

Sistemas Operacionais

Interação entre tarefas - coordenação

Prof. Carlos Maziero

DInf UFPR, Curitiba PR

Fevereiro de 2019

Conteúdo

- 1 Condições de disputa
- 2 Exclusão mútua
- 3 Mecanismos de coordenação
 - Inibição de interrupções
 - A solução trivial
 - Alternância de uso
 - Os algoritmos de Dekker & Peterson
 - Operações atômicas
- 4 Problemas

Introdução

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

- **comunicar** informações entre as tarefas
- **coordenar** as tarefas para que os resultados sejam coerentes

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

Condições de disputa

Operação de depósito em uma conta bancária (simplificada):

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

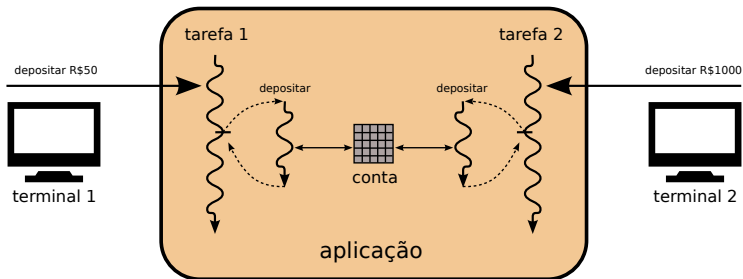
Em assembly (Intel 64 bits)

```

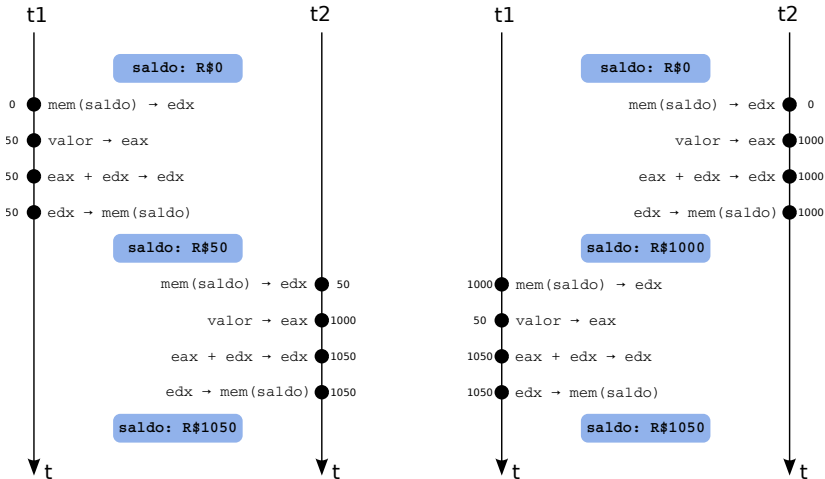
1  00000000000000000000 <depositar>:
2      ; carrega o conteúdo da memória apontada por "saldo" em EDX
3      mov    -0x8(%rbp),%rax      ; saldo → rax (endereço do saldo)
4      mov    (%rax),%edx         ; mem[rax] → edx
5
6      ; carrega o conteúdo de "valor" no registrador EAX
7      mov    -0xc(%rbp),%eax     ; 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]
15
16     ; finaliza a função
17     nop
18     pop    %rbp
19     retq
  
```

Condições de disputa

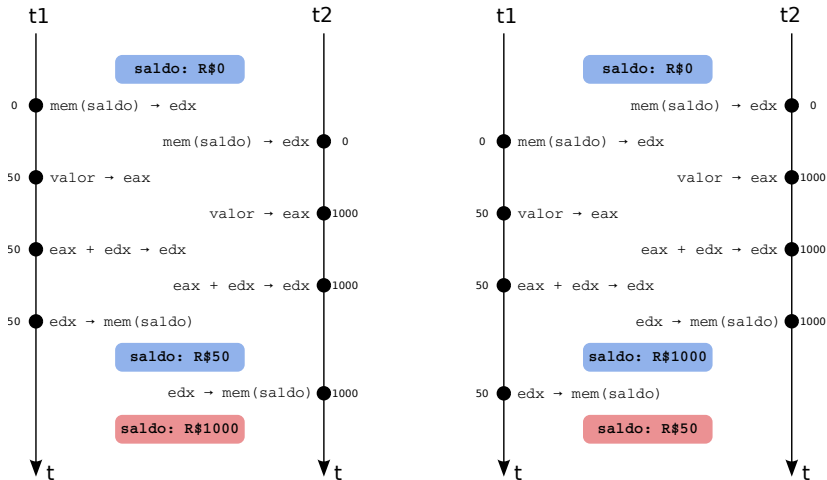
Dois clientes fazendo depósitos simultâneos:



Execuções independentes



Execuções concorrentes



Condição de disputa

Condições de disputa:

- 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 ($t_1 \parallel t_2$)
- $\mathcal{R}(t_i)$: conjunto de variáveis lidas por t_i
- $\mathcal{W}(t_i)$: conjunto de variáveis escritas por t_i

$$t_1 \parallel t_2 \iff \left\{ \begin{array}{l} // t_1 \text{ não lê as variáveis escritas por } t_2 \\ \mathcal{R}(t_1) \cap \mathcal{W}(t_2) = \emptyset \\ // t_2 \text{ não lê as variáveis escritas por } t_1 \\ \mathcal{R}(t_2) \cap \mathcal{W}(t_1) = \emptyset \\ // t_1 \text{ e } t_2 \text{ não escrevem nas mesmas variáveis} \\ \mathcal{W}(t_1) \cap \mathcal{W}(t_2) = \emptyset \end{array} \right.$$

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 conta.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 fica esperando 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 }
  
```

Exclusão mútua

Os mecanismos de exclusão mútua devem garantir:

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

Espera limitada : uma tarefa que pede acesso a uma seção crítica deve recebê-lo em um tempo finito.

Independência de outras tarefas : a decisão sobre o uso de uma seção crítica deve depender somente das tarefas que estão tentando usá-la.

Independência de fatores físicos : o mecanismo não deve depender da velocidade do sistema, de temporizações, do número de processadores ou de outros fatores físicos.

Inibição de 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

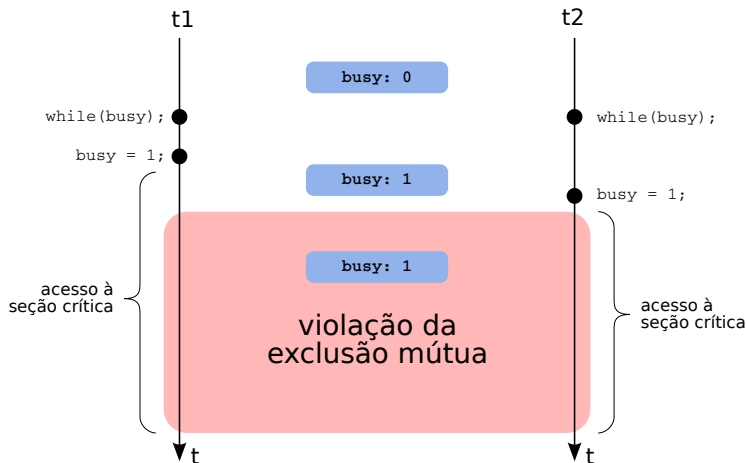
A solução trivial

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 }
```

A solução trivial

Problema: o teste da variável *busy* e sua atribuição são feitos em momentos distintos, gerando condições de disputa:



Alternância

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     if (turn < num_tasks-1) // o turno é da próxima tarefa
12         turn ++ ;
13     else
14         turn = 0 ;          // volta à primeira tarefa
15 }
```

Acesso em sequência: $t_0 \rightarrow t_1 \rightarrow t_2 \rightarrow \dots \rightarrow t_{n-1} \rightarrow t_0$

Os algoritmos de Dekker & Peterson

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

```

1  int turn = 0 ;           // indica de quem é a 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 = task ;
9     while ((turn == task) && wants[other]) {} ;
10 }
11
12 void leave (int task)
13 {
14     wants[task] = 0 ;      // task libera a seção crítica
15 }
  
```

Operações atômicas

Instruções de máquina específicas:

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

Comportamento da instrução TSL:

```
TSL(x) :   old ← x  
           x ← 1  
           return(old)
```

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 }
```

Spinlocks: usados para proteger seções críticas no núcleo do SO (descritores de tarefas, buffers de arquivos, ...)

Problemas

As soluções vistas até agora têm problemas:

Ineficiência : processador testa continuamente uma condição (**espera ocupada**); o ideal seria suspender as tarefas

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

Essas soluções são usadas apenas dentro do núcleo do sistema operacional e em sistemas mais simples.