if (SemAviso) {  
        Deixa Aviso;  
        Compra Leite;  
        Remove Aviso;  
    }  
}
```

Esta “solução” funciona?

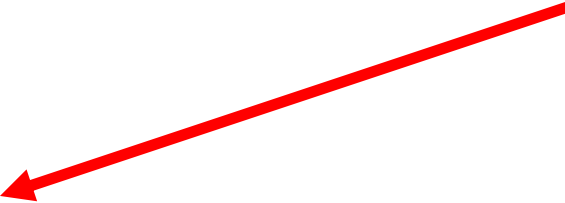


# Exclusão Mútua

Não, por causa da **troca de contexto**.

## Processos A e B

```
if (SemLeite) {  
    if (SemAviso) {  
        Deixa Aviso;  
        Compra Leite;  
        Remove Aviso;  
    }  
}
```



A “solução” piora o problema! Agora, falha só de vez em quando, ou seja, a depuração fica muito mais difícil.

- **Heisenbug**
- Não importa se é raro, na prática vai acontecer nos primeiros 5 minutos. A não ser quando vc estiver querendo que aconteça!

# Exclusão Mútua

Segunda tentativa de resolver o *Problema do Espaço na Geladeira* –  
mudar o significado do aviso:

## **Processo A**

```
if (SemAviso) {  
    if (SemLeite) {  
        Compra Leite;  
    }  
    Deixa Aviso;  
}
```

## **Processo B**

```
if (Aviso) {  
    if (SemLeite) {  
        Compra Leite;  
    }  
    Remove Aviso;  
}
```

Funciona? Porque?

# Exclusão Mútua

Que tal o seguinte argumento?

- Somente A deixa um aviso, e somente se já não existe um aviso;
- Somente B remove um aviso, e somente se houver um aviso;
- Portanto, ou existe um aviso, ou nenhum;
- Se houver aviso, B compra leite;
- Se não houver aviso, A compra leite;
- Portanto, apenas uma pessoa (processo) vai comprar leite.

Certo?

# Exclusão Mútua

Suponha que B saia de férias:

- A vai comprar leite uma vez e não vai comprar mais até que B retorne.
- Portanto, esta solução não é boa; em particular, ela pode levar a uma **inanição** (*starvation*).

(Tá bom, trapaça. Eu nunca disse nada sobre inanição.  
Mas não é importante?)

# Exclusão Mútua

Terceira tentativa – usar dois avisos diferentes:

## **Processo A**

```
Deixa AvisoA;  
while (AvisoB);  
if (SemLeite) {  
    Compra Leite;  
}  
Remove AvisoA;
```

## **Processo B**

```
Deixa AvisoB;  
if (SemAvisoA) {  
    if (SemLeite) {  
        Compra Leite;  
    }  
}  
Remove AvisoB;
```

Funciona?

# Exclusão Mútua

SIM!

## Processo A

```
Deixa AvisoA;  
while (AvisoB);  
if (SemLeite) {  
    Compra Leite;  
}  
Remove AvisoA;
```

## Processo B

```
Deixa AvisoB;  
if (SemAvisoA) {  
    if (SemLeite) {  
        Compra Leite;  
    }  
}  
Remove AvisoB;
```

- Se SemAvisoA B pode comprar porque A ainda não começou.
- Se AvisoA, A está comprando ou esperando até que B desista. B pode desistir.
- Se SemAvisoB A pode comprar.
- Se AvisoB, não se sabe:
  - Se B comprar, remove AvisoB, fim.
  - Se B não comprar, remove AvisoB, A pode comprar

# Exclusão Mútua

Esta solução funciona, mas não é boa (cara chato!):

- Muito complicado. Difícil de entender e se convencer de que está correto.
- Código de A é diferente do do B. E se houver mais de dois processos?
- Enquanto A está esperando, está consumindo CPU (*busy waiting* – espera ocupada)

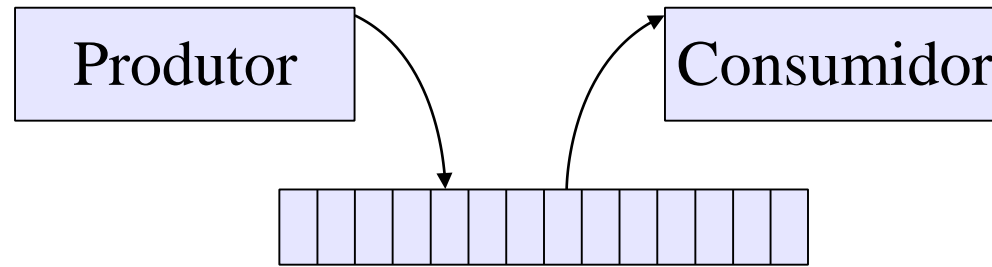
# Exclusão Mútua

Pontos importantes:

- Comportamento **muito** sutil. Difícil de programar e entender.
- Como provar que está correto?
- Quais os critérios para uma boa solução?



# Produtor & Consumidor



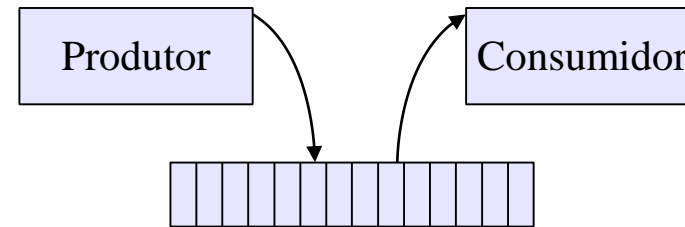
- Produtor gera itens continuamente e os coloca no *buffer*.
- Consumidor usa itens, lendo-os do *buffer*.
- *Buffer* é necessário por causa da velocidade relativa entre produtor e consumidor.
- Sincronização é necessária para acesso ao *buffer*:
  - Produtor não pode colocar mais itens em *buffer* cheio.
  - Consumidor não pode ler itens de *buffer* vazio.

# Produtor & Consumidor

Solução “ideal” – usa todas as posições do *buffer*:

```
Produtor() {  
    while (true) {  
        while (counter == n);  
        buffer[in]= item produzido;  
        in= in+1 mod n;  
        counter++;  
    }  
}  
Consumidor() {  
    while (true) {  
        while (counter == 0);  
        item consumido= buffer[out];  
        out= out+1 mod n;  
        counter--;  
    }  
}
```

- *Buffer* circular



# Produtor & Consumidor

```
Produtor() {  
    while (true) {  
        while (counter == n);  
        buffer[in]= item produzido;  
        in= in+1 mod n;  
        R1= counter;    INC(R1);    counter= R1;  
    }  
}  
Consumidor() {  
    while (true) {  
        while (counter == 0);  
        item consumido= buffer[out];  
        out= out+1 mod n;  
        R1= counter;    DEC(R1);    counter= R1;  
    }  
}
```

# Produtor & Consumidor

Solução não ideal – não usa todas as posições do *buffer*:

```
Produtor() {  
    while (true) {  
        while (in+1 mod n == out);  
        buffer[in]= item produzido;  
        in= in+1 mod n;  
    }  
}
```

```
Consumidor() {  
    while (true) {  
        while (in == out);  
        item consumido= buffer[out];  
        out= out+1 mod n;  
    }  
}
```

Como escrever uma solução correta que usa  $n$  posições do *buffer*?

# Produtor & Consumidor

Simples! Declare um *buffer* com  $n+1$  posições:

- Uma posição de memória não vale o esforço de reescrever o programa!

# Sincronização – Requisitos

Requisitos para uma primitiva de exclusão mútua:

- Deve permitir apenas um processo dentro da região crítica a cada instante;
- Se várias requisições são feitas ao mesmo tempo, deve permitir que um processo prossiga;
- Processos podem “entrar de férias” somente fora das regiões críticas.

# Sincronização – Requisitos

Propriedades desejáveis de um mecanismo de exclusão mútua:

- Justiça (*fairness*): se vários processos estão esperando, dar acesso a todos, eventualmente;
- Eficiência: não utilizar quantidades substanciais de recursos quando estiver esperando. Em particular, evitar a *espera ocupada*;
- Simples: deve ser fácil de utilizar.

# Sincronização – Requisitos

Propriedades dos processos utilizando os mecanismos necessários para manter coerência:

- Trancar sempre antes de utilizar dado compartilhado;
- Destrancar sempre que terminar o uso do dado compartilhado;
- Não trancar de novo se já tiver trancado o recurso;
- Não ficar muito tempo dentro das seções críticas:

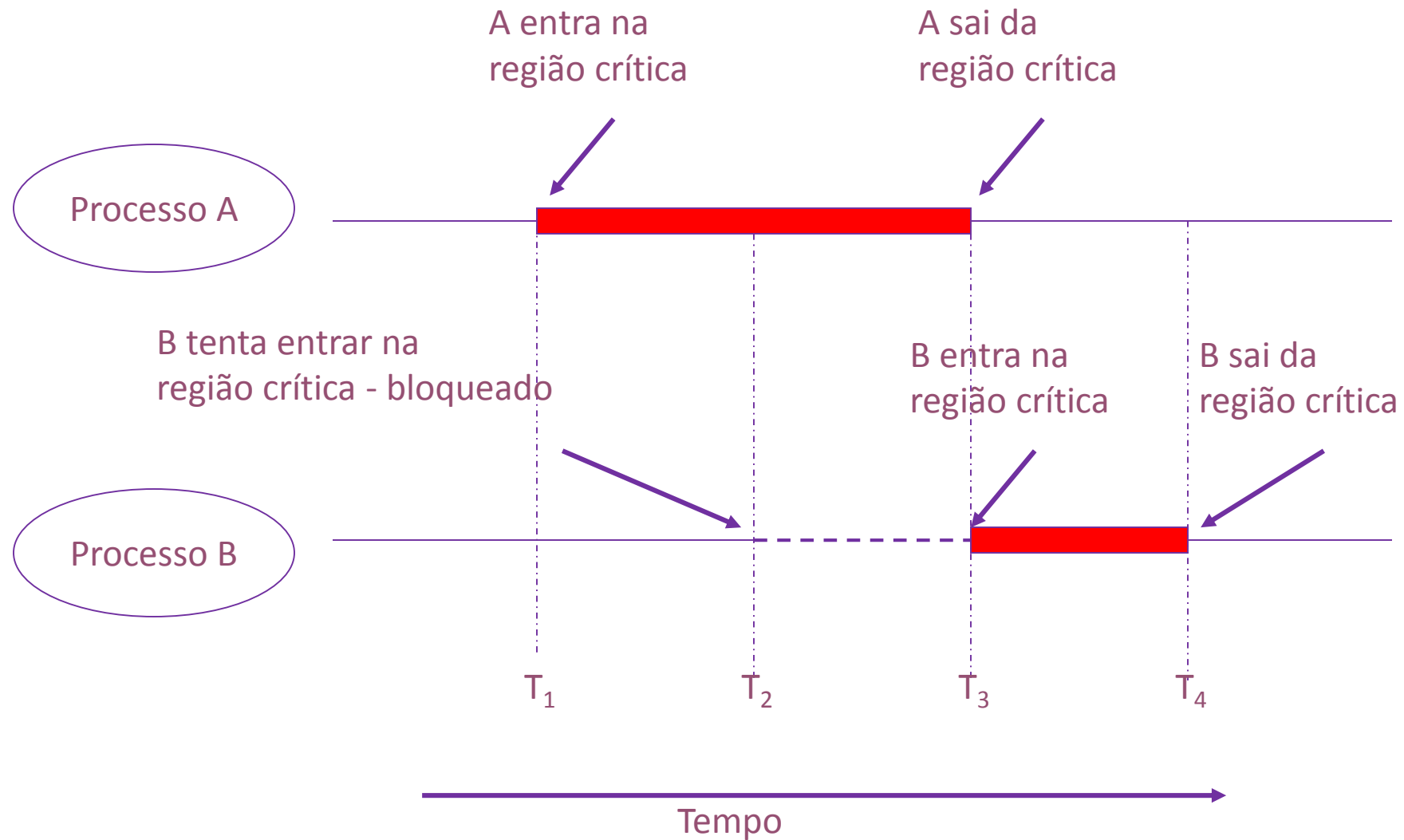
```
while (!fim) {  
    seção_não_crítica;  
    lock();  
    seção_crítica;  
    unlock();  
}
```



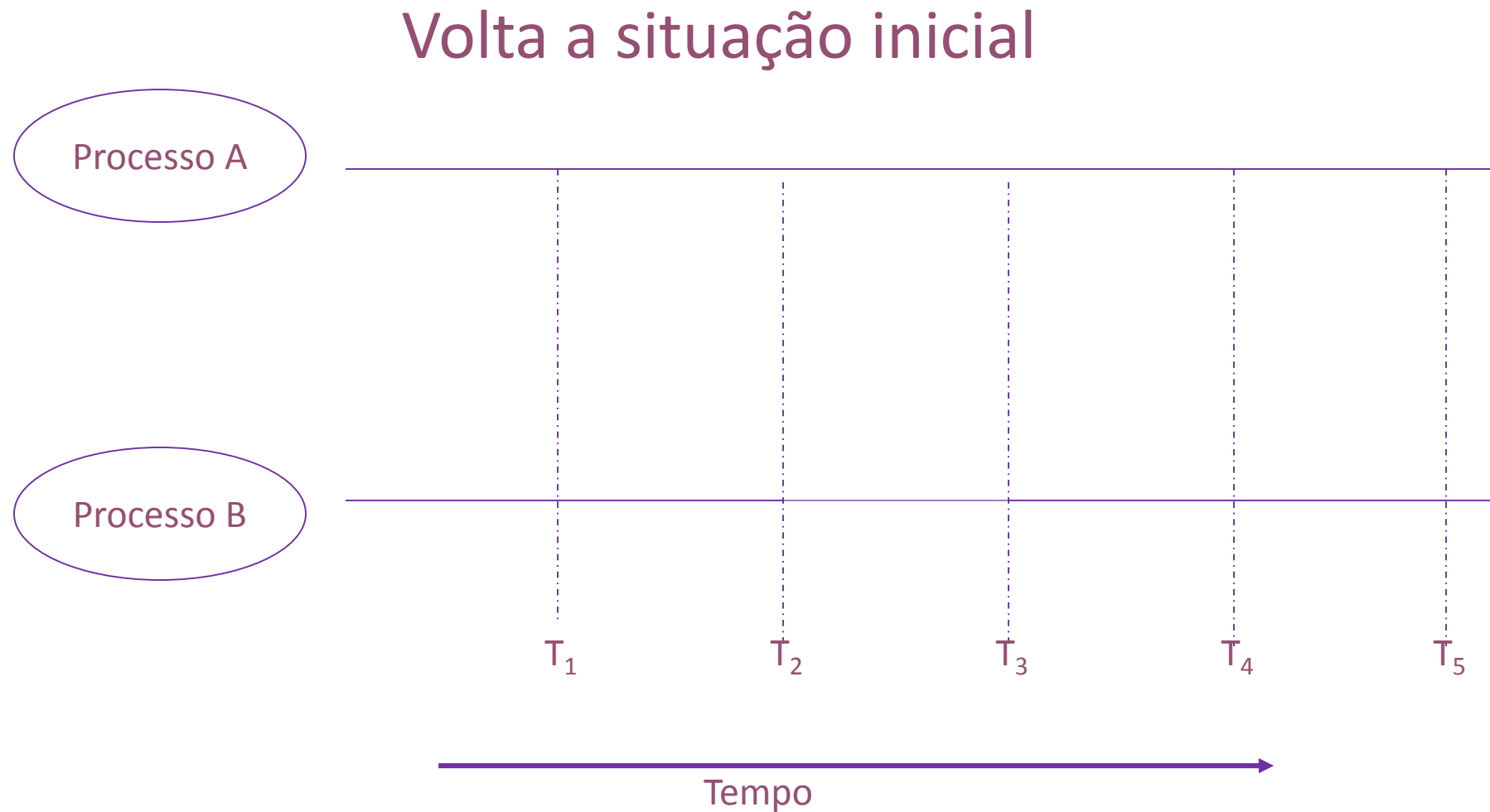
# Conclusão

- 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;
  4. Processos não podem esperar para sempre para acessarem regiões críticas;

# Exclusão Mútua



# Exclusão Mútua



# Soluções

- Exclusão Mútua:
  - **Espera Ocupada;**
  - Primitivas *Sleep/Wakeup*;
  - Semáforos;
  - Monitores;
  - Passagem de Mensagem;

# Exclusão Mútua

- Espera Ocupada (*Busy Waiting*): constante checagem por algum valor;
- Algumas soluções para Exclusão Mútua com Espera Ocupada:
  - Desabilitar interrupções;
  - Variáveis de Travamento (*Lock*);
  - Estrita Alternância (*Strict Alternation*);
  - Solução de Peterson e Instrução TSL;

# Exclusão Mútua

- 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;
  - Não é uma solução segura, pois um processo pode não habilitar novamente suas interrupções e não ser finalizado;

# Exclusão Mútua

- 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

- *Strict Alternation*:
  - Fragmentos de programa controlam o acesso às regiões críticas;
  - Variável `turn`, inicialmente em 0, estabelece qual processo pode entrar na região crítica;

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

(Processo A)

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

(Processo B)



# Exclusão Mútua

- Problema do *Strict Alternation*:
  1. Processo B é mais rápido e sai da região crítica (neste caso);
  2. Ambos os processos estão fora da região crítica e `turn` com valor 0;
  3. O processo A termina antes de executar sua região não crítica e retorna ao início do *loop*;  
Como o `turn` está com valor zero, o processo A entra novamente na região crítica, enquanto o processo B ainda está na região não crítica;
  4. Ao sair da região crítica, o processo A atribui o valor 1 à variável `turn` e entra na sua região não crítica;
  5. Novamente ambos os processos estão na região não crítica e a variável `turn` está com valor 1;
  6. Quando o processo A tenta novamente entrar na região crítica, não consegue, pois `turn` ainda está com valor 1;
  7. **Assim, o processo A fica bloqueado pelo processo B que está na sua região não crítica (condição 3);**

# Exclusão Mútua

- Solução de Peterson e Instrução TSL (*Test and Set Lock*):
  - Uma variável 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;

# Exclusão Mútua

- Instrução TSL:
  - Utiliza os registradores do hardware;
  - TSL RX, LOCK;
  - *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
```

# Soluções

- Exclusão Mútua:
  - Espera Ocupada;
  - **Primitivas *Sleep/Wakeup***;
  - Semáforos;
  - Monitores;
  - Passagem de Mensagem;

# *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;

# *Sleep/Wakeup*

- Para solucionar esse problema de espera, um par de primitivas *Sleep* e *Wakeup* é utilizado → BLOQUEIO E DESBLOQUEIO de processos.
- 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*;

# *Sleep/Wakeup*

- Problemas que podem ser solucionados com o uso dessas primitivas:
  - Problema do Produtor/Consumidor (*bounded buffer*): 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;

# *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.



# *Sleep/Wakeup*

- 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 com 0, o processo vai “dormir”, senão ele retira os dados do *buffer* e decrementa a variável;

# 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);
    }
}
```

# *Sleep/Wakeup*

- Problemas desta solução: Acesso à variável `count` é irrestrita
  - O *buffer* está vazio e o consumidor acabou de checar a variável `count` com valor 0;
  - O escalonador (por meio de uma interrupção) decide que o processo produtor será executado;
    - Então o processo produtor insere um item no *buffer* e incrementa a variável `count` com valor 1; Imaginando que o processo consumidor está dormindo, o processo produtor envia um sinal de *wakeup* para o consumidor;
  - No entanto, o processo consumidor não está dormindo, e não recebe o sinal de *wakeup*;

# *Sleep/Wakeup*

- Assim que o processo consumidor é executado novamente, a variável `count` já tem o valor zero; Nesse instante, o consumidor é colocado para dormir, pois acha que não existem informações a serem lidas no *buffer*;
  - Assim que o processo produtor acordar, ele insere outro item no *buffer* e volta a dormir. Ambos os processos dormem 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!!!!

# Soluções

- Exclusão Mútua:
  - Espera Ocupada;
  - Primitivas *Sleep/Wakeup*;
  - **Semáforos**;
  - Monitores;
  - Passagem de Mensagem;

# Semáforos

- Idealizados por E. W. Dijkstra (1965);
- Variável inteira que armazena o número de sinais *wakeups* enviados;
- Um semáforo pode ter valor 0 quando não há sinal armazenado ou um valor positivo referente ao número de sinais armazenados;
- Duas primitivas de chamadas de sistema: *down (sleep)* e *up (wake)*;
- Originalmente P (*down*) e V (*up*) em holandês;

# Semáforos

- *Down*: verifica se o valor do semáforo é maior do que 0; se for, o semáforo é decrementado; Se o valor for 0, o processo é colocado para dormir sem completar sua operação de *down*;
- Todas essas ações são chamadas de **ações atômicas**;
  - **Ações atômicas** garantem que quando uma operação no semáforo está sendo executada, nenhum processo pode acessar o semáforo até que a operação seja finalizada ou bloqueada;

# Semáforos

- $Up$ : incrementa o valor do semáforo, fazendo com que algum processo que esteja dormindo possa terminar de executar sua operação *down*;



# Semáforos

- Problema produtor/consumidor: resolve o problema de perda de sinais enviados;
- Solução utiliza três semáforos:
  - *Full*: conta o número de *slots* no *buffer* que estão ocupados; iniciado com 0; resolve sincronização;
  - *Empty*: conta o número de *slots* no *buffer* que estão vazios; iniciado com o número total de *slots* no *buffer*; resolve sincronização;
  - *Mutex*: garante que os processos produtor e consumidor não acessem o *buffer* ao mesmo tempo; iniciado com 1; também chamado de semáforo binário; Permite a exclusão mútua;

# Semáforos

```
# include "prototypes.h"
# define N 100
```

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

```
void producer (void){
    int item;
    while (TRUE){
        produce_item(&item);
        down(&empty);
        down(&mutex);
        enter_item(item);
        up(&mutex);
        up(&full);
    }
}
```

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

# Semáforos

- Problema: erro de programação pode gerar um *deadlock*;
  - Suponha que o código seja trocado no processo produtor;

..	..
down (&empty) ;	<b>down (&amp;mutex) ;</b>
<b>down (&amp;mutex) ;</b>	down (&empty) ;
..	..

- Se o *buffer* estiver cheio, o produtor será bloqueado com `mutex = 0`; Assim, a próxima vez que o consumidor tentar acessar o *buffer*, ele tenta executar um `down` sobre o `mutex`, ficando também bloqueado.

# Soluções

- Exclusão Mútua:
  - Espera Ocupada;
  - Primitivas *Sleep/Wakeup*;
  - Semáforos;
  - **Monitores**;
  - Passagem de Mensagem;

# Monitores

- Idealizado por Hoare (1974) e Brinch Hansen (1975)
- Monitor: primitiva de alto nível para sincronizar processos; é um conjunto de procedimentos, variáveis e estruturas de dados agrupados em um único módulo;
- Somente um processo pode estar ativo dentro do monitor em um mesmo instante; outros processos ficam bloqueados até que possam estar ativos no monitor;
- Alguns compiladores implementam a exclusão mútua por meio dos monitores
  - **Java**;

# Monitores

```
monitor example
  int i;
  condition c;

  procedure producer();
  .
  end;
  procedure consumer();
  .
  end;
end monitor;
```

Estrutura básica de um Monitor

# Monitores

- Execução:
  - Chamada a uma rotina do monitor;
  - Instruções iniciais: teste para detectar se um outro processo está ativo dentro do monitor;
  - Se positivo, o processo novo ficará bloqueado até que o outro processo deixe o monitor;
  - Caso contrário, o processo novo entra no monitor;

# Monitores

- *Condition Variables* → variáveis que indicam uma condição (*full* e *empty*)

+

- Operações básicas: *WAIT* e *SIGNAL*

*wait* (*condition*) → TRUE → bloqueado  
→ FALSE → executando

*signal* (*condition*) → “acorda” o processo que executou um *wait* na variável *condition* e foi bloqueado;



# Monitores

- Como evitar dois processos ativos no monitor ao mesmo tempo?
- (1) Hoare → colocar o processo mais novo para rodar; suspende-se o outro
- (2) B. Hansen → um processo que executa um `SIGNAL` deve deixar o monitor imediatamente;

A condição (2) é mais simples e mais fácil de se implementar.

# Monitores

```
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

- Limitações de semáforos e monitores:
  - Ambos são boas soluções somente para CPUs com memória compartilhada. Não são boas soluções para sistema distribuídos;
  - Nenhuma das soluções provê troca de informações entre processo que estão em diferentes máquinas;
  - Monitores dependem de uma linguagem de programação – poucas linguagens suportam Monitores;

# Soluções

- Exclusão Mútua:
  - Espera Ocupada;
  - Primitivas *Sleep/Wakeup*;
  - Semáforos;
  - Monitores;
  - **Passagem de Mensagem**;

# Passagem de Mensagem

- Provê troca de mensagens entre processos rodando em máquinas diferentes;
- Utiliza-se de duas primitivas de chamadas de sistema: *send* e *receive*;

# Passagem de Mensagem

- Podem ser implementadas como procedimentos:
  - *send (destination, &message);*
  - *receive (source, &message);*
- O procedimento `send` envia para um determinado destino uma mensagem, enquanto que o procedimento `receive` recebe essa mensagem em uma determinada fonte;
- Se nenhuma mensagem está disponível, o procedimento `receive` é bloqueado até que uma mensagem chegue.

# Passagem de Mensagem

- Problemas desta solução:
  - Mensagens são enviadas para/por máquinas conectadas em rede; assim mensagens podem ser perder a longo da transmissão;
  - Mensagem especial chamada ***acknowledgement*** → o procedimento `receive` envia um ***acknowledgement*** para o procedimento `send`.
  - Se esse ***acknowledgement*** não chega no procedimento `send`, esse procedimento retransmite a mensagem já enviada;

# Passagem de Mensagem

- Problemas:
  - A mensagem é recebida corretamente, mas o ***acknowledgement*** se perde.
  - Então o `receive` deve ter uma maneira de saber se uma mensagem recebida é uma retransmissão → cada mensagem enviada pelo `send` possui uma identificação – seqüência de números;
  - Assim, ao receber uma nova mensagem, o `receive` verifica essa identificação, se ela for semelhante a de alguma mensagem já recebida, o `receive` descarta a mensagem!



# Passagem de Mensagem

- Problemas:
  - Desempenho: copiar mensagens de um processo para o outro é mais lento do que operações com semáforos e monitores;
  - Autenticação → Segurança;

# Passagem de Mensagem

```
#define N 100                                /* number of slots in the buffer */

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

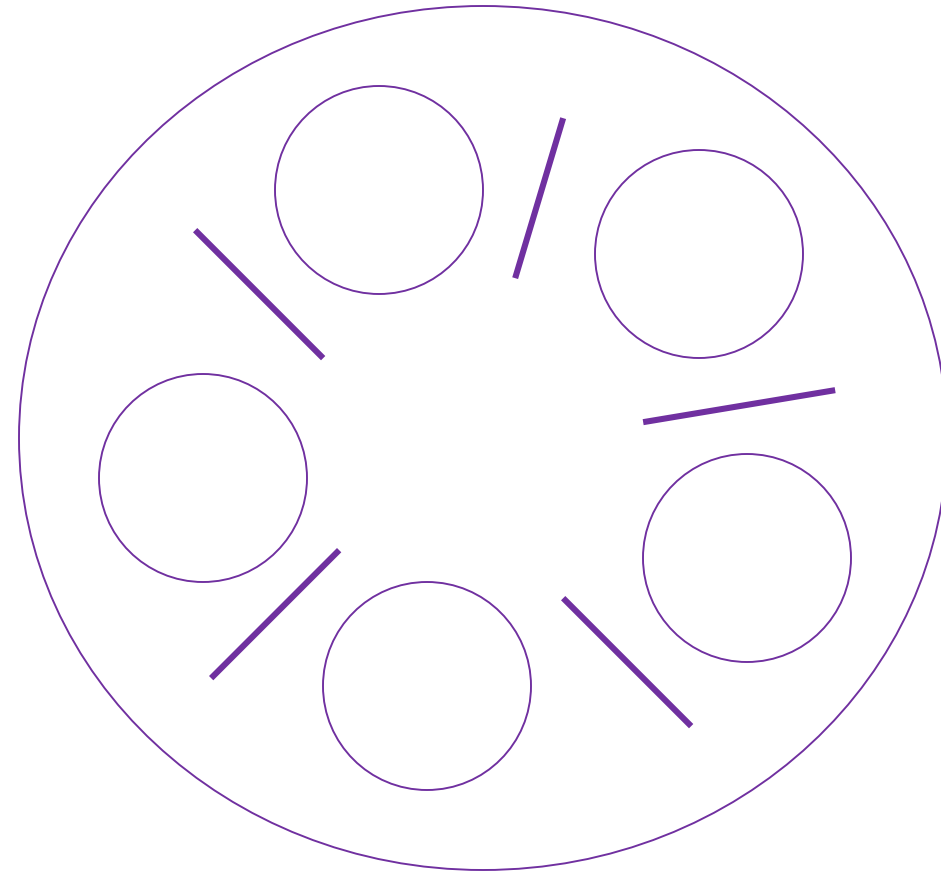
    while (TRUE) {
        item = produce_item( );              /* generate something to put in buffer */
        receive(consumer, &m);               /* wait for an empty to arrive */
        build_message(&m, item);             /* construct a message to send */
        send(consumer, &m);                  /* send item to consumer */
    }
}

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

    for (i = 0; i < N; i++) send(producer, &m); /* send N empties */
    while (TRUE) {
        receive(producer, &m);               /* get message containing item */
        item = extract_item(&m);             /* extract item from message */
        send(producer, &m);                 /* send back empty reply */
        consume_item(item);                 /* do something with the item */
    }
}
```

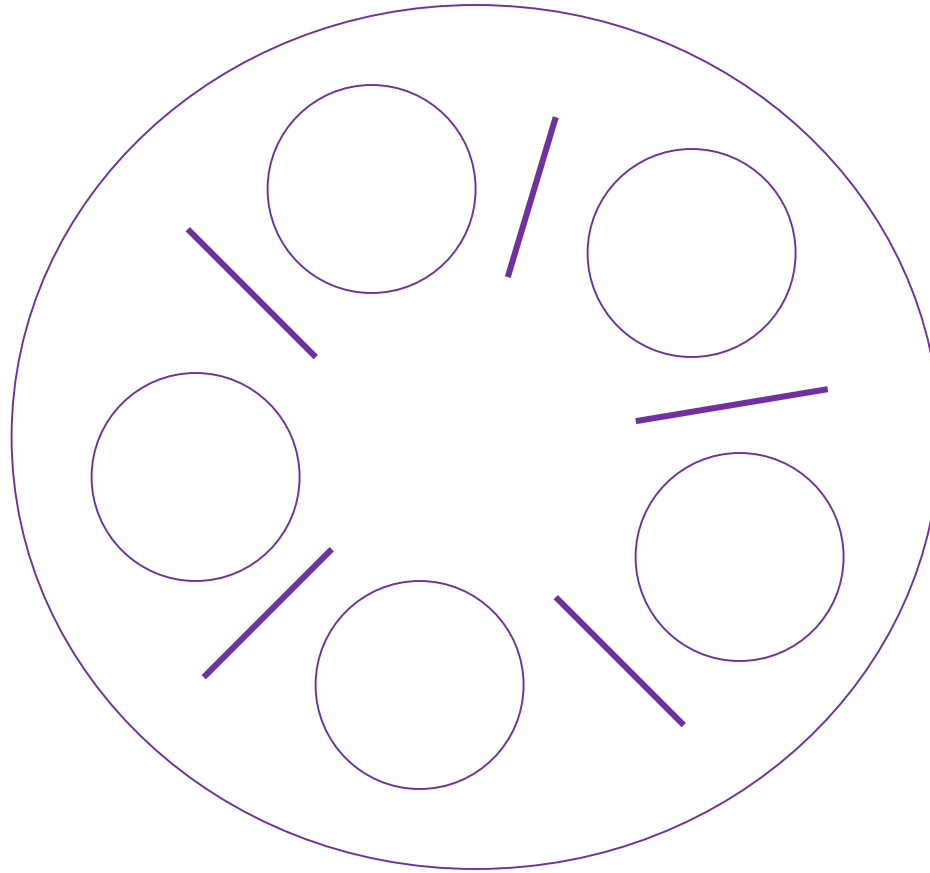
# Problemas clássicos de comunicação entre processos

- Problema do Jantar dos Filósofos
  - Cinco filósofos desejam comer espaguete; No entanto, para poder comer, cada filósofo precisa utilizar dois garfo e não apenas um. Portanto, os filósofos precisam compartilhar o uso do garfo de forma sincronizada.
  - Os filósofos comem e pensam;



# Problemas clássicos de comunicação entre processos

- Problemas que devem ser evitados:
  - *Deadlock* – todos os filósofos pegam um garfo ao mesmo tempo;
  - *Starvation* – os filósofos podem ficar indefinidamente pegando os garfos simultaneamente;



# Solução 1 para Filósofos (1/2)

```
#define N 5                                /* number of philosophers */

void philosopher(int i)                    /* i: philosopher number, from 0 to 4 */
{
    while (TRUE) {
        think( );                        /* philosopher is thinking */
        take_fork(i);                    /* take left fork */
        take_fork((i+1) % N);            /* take right fork; % is modulo operator */
        eat( );                          /* yum-yum, spaghetti */
        put_fork(i);                     /* put left fork back on the table */
        put_fork((i+1) % N);             /* put right fork back on the table */
    }
}
```

# Solução 1 para Filósofos (2/2)

- Problemas da solução 1:
  - *Deadlock*:
    - Execução do `take_fork(i)` → Se todos os filósofos pegarem o garfo da esquerda, nenhum pega o da direita;
  - *Starvation*:
    - Verificar antes se o garfo da direita está disponível. Se não está, devolve o da esquerda e começa novamente;
  - Somente um filósofo come em um determinado tempo;

# Solução 2 para Filósofos usando Semáforos (1/3)

- Não apresenta:
  - *Deadlocks*;
  - *Starvation*;
- Permite o máximo de “paralelismo”;

# Solução 2 para Filósofos usando Semáforos (2/3)

```
#define N          5          /* number of philosophers */
#define LEFT      (i+N-1)%N   /* number of i's left neighbor */
#define RIGHT     (i+1)%N     /* number of i's right neighbor */
#define THINKING  0          /* philosopher is thinking */
#define HUNGRY    1          /* philosopher is trying to get forks */
#define EATING    2          /* philosopher is eating */
typedef int semaphore;        /* semaphores are a special kind of int */
int state[N];                /* array to keep track of everyone's state */
semaphore mutex = 1;         /* mutual exclusion for critical regions */
semaphore s[N];              /* one semaphore per philosopher */

void philosopher(int i)      /* i: philosopher number, from 0 to N-1 */
{
    while (TRUE) {           /* repeat forever */
        think( );           /* philosopher is thinking */
        take_forks(i);      /* acquire two forks or block */
        eat( );             /* yum-yum, spaghetti */
        put_forks(i);       /* put both forks back on table */
    }
}
```



# Solução 2 para Filósofos usando Semáforos (3/3)

```
void take_forks(int i)                /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                      /* enter critical region */
    state[i] = HUNGRY;                 /* record fact that philosopher i is hungry */
    test(i);                          /* try to acquire 2 forks */
    up(&mutex);                        /* exit critical region */
    down(&s[i]);                       /* block if forks were not acquired */
}

void put_forks(i)                    /* i: philosopher number, from 0 to N-1 */
{
    down(&mutex);                      /* enter critical region */
    state[i] = THINKING;              /* philosopher has finished eating */
    test(LEFT);                       /* see if left neighbor can now eat */
    test(RIGHT);                      /* see if right neighbor can now eat */
    up(&mutex);                       /* exit critical region */
}

void test(i)                         /* i: philosopher number, from 0 to N-1 */
{
    if (state[i] == HUNGRY && state[LEFT] != EATING && state[RIGHT] != EATING) {
        state[i] = EATING;
        up(&s[i]);
    }
}
```

# Referências

CAMPOS, Sérgio; ROCHA, Marcus. Sistemas Operacionais – Operações Atômicas. Notas de aula. UFMG, 2013

DA SILVA CARISSIMI, Alexandre; TOSCANI, Simão Sirineo; OLIVEIRA, Rômulo Silva de. Sistemas operacionais e programação concorrente. 1ª ed. Porto Alegre: Sagra Luzzatto, 2003

SANTANA, Marcos José. Sistemas Operacionais – Processos. Notas de Aula. USP, 2014