Arquiteturas Paralelas e Distribuídas

Programação com variáveis compartilhadas

Eduardo Furlan Miranda

Baseado em: MAZIERO, C. A. Sistemas Operacionais: Conceitos e Mecanismos. 2019.

Coordenação entre tarefas

- Sistemas complexos envolvem tarefas interdependentes
  - Destaca a complexidade dos sistemas com múltiplas tarefas
  - Enfatiza a necessidade de cooperação entre as tarefas
- Tarefas precisam cooperar, comunicar e coordenar
  - Sublinha a importância da comunicação entre tarefas
  - Ressalta a necessidade de ações coordenadas para resultados consistentes

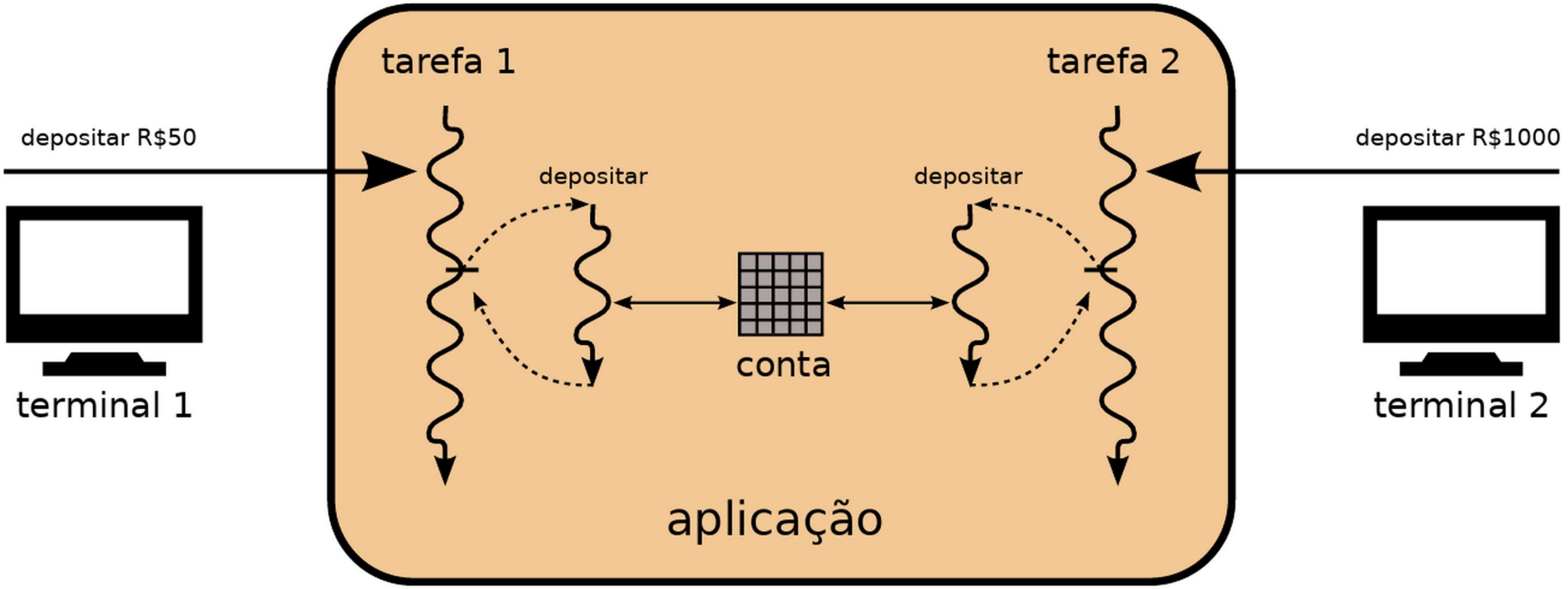
O Problema da Concorrência

- Concorrência: acesso simultâneo a recursos compartilhados
  - Define concorrência como acessos simultâneos
  - Enfatiza o compartilhamento de recursos como causa de problemas
- Pode levar a inconsistências nos dados ou no estado do recurso
  - Indica que a concorrência pode gerar dados inconsistentes
- Um exemplo é o acesso simultâneo a uma conta bancária

Uma aplicação concorrente


```
void depositar (long * saldo, long valor) {
      (*saldo) += valor ;
}
```

- A função faz parte de um sistema amplo de gestão de contas em um banco
- Caso 2 clientes em terminais diferentes tentem depositar valores na mesma conta ao mesmo tempo, existirão duas tarefas t1 e t2 acessando os dados da conta de forma concorrente



Condições de disputa (corrida)

race conditions

- Caso o depósito da tarefa t1 execute integralmente antes ou depois do depósito efetuado por t2, não há problema

- No entanto, caso as operações de depósito de t1 e de t2 se entrelacem, podem ocorrer interferências entre ambas, levando a resultados incorretos, ex.:
  - um dos depósitos é perdido
  - o que depositou por último fica sendo “o que valeu”

- Ocorre com tarefas acessam de forma concorrente recursos compartilhados (variáveis, áreas de memória, arquivos abertos, etc.)
  - são erros dinâmicos, ou seja, erros que não aparecem no código fonte e que só se manifestam durante a execução

- Erros dessa natureza não se manifestam a cada execução, mas apenas quando certos entrelaçamentos ocorrerem
- Uma condição de disputa poderá permanecer latente no código durante anos, ou mesmo nunca se manifestar
- A depuração pode ser muito complexa
- Por isso, é importante conhecer técnicas que previnam a ocorrência de condições de disputa

Condições de Bernstein (1966)

- Dadas duas tarefas, t1 e t2, com R(ti) representando o conjunto de variáveis lidas por ti e W(ti) o conjunto de variáveis escritas por ti, as tarefas podem executar em paralelo (t1 || t2) se e somente se as seguintes três condições forem atendidas
  - $R(t1) \cap W(t2) = \emptyset$: significa que a tarefa t1 não pode ler nenhuma variável que seja escrita pela tarefa t2. Em outras palavras, t1 não pode ler dados que t2 está modificando
  - $R(t2) \cap W(t1) = \emptyset$: similarmente, a tarefa t2 não pode ler nenhuma variável que seja escrita pela tarefa t1. Ou seja, t2 não pode ler dados que t1 está modificando
  - $W(t1) \cap W(t2) = \emptyset$: significa que as tarefas t1 e t2 não podem escrever na mesma variável simultaneamente. Ambas não podem tentar modificar os mesmos dados

- Um ponto importante evidenciado pelas condições de Bernstein é que as condições de disputa somente ocorrem se pelo menos uma das operações envolvidas for de escrita

- Acessos de leitura concorrentes às mesmas variáveis respeitam as condições de Bernstein e portanto não geram condições de disputa entre si

Seções (ou regiões) críticas

- Trechos de código que acessam dados compartilhados em cada tarefa, onde podem ocorrer condições de disputa. Ex.:
  - (*saldo) += valor ; (do código anterior)
- A várias seções críticas podem ser relacionadas entre si ou não (caso manipulem dados compartilhados distintos)

Exclusão mútua

- Dado um conjunto de regiões críticas relacionadas, apenas uma tarefa pode estar em sua seção crítica a cada instante, excluindo o acesso das demais

- Diversos mecanismos podem ser definidos para garantir a exclusão mútua
  - Todos eles exigem que o programador defina os limites (início e o final) de cada seção crítica

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

- Dada uma seção crítica csi , podem ser definidas as primitivas
  - enter(csi) : para que uma tarefa indique sua intenção de entrar na seção crítica csi
    - bloqueante: caso uma tarefa já esteja ocupando a seção crítica csi, as demais tarefas que tentarem entrar deverão aguardar até que a primeira libere a csi através da primitiva leave(csi)
  - leave(csi) : para que uma tarefa que está na seção crítica csi informe que está saindo da mesma

- Exclusão mútua
  - somente uma tarefa pode estar dentro da seção crítica em cada instante
- Espera limitada
  - uma tarefa que aguarda acesso a uma seção crítica deve ter esse acesso garantido 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 entrar na mesma
  - Outras tarefas, que no momento não estejam interessadas em entrar na região crítica, não podem influenciar sobre essa decisão

- Independência de fatores físicos
  - A solução deve ser puramente lógica e não depender da velocidade de execução das tarefas, de temporizações, do número de processadores no sistema ou de outros fatores físicos

Inibição de interrupções

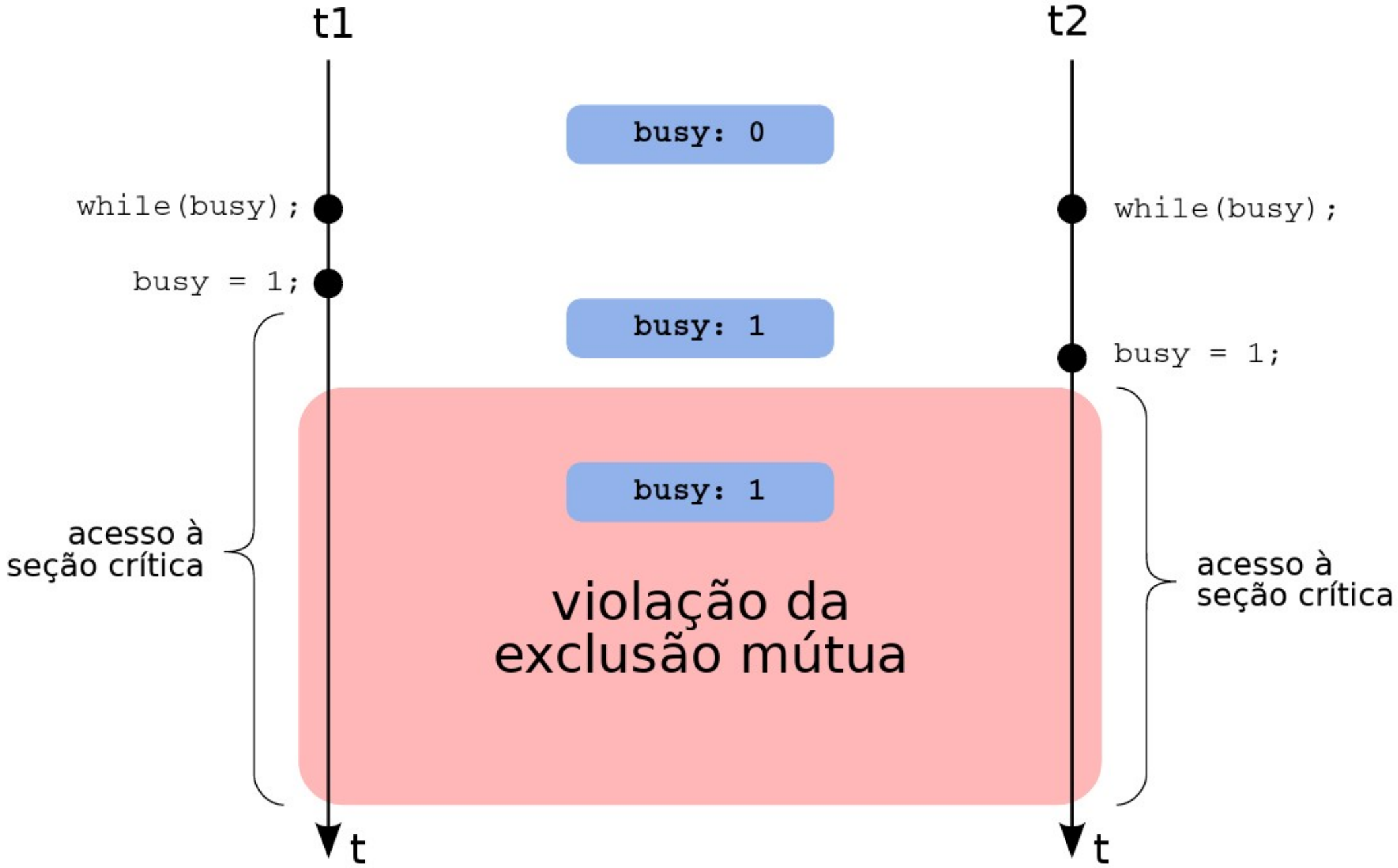
- Solução simples para a implementação da exclusão mútua
  - impedir as trocas de contexto dentro da seção crítica
  - reativar ao sair da seção crítica
- Raramente é usada
  - preempção por tempo ou por recursos deixa de funcionar
    - se “travar” não tem como sair
  - i/o deixa de funcionar
  - só funciona em uma única cpu (sistemas monoprocessados)

A solução trivial

- Usar uma variável busy para indicar se a seção crítica está livre ou ocupada

```
int busy = 0 ;
void enter () {
    while (busy) {} ;
    busy = 1 ;
}
void leave () {
    busy = 0 ;
}
```

  - Porém, não funciona: o teste da variável busy e sua atribuição são feitos em momentos distintos
    - caso ocorra uma troca de contexto entre (while (busy) {} ;) e (busy = 1), pode ocorrer uma condição de disputa envolvendo a variável busy
      - (while (busy) {} ;) e (busy = 1) formam uma seção crítica


t1
t2
busy: 0
while(busy);
busy = 1;
while(busy);
busy = 1;
busy: 1
busy: 1
acesso à
seção crítica
violação da
exclusão mútua
acesso à
seção crítica
t
t

Espera Ocupada

(busy wait)

- Teste contínuo de uma condição
  - Ex.: (while (busy) {} ;) acaba consumindo processamento

Alternância de uso


```
int num_tasks ;
int turn = 0 ;                    // inicia pela tarefa 0
void enter (int task) {           // task vale 0, 1, ..., num_tasks-1
   while (turn != task) {} ;  // a tarefa espera seu turno
}
void leave (int task) {
   turn = (turn + 1) % num_tasks ; // passa para a próxima tarefa
}
```

- cada tarefa aguarda seu turno de usar a seção crítica, em uma sequência circular: $t0 \rightarrow t1 \rightarrow t2 \rightarrow \cdots \rightarrow tn-1 \rightarrow t0$
- garante a exclusão mútua entre as tarefas e independe de fatores externos
- não atende os demais critérios: caso uma tarefa ti não deseje usar a seção crítica, todas as tarefas tj com $j > i$ ficarão impedidas de fazê-lo, pois a variável turno não evoluirá

Algoritmo de Peterson (1981)


```
int turn = 0 ;                  // indica de quem é a vez
int wants[2] = {0, 0} ;  // a tarefa i quer acessar a seção crítica
void enter (int task) {    // task pode valer 0 ou 1
    int other = 1 - task ;  // indica a outra tarefa
    wants[task] = 1 ;       // task quer acessar a seção crítica
    turn = other ;
    while ((turn == other) && wants[other]) {} ; // espera ocupada
}
void leave (int task) {
    wants[task] = 0 ;       // task libera a seção crítica
}
```

- Diversas generalizações para n > 2 tarefas podem ser encontradas na literatura
  - a mais conhecida delas o algoritmo do padeiro, proposto por Leslie Lamport (1974)

Algoritmo de Peterson (continuação)

- Tenta garantir também o critério de espera limitada,
  - porém pode falhar em arquiteturas que permitam execução fora de ordem
- Nesse caso, é necessário incluir uma instrução de barreira de memória logo antes do laço while