Sistemas Operacionais

Interação entre tarefas - coordenação

Prof. Carlos Maziero

DInf UFPR, Curitiba PR

Julho de 2020

Conteúdo

Introdução

Sistemas complexos são implementados como **várias tarefas que cooperam entre si** para atingir os objetivos da aplicação.

A cooperação exige:

- **comunicar** informações entre as tarefas
- **coordenar** as tarefas para ter resultados coerentes

Este módulo apresenta os principais conceitos, problemas e soluções referentes à coordenação entre tarefas.

Condições de disputa

Função de depósito em uma conta bancária (simplificada):

```
1  void depositar (long * saldo, long valor)
2  {
3     (*saldo) += valor ;
4  }
```

Como fica essa função em código de máquina?

```
gcc -Wall -c -O0 depositar.c
objdump --no-show-raw-insn -d depositar.o
```

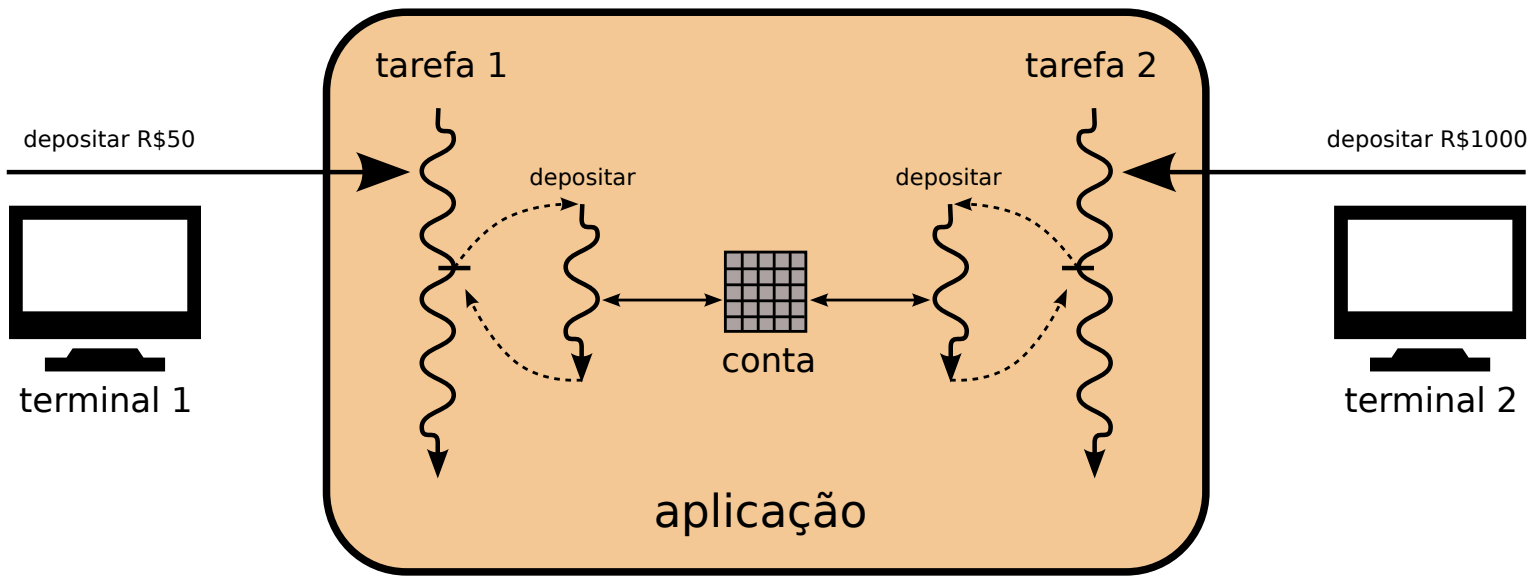
Em assembly (Intel 64 bits)

```
1  0000000000000000 <depositar>:
2        ; carrega o conteúdo da memória apontada por "saldo" em EDX
3        mov    -0x8(%rbp),%rax        ; saldo → rax (endereço do saldo)
4        mov    (%rax),%edx            ; mem[rax] → edx
5
6        ; carrega o conteúdo de "valor" no registrador EAX
7        mov    -0xc(%rbp),%eax        ; valor → eax
8
9        ; soma EAX ao valor em EDX
10       add    %eax,%edx              ; eax + edx → edx
11
12       ; escreve o resultado em EDX na memória apontada por "saldo"
13       mov    -0x8(%rbp),%rax        ; saldo → rax
14       mov    %edx,(%rax)            ; edx → mem[rax]
```

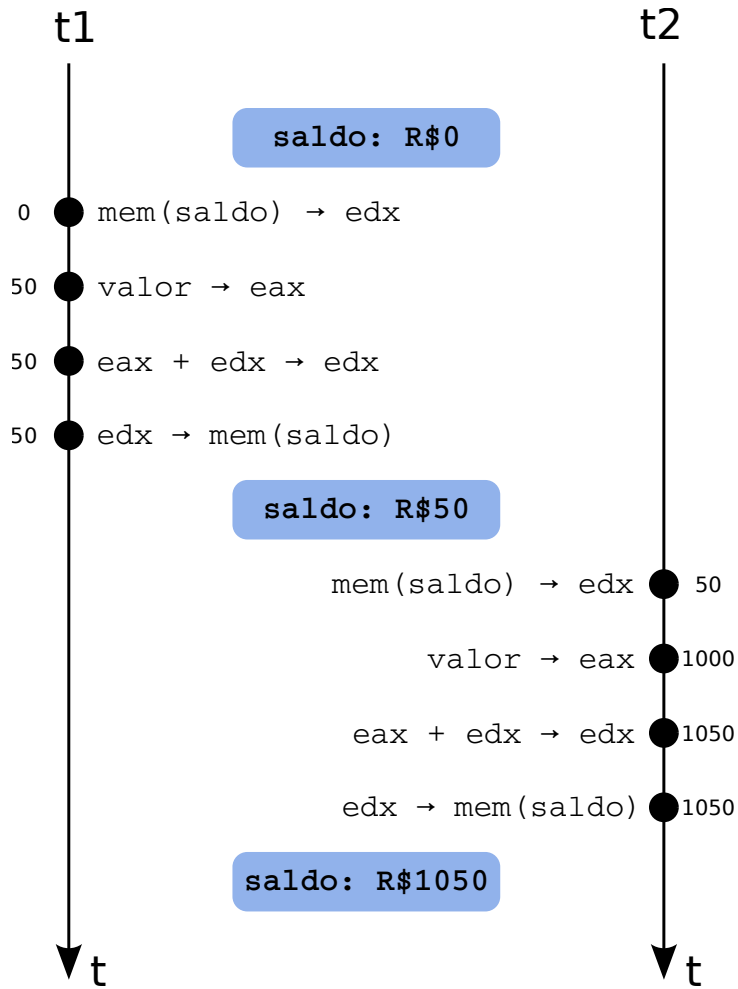
Atenção: as operações aritméticas ocorrem **nos registradores**

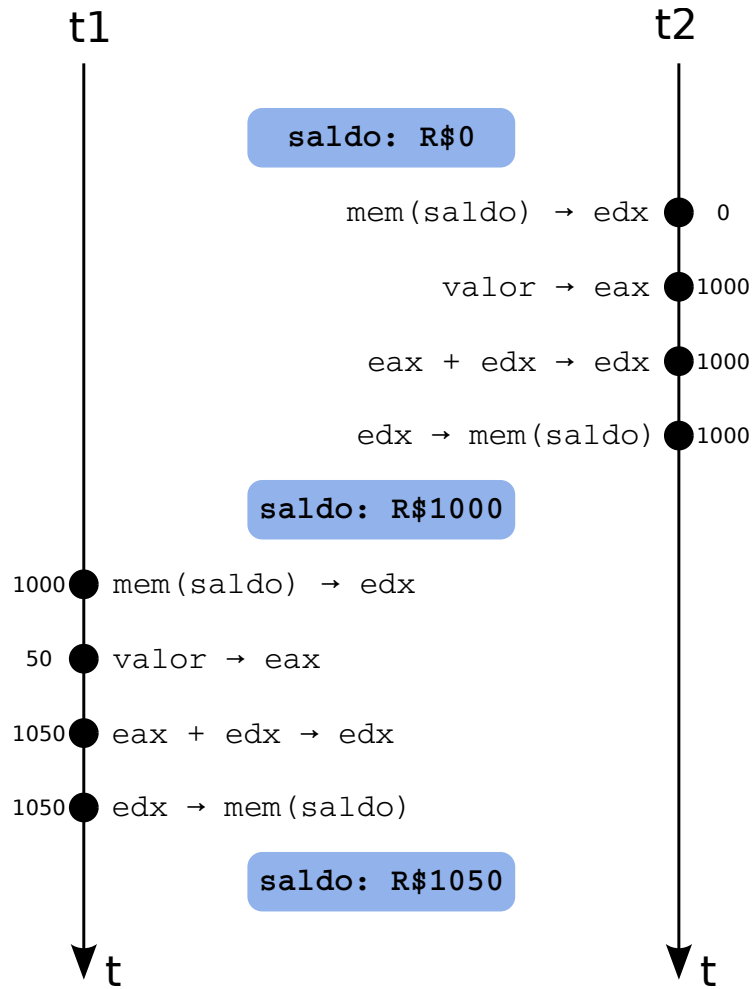
Condições de disputa

Dois clientes fazendo depósitos simultâneos:

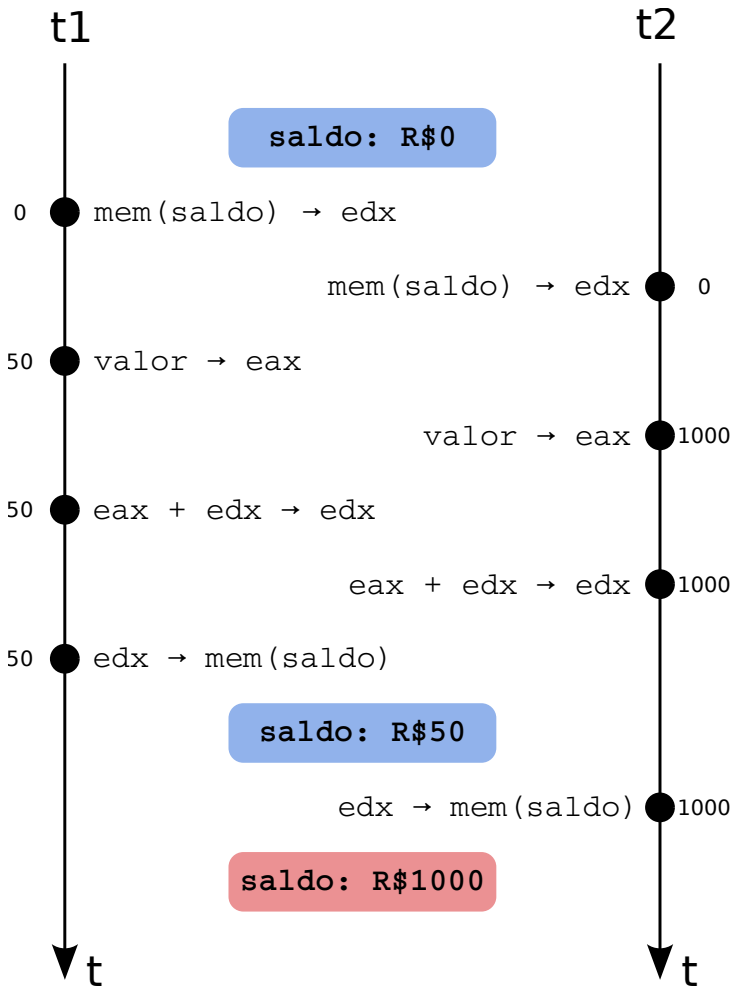


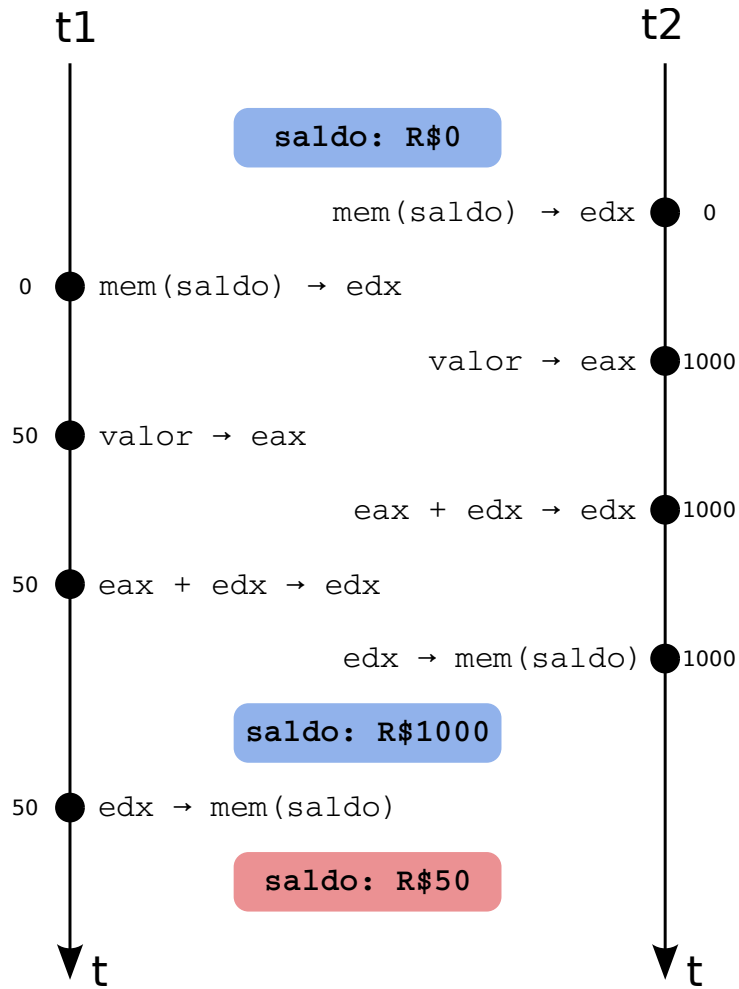
Execuções independentes





Execuções concorrentes





Condição de disputa

Definição:

- Erros gerados por acessos concorrentes a dados
- Podem ocorrer em qualquer sistema concorrente
- Envolvem ao menos **uma operação de escrita**
- Termo vem do inglês *Race Condition*

Condição de disputa

Condições de disputa são **erros dinâmicos**:

- Não aparecem no código fonte
- Só se manifestam durante a execução
- São difíceis de detectar
- Podem ocorrer raramente ou mesmo nunca
- Sua depuração pode ser muito complexa

É melhor **prevenir** condições de disputa que *consertá-las*.

Condições de Bernstein

Permitem formalizar as condições de disputa:

- t_1 e t_2 : duas tarefas executando em paralelo
- $\mathcal{R}(t_i)$: conjunto de variáveis lidas por t_i
- $\mathcal{W}(t_i)$: conjunto de variáveis escritas por t_i

t_1 e t_2 são independentes (sem condições de disputa) sse:

$$t_1 \parallel t_2 \iff \begin{cases} \text{// } t_1 \text{ não lê as variáveis escritas por } t_2 \\ \mathcal{R}(t_1) \cap \mathcal{W}(t_2) = \varnothing \\ \text{// } t_2 \text{ não lê as variáveis escritas por } t_1 \\ \mathcal{R}(t_2) \cap \mathcal{W}(t_1) = \varnothing \\ \text{// } t_1 \text{ e } t_2 \text{ não escrevem nas mesmas variáveis} \\ \mathcal{W}(t_1) \cap \mathcal{W}(t_2) = \varnothing \end{cases}$$

Seções críticas

Seção crítica

Trecho de código de cada tarefa que acessa dados compartilhados, onde podem ocorrer condições de disputa.

As seções críticas das tarefas t_1 e t_2 são:

```
1  (*saldo) += valor ;
```

Exclusão mútua

Impedir o entrelaçamento de seções críticas, de modo que apenas uma tarefa esteja na seção crítica a cada instante.

Exclusão mútua

Cada seção crítica i pode ser associada a um identificador cs_i.

Primitivas de controle:

- $enter(t_a, cs_i)$: a tarefa t_a deseja entrar na seção crítica cs_i
- $leave(t_a, cs_i)$, a tarefa t_a está saindo da seção crítica cs_i

A primitiva $enter(t_a, cs_i)$ é bloqueante: t_a aguarda até que cs_i esteja livre.

Exclusão mútua

O código da operação de depósito pode ser reescrito assim:

```
1  void depositar (long conta, long *saldo, long valor)
2  {
3     enter (conta) ;          // entra na seção crítica "conta"
4     (*saldo) += valor ;      // usa as variáveis compartilhadas
5     leave (conta) ;          // sai da seção crítica
6  }
```

A variável `conta` representa uma seção crítica.

Exclusão mútua

Os mecanismos de exclusão mútua devem garantir:

Exclusão mútua : só uma tarefa pode estar na seção crítica.

Espera limitada : a seção crítica é acessível em tempo finito.

Independência de outras tarefas : o acesso à seção crítica depende somente das tarefas que querem usá-la.

Independência de fatores físicos : o acesso não deve depender do tempo ou de outros fatores físicos.

Solução: inibir interrupções

Inibir as interrupções durante o acesso à seção crítica

Impedir as trocas de contexto dentro da seção crítica

Solução simples, mas raramente usada em aplicações:

- A preempção por tempo para de funcionar
- As interrupções de entrada/saída não são tratadas
- A tarefa que está na seção crítica não pode realizar E/S
- Só funciona em sistemas mono-processados

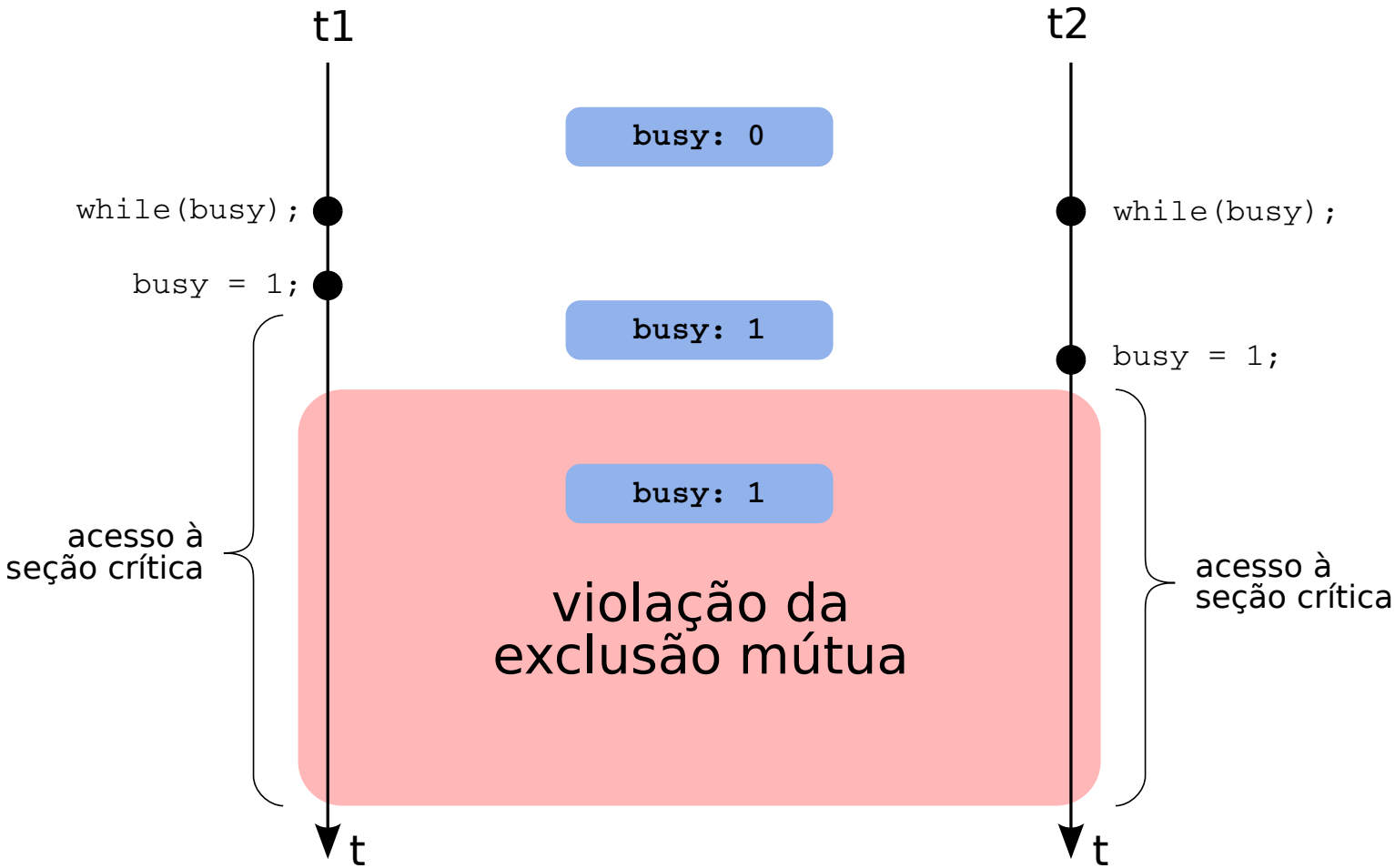
Usada em situações específicas dentro do núcleo do SO

Uma solução trivial

Usar uma variável *busy* para o status da seção crítica:

```
1  int busy = 0 ;          // a seção está inicialmente livre
2
3  void enter ()
4  {
5     while (busy) ;       // espera enquanto a seção estiver ocupada
6     busy = 1 ;           // marca a seção como ocupada
7  }
8
9  void leave ()
10 {
11    busy = 0 ;           // libera a seção (marca como livre)
12 }
```

Uma solução trivial



Solução: alternar o uso

Uma variável *turno* indica quem pode entrar na seção crítica:

```
1  int turn = 0 ;
2  int num_tasks ;
3
4  void enter (int task)       // task vale 0, 1, ..., num_tasks-1
5  {
6     while (turn != task) ;   // a tarefa espera seu turno
7  }
8
9  void leave (int task)
10 {
11    turn = (turn + 1) % num_tasks ; // turno da próxima tarefa
12 }
```

Sequência de acessos: $t_0 \longrightarrow t_1 \longrightarrow t_2 \longrightarrow \cdots \longrightarrow t_{n-1} \longrightarrow t_0 \longrightarrow \cdots$

A solução de Dekker & Peterson

Proposto por Dekker (1965) e melhorado por Peterson (1981):

```
1  int turn = 0 ;              // indica de quem éa vez
2  int wants[2] = {0, 0} ;     // a tarefa i quer acessar?
3
4  void enter (int task)       // task pode valer 0 ou 1
5  {
6    int other = 1 - task ;    // indica a outra tarefa
7    wants[task] = 1 ;         // task quer acessar a seção crítica
8    turn = other ;
9    while ((turn == other) && wants[other]) {} ; // espera ocupada
10 }
11
12 void leave (int task)
13 {
14   wants[task] = 0 ;         // task libera a seção crítica
15 }
```

Solução: Operações atômicas

Instruções de máquina específicas para exclusão:

- TSL - *Test and Set Lock*
- CAS - *Compare and Swap*
- XCHG - *Exchange*

Comportamento da instrução TSL:

$$TSL(x) = \begin{cases} old \leftarrow x & \text{// guarda o valor anterior} \\ x \leftarrow 1 & \text{// ativa o flag} \\ return(old) & \text{// devolve o valor anterior} \end{cases}$$

Solução: operações atômicas

Implementação de *enter* e *leave* usando TSL:

```
1  int lock = 0 ;                  // variável de trava
2
3  void enter (int *lock)          // passa o endereço da trava
4  {
5    while ( TSL (*lock) ) ;       // espera ocupada
6  }
7
8  void leave (int *lock)
9  {
10   (*lock) = 0 ;                 // libera a seção crítica
11 }
```

Usados para seções críticas no núcleo do SO (*Spinlocks*)

Problemas dessas soluções

As soluções vistas até agora têm problemas:

Ineficiência : teste contínuo de uma condição (**espera ocupada**); o ideal seria suspender as tarefas.

Injustiça : não garantem ordem no acesso; uma tarefa pode entrar e sair da seção crítica várias vezes em sequência.

Essas soluções são usadas apenas dentro do núcleo do SO e em sistemas simples.