

# Sistemas Operacionais

## Sincronização de Processos

### Parte 1

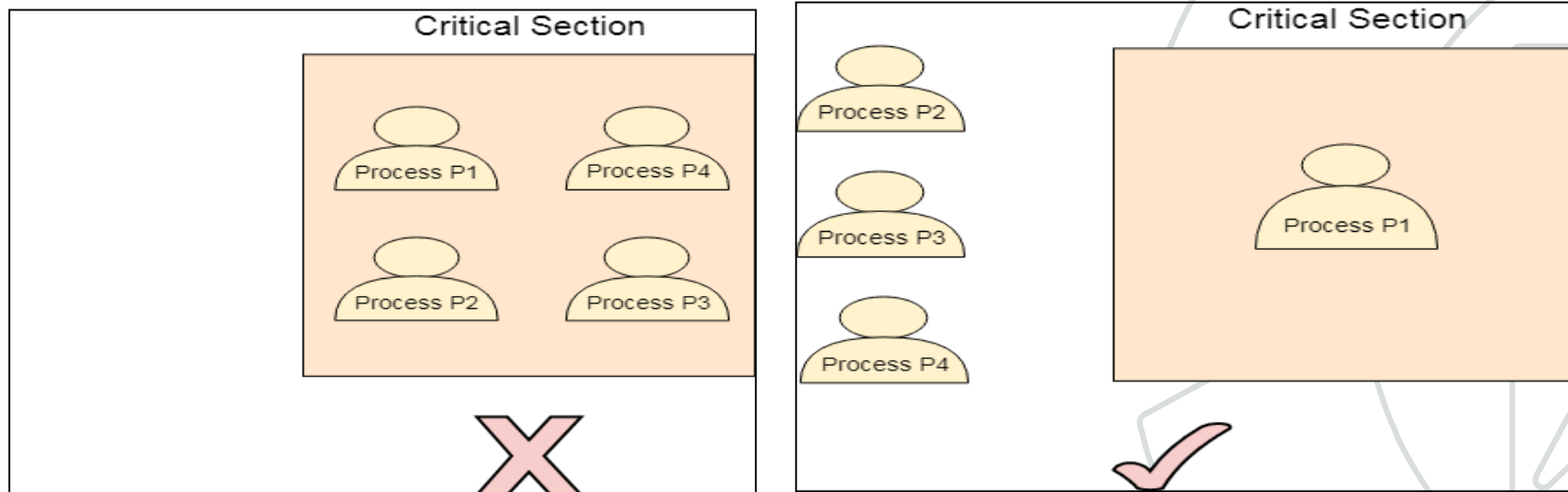
Prof. Otávio Gomes

[otavio.gomes@unifei.edu.br](mailto:otavio.gomes@unifei.edu.br)



# Condições de corrida e o problema da Seção Crítica

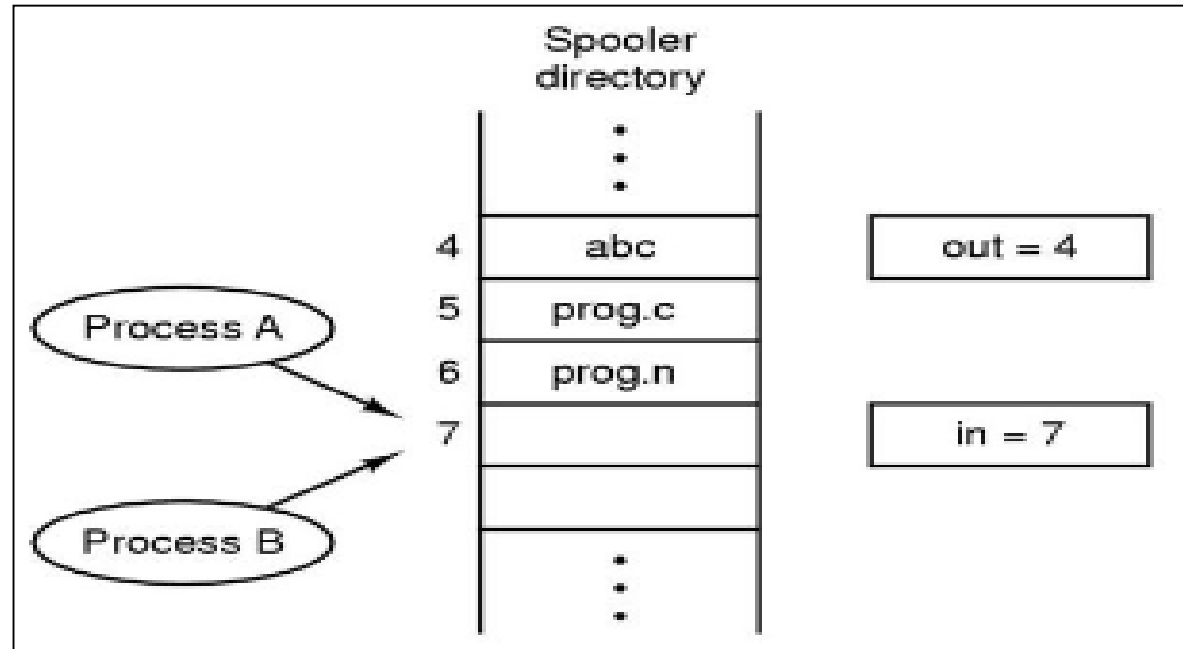
- Acesso concorrente a dados compartilhados pode resultar em **inconsistências**;
- **Condições de corrida** (*race conditions*):
  - Situação onde dois ou mais processos acessam e manipulam recursos compartilhados simultaneamente.
  - **Seção crítica:** N processos competem para usar alguma estrutura de dados compartilhada.



# Condições de corrida

e o problema da Seção Crítica

- Acesso concorrente a dados compartilhados pode resultar em **inconsistências**;
- **Condições de corrida** (*race conditions*):
  - Situação onde dois ou mais processos acessam e manipulam recursos compartilhados simultaneamente.
  - **Seção crítica**: N processos competem para usar alguma estrutura de dados compartilhada.



# Condições de corrida

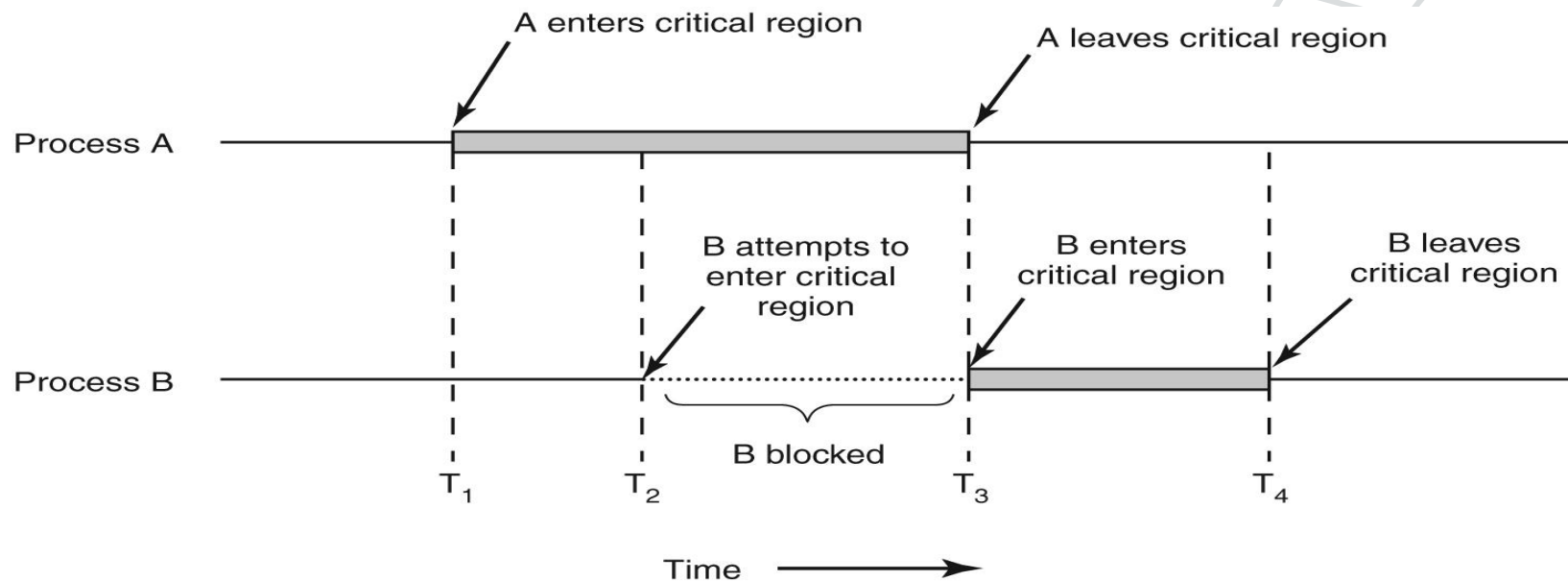
## e o problema da Seção Crítica

- Acesso concorrente a dados compartilhados pode resultar em **inconsistências**;
- **Condições de corrida** (*race conditions*):
  - Situação onde dois ou mais processos acessam e manipulam recursos compartilhados simultaneamente.
  - **Seção crítica:** N processos competem para usar alguma estrutura de dados compartilhada.
    - Cada processo possui uma seção crítica de código, onde há a **manipulação dos seus dados**.
    - Para resolver a questão de seção crítica cada processo deve pedir **permissão para entrar na região crítica**, após a utilização da seção crítica, seguir com a execução das ações.
    - Especialmente difícil resolver este problema em **kernel preemptivo**.

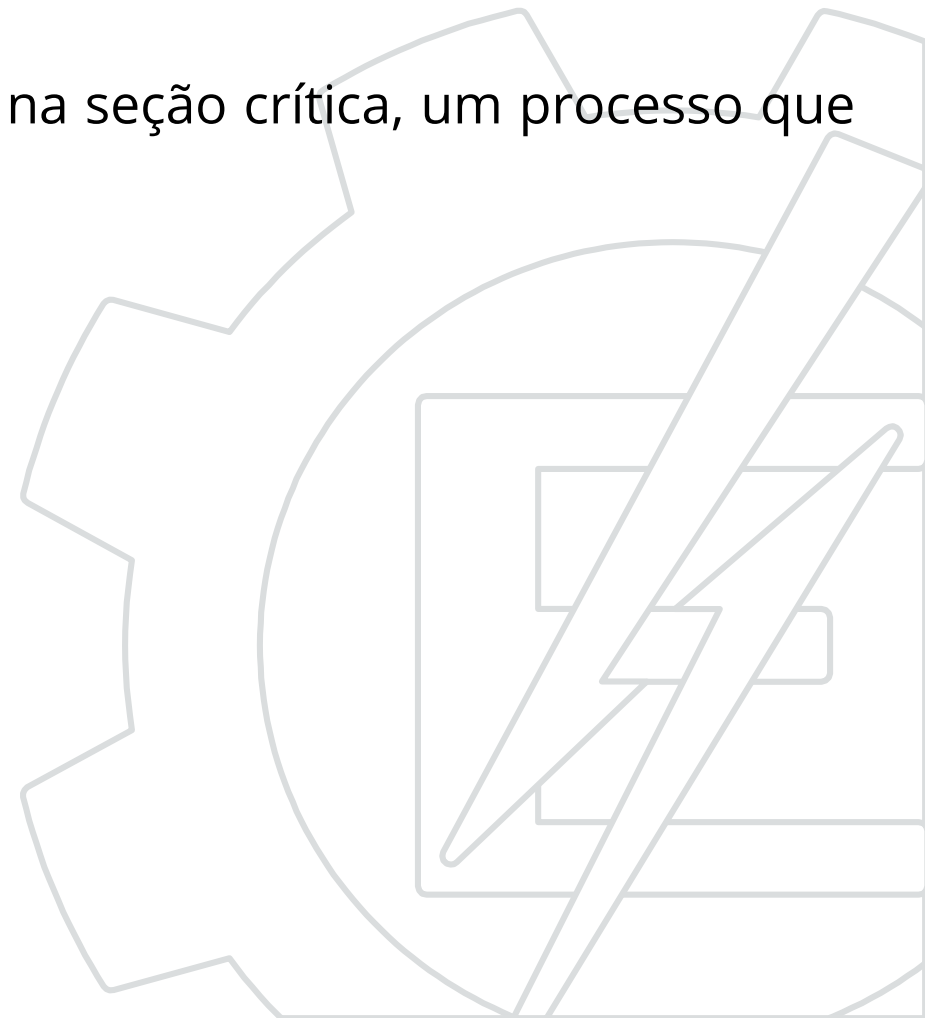
- Uma boa solução para o problema da seção crítica deve satisfazer os seguintes requisitos:
  - 1) Exclusão mútua:** se um processo  $i$  ( $P_i$ ) está na seção crítica, nenhum outro processo pode entrar nela;



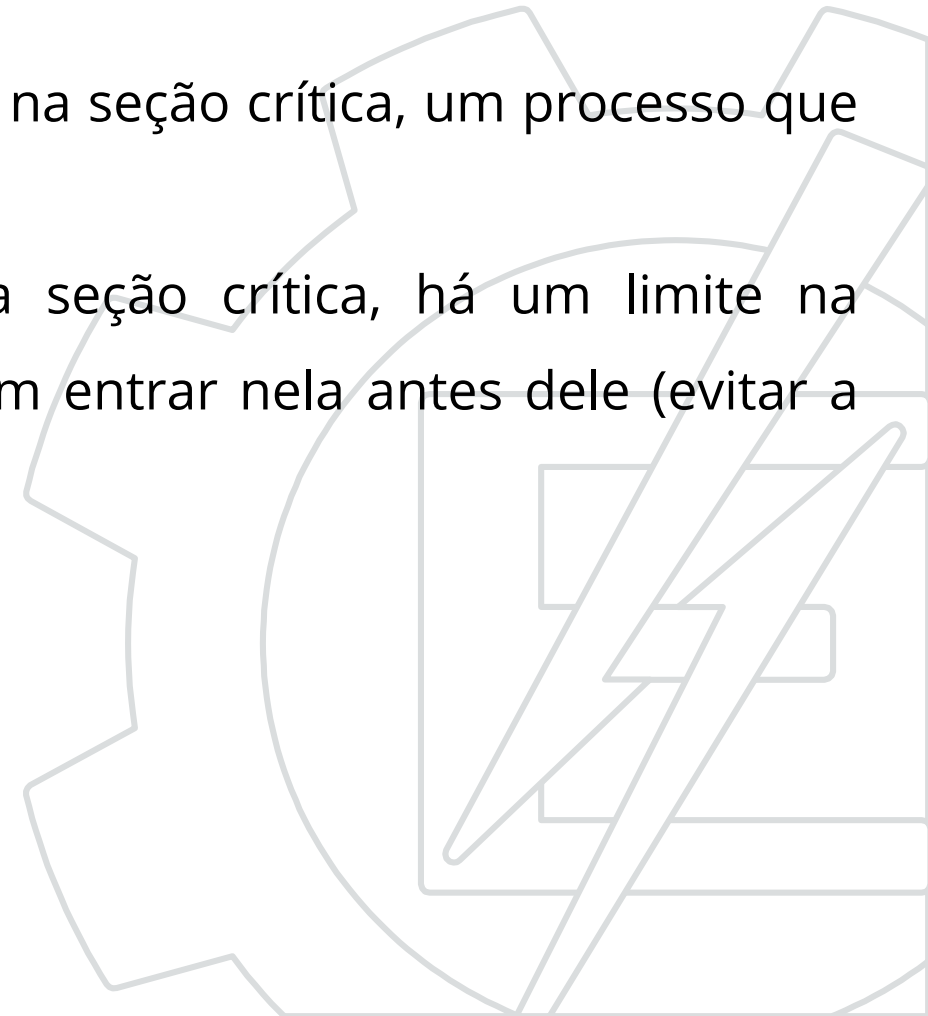
- Uma boa solução para o problema da seção crítica deve satisfazer os seguintes requisitos:
  - 1) Exclusão mútua:** se um processo  $i$  ( $P_i$ ) está na seção crítica, nenhum outro processo pode entrar nela;
    - Exclusão mútua é garantir que um processo não terá acesso a uma região crítica quando outro processo estiver utilizando esta região.
    - Pode gerar uma fila de clientes para acessar a Região Crítica (*overhead*).



- Uma boa solução para o problema da seção crítica deve satisfazer os seguintes requisitos:
  - 1) Exclusão mútua:** se um processo  $i$  ( $P_i$ ) está na seção crítica, nenhum outro processo pode entrar nela;
  - 2) Progresso garantido:** se nenhum outro processo está na seção crítica, um processo que tente fazê-lo não pode ser detido indefinidamente;



- Uma boa solução para o problema da seção crítica deve satisfazer os seguintes requisitos:
  - 1) Exclusão mútua:** se um processo  $i$  ( $P_i$ ) está na seção crítica, nenhum outro processo pode entrar nela;
  - 2) Progresso garantido:** se nenhum outro processo está na seção crítica, um processo que tente fazê-lo não pode ser detido indefinidamente;
  - 3) Espera limitada:** se um processo deseja entrar na seção crítica, há um limite na quantidade de vezes que outros processos que podem entrar nela antes dele (evitar a inanição - *starvation*);





- Uma boa solução para o problema da seção crítica deve satisfazer os seguintes requisitos:
  - 1) Exclusão mútua:** se um processo  $i$  ( $P_i$ ) está na seção crítica, nenhum outro processo pode entrar nela;
  - 2) Progresso garantido:** se nenhum outro processo está na seção crítica, um processo que tente fazê-lo não pode ser detido indefinidamente;
  - 3) Espera limitada:** se um processo deseja entrar na seção crítica, há um limite na quantidade de vezes que outros processos que podem entrar nela antes dele (evitar a inanição - *starvation*);
  - 4) Independência da arquitetura:** o processo não pode funcionar somente se estiver sendo executado em uma configuração específica de dispositivo, por exemplo: quantidade de núcleos e/ou frequência determinados.

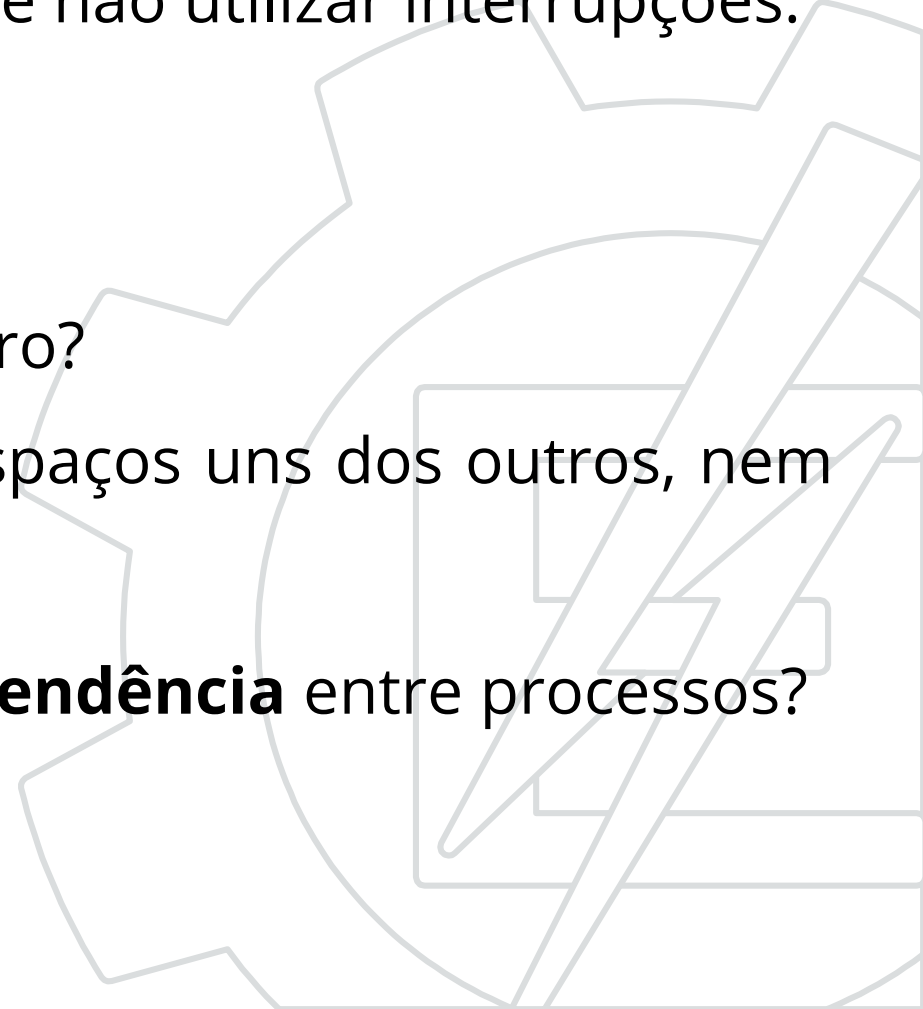
Observação: Alguns autores omitem este último requisito.

- Uma boa solução para o problema da seção crítica deve satisfazer os seguintes requisitos:
  - 1) Exclusão mútua:** se um processo  $i$  ( $P_i$ ) está na seção crítica, nenhum outro processo pode entrar nela;
  - 2) Progresso garantido:** se nenhum outro processo está na seção crítica, um processo que tente fazê-lo não pode ser detido indefinidamente;
  - 3) Espera limitada:** se um processo deseja entrar na seção crítica, há um limite na quantidade de vezes que outros processos que podem entrar nela antes dele (evitar a inanição - *starvation*);
  - 4) Independência da arquitetura:** o processo não pode funcionar somente se estiver sendo executado em uma configuração específica de dispositivo, por exemplo: quantidade de núcleos e/ou frequência determinados.

O acesso à região crítica pode envolver tanto processos, quanto *threads*.

# Comunicação entre Processos

*InterProcess Communication (IPC)*

- Frequentemente processos precisam se comunicar.
  - A comunicação é mais eficiente se for estruturada e não utilizar interrupções.
  - Questões importantes:
    - **Como** um processo passa informação para outro?
    - Como garantir que processos não **invadam** espaços uns dos outros, nem entrem em **conflito**?
    - Qual a sequência adequada quando existe **dependência** entre processos?
- 

# Sincronização

Hora	Pessoa A	Pessoa B
6h00	Olha a geladeira: sem leite	
6h05	Sai para a padaria	
6h10	Chega na padaria	Olha a geladeira: sem leite
6h15	Sai da padaria	Sai para a padaria
6h20	Chega em casa: guarda o leite	Chega na padaria
6h25		Sai da padaria
6h30		Chega em casa: Ops!

# Sincronização

Hora	Pessoa A	Pessoa B
6h00	Olha a geladeira: sem leite	
6h05	Sai para a padaria	
6h10	Chega na padaria	Olha a geladeira: sem leite
6h15	Sai da padaria	Sai para a padaria
6h20	Chega em casa: guarda o leite	Chega na padaria
6h25		Sai da padaria
6h30		Chega em casa: Ops!

Regra	Exemplo da geladeira
1. Trancar antes de utilizar	Deixar o aviso
2. Destancar quando terminar	Retirar o aviso
3. Esperar se estiver trancado	Não sai para comprar se houver aviso

- A sincronização dos recursos do computador é realizada pelo S.O. multitarefas e é feita tanto para **dados** (para manter dados em integridade) quanto para **processos** (evitar conflito na utilização dos recursos).
- Os mecanismos para controle de sincronização, dentre outros, podem ser:
  - 1. Barreiras:** criadas e gerenciadas pelo programa – o programa entra em estado de espera até que todos os processos pertencentes ao mesmo programa também entrem neste estado;

**2. Semáforos:** podem ser implementados tanto pelo S.O. quanto pelo programa; consistem em variáveis de controle que indicam quantos processos podem compartilhar um recurso;

### 3. Trava (*Lock*):

**simples** - impede a utilização do recurso por outro processo;

**especial** - sinaliza quando se tenta utilizar um recurso já em uso; e

**compartilhada** - um único processo recebe permissão de leitura e escrita e os demais somente de leitura.

- A barreira e o semáforo permitem compartilhamento de recurso, a trava simples não. A sincronia exige a existência de uma instrução capaz de ao mesmo tempo "verificar e, se possível, travar".

- Espera ocupada (*busy waiting*)
- *Sleep / WakeUp* (primitivas - chamadas de sistema)
- Semáforos (variável de controle)
- Monitores (primitiva de alto nível)
- Troca de Mensagens





- Uma boa solução para o problema da seção crítica deve satisfazer os seguintes requisitos:
  - 1) Exclusão mútua:** se um processo  $i$  ( $P_i$ ) está na seção crítica, nenhum outro processo pode entrar nela;
  - 2) Progresso garantido:** se nenhum outro processo está na seção crítica, um processo que tente fazê-lo não pode ser detido indefinidamente;
  - 3) Espera limitada:** se um processo deseja entrar na seção crítica, há um limite na quantidade de vezes que outros processos que podem entrar nela antes dele (evitar a inanição - *starvation*);
  - 4) Independência da arquitetura:** o processo não pode funcionar somente se estiver sendo executado em uma configuração específica de dispositivo, por exemplo: quantidade de núcleos e/ou frequência determinados.

- **Espera ocupada** (*busy waiting*)
- *Sleep / WakeUp* (primitivas - chamadas de sistema)
- Semáforos (variável de controle)
- Monitores (primitiva de alto nível)
- Troca de Mensagens



- Consiste na constante verificação de um valor.
- Gera desperdício de tempo da CPU.
- Soluções para exclusão mútua através de espera ocupada:
  - 1) Desabilitar Interrupções;
  - 2) Variáveis de trancamento (**Lock**);
  - 3) Estrita alternância (**Turn**);
  - 4) Solução de Peterson e instrução **TSL**.



# Espera Ocupada

## 1) Desabilitar Interrupções

- Cada processo desabilita todas as interrupções (inclusive a do relógio) ao entrar na região crítica;
- Com as interrupções desabilitadas, a CPU não realiza chaveamento entre os processos;



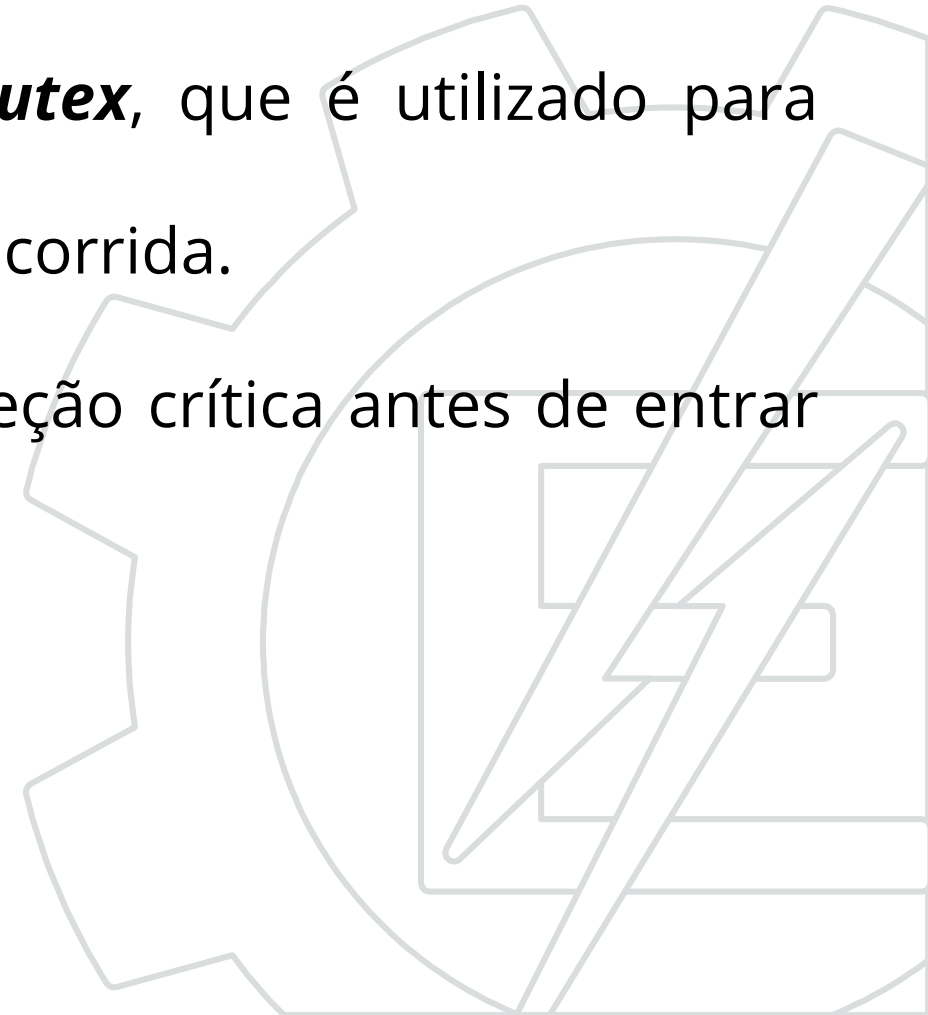
# Espera Ocupada

## 1) Desabilitar Interrupções

- Cada processo desabilita todas as interrupções (inclusive a do relógio) ao entrar na região crítica;
- Com as interrupções desabilitadas, a CPU não realiza chaveamento entre os processos;
- Análise de situações-problema:
  - Em **sistemas com várias CPUs**, desabilitar interrupções de uma CPU não evita que outras acessem a memória compartilhada;
  - Não é uma solução segura, pois um processo pode **esquecer de reabilitar** suas interrupções, gerando inanição;
  - Funciona bem somente em ambientes **monoprocessados**.

- 1) Exclusão mútua;
- 2) Progresso garantido;
- 3) Espera limitada;
- 4) Independência da arquitetura.

- Os sistemas operacionais oferecem soluções em *software* para o problema da região crítica. O mais simples é a trava ***mutex***, que é utilizado para proteger regiões críticas e prevenir condições de corrida.
- O processo precisa realizar o trancamento da seção crítica antes de entrar nela e realiza seu destravamento após sair dela.



# Espera Ocupada

## 2) Variáveis de trancamento (*mutex lock*)

### Processo A

```
while(true){  
    while (lock!=0);  
    lock=1;  
    critical_region();  
    lock=0;  
    non-critical_region();  
}
```

### Processo B

```
while(true){  
    while (lock!=0);  
    lock=1;  
    critical_region();  
    lock=0;  
    non-critical_region();  
}
```



# Espera Ocupada

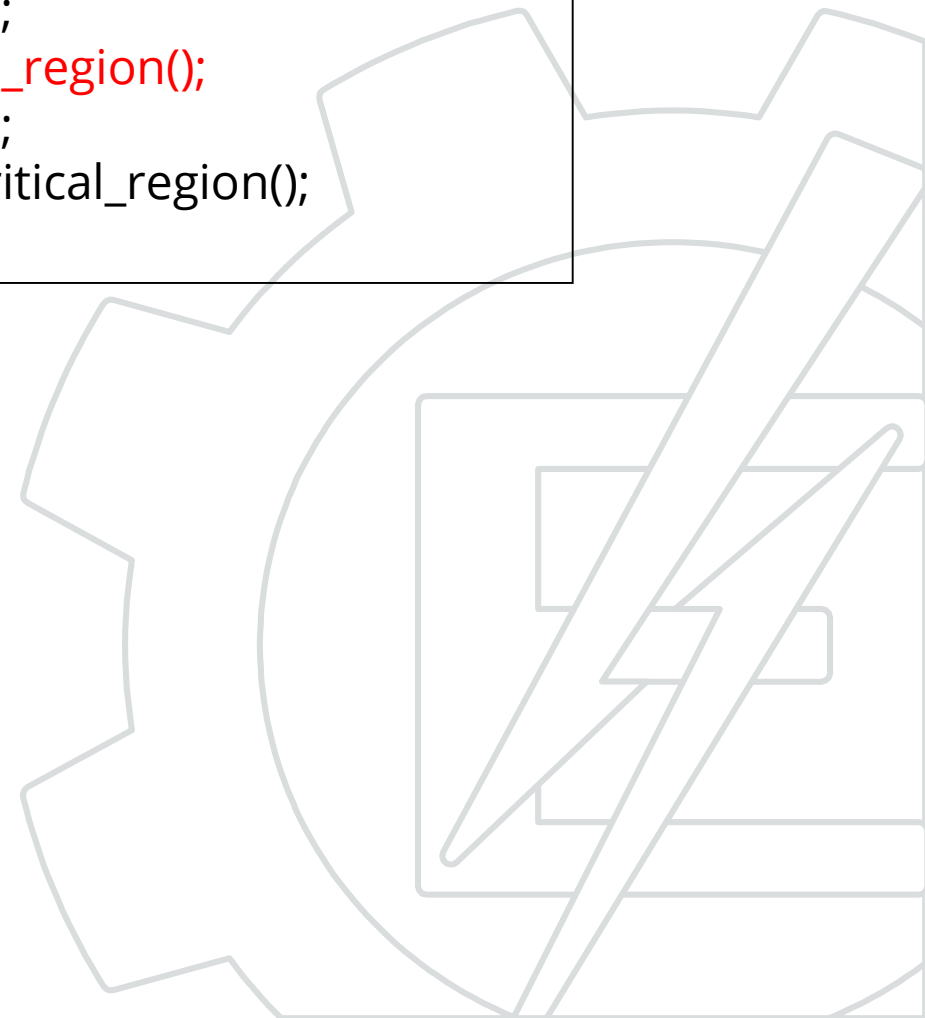
## 2) Variáveis de trancamento (*mutex lock*)

Processo A

```
while(true){  
    while (lock!=0);  
    lock=1;  
    critical_region();  
    lock=0;  
    non-critical_region();  
}
```

Processo B

```
while(true){  
    while (lock!=0);  
    lock=1;  
    critical_region();  
    lock=0;  
    non-critical_region();  
}
```





# Espera Ocupada

## 2) Variáveis de trancamento (*mutex lock*)

### Processo A

```
while(true){  
    while (lock!=0);  
    lock=1;  
    critical_region();  
    lock=0;  
    non-critical_region();  
}
```

### Processo B

```
while(true){  
    while (lock!=0);  
    lock=1;  
    critical_region();  
    lock=0;  
    non-critical_region();  
}
```

- Análise de uma situação-problema:
  - 1) Mudança de contexto no processo A após a verificação de *lock==0*;
  - 2) Execução do processo B;
  - 3) Retorno do processo A ao processador;
  - 4) Os dois processos acessam a região crítica.

- 1) **Exclusão mútua;**
- 2) **Progresso garantido;**
- 3) **Espera limitada;**
- 4) **Independência da arquitetura.**

# Espera Ocupada

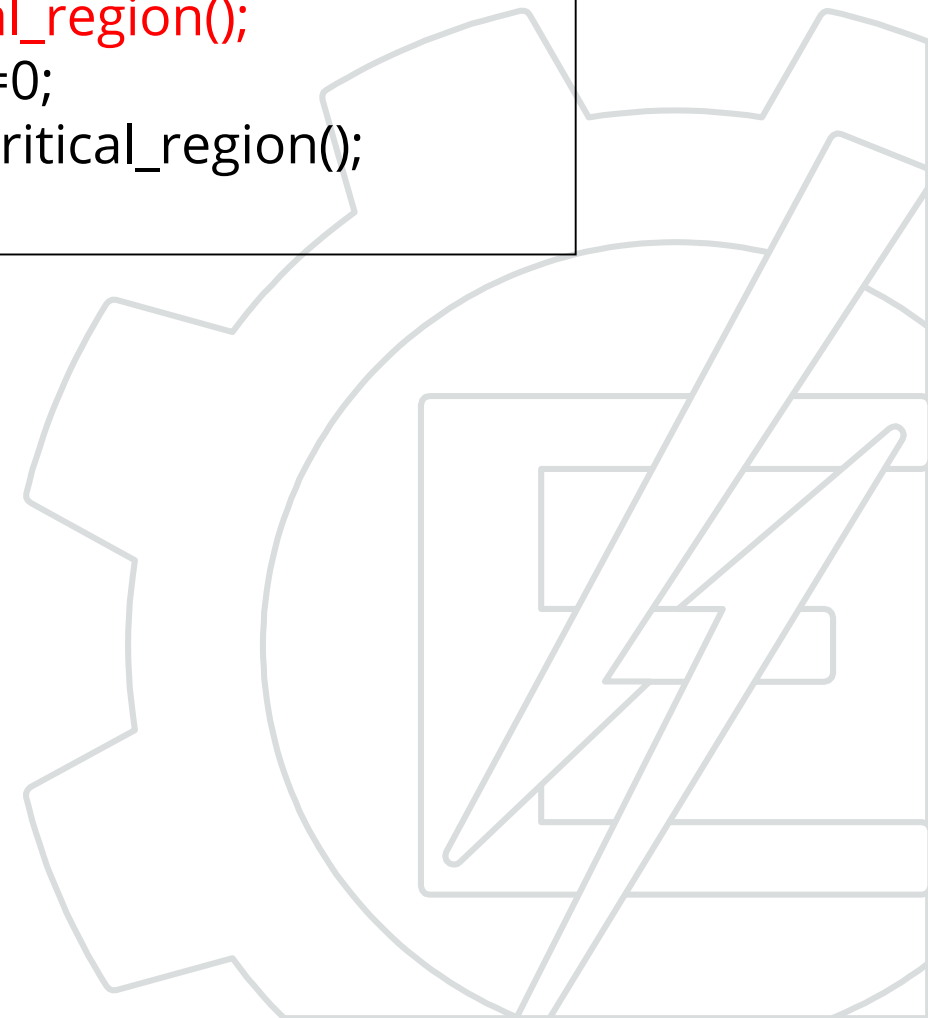
3) Estrita alternância

Processo A (*turn*=0)

```
while(true){  
    while (turn!=0);  
    critical_region();  
    turn=1;  
    non-critical_region();  
}
```

Processo B (*turn*=1)

```
while(true){  
    while (turn!=1);  
    critical_region();  
    turn=0;  
    non-critical_region();  
}
```



# Espera Ocupada

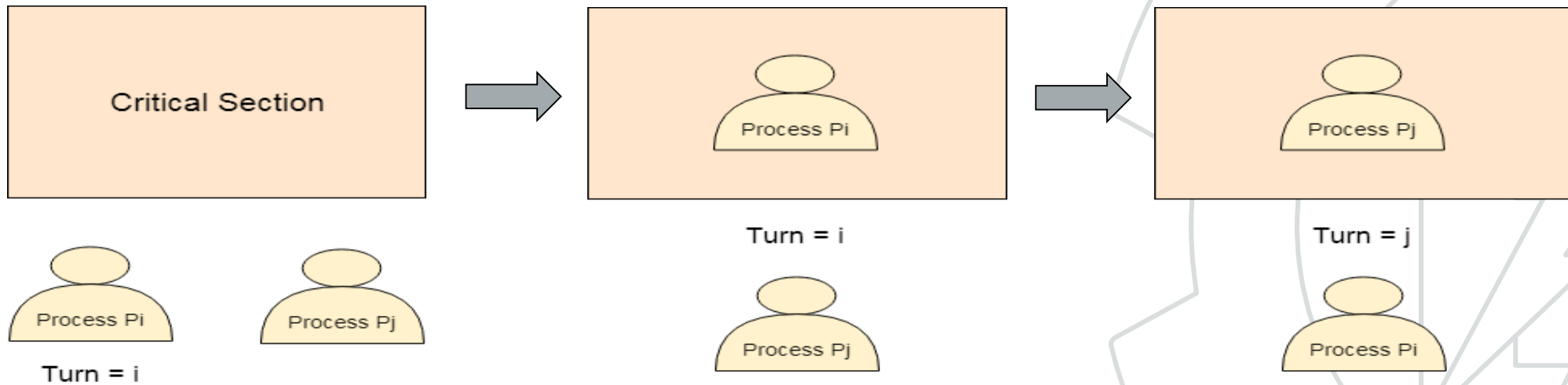
3) Estrita alternância

Processo A ( $turn=0$ )

```
while(true){  
    while (turn!=0);  
    critical_region();  
    turn=1;  
    non-critical_region();  
}
```

Processo B ( $turn=1$ )

```
while(true){  
    while (turn!=1);  
    critical_region();  
    turn=0;  
    non-critical_region();  
}
```



# Espera Ocupada

3) Estrita alternância

Processo A (*turn*=0)

```
while(true){  
    while (turn!=0);  
    critical_region();  
    turn=1;  
    non-critical_region();  
}
```

Processo B (*turn*=1)

```
while(true){  
    while (turn!=1);  
    critical_region();  
    turn=0;  
    non-critical_region();  
}
```

- Conhecida como **alternância rigorosa**.
- Nunca um processo entra duas vezes seguidas em uma região crítica.



# Espera Ocupada

## 3) Estrita alternância

Processo A (*turn*=0)

```
while(true){  
    while (turn!=0);  
    critical_region();  
    turn=1;  
    non-critical_region();  
}
```

Processo B (*turn*=1)

```
while(true){  
    while (turn!=1);  
    critical_region();  
    turn=0;  
    non-critical_region();  
}
```

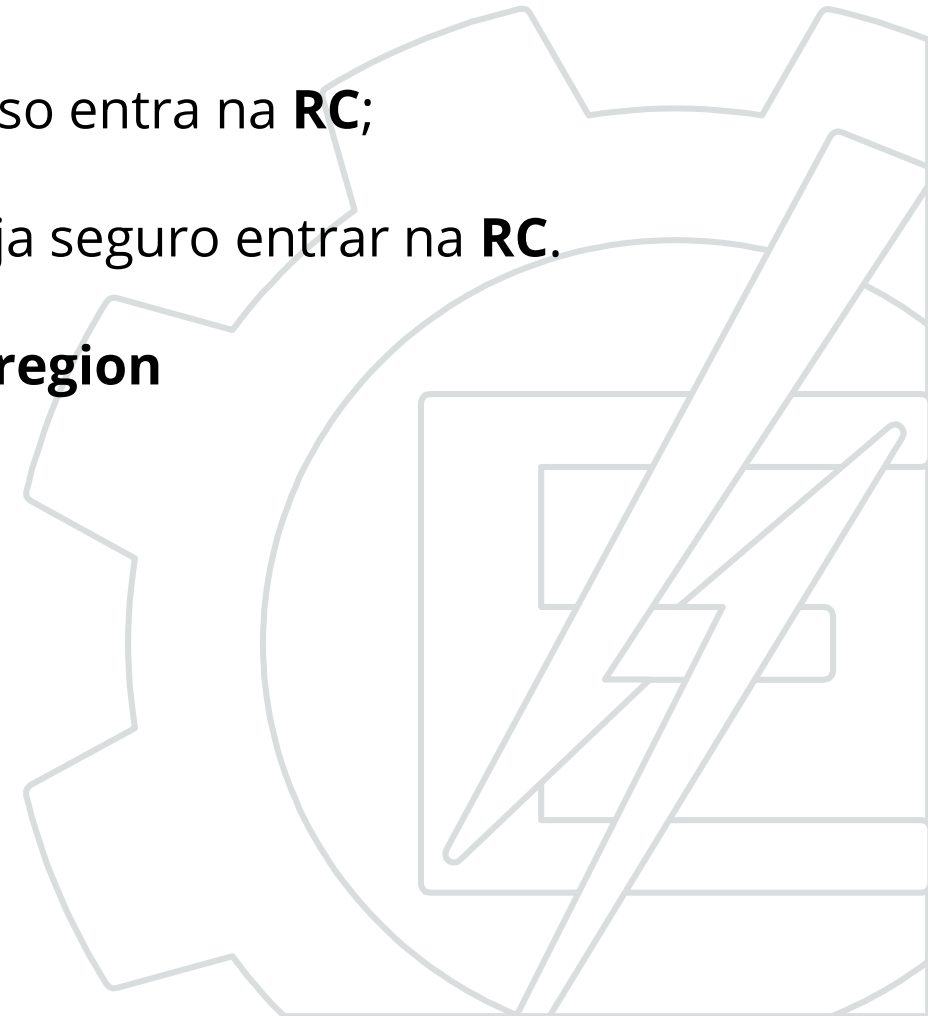
- Conhecida como **alternância rigorosa**.
- Nunca um processo entra duas vezes seguidas em uma região crítica.

- 1) Exclusão mútua;
- 2) **Progresso garantido;**
- 3) Espera limitada;
- 4) Independência da arquitetura.

- Antes de entrar na Região Crítica (**RC**), cada processo chama a função **enter\_region**, demonstrando seu interesse em entrar na **RC**;
- Somente no retorno da função **enter\_region** é que o processo entra na **RC**;
- Com a utilização desta função, o processo espera até que seja seguro entrar na **RC**.
- Após terminar a execução na **RC**, ele chama a função **leave\_region**

Processo

```
...  
enter_region(i);  
...  
Região crítica  
...  
leave_region(i);  
...
```



# Espera Ocupada

## 4) Solução de Peterson

Processo

```
...  
enter_region(i);  
...  
Região crítica  
...  
leave_region(i);  
...
```

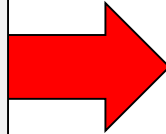


# Espera Ocupada

## 4) Solução de Peterson

Processo

```
...  
enter_region(i);  
...  
Região crítica  
...  
leave_region(i);  
...
```



```
#define FALSE 0  
#define TRUE 1  
#define N 2  
  
int turn;  
int interested[N] = {0,0};  
  
void enter_region(int procnr){  
    int other;  
    other = 1- procnr;  
    interested[procnr] = TRUE;  
    turn = procnr;  
    while (turn == procnr && interested[other] == TRUE);  
}  
  
void leave_region(int procnr){  
    interested[procnr] = FALSE;  
}
```



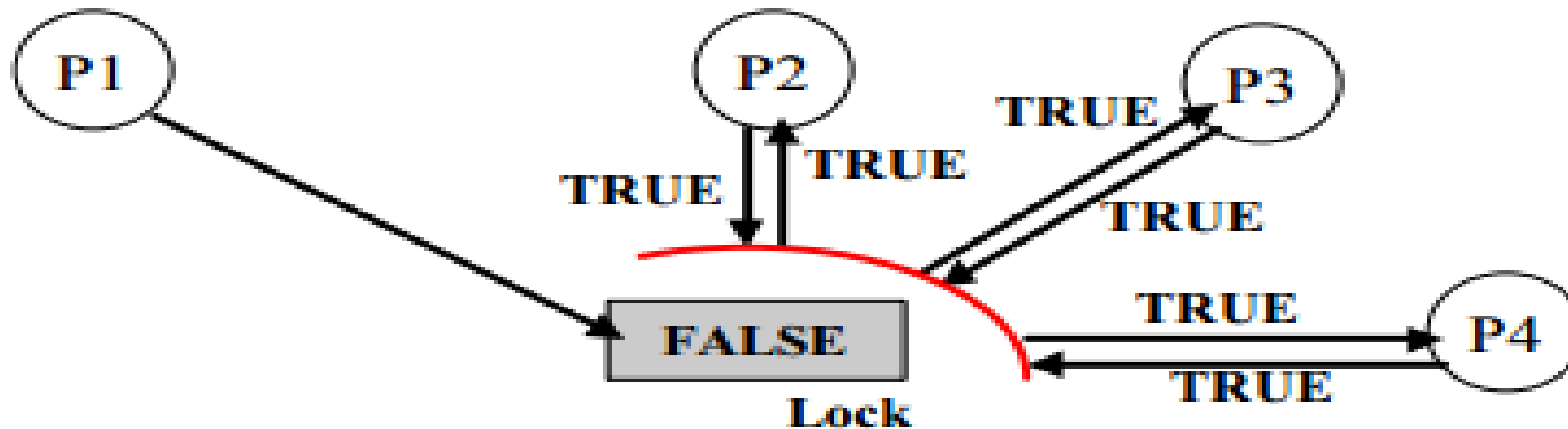
# Espera Ocupada

## 4) Solução de Peterson + TSL *Test-and-Set Lock*

### Processo

```
...  
call enter_region  
...  
Região crítica  
...  
call leave_region  
...
```

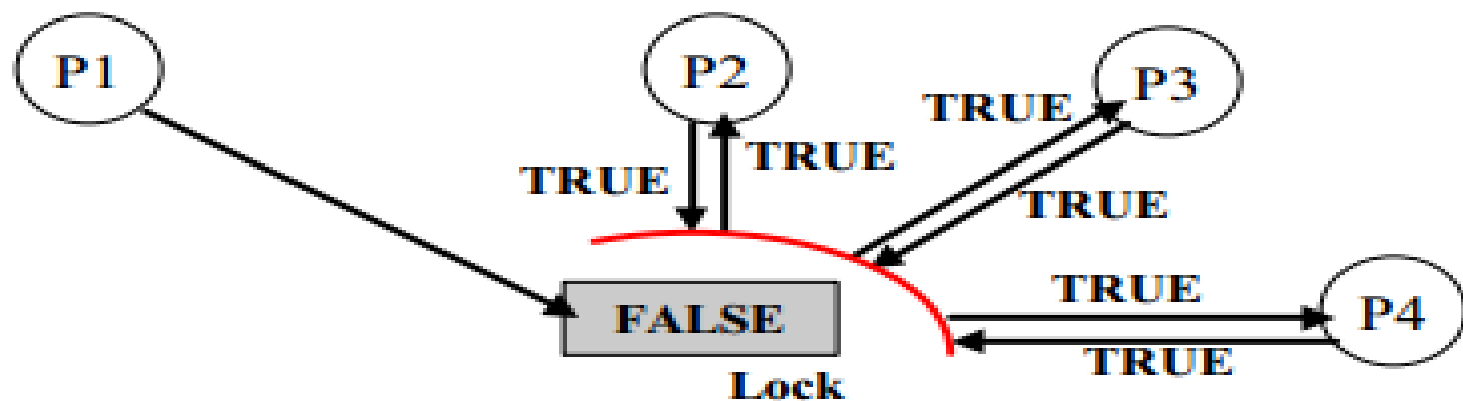
```
enter_region:  
    TSL REGISTER, LOCK  
    CMP REGISTER, #0  
    JNE enter_region  
    RET  
  
leave_region:  
    MOVE LOCK, #0  
    RET
```



# Espera Ocupada

## 4) Solução de Peterson + TSL

- Utiliza suporte de *hardware* – TSL (**Test-And-Set Lock**).
- Operação indivisível (**atômica**) - Bloqueia o barramento de memória.
- No *Intel x86* TSL é XCHG (*Exchange Data*).



- 1) Exclusão mútua;
- 2) Progresso garantido;
- 3) Espera limitada;
- 4) **Independência da arquitetura.**

# Seção crítica

## Espera Ocupada (**Resumo**)

- Consiste na constante verificação de um valor (laço de espera).
- Gera **desperdício de tempo da CPU**.
- Soluções para exclusão mútua através de espera ocupada:
  - 1) Desabilitar Interrupções;
  - 2) Variáveis de trancamento (**Lock**);
  - 3) Estrita alternância (**Turn**);
  - 4) Solução de Peterson e a instrução **TSL**.

- |   |
|---|
| <ol style="list-style-type: none"><li>1) Exclusão mútua;</li><li>2) Progresso garantido;</li><li>3) Espera limitada;</li><li>4) Independência da arquitetura.</li></ol> |
|---|



# Bibliografia

- TANENBAUM, Andrew S; BOS, Herbert. Sistemas operacionais modernos. 4a ed. São Paulo: Pearson Education do Brasil, 2016.

## **Capítulo 2.**

<https://plataforma.bvirtual.com.br/Acervo/Publicacao/1233>

- DEITEL, H.M; DEITEL, P.J; CHOFFNES,D.R. Sistemas Operacionais. 3a ed. São Paulo: Pearson Prentice Hall, 2005. **Capítulos 5 e 6.**

<https://plataforma.bvirtual.com.br/Acervo/Publicacao/315>



# Sistemas Operacionais

Prof. Otávio Gomes

[otavio.gomes@unifei.edu.br](mailto:otavio.gomes@unifei.edu.br)

