Sistemas Operacionais

Interação entre tarefas - coordenação

Prof. Carlos Maziero

DInf UFPR, Curitiba PR

Julho de 2020

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  }
```

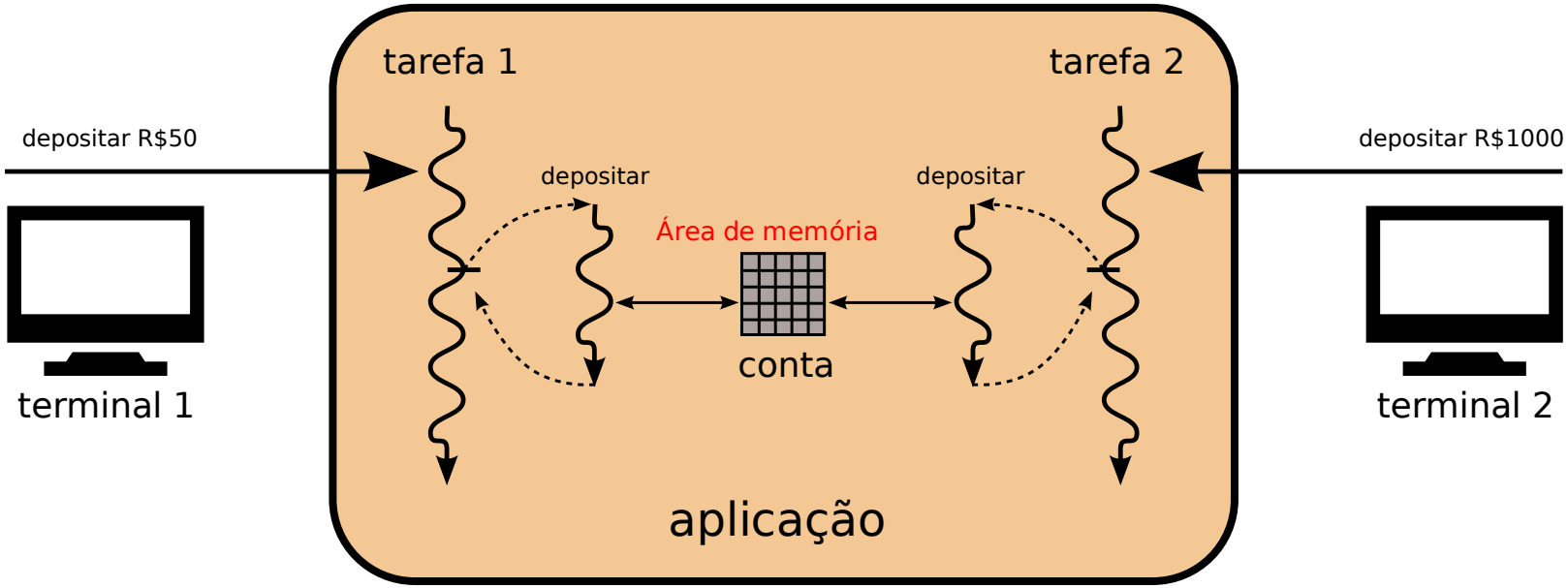
Como fica essa função em código de máquina?

~~gcc -Wall -c -O0 depositar.c~~

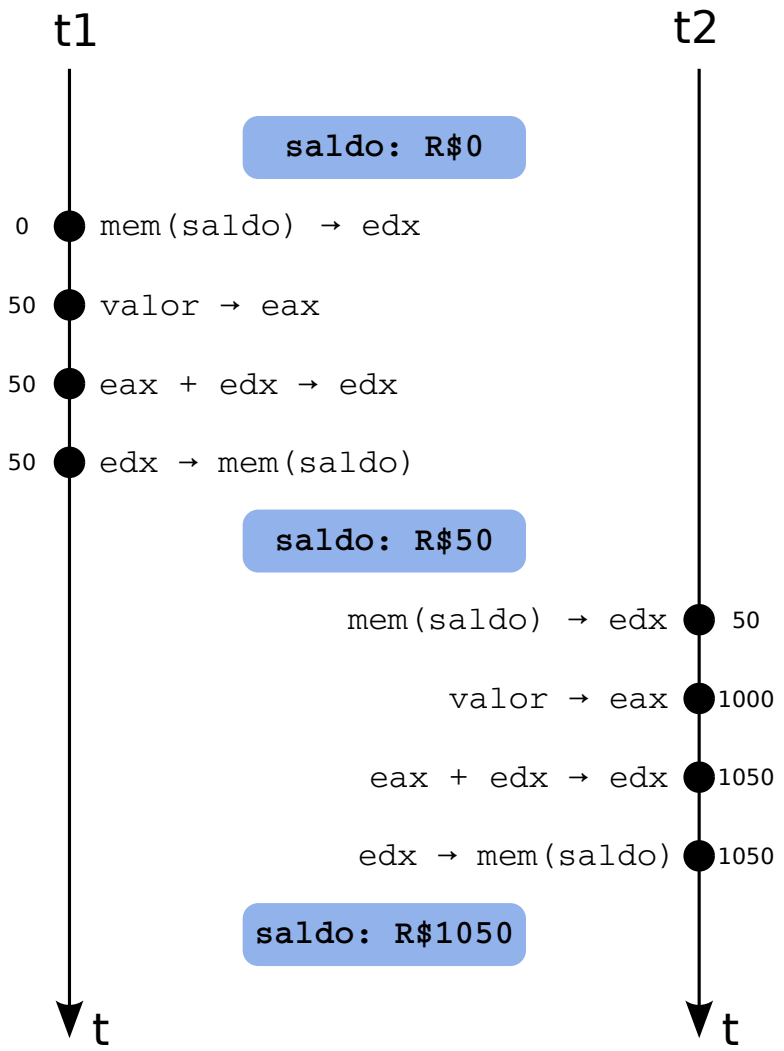
~~objdump --no-show-raw-insn -d depositar.o~~

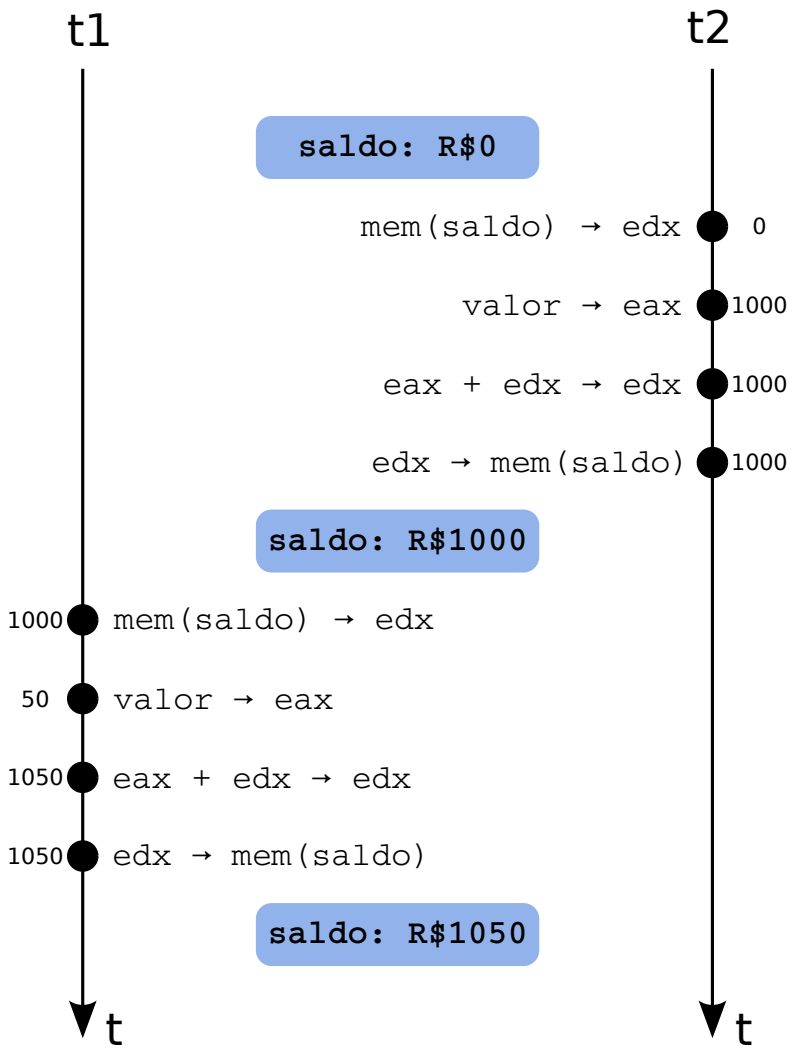
Condições de disputa

Dois clientes fazendo depósitos simultâneos:

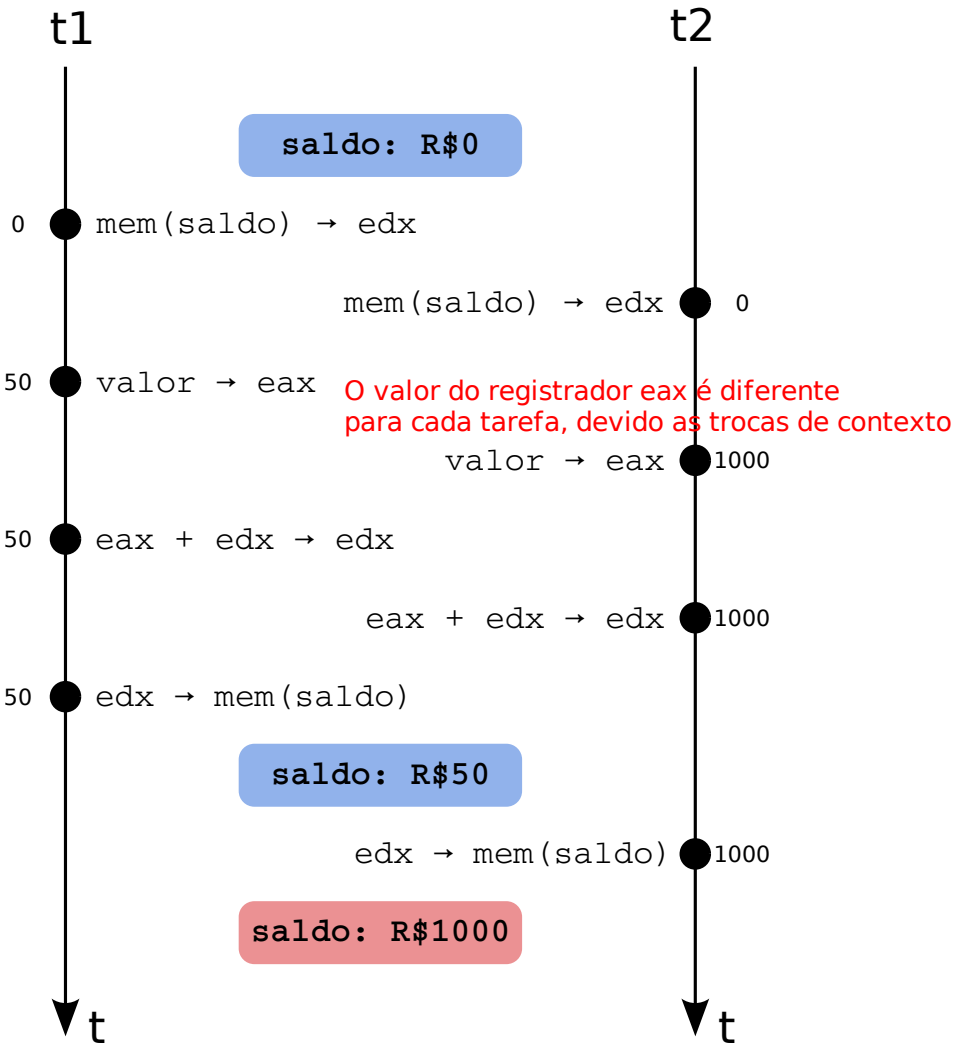


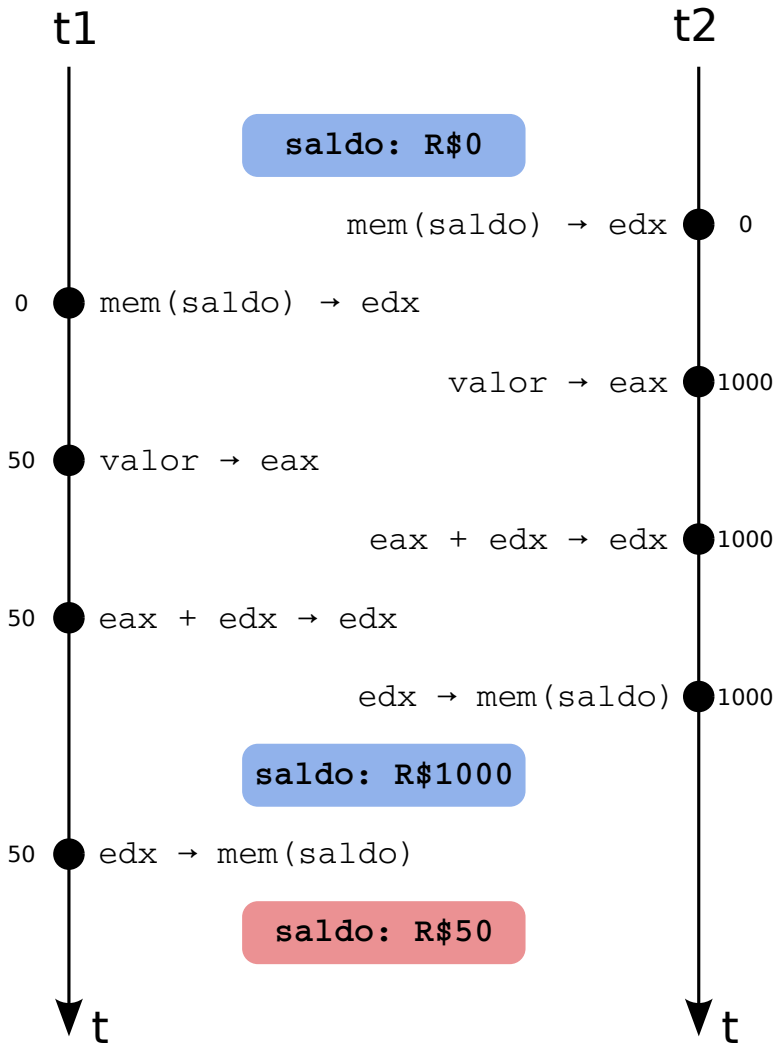
Execuções independentes





Execuções concorrentes





7.4 Problemas de Compartilhamento de Recursos

1ª Situação - compartilhamento de um arquivo em disco

```
PROGRAM Conta_Corrente;
    .
    .
    READ (Arq_Contas, Reg_Cliente);
    READLN (Valor_Dep_Ret);
    Reg_Cliente.Saldo := Reg_Cliente.Saldo + Valor_Dep_Ret;
    WRITE (Arq_Contas, Reg_Cliente);
    .
    .
END.
```

arquivo em disco

Caixa	Comando	Saldo arquivo	Valor dep/ret	Saldo memória
1	READ	1.000	*	1.000
1	READLN	1.000	-200	1.000
1	:=	1.000	-200	800
2	READ	1.000	*	1.000
2	READLN	1.000	+300	1.000
2	:=	1.000	+300	1.300
1	WRITE	800	-200	800
2	WRITE	1.300	+300	1.300

7.4 Problemas de Compartilhamento de Recursos

2ª Situação - variável na MP compartilhada por dois processos.

Inicialmente X=2;

Processo A	Processo B
X := X + 1 ;	X := X - 1;

Processo A	Processo B
LOAD x,R_a	LOAD x,R_b
ADD 1,R_a	SUB 1,R_b
STORE R_a,x	STORE R_b,x

Processo	Comando	X	R_a	R_b
A	LOAD X,R_a	2	2	*
A	ADD 1,R_a	2	3	*
B	LOAD X,R_b	2	*	2
B	SUB 1,R_b	2	*	1
A	STORE R_a,X	3	3	*
B	STORE R_b,X	1	*	1

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*.

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

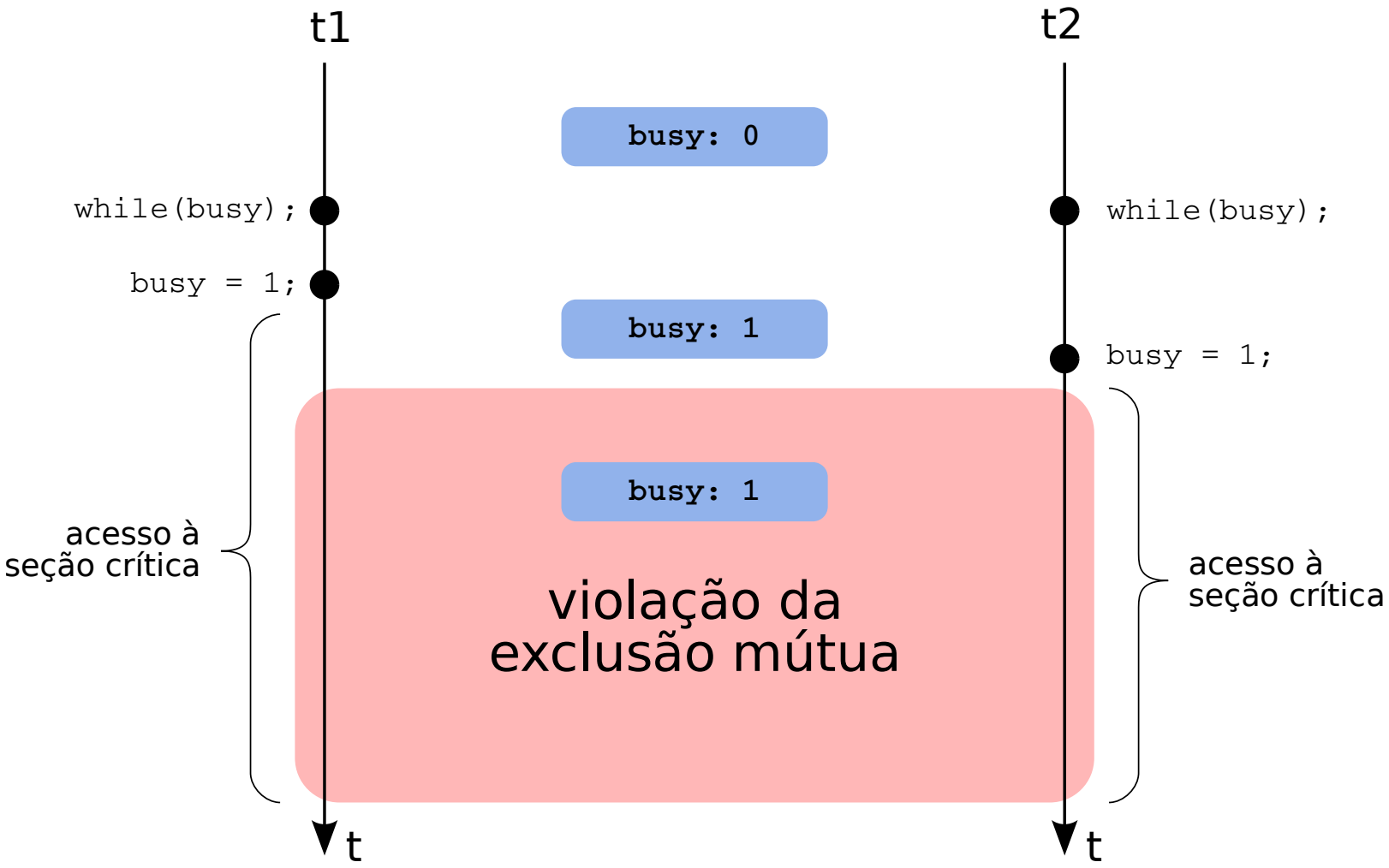
Usada em situações específicas dentro do núcleo do SO

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



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 }
```

7.5.2 Soluções de Software

Algoritmo de Dekker

Primeira solução de software que garantiu a exclusão mútua entre dois processos sem apresentar problemas, porém possui uma lógica bastante complexa.

7.5.2 Soluções de Software

Algoritmo de Peterson - 2 processos, CA, CB e Vez

```
PROGRAM Algoritmo_Peterson;
    VAR CA,CB: BOOLEAN;
        Vez : CHAR;
PROCEDURE Processo_A;
BEGIN
    REPEAT
        CA := true;
        Vez := 'B';
        WHILE (CB and Vez = 'B') DO (* Nao faz nada *);
        Regiao_Critica_A;
        CA := false;
        Processamento_A;
    UNTIL False;
END;

PROCEDURE Processo_B;
BEGIN
    REPEAT
        CB := true;
        Vez := 'A';
        WHILE (CA and Vez = 'A') DO (* Nao faz nada *);
        Regiao_Critica_B;
        CB := false;
         Processamento_B;
    UNTIL False;
END;

BEGIN
    CA := false;
    CB := false;
    PARBEGIN
        Processo_A;
        Processo_B;
    PAREND;
END.
```

7.5.2 Soluções de Software

Algoritmo para a exclusão mútua entre N processos

O algoritmo de Peterson depois foi generalizado para o caso de N processos e outros algoritmos foram desenvolvidos.

Deficiência: *espera ocupada (busy wait)*

Toda vez que um processo não consegue entrar em sua RC, por já existir outro processo com o recurso, o processo permanece em *looping*, testando uma condição, até que lhe seja permitido o acesso.

7.5.2 Soluções de Software

Deficiência: *espera ocupada (busy wait)*

Solução: introdução dos *mecanismos de sincronização* (semáforos e monitores). Coloca os processos no estado de espera.

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.

Sistemas Operacionais

Interação entre tarefas - mecanismos de coordenação

Prof. Carlos Maziero

DInf UFPR, Curitiba PR

Julho de 2020

Semáforo

Mecanismo proposto por Esdger **Dijkstra** em 1965.

Usado em várias situações de coordenação entre tarefas.

O semáforo provê:

- Eficiência: baixo consumo de CPU.
- Justiça: respeita a ordem das requisições.
- Independência de outras tarefas.

Base de muitos mecanismos de comunicação e coordenação.

Estrutura de um semáforo

Um semáforo s é uma **variável composta** por:

- Contador inteiro $s.counter$
- Fila de tarefas $s.queue$ (inicia vazia)
- Operações de acesso $down(s)$ e $up(s)$

O conteúdo interno do semáforo não é acessivel diretamente.

Imagine um “objeto semáforo” com atributos e métodos.

Acesso ao semáforo

É feito usando **operações atômicas**:

Operação *Down(s)* ou *P(s)*:

- Decrementa o contador do semáforo.
- Se < 0, **suspende** a tarefa e a põe na fila de s.

Operação *Up(s)* ou *V(s)*:

- Incrementa o contador do semáforo.
- Se ≤ 0, **acorda** uma tarefa da fila.

Operações

```
 1: procedure DOWN(t, s)
 2:     s.counter ← s.counter − 1
 3:     if s.counter < 0 then
 4:         append (t, s.queue)          ▷ põe t no final de s.queue
 5:         suspend (t)                  ▷ a tarefa t perde o processador
 6:     end if
 7: end procedure

 8: procedure UP(s)
 9:     s.counter ← s.counter + 1
10:     if s.counter ≤ 0 then
11:         u = first (s.queue)          ▷ retira a primeira tarefa de s.queue
12:         awake(u)                     ▷ devolve u à fila de tarefas prontas
13:     end if
14: end procedure

15: procedure INIT(s, v)
16:     s.counter ← v                    ▷ valor inicial do contador
17:     s.queue ← [ ]                    ▷ a fila inicia vazia
18: end procedure
```

Exclusão mútua usando semáforo

Cada recurso é representado por um semáforo:

- Operações *down*(s) para obter e *up*(s) para liberar.
- Iniciar o semáforo com o contador em 1.
- Somente uma tarefa obtém o semáforo por vez.

Código do depósito em conta bancária usando semáforos:

```
1  init (s, 1) ;                    // semáforo que representa a conta
2
3  void depositar (semaphore s, int *saldo, int valor)
4  {
5     down (s) ;                    // solicita acesso à conta
6     (*saldo) += valor ;           // seção crítica
7     up (s) ;                      // libera o acesso à conta
8  }
```

Mutex

Semáforo simplificado: *livre* ou *ocupado*. Binário

```c
#include <pthread.h>

// inicializa mutex, usando um struct de atributos
int pthread_mutex_init (pthread_mutex_t *restrict mutex,
                        const pthread_mutexattr_t *restrict attr);

// destrói uma variável do tipo mutex
int pthread_mutex_destroy (pthread_mutex_t *mutex);

// solicita acesso à seção crítica protegida pelo mutex;
// se a seção estiver ocupada, bloqueia a tarefa
int pthread_mutex_lock (pthread_mutex_t *mutex);

// libera o acesso à seção crítica protegida pelo mutex
int pthread_mutex_unlock (pthread_mutex_t *mutex);
```

Monitor

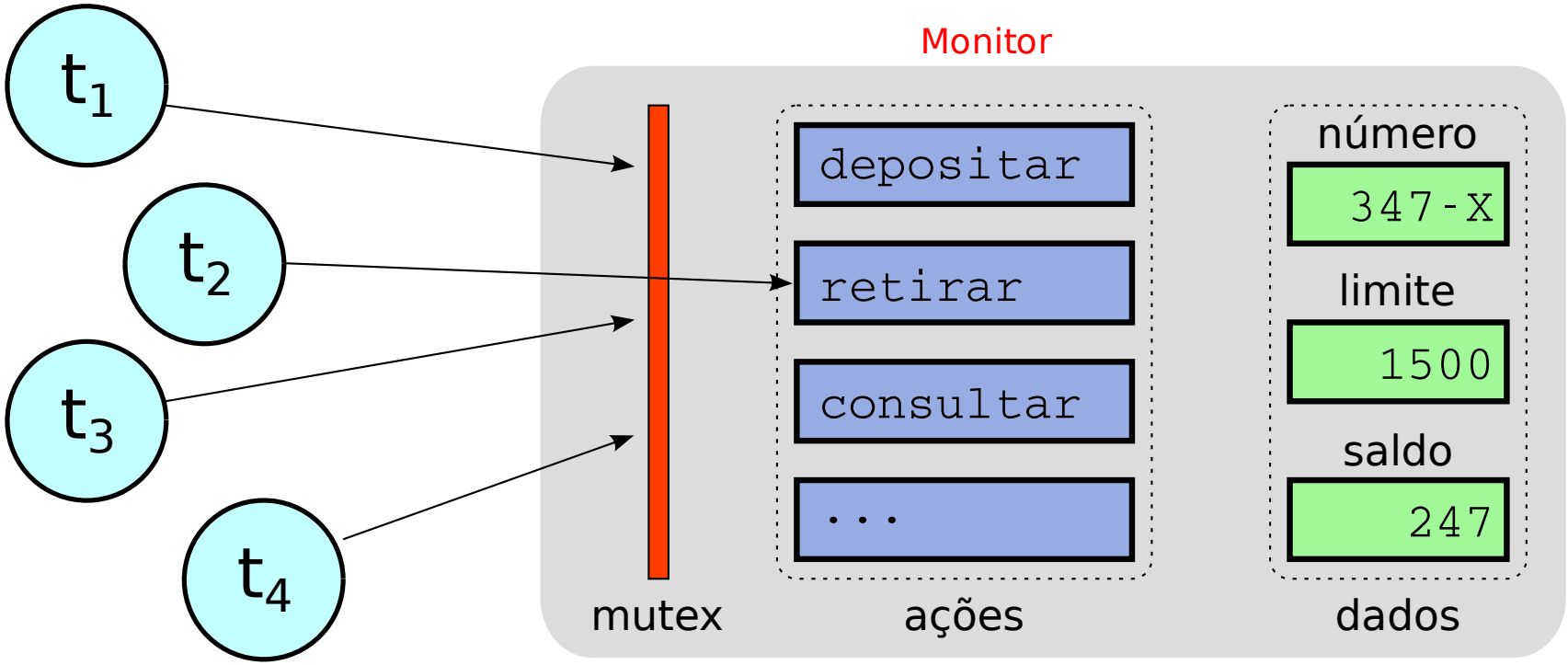
Definição

- Estrutura de sincronização associada a um recurso.
- Requer e libera a seção crítica de forma **transparente**.
- O programador não precisa mais se preocupar com isso.

Componentes de um monitor:

- Recurso compartilhado (conjunto de variáveis).
- Procedimentos para acessar essas variáveis.
- Um *mutex*, usado em cada acesso ao monitor.

Estrutura de um monitor



Pense em um monitor como um “objeto sincronizado”.

Estrutura de um Monitor

```
1  monitor conta
2  {
3     float saldo = 0 ;                          // recurso (variáveis)
4     float limite ;
5
6     void depositar (float valor)               // ação sobre o recurso
7     {
8        if (valor >= 0)
9           conta->saldo += valor ;
10       else
11          error ("erro: valor negativo\n") ;
12    }
13
14    void retirar (float saldo)                 // ação sobre o recurso
15    {
16       if (valor >= 0)
17          conta->saldo -= valor ;
18       else
19          error ("erro: valor negativo\n") ;
20    }
21 }
```

Um monitor em Java

```
1  class Conta
2  {
3     private float saldo = 0;
4
5     public synchronized void depositar (float valor)
6     {
7        if (valor >= 0)
8           saldo += valor ;
9        else
10          System.err.println("valor negativo");
11    }
12
13    public synchronized void retirar (float valor)
14    {
15       if (valor >= 0)
16          saldo -= valor ;
17       else
18          System.err.println("valor negativo");
19    }
20 }
```