IFSP – SÃO JOÃO DA BOA VISTA
CIÊNCIA DA COMPUTAÇÃO

Sistemas Operacionais

SEMANA 11

Prof.: Ederson Borges

Sistemas Operacionais

Tópicos

- DeadLock
  - Coordenação entre tarefas
    - Concorrência
      - Seção Crítica
    - Exclusão Mútua
    - Problemas na coordenação
  - Mecanismos de coordenação
    - Semáforos
    - Mutex
    - Monitores
  - DeadLocks (impasses)

Coordenação entre tarefas

- Coordenação entre tarefas
  - Já discutimos a questão de comunicação entre tarefas
    - Existem tarefas que precisam trabalhar juntas
    - Elas precisam cooperar
    - Exemplo:
      - Dentro de um navegador Web diversas tarefas executam diferentes atividades
      - As diferentes atividades se relacionam
        - » Botões do navegador
        - » Tela de navegação

Coordenação entre tarefas

- Coordenação entre tarefas
  - Como as tarefas precisam se comunicar e, muitas vezes, utilizar os mesmos recursos, como controlar que um mesmo recurso não seja “disputado” (e corrompido) pelas diferentes tarefas
    - Coordenação entre tarefas

Sistemas Operacionais

Coordenação entre tarefas

- Concorrência
  - Recursos em um computador são compartilhados
    - Arquivos
    - Rede
    - Periféricos...
  - O que pode ocorrer se duas ou mais tarefas acessarem um recurso simultaneamente?
    - Inconsistências
    - Mudança de estado do recurso provoca resultados inválidos
  - Temos o problema da ***CONCORRÊNCIA***

Coordenação entre tarefas

- Concorrência
  - Vejamos o exemplo abaixo:

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

Coordenação entre tarefas

- Concorrência
  - Assembly:

```
0000000000000000 <depositar>:
       ; inicializa a função
       push  %rbp
       mov   %rsp,%rbp
       mov   %rdi,-0x8(%rbp)
       mov   %esi,-0xc(%rbp)

       ; carrega o conteúdo da memória apontada por "saldo" em EDX
       mov   -0x8(%rbp),%rax       ; saldo → rax (endereço do saldo)
       mov   (%rax),%edx           ; mem[rax] → edx

       ; carrega o conteúdo de "valor" no registrador EAX
       mov   -0xc(%rbp),%eax       ; valor → eax

       ; soma EAX ao valor em EDX
       add   %eax,%edx             ; eax + edx → edx

       ; escreve o resultado em EDX na memória apontada por "saldo"
       mov   -0x8(%rbp),%rax       ; saldo → rax
       mov   %edx,(%rax)           ; edx → mem[rax]

       ; finaliza a função
       nop
       pop   %rbp
       retq
```

Coordenação entre tarefas

- Concorrência
  - Vamos imaginar a situação
    - Cliente 1 quer depositar na conta 1
    - Cliente 2 quer depositar na conta 2

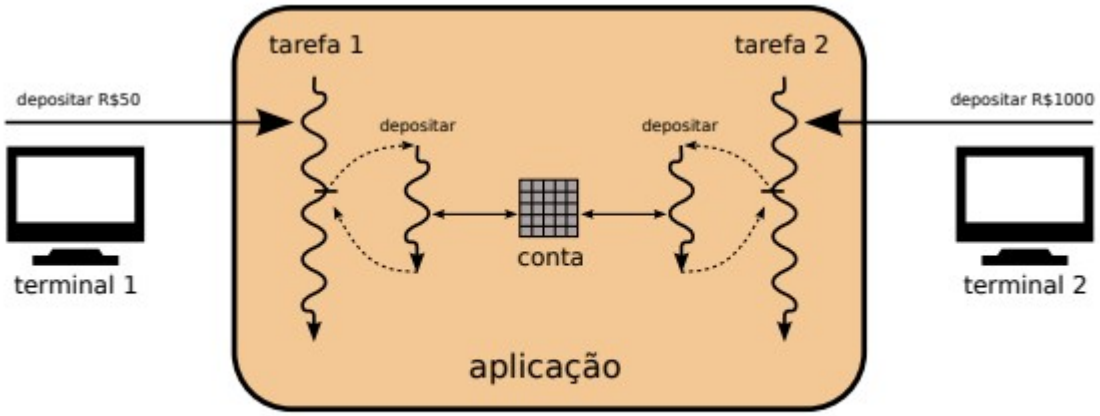

Figura 10.1: Acessos concorrentes a variáveis compartilhadas.

Coordenação entre tarefas

- Concorrência
  - A situação ideal será:

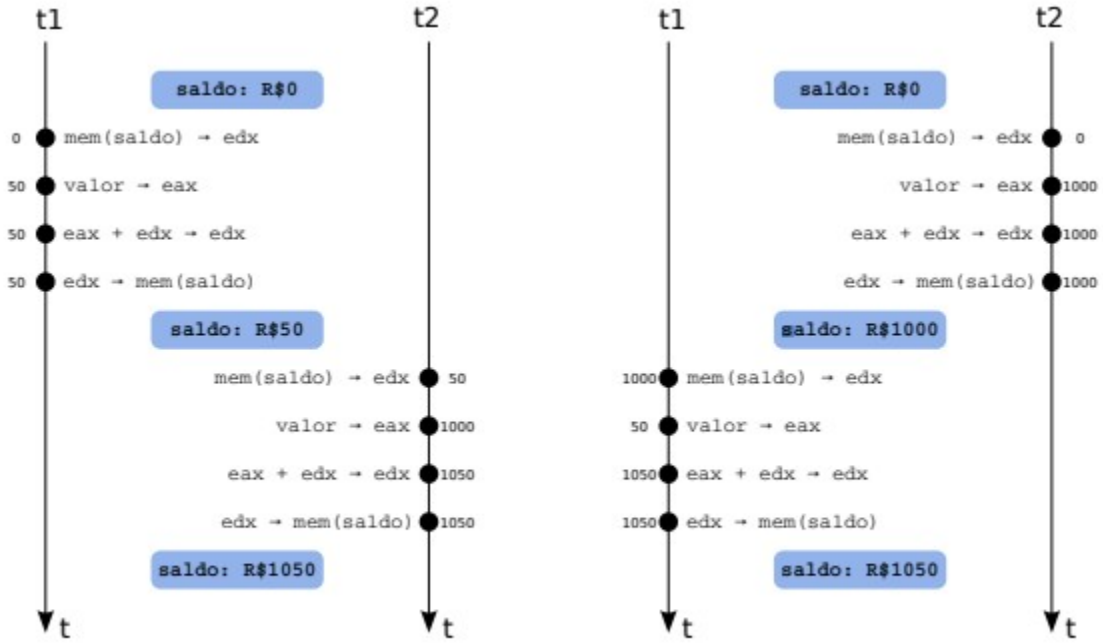

Figura 10.2: Operações de depósitos não-concorrentes.

Sistemas Operacionais

Coordenação entre tarefas

- Concorrência
  - Mas em caso de concorrência:

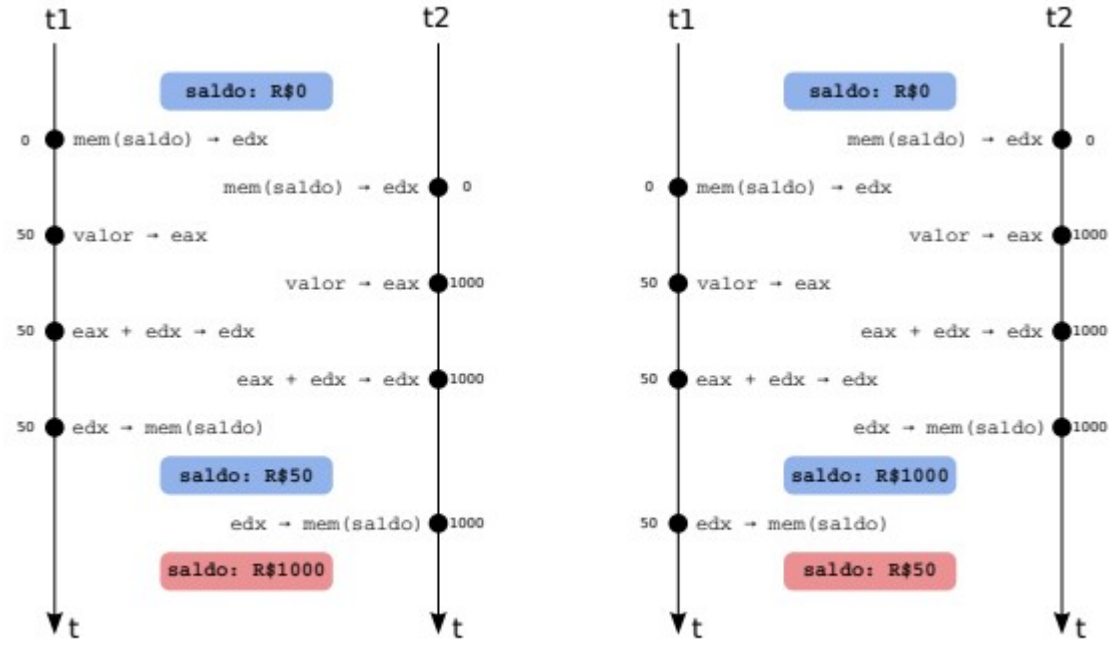

Figura 10.3: Operações de depósito concorrentes.

Sistemas Operacionais

Coordenação entre tarefas

- Concorrência
  - Erros e inconsistências dessa forma são chamados de **condições de disputa**
    - Várias tarefas acessando de forma concorrente um mesmo recurso
      - Memória (variáveis), arquivos, etc
    - Os erros são dinâmicos
      - Não existe erro no código fonte
      - Erros que não se manifestam a toda execução do programa

Coordenação entre tarefas

- Concorrência
  - ***SEÇÃO CRÍTICA***
    - Trechos de código que acessam dados compartilhados em cada tarefa

      (*saldo) += valor;
    - Esses trechos de código manipulam dados compartilhados que sofrem da condição de disputa
    - Para evitar o erro anterior, uma seção crítica só pode ser acessada por uma tarefa
      - A tarefa inicia a sua execução na seção crítica, enquanto não finalizar, outra tarefa não pode executar instruções da mesma seção crítica
      - Para isso ocorrer veremos o que é a ***EXCLUSÃO MÚTUA***

Coordenação entre tarefas

- Exclusão Mútua
  - Consiste em garantir que se um processo está usando um recurso compartilhado os outros processos estão impedidos de fazê-lo
    - Existem diversas formas de implementar a Exclusão Mútua

Coordenação entre tarefas

- Exclusão Mútua

```
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  }
```

  - No exemplo anterior
    - enter(conta) é um seção crítica
    - leave(conta) é a saída da seção crítica

Coordenação entre tarefas

- Exclusão Mútua
  - Critérios para implementação de Exclusão Mútua
    - Apenas uma tarefa na seção crítica
    - A espera para entrar deve ser finita
    - Não existe influência de tarefas externas
      - Tarefas que não queiram entrar na seção crítica
    - Independência de fatores físicos
      - Velocidade de processamento
      - Temporizadores
      - Número de processadores

Coordenação entre tarefas

- Exclusão Mútua
  - Soluções para Exclusão Mútua
    - Inibição de interrupções
    - Solução trivial (variável)
    - Alternância de uso
    - Algoritmo de Peterson
    - Operações atômicas

Sistemas Operacionais

Coordenação entre tarefas

- Inibição de interrupções
  - Se impedirmos a troca de contexto, uma tarefa não irá "invadir" a execução de outra
    - Exclusão Mútua
  - Problemas
    - Preempção por tempo para de funcionar
    - Dispositivos de entrada/saída são ignorados
    - Tarefa na seção crítica não faz operações de entrada/saída
    - Não pode existir multiprocessadores

Coordenação entre tarefas

- Solução trivial
  - Utilizar uma variável compartilhada indicando a presença de uma tarefa na 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 }
```

Coordenação entre tarefas

- Solução trivial
  - Problemas
    - Troca de contexto entre passo 5 e 6

```
5 | while (busy) {} ;   // espera enquanto a seção estiver ocupada
6 | busy = 1 ;          // marca a seção como ocupada
```

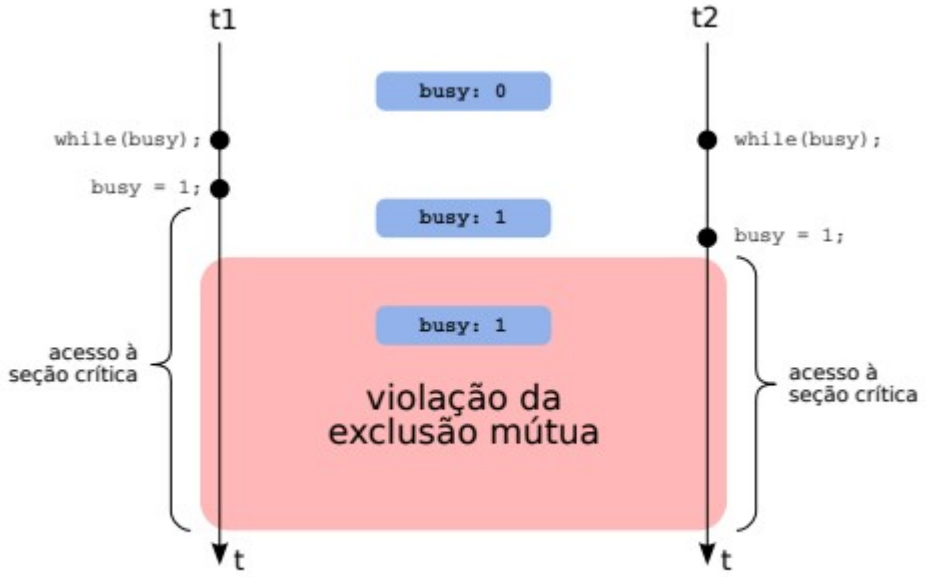

Figura 10.4: Condição de disputa no acesso à variável *busy*.

Coordenação entre tarefas

- Alternância de uso

```
1  int num_tasks ;
2  int turn = 0 ;                    // inicia pela tarefa 0
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 ; // passa para a próxima tarefa
12 }
```

  - E se a tarefa t2 não quiser usar a seção crítica?
    - Todas depois de t2 ficam impedidas de entrar na seção crítica
    - Variável *turn* não é alterada

Sistemas Operacionais

Coordenação entre tarefas

- Algoritmo de Peterson
  - *Solução considerada ideal para 2 TAREFAS*
    - *Existem generalizações para mais tarefas*

```
1  int turn = 0 ;           // indica de quem éa vez
2  int wants[2] = {0, 0} ; // indica se a tarefa i quer acessar a seção crítica
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 }
```

Sistemas Operacionais

Coordenação entre tarefas

- Operações atômicas
  - Vamos lembrar do problema da solução trivial

```
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 }
```

  - As linhas 5 e 6 podem ocorrer em tempos diferentes
    - Troca de contexto
  - Mas e se fosse possível garantir que fossem executadas ao mesmo tempo?
    - Execução atômica

Sistemas Operacionais

Coordenação entre tarefas

- Operações atômicas
  - Existem instruções de hardware para isso
    - TSL
      - *Test-and-Set Lock*
    - XCHG
      - Exchange
        - » Processadores AMD e Intel

Coordenação entre tarefas

- Operações atômicas
  - XCHG

```
1  int lock ;                          // variável de trava
2
3  enter (int *lock)
4  {
5     int key = 1 ;                    // variável auxiliar (local)
6     while (key)                      // espera ocupada
7       XCHG (lock, &key) ;            // alterna valores de lock e key
8  }
9
10 leave (int *lock)
11 {
12    (*lock) = 0 ;                    // libera a seção crítica
13 }
```

Coordenação entre tarefas

- Operações atômicas
  - Instruções atômicas são usadas no interior do sistema operacional
    - Controlam o acesso a seções críticas dentro do núcleo
    - Muitas vezes denominados *spinlocks*

Coordenação entre tarefas

- Problemas para coordenação entre tarefas
  - A exclusão mútua, como visto, é uma forma de coordenar acessos múltiplos a um mesmo recurso
  - Mas mesmo utilizando as formas descritas aqui, existem problemas
    - **Ineficiência**: Testes contínuos para acesso a SC
    - **Injustiça**: Não há garantia de acesso na ordem
    - **Dependência**: Tarefas que não querem acesso a SC podem afetar outras tarefas (solução de alternância)

Mecanismos de coordenação

- Mecanismos de coordenação
  - Já que as soluções anteriores não são suficiente para os problemas de concorrência
    - **Semáforos**
    - **Mutex**
    - Variáveis de condição
    - **Monitores**

Sistemas Operacionais

Mecanismos de coordenação

- Semáforos
  - Edsger Dijkstra
  - 1965
  - Exclusão Mútua entre *n* tarefas
  - Apesar do ano, semáforo é utilizado até hoje em diversos algoritmos

Sistemas Operacionais

Mecanismos de coordenação

- Semáforos
  - S: estrutura de dados que contém
    - Counter: contador inteiro interno
    - Queue: fila de tarefas, inicialmente vazia
    - Operações
      - Down: decrementa o contador e o testa
        - » Se negativo a tarefa solicitante entra na fila e fica suspensa
        - » Se positivo down retorna e a tarefa executa
      - Up: incrementa o contador interno e testa
        - » Se negativo (ou nulo): existem tarefas suspensas, devolver a tarefa para fila de pronta

Sistemas Operacionais

Mecanismos de coordenação

- Semáforos

Algoritmo 1 Operações sobre semáforos

Require: as operações devem executar atomicamente

t: tarefa que invocou a operação
s: semáforo, contendo um contador e uma fila
v: valor inteiro

```
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 a tarefa 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
```

Sistemas Operacionais

Mecanismos de coordenação

- Semáforos
  - Operações no semáforo são atômicas
    - Executadas como chamada de sistema
  - Semáforos podem ser utilizados para Exclusão Mútua
    - Down indica a entrada na seção crítica (tentativa)
    - Up indica a saída da seção crítica (final de uso do recurso)

Mecanismos de coordenação

- Semáforos
  - Características
    - Eficiente: Tarefas aguardando a seção crítica são suspensas, não consomem processador
    - Justiça: Existe uma fila, apenas a próxima tarefa é liberada para fila de tarefa pronta
    - Independência: Só quem solicita o semáforo que são consideradas para decisão da próxima a obter a seção crítica

Sistemas Operacionais

Mecanismos de coordenação

- Semáforos
  - Outras formas de utilizar semáforo
    - Estacionamento
      - Controle de vagas
        - » Contador do semáforo tem o número de vagas
        - » Ao entrar um carro o contador é decrementado
        - » Chegando a negativo, não existem mais vagas (espera)
        - » Ao sair um carro o contador é incrementado (nova vaga)

Sistemas Operacionais

Mecanismos de coordenação

- Mutexes
  - Mutual exclusion
  - Semáforos simplificados
    - Contador interno é 0 (livre) ou 1 (ocupado)
  - Diversas linguagens possuem mutex
    - C/C++
    - Python
    - Java
    - C#

Sistemas Operacionais

Mecanismos de coordenação

- Monitores
  - Quando se usa semáforo (ou mutex) o programador deve indicar os pontos exatos de uso destes mecanismos
  - Caso esqueça, algum problema poderá ocorrer
    - Inconsistência de dados
    - Bloqueio permanente de tarefas
    - Outros...

Mecanismos de coordenação

- Monitores
  - É uma estrutura de sincronização que requisita e libera a seção crítica associada a um recurso de forma transparente
    - 1972
    - Per Brinch e Charles Hoare

Mecanismos de coordenação

- Monitores
  - Consiste de:
    - Recurso compartilhado (conjunto de variáveis internas ao monitor)
    - Procedimentos e funções que acessam as variáveis internas
    - Mutex (semáforo) para exclusão mútua

Mecanismos de coordenação

- Monitores
  - Conta: recurso compartilhado

```
1  monitor conta
2  {
3     string numero ;
4     float saldo = 0.0 ;
5     float limite ;
6
7     void depositar (float valor)
8     {
9        if (valor >= 0)
10          conta->saldo += valor ;
11       else
12          error ("erro: valor negativo\n") ;
13    }
14
15    void retirar (float saldo)
16    {
17       if (valor >= 0)
18          conta->saldo -= valor ;
19       else
20          error ("erro: valor negativo\n") ;
21    }
22 }
```

Mecanismos de coordenação

- Monitores

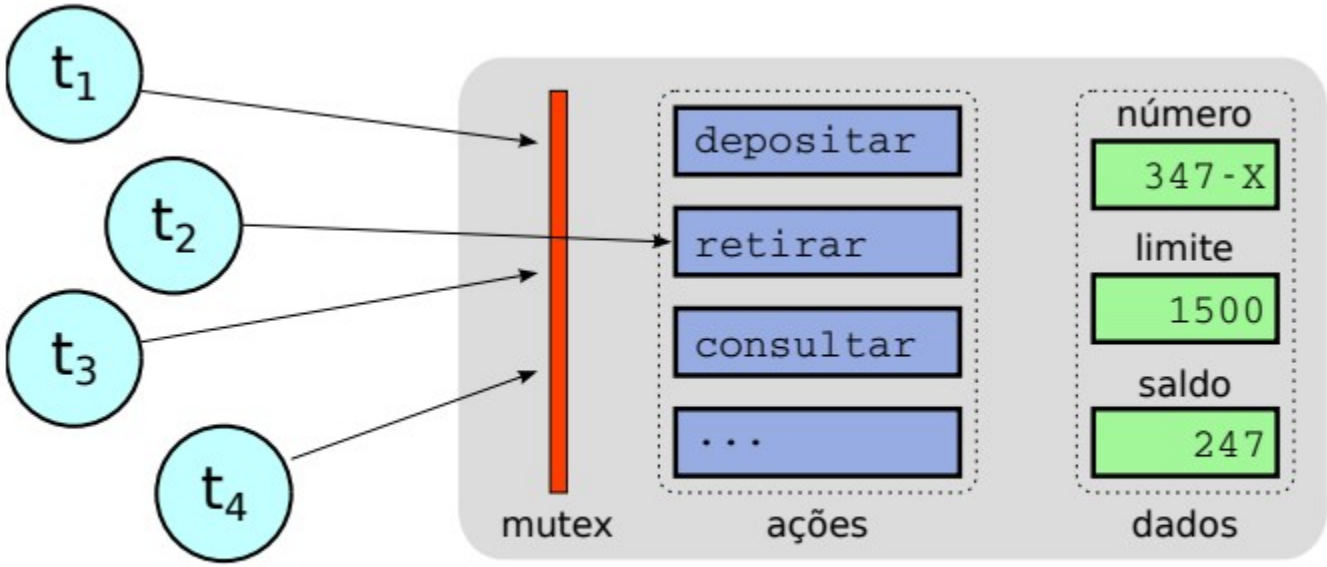

Figura 11.1: Estrutura básica de um monitor de sincronização.

Sistemas Operacionais

Mecanismos de coordenação

- Monitores

```
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 }
```

Sistemas Operacionais

Mecanismos de coordenação

- Atividades
  - Em que situações um semáforo deve ser inicializado em 0, 1 ou n > 1?
  - Sobre as afirmações a seguir, relativas aos mecanismos de coordenação, indique quais são incorretas, justificando sua resposta:
    - A estratégia de inibir interrupções para evitar condições de disputa funciona em sistemas multi-processados.
    - Os mecanismos de controle de entrada nas regiões críticas provêem exclusão mútua no acesso às mesmas.
    - Condições de disputa ocorrem devido às diferenças de velocidade na execução dos processos.
    - Condições de disputa ocorrem quando dois processos tentam executar o mesmo código ao mesmo tempo.
    - Instruções do tipo *Test&Set Lock* devem ser implementadas pelo núcleo do SO.
    - O algoritmo de Peterson garante justiça no acesso à região crítica.
    - Uma forma eficiente de resolver os problemas de condição de disputa é introduzir pequenos atrasos nos processos envolvidos.

DEADLOCK

DeadLock

- Na Concorrência existe a suspensão de tarefas ao tentar usar recurso compartilhado que já está sendo acessado
  - São associados semáforos (mutex) para os recursos compartilhados
- Semáforos (mutex) podem levar a situações em que as tarefas envolvidas são bloqueadas para sempre
  - **DEADLOCK**
    - Traduções: ***Impasse***

Sistemas Operacionais

DeadLock

Figura 13.2: Uma situação de impasse no trânsito.

Sistemas Operacionais

DeadLock

```
1  typedef struct conta_t
2  {
3     int saldo ;                    // saldo atual da conta
4     mutex m ;                      // mutex associado à conta
5     ...                            // outras informações da conta
6  } conta_t ;
7
8  void transferir (conta_t* contaDeb, conta_t* contaCred, int valor)
9  {
10    lock (contaDeb->m) ;           // obtém acesso a contaDeb
11    lock (contaCred->m) ;          // obtém acesso a contCred
12
13    if (contaDeb->saldo >= valor)
14    {
15       contaDeb->saldo -= valor ;  // debita valor de contaDeb
16       contaCred->saldo += valor ; // credita valor em contaCred
17    }
18    unlock (contaDeb->m) ;         // libera acesso a contaDeb
19    unlock (contaCred->m) ;        // libera acesso a contaCred
20 }
```

DeadLock

- Situação
  - Cliente c1 (tarefa t1) e Cliente c2 (tarefa t2) querem executar uma transferência entre contas
    - t1 transfere um valor v1 de c1 para c2
    - t2 transfere um valor v2 de c2 para c1

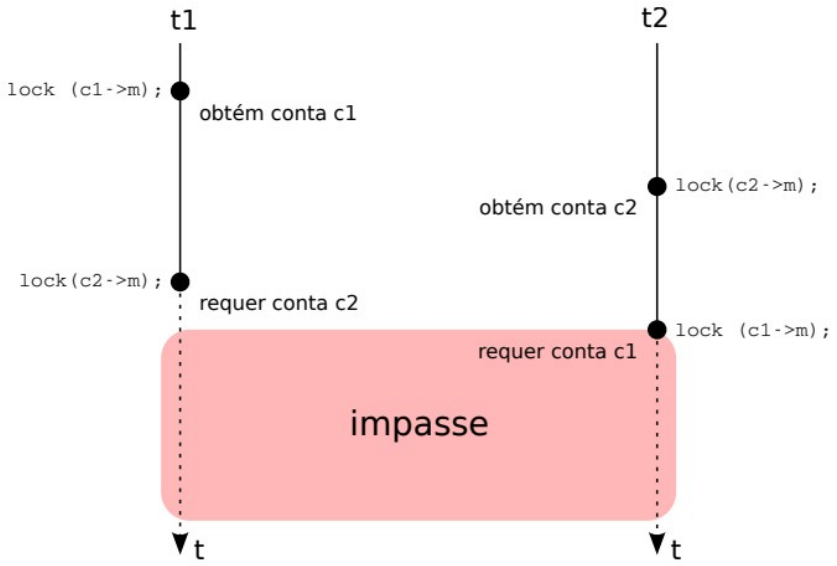

Figura 13.1: Impasse entre duas transferências.

DeadLock

- DeadLock
  - Duas ou mais tarefas se encontram bloqueadas
  - Não existe influência de entidades externas

Conjunto *N* de tarefas se encontra em um impasse se cada uma das tarefas aguarda um evento que somente outra tarefa do conjunto poderá produzir

DeadLock

- Condições para o DeadLock
  - **Exclusão Mútua**: acesso aos recursos deve ser feito de forma única (semáforos)
  - **Posse e espera:** Tarefa solicita acesso a recursos, mas sem ter que liberar os recursos que detém
  - **Não-preempção:** Tarefa libera o recurso quando ela decidir o momento, não existe um "terceiro" que fará isso
  - **Espera circular:** Ciclo de esperas: t1->t2->t3...->t1

DeadLock

- Grafos de alocação de recursos
  - Representação de DeadLock

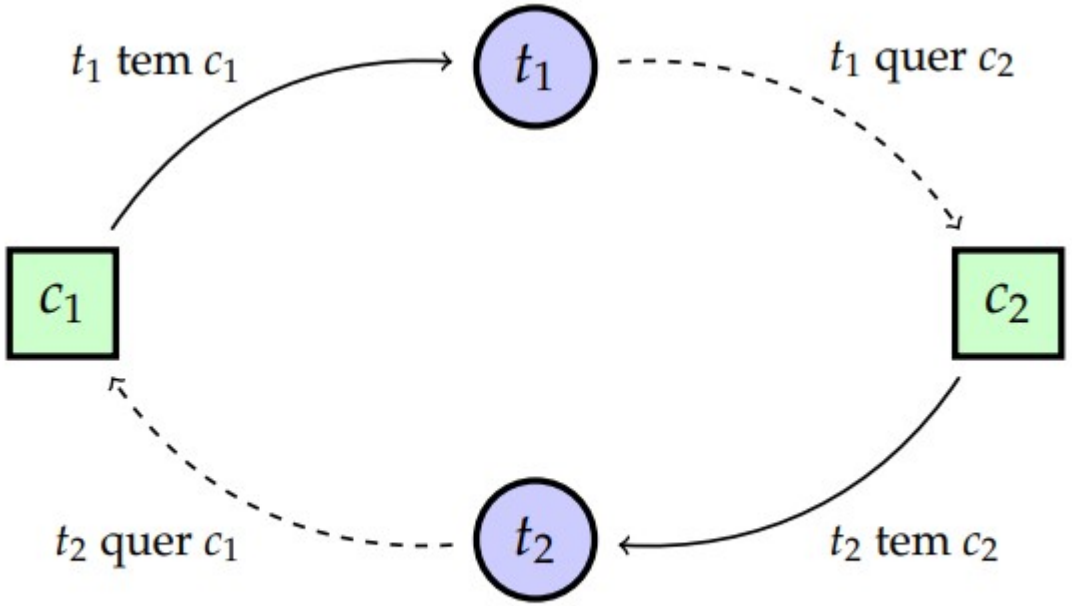

Figura 13.3: Grafo de alocação de recursos com impasse.

DeadLock

- Grafos de alocação de recursos
  - Recursos duplicados (**)

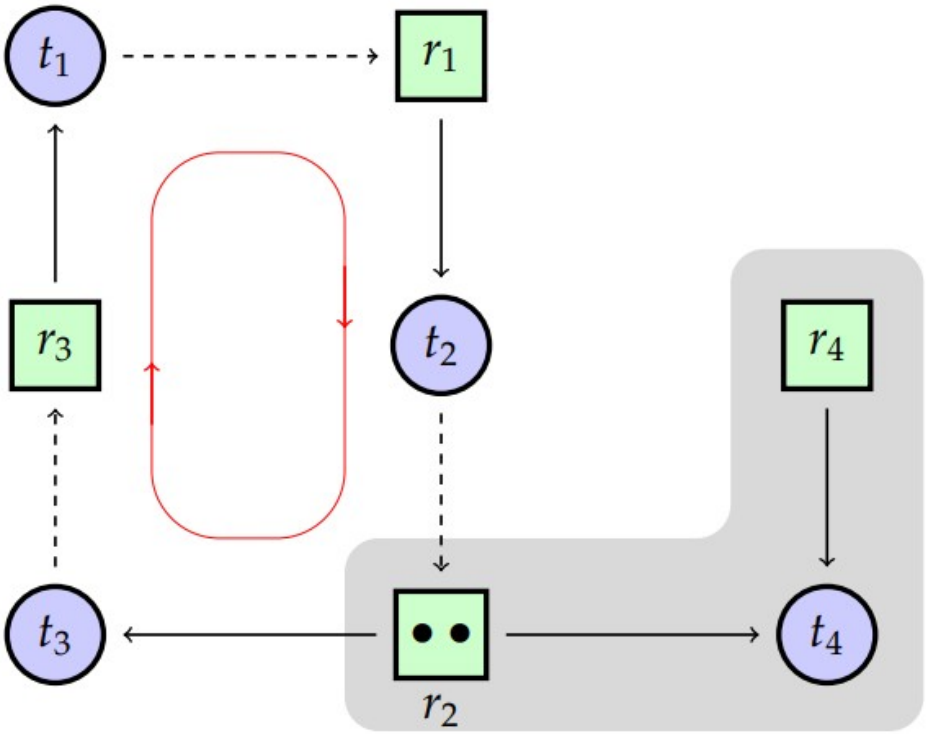

Figura 13.4: Grafo de alocação com múltiplas instâncias de recursos.

DeadLock

- Tratamento de DeadLock
  - Ocorrência de deadlock pode gerar consequências graves
  - Exemplo:
    - Paralisação de várias tarefas
      - O recurso solicitado por uma tarefa pode ser essencial no sistema operacional, outras tarefas vão solicitar esse recurso

DeadLock

- Tratamento de DeadLock
  - Existem técnicas de tratamento:
    - **Proativas**: antecipam a ocorrência do deadlock, impedindo que ocorram
    - **Reativas**: detectam deadlock ocorrido e toma medidas para resolvê-los

DeadLock

- Prevenção
  - Tentam garantir que não ocorra o deadlock
  - Uma das condições deve ser quebrada:
    - Exclusão Mútua
    - Posse e espera
    - Não-preempção
    - Espera circular

DeadLock

- Prevenção
  - Exclusão Mútua
    - Serviços externos (servidor de exclusão mútua)
      - Exemplo: Servidor de impressão
  - Posse e espera
    - Um recurso por vez pode ser utilizado
    - Aguardar todos os recursos de uma vez
    - Timeout

DeadLock

- Prevenção
  - Não-preempção
    - Liberar um recurso que uma tarefa detém
    - Difícil para arquivos e áreas de memória compartilhada
  - Espera circular
    - Ordem universal de recursos
      - Conta c1 é menor que Conta c2
      - Tarefa deve solicitar primeiro c1 e depois c2

DeadLock

- Impedimento
  - Acompanha a alocação de recursos
  - Nega acesso que poderiam levar ao deadlock
  - Trabalha com o conceito de **ESTADO SEGURO**
    - O estado do sistema é levado em consideração
    - A distribuição de recursos entre tarefas é a definição do estado do sistema

DeadLock

- Impedimento
  - Um grafo de estados é analisado pelo sistema operacional
    - Caso exista a possibilidade de uma alocação de recursos levar a uma deadlock, é considerado um estado **inseguro**
      - Sistema impede este estado
  - Algoritmo do banqueiro
    - Dijkstra

Sistemas Operacionais

DeadLock

- Impedimento

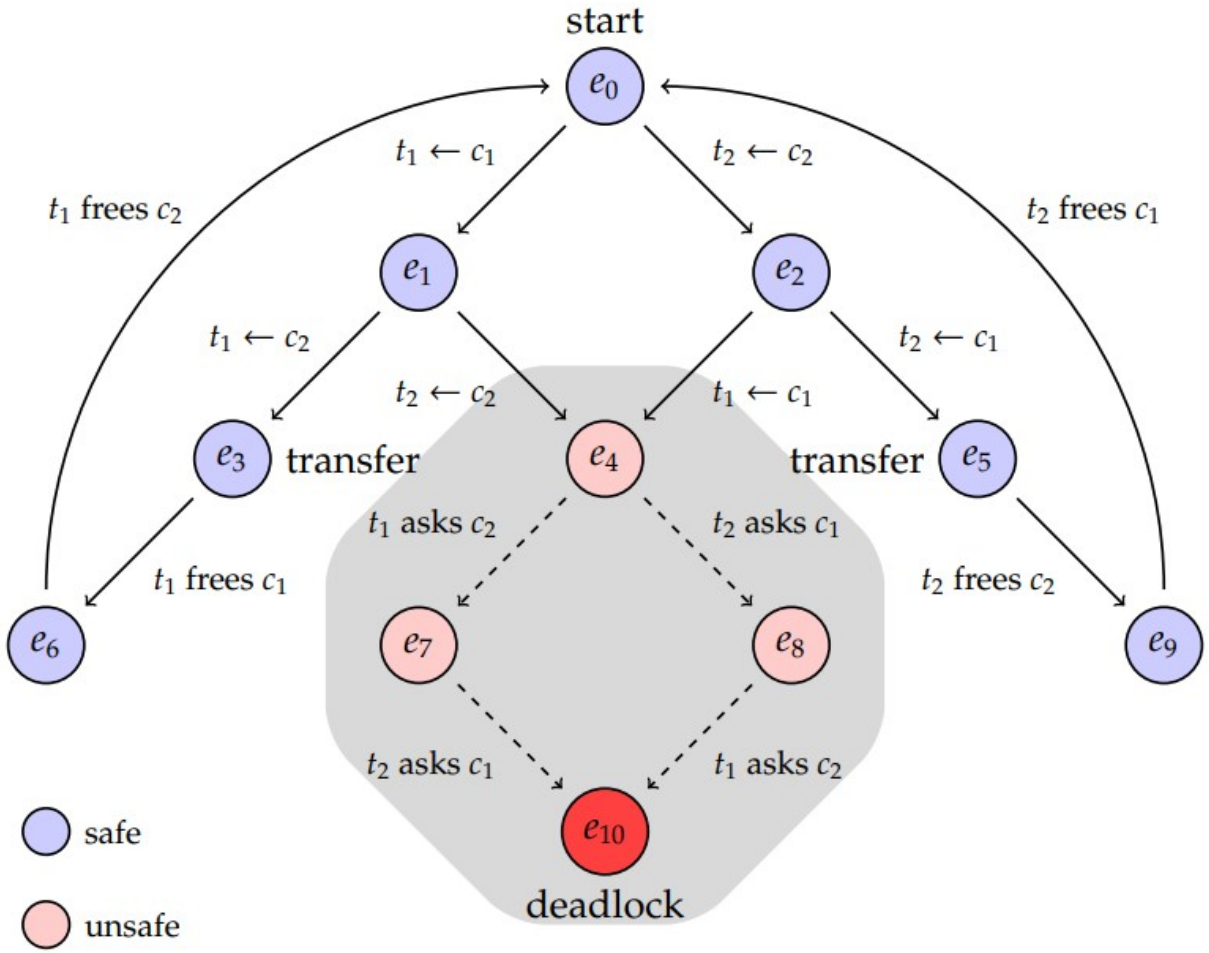

Figura 13.5: Grafo de estados do sistema de transferências com duas tarefas.

DeadLock

- Detecção e resolução
  - Não existem medidas preventivas
  - Tarefas executam normalmente
    - Ao chegar a um deadlock sistema deve detectar
      - Serão tomadas medidas para desfazer o problema

DeadLock

- Detecção e resolução
  - Detecção
    - Análise do grafo de alocação
    - Atualizado a cada alocação ou liberação de recurso
    - Algoritmo de detecção de ciclos é utilizado no grafo
    - Problemas
      - Custo do algoritmo
      - Manutenção do grafo

DeadLock

- Detecção e resolução
  - Resolução
    - Eliminação de tarefas
      - As tarefas envolvidas são descartadas
      - Liberam os recursos presos a elas
    - Retroceder tarefas
      - Rollback
      - Execução é desfeita de forma parcial
        - » Retornar a um estado seguro do sistema
      - Problema
        - » Salvar o estado da tarefa e do sistema de forma a ser possível retornar a um estado anterior

Sistemas Operacionais

DeadLock

- Detecção e resolução
  - Abordagem interessante
  - Técnicas pouco utilizadas no cotidiano de SO
  - Aplicada no gerenciamento de transações em sistemas de bancos de dados
    - Mecanismos para criar *checkpoints* (*transactions*) dos registros envolvidos antes da transação e para efetuar o *rollback* da mesma em caso de impasse

DeadLock

- Atividades
  - Na prevenção de impasses:
    - Como pode ser feita a quebra da condição de posse e espera?
    - Como pode ser feita a quebra da condição de exclusão mútua?
    - Como pode ser feita a quebra da condição de espera circular?
    - Como pode ser feita a quebra da condição de não-preempção?
  - Impasses: indique quais alternativas são incorretas:
    - Impasses ocorrem porque vários processos tentam usar o processador ao mesmo tempo.
    - Os sistemas operacionais atuais provêm vários recursos de baixo nível para o tratamento de impasses.
    - Podemos encontrar impasses em sistemas de processos que interagem unicamente por mensagens.
    - As condições necessárias para a ocorrência de impasses são também suficientes se houver somente um recurso de cada tipo no conjunto de processos considerado.