

Sincronização entre processos

- Introdução
- Condição de corrida
- Exclusão mútua
- Soluções para exclusão mútua com Espera Ocupada

Conceitos de programação

- Programação sequencial
 - Programas com um único fluxo de execução
- Programação concorrente
 - Possui dois ou mais fluxos de execução sequenciais, que podem ser executados concorrentemente
 - Necessidade de comunicação p/ troca de informação e sincronização
 - ☺ maior desempenho
 - ☹ maior complexidade
 - ☹ execução não determinística (p/ um mesmo conjunto de entrada, o programa não produz necessariamente a mesma saída)

Paralelismo x concorrência

- Paralelismo real
 - Só ocorre em máquinas multiprocessadas
- Paralelismo aparente (concorrência)
 - Mecanismo que executa “simultaneamente” M processos em N processadores, quando $M > N$
 - $N = 1$, caso particular de monoprocessamento

Programação concorrente

- Paradigma de programação que possibilita a implementação computacional de vários programas sequenciais, que executam “simultaneamente” trocando informações e disputando recursos comuns
- Programas concorrentes
 - Podem afetar ou serem afetados entre si

Programação concorrente: especificação das tarefas

- Quantos processos concorrentes haverá no sistema ?
- O que cada processo fará ?
- Como os processos irão cooperar entre si ?
 - Comunicação entre processos
- Que recursos os processos irão disputar ?
 - exige mecanismo de controle de acesso a recursos (memória, CPU, devices, ...)
- Qual é a ordem de execução dos processos ?
 - Sincronização entre processos

Compartilhamento de recursos: problemas

- Programação concorrente → compartilhamento de recursos
 - Como manter o estado (dados) de cada processo (fluxo) consistente mesmo quando diversos fluxos interagem ?
 - Como garantir o acesso a um determinado recurso a todos os processos que necessitam dele ? (uso de CPU, p. ex)
 - Exemplos:
 - Produtor / consumidor
 - Condição de corrida
 - Problema de exclusão mútua

Problema de Condição de Corrida (Race Condition)

- Ocorre quando dois ou mais processos manipulam o mesmo conjunto de dados concorrentemente e o resultado depende da ordem em que os acessos são efetuados (“timing” do escalonamento)
- Ex.: 2 processos que compartilham as variáveis A e B:

P1	P2
A = 1	B = 2

Qual é o resultado final ?
Importa a ordem de execução dos processos ?

P1	P2
A = B + 1	B = 2 * B

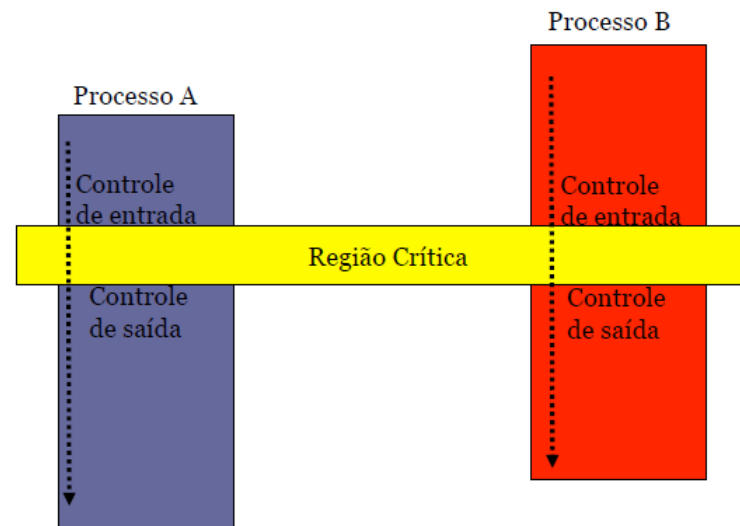
Qual é o resultado final ?
Importa a ordem de execução dos processos ?

P1	P2
A = 1	A = 2

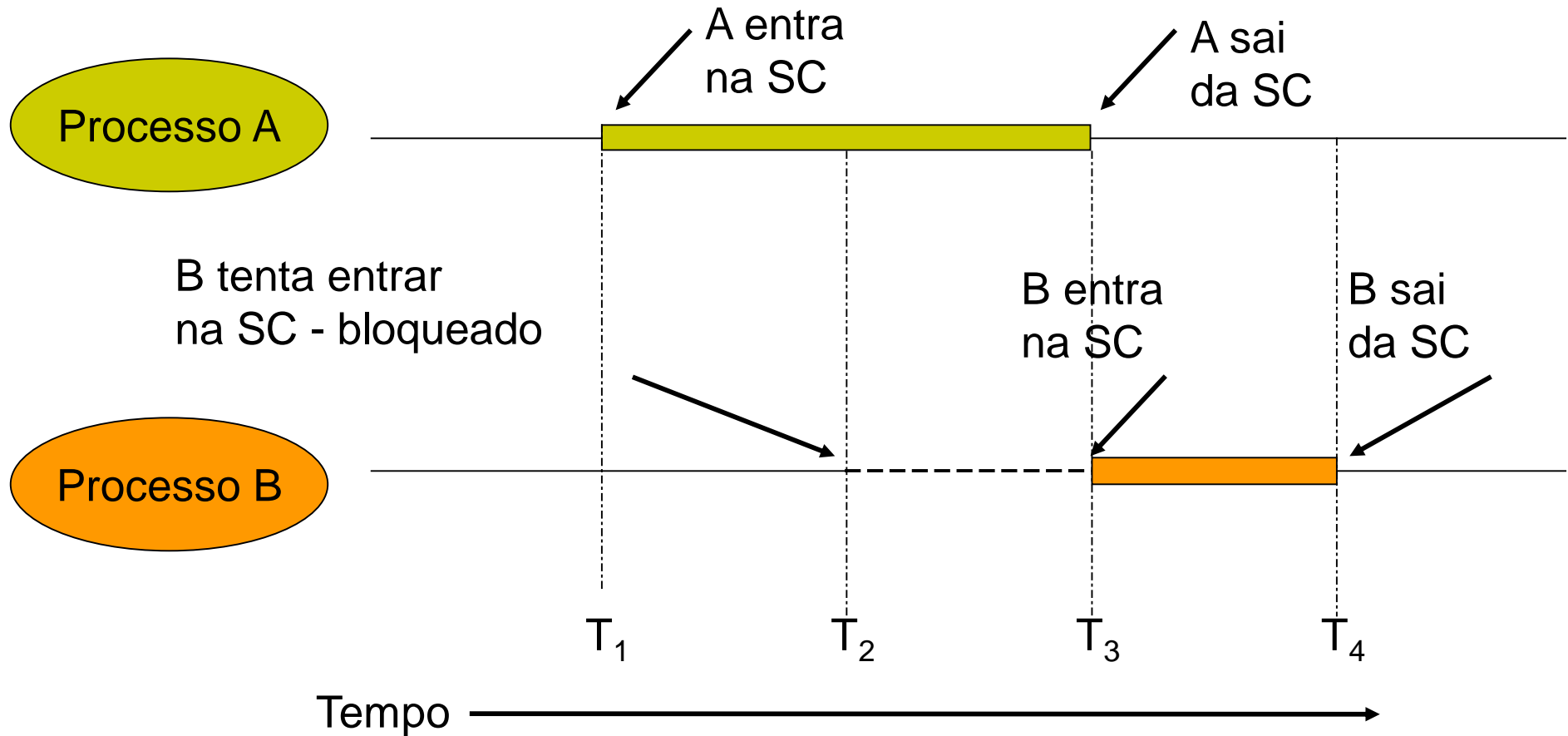
Qual é o resultado final ?
E em um computador c/ múltiplos processadores ?

Problema de Exclusão Mútua (1)

- Ocorre quando dois ou mais processos necessitam de recursos de forma exclusiva
 - CPU, impressora, dados, ...



Problema de Exclusão Mútua (2)



Propriedades da Seção/Região Crítica (1)

- evitar que mais de um processo acesse recursos compartilhados “concorrentemente” → sincronizar
 - recursos compartilhados → **seções críticas - SC** (ou regiões críticas)
- **Seção crítica**
 - código de um processo que acessa dados compartilhados
 - 3 condições:
 - Exclusão mútua
 - Progresso
 - Espera limitada

Propriedades da Seção/Região Crítica (2)

- 1) **Exclusão mútua**: 2 processos não podem estar simultaneamente numa mesma SC
- 2) **Progresso**: nenhum processo fora da SC pode bloquear a execução de outro processo
- 3) **Espera limitada**: nenhum processo pode ter seu acesso a SC postergado indefinidamente

Exclusão Mútua: implementações

- **Espera ocupada**: processo espera permissão p/ entrar na SC em um loop de teste

```
while (vez != minha);
```

 - desperdício de CPU
 - uso: quando se sabe que a espera é pequena
- **Bloqueio de processos**: processo que espera permissão p/ entrar na SC executa primitiva que gera seu bloqueio até a liberação da SC

```
if (vez != minha)
    sleep();
```

 - bloqueio → troca de contexto → esperas longas

Soluções para exclusão mútua

Espera Ocupada


Primitivas *Sleep/Wakeup*

Semáforos

Soluções erradas podem gerar:

- Starvation
- Deadlock

Soluções para exclusão mútua com Espera ocupada (*busy waiting*)

- verificação constante de um valor
- soluções p/ exclusão mútua com espera ocupada:
 - 1) Desabilitar interrupções *solução por hw*
 - 2) Variáveis de Travamento (*Lock*)
 - 3) Alternância obrigatória
 - 4) Solução de Peterson

1) Desabilitar interrupções

- processo desabilita todas as suas interrupções ao entrar na SC e habilita ao sair
 - interrupções desabilitadas → processo não é preemptado
 - uso: sistemas pequenos e dedicados (*embedded systems*)
 - garante exclusividade, progresso e espera limitada
 - problemas:
 - e se o processo não reabilitar as interrupções ?
 - tarefa privilegiada: executada em modo kernel
 - solução adequada p/ kernel, mas não p/ processos do usuário
-

2) Variáveis de travamento- *lock* (1)

- variável compartilhada com valor inicial 0
- processo quer acessar SC, verifica valor de *lock*
 - *lock* = 0 (livre) → processo altera p/ 1 e entra na SC
 - *lock* = 1 (trancada) → processo aguarda

```
while(true) {  
    while(lock!=0);    //loop  
    lock=1;  
    critical_section();  
    lock=0;  
    no_critical_section();  
}
```


2) Variáveis de travamento- *lock* (2)

- Supondo que ...
 - **pA** verifica que *lock* é 0 e perde processador antes de alterar p/ 1

pA lê *lock*=0
e perde
processador

```
while(true) {  
    while(lock!=0);    //loop  
    lock=1;  
    critical_section();  
    lock=0;  
    no_critical_section();  
}
```

2) Variáveis de travamento- *lock* (3)

- Supondo que ...
 - ...
 - **pB** verifica que *lock* é 0 e altera p/ 1

```
while(true) {  
    while(lock!=0) ; } //loop  
    lock=1;  
    critical_section();  
    lock=0;  
    no_critical_section();  
}
```

pB lê *lock=0*
e altera p/ 1

2) Variáveis de travamento- *lock* (4)

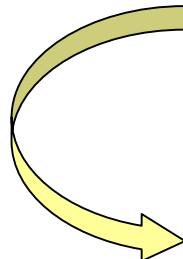
- Supondo que ...
 - ...
 - Se **pA** for escalonado antes de **pB** alterar *lock* p/ 0
 - **pA** e **pB** entram na SC

Se pA for escalonado
antes de pB alterar
lock p/ 0
→ ambos na SC

```
while(true) {  
    while(lock!=0);    //loop  
    lock=1;  
    critical_section();  
    lock=0;  
    no_critical_section();  
}
```

2) Variáveis de travamento- *lock* (5)

- Busy waiting
- Solução exige que teste e alteração da variável *lock* sejam atômicos



```
while(true) {  
    while(lock!=0);           //loop  
    lock=1;  
    critical_section();  
    lock=0;  
    no_critical_section();  
}
```

3) Alternância obrigatória (1)

- variável *turn*, inicializada em **0**, estabelece qual processo pode entrar na SC

```
while (true) {  
    while(turn != 0); //loop  
    critical_section();  
    turn = 1;  
    no_critical_section();  
}
```

(processo pA)
turn 0

```
while (true){  
    while(turn != 1); //loop  
    critical_section();  
    turn = 0;  
    no_critical_section();  
}
```

(processo pB)
turn 1

3) Alternância obrigatória (2)

- Busy waiting
- Solução não satisfaz progresso
 - exige alternância na execução dos processos
- Só permite entrada alternada de dois processos na SC

4) Solução de Peterson (1)

- **flag**: array de 2 posições, indica quando o processo quer entrar na SC (1 → quer)
- **turn**: nº do processo com preferência no momento
- **p0** pode entrar na SC quando **p1** não quer entrar ou quando a preferência é p/ **p0**

```
flag[0]=1;
turn=1;
while (flag[1] && turn == 1);
    // loop
critical_section();
flag[0] = 0;
```

(processo p0)

```
flag[1]=1;
turn=0;
while (flag[0] && turn == 0);
    // loop
critical_section();
flag[1] = 0;
```

(processo p1)

4) Solução de Peterson (2)

- Garante exclusividade, progresso e espera limitada
- Busy waiting
- Resolve o problema da SC somente p/ 2 processos