Computação Concorrente (DCC/UFRJ)

Aula 4: Comunicação entre threads via memória compartilhada e sincronização com espera ocupada

Prof. Silvana Rossetto

27 de março de 2012

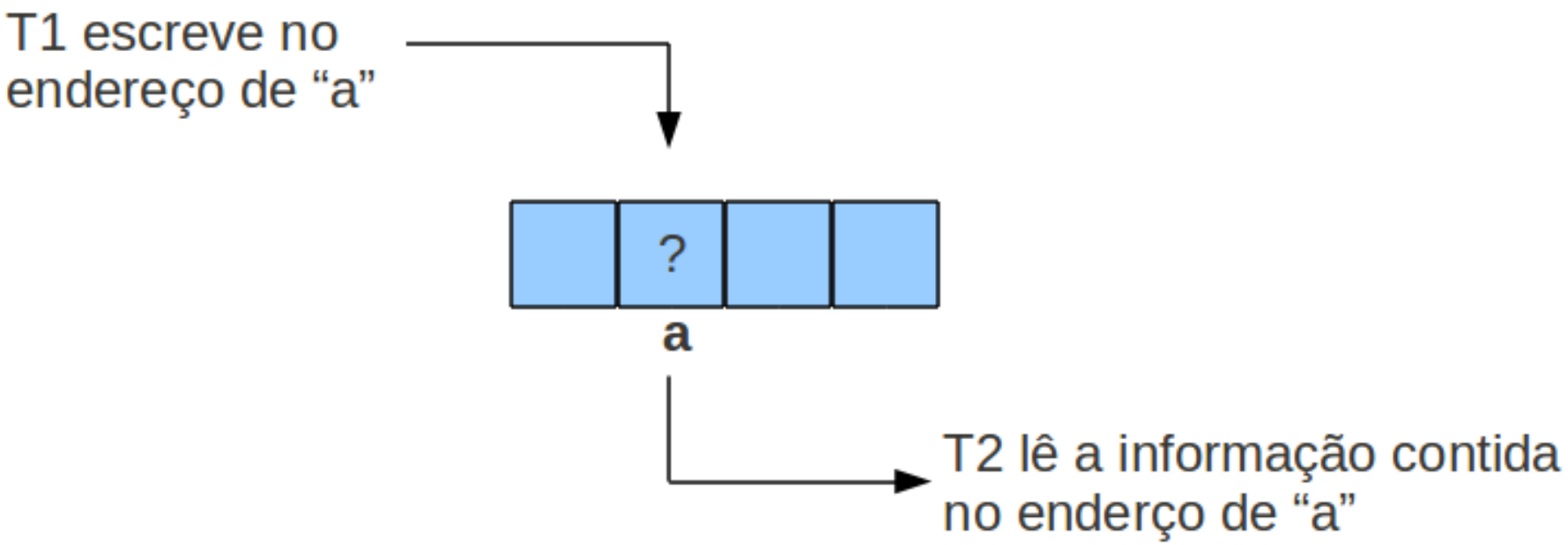
Comunicação entre threads

Para que as threads de uma aplicação trabalhem juntas, elas precisam **trocar informações entre si**

A facilidade de **espaço de endereçamento físico único** pode ser usada para implementar comunicação entre threads via **memória compartilhada**

Comunicação via memória compartilhada

- Quando uma thread tem um valor para ser comunicado para as demais threads, ela simplesmente **escreve esse valor na variável compartilhada**
- Quando outra thread precisa saber qual é o valor atual dessa informação, ela simplesmente **lê o conteúdo atual da variável compartilhada**



Comunicação e sincronização

Com memória compartilhada, a comunicação entre as threads é **assíncrona**: as threads escrevem/lêem valores nas variáveis compartilhadas a qualquer tempo

Para garantir que a comunicação ocorra de forma correta, a interação entre threads via memória compartilhada gera a necessidade de **sincronização** *ex.: uma variável não pode ser alterada enquanto outra thread não leu seu valor atual*

Sincronização por exclusão mútua e por condição

Sincronização refere-se a qualquer mecanismo que permite ao programador controlar a ordem relativa na qual as operações ocorrem em diferentes threads

Duas formas de sincronização aparecem em programas concorrentes de memória compartilhada:

1. **sincronização condicional**
2. **sincronização por exclusão mútua**

Sincronização por condição

- Visa garantir que **uma thread seja retardada enquanto uma determinada condição lógica da aplicação não for satisfeita**
- Exemplo: **problema produtor/consumidor**

A solução para a sincronização por condição é definida **impedindo a continuação da execução de uma thread até que o estado da aplicação seja correto para a sua execução**

Sincronização por exclusão mútua

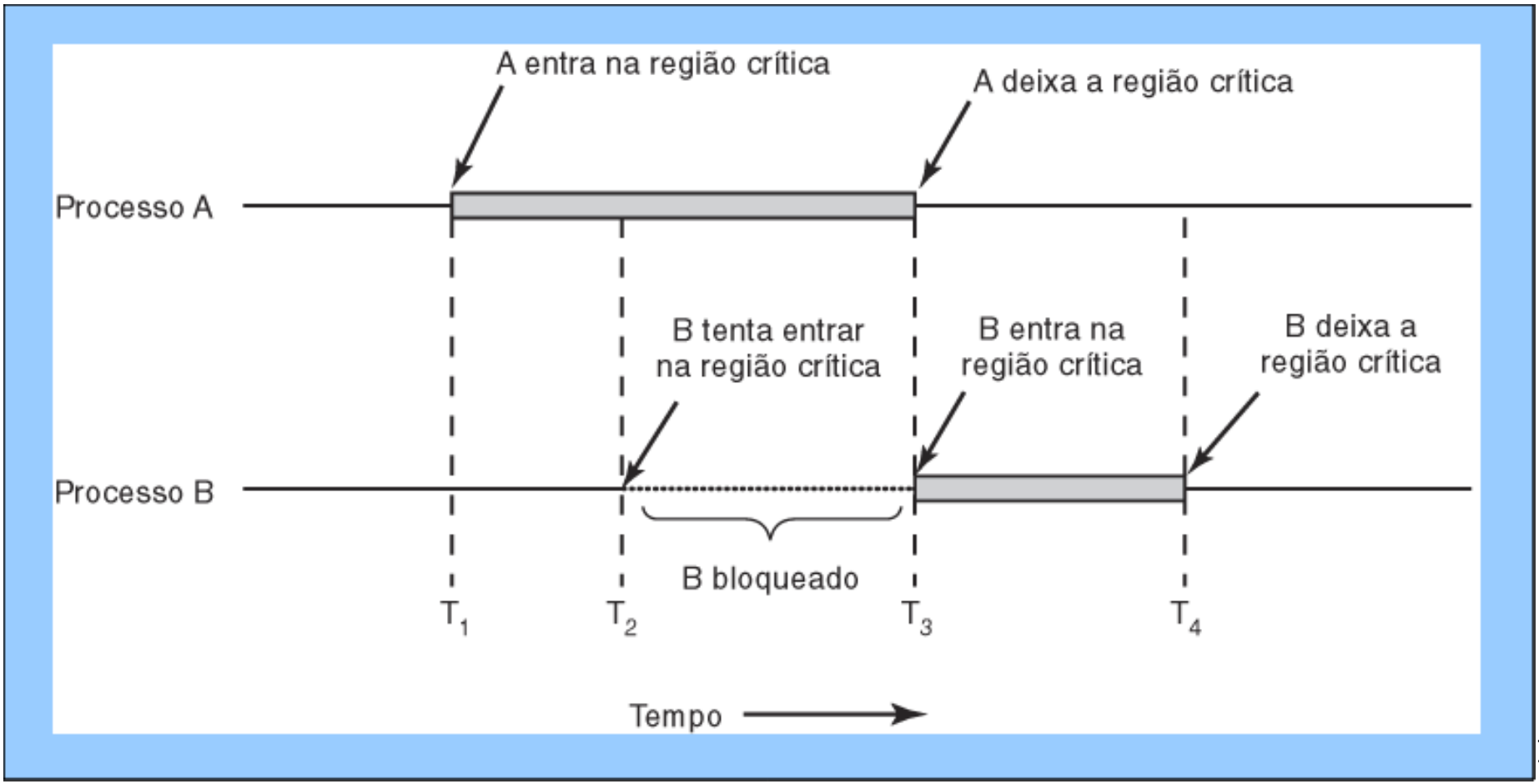
- Visa garantir que os **trechos de código em cada thread que acessam objetos compartilhados não sejam executados ao mesmo tempo**
- Restrição necessária para lidar com a possibilidade de inconsistência dos valores das variáveis compartilhadas (ex.: s++)

Solução para exclusão mútua

- Uma solução para a **exclusão mútua** consiste em combinar sequências contínuas de ações atômicas de hardware em seções críticas de software
- As **seções críticas** (trechos de código que acessam objetos compartilhados) devem ser transformadas em **ações atômicas**

Assim a execução de uma seção crítica NÃO ocorre simultaneamente com outra seção crítica que referencia a mesma variável

Controle de acesso à seção crítica



[1] *Fonte: Pearson*

Seções de entrada e saída da seção crítica

```
while (true) {
    requisita a entrada na seção crítica  //seção de entrada
    executa a seção crítica               //seção crítica
    sai da seção crítica                  //seção de saída
    executa fora da seção crítica
 }
```

Espera ocupada versus escalonamento

Há duas abordagens básicas para implementar a sincronização:

1. **por espera ocupada**: a thread fica continuamente testando o valor de uma determinada variável até que esse valor lhe permita executar a sua seção crítica com exclusividade
2. **por escalonamento**: são a alternativa mais usual, nas formas **semáforos** e **monitores**

Condições para implementar a exclusão mútua

1. Apenas uma thread na seção crítica a cada instante
2. Nenhuma suposição sobre velocidade das threads
3. Nenhuma thread fora da seção crítica pode impedir outra thread de continuar
4. Nenhuma thread deve esperar indefinidamente para executar a sua seção crítica

Casos de uso de espera ocupada

O problema da "espera ocupada" é o **desperdício de ciclos de CPU**, e só faz sentido nos seguintes casos:

- não há nada melhor para a CPU fazer enquanto espera
- o tempo de espera é menor que o tempo requerido para a troca de contexto entre threads

Solução (incorreta) 1

```
boolean queroEntrar_0 = false, queroEntrar_1 = false;
```

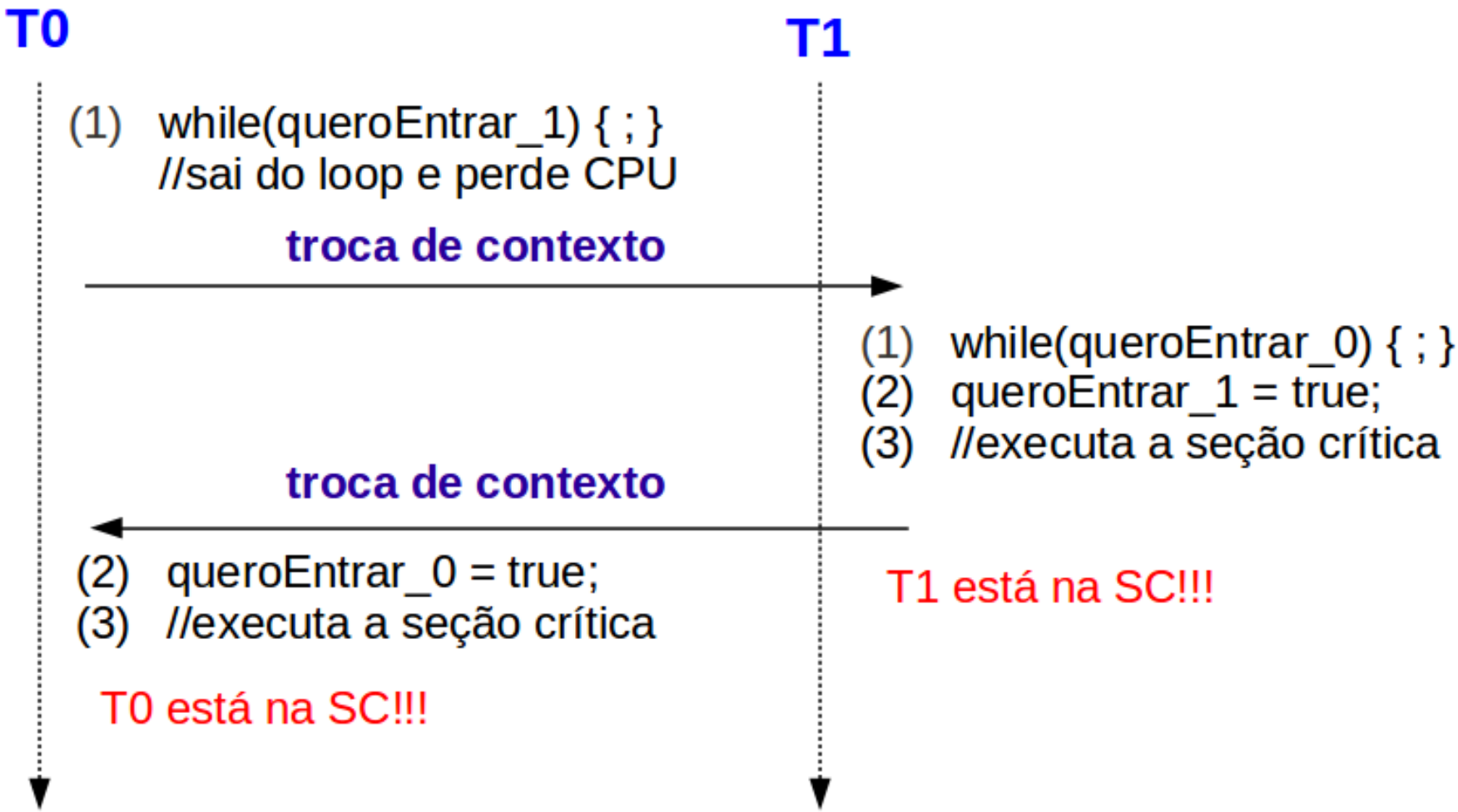
T0

```
while(true) {
(1)  while(queroEntrar_1) { ; }
(2)  queroEntrar_0 = true;
(3)  //executa a seção crítica
(4)  queroEntrar_0 = false;
(5)  //executa fora da seção crítica
}
```

T1

```
while(true) {
(1)  while(queroEntrar_0) { ; }
(2)  queroEntrar_1 = true;
(3)  //executa a seção crítica
(4)  queroEntrar_1 = false;
(5)  //executa fora da seção crítica
}
```

Exemplo de sequência de execução com erro



Solução (incorreta) 2

```
int TURN = 1;
```

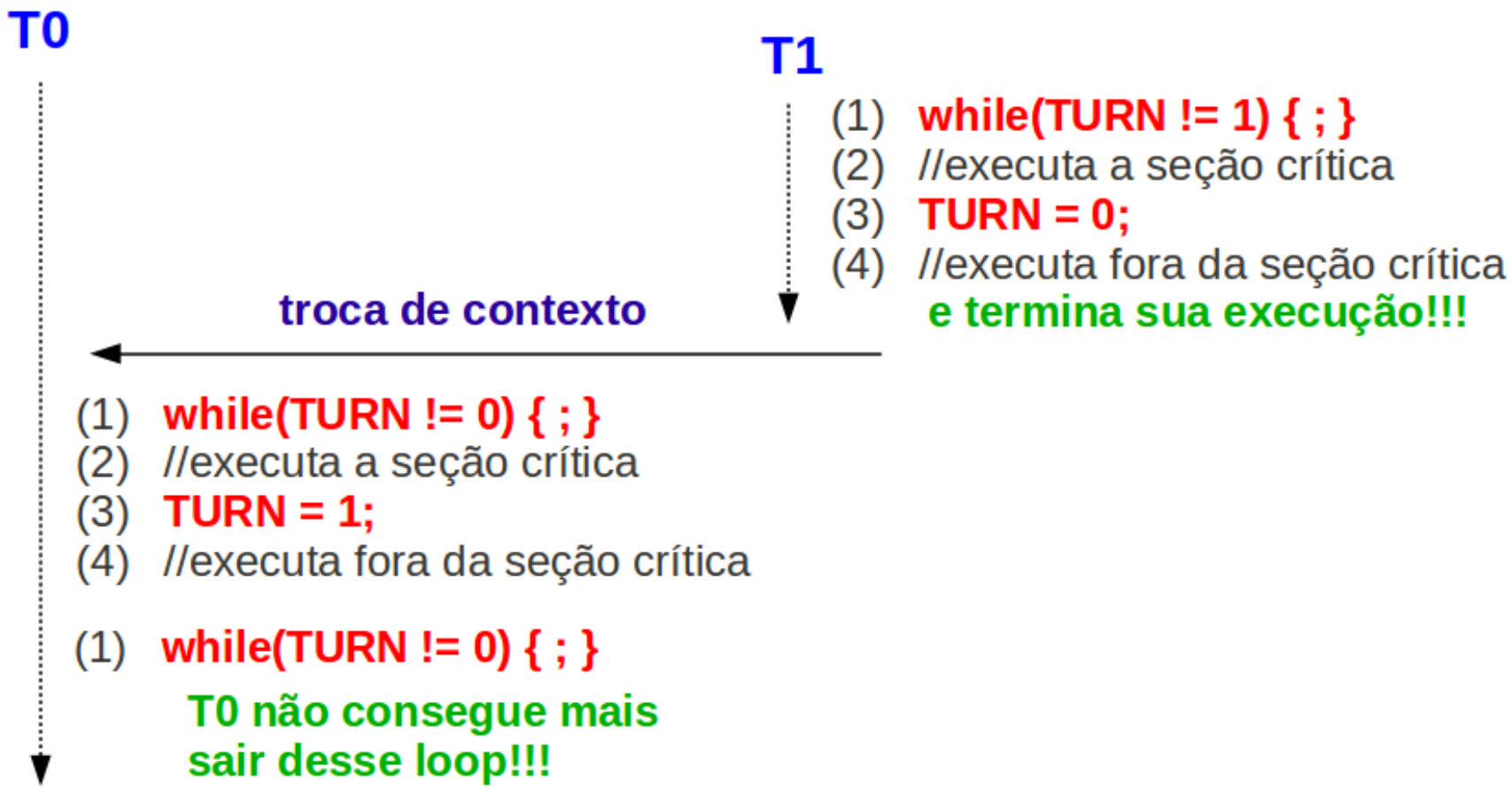
T0

```
while(true) {
(1)  while(TURN != 0) { ; }
(2)  //executa a seção crítica
(3)  TURN = 1;
(4)  //executa fora da seção crítica
}
```

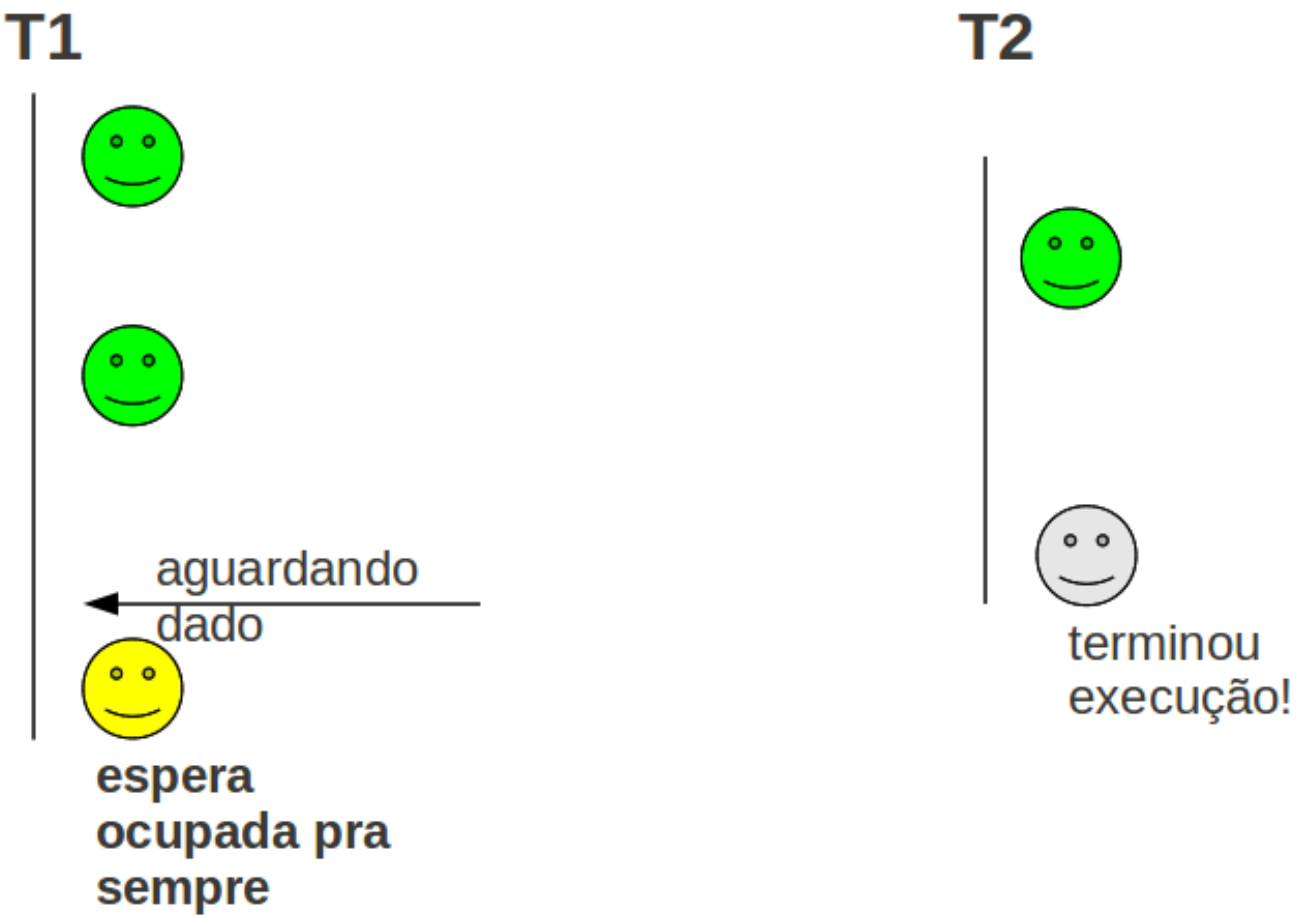
T1

```
while(true) {
(1)  while(TURN != 1) { ; }
(2)  //executa a seção crítica
(3)  TURN = 0;
(4)  //executa fora da seção crítica
}
```

Exemplo de sequência de execução (livelock)



Livelock


T1
T2
aguardando
dado
terminou
execução!
espera
ocupada pra
sempre

Solução (incorreta) 3

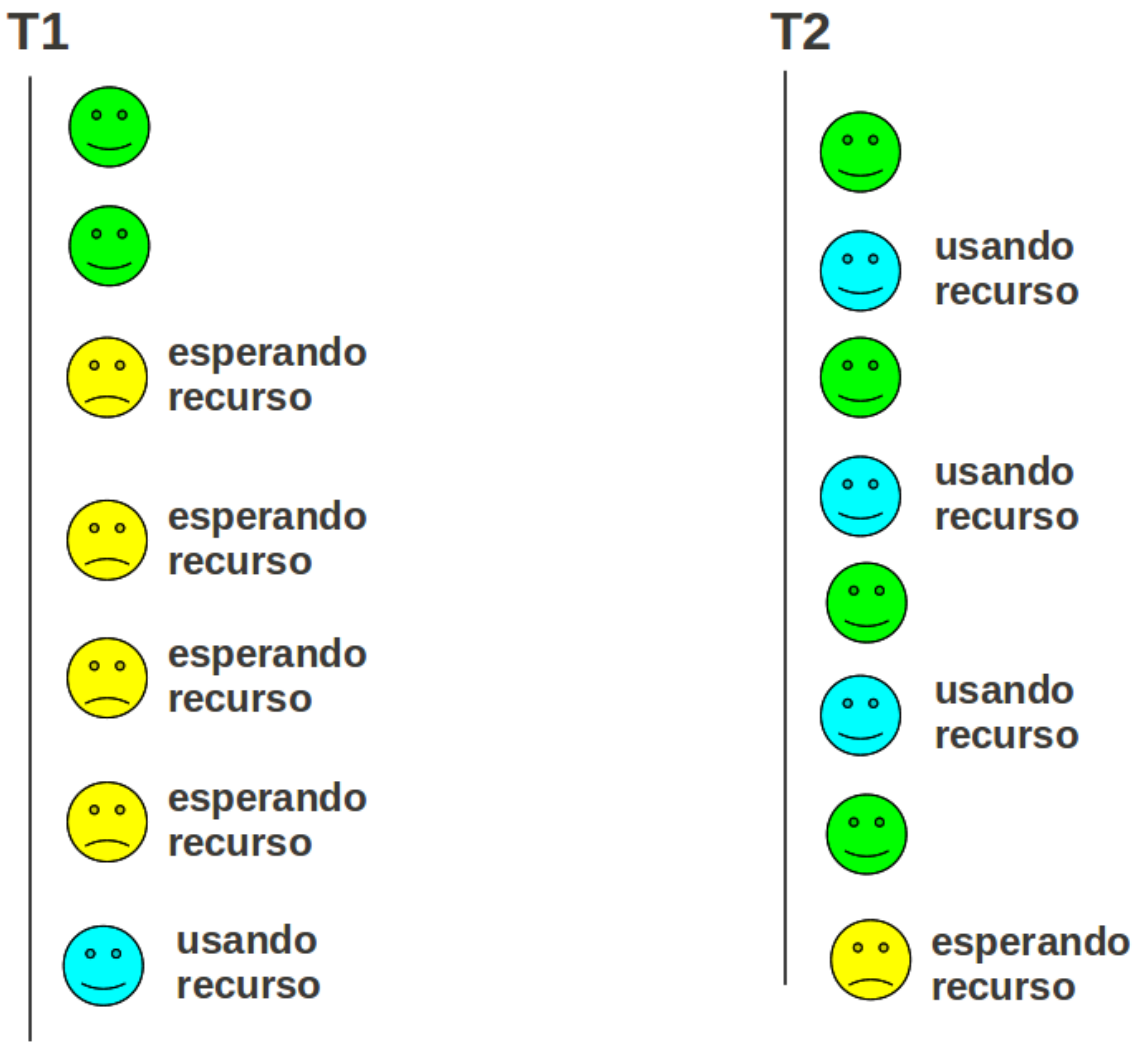
```
T0: while(true) {
(1)     queroEntrar-T0 = true;
(2)     while(queroEntrar-T1) {
(3)       queroEntrar-T0 = false;
(4)       while(queroEntrar-T1) {;}
(5)       queroEntrar-T0 = true;  }
(6)     //executa na seção crítica
(7)     queroEntrar-T0 = false;
(8)     //executa fora da seção crítica }
```

```
T1: while(true) {
(1)     queroEntrar-T1 = true;
(2)     while(queroEntrar-T0) {
(3)       queroEntrar-T1 = false;
(4)       while(queroEntrar-T0) {;}
(5)       queroEntrar-T1 = true;  }
(6)     //executa na seção crítica
(7)     queroEntrar-T1 = false;
(8)     //executa fora da seção crítica }
```

Exemplo de sequência de execução (starvation)

- T0 executa **(1), (2) e (6)**, T0 está na seção crítica e `queroEntrar-T0` é true
- T1 executa **(1), (2), (3) e (4)**, `queroEntrar-T1` é false e T1 está esperando `queroEntrar-T0` ficar false
- T0 executa **(7), (8), (1), (2) e (6)**, T0 está na seção crítica e `queroEntrar-T0` é true
- T1 retoma a execução em **(4)** e continua a esperar `queroEntrar-T0` ficar false
- T0 executa **(7), (8), (1), (2) e (6)**, T0 está na seção crítica e `queroEntrar-T0` é true
- T1 retoma a execução em **(4)** e continua a esperar `queroEntrar-T0` ficar false
- (...)

Starvation


T1
esperando recurso
esperando recurso
esperando recurso
esperando recurso
usando recurso
T2
usando recurso
usando recurso
usando recurso
esperando recurso

Solução de Peterson

```
boolean queroEntrar_0 = false, queroEntrar_1 = false; int TURN;
```

T0

```
while(true) {
(1)  queroEntrar_0 = true;
(2)  TURN = 1;
(3)  while(queroEntrar_1 &&
                TURN == 1) { ; }
(4)  //executa a seção crítica
(5)  queroEntrar_0 = false;
(6)  //executa fora da seção crítica
}
```

T1

```
while(true) {
(1)  queroEntrar_1 = true;
(2)  TURN = 0;
(3)  while(queroEntrar_0 &&
                TURN == 0) { ; }
(4)  //executa a seção crítica
(5)  queroEntrar_1 = false;
(6)  //executa fora da seção crítica
}
```

Solução de Peterson

Essa solução atende a todos os requisitos de uma solução para o problema de exclusão mútua?

Restrições da solução de Peterson

- A solução de Peterson pode não funcionar na presença de certas otimizações de compilação e de hardware (ex., o compilador pode permitir que cada thread mantenha cópias privadas das variáveis compartilhadas, alterações feitas nessas variáveis por uma thread não serão visíveis para a outra thread)
- Uma solução é declarar as variáveis compartilhadas como **volatile** (informa que a variável pode ser alterada por outras linhas de execução e as opções de otimização devem ser reduzidas)

Restrições da solução de Peterson

- O uso de níveis dintintos de memória cache pelo hardware da máquina também pode fazer a solução de Peterson falhar
- Mesmo com o uso do qualificador `volatile`, cópias temporárias da variável compartilhada podem ser mantidas em caches distintas, fazendo com que alterações no seu valor por uma thread não sejam vistas imediatamente por outras threads

Referências bibliográficas

- *Programming Language Pragmatics*, M.L.Scott, Morgan-Kaufmann, ed. 2, 2006
- *Modern Multithreading*, Carver e Tai, Wiley, 2006