Sistemas Distribuídos
Aula 4

Aula passada

- Threads
- Kernel level
- User level
- Escalonamento

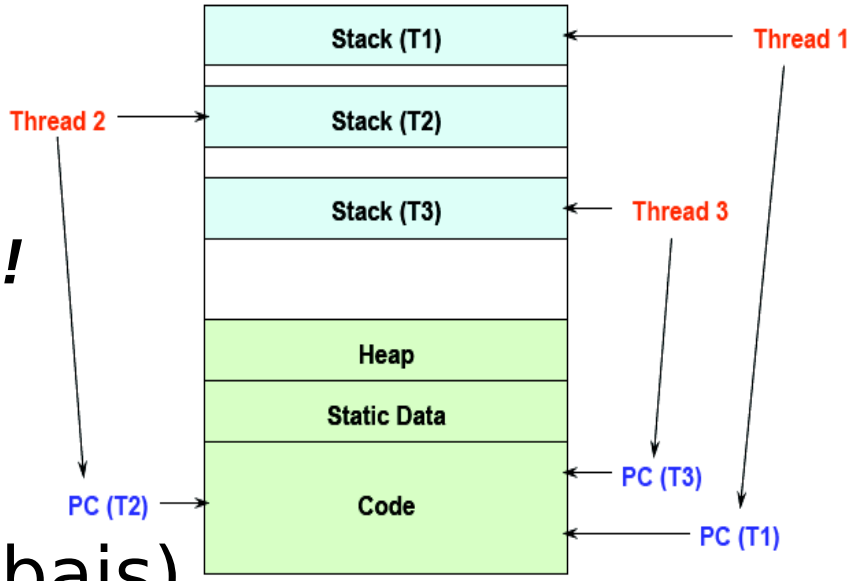
Aula de hoje

- Sincronização
- *Race condition*
- Região crítica
- Locks
- Algoritmo de Peterson

Threads Compartiham

- O que é compartilhado por diferentes threads?

Todo espaço de endereçamento!



- Código, e variáveis estáticas
  - declaradas no programa (globais)
- Variáveis dinâmicas
  - declaradas por *malloc()* (heap)
- Variáveis locais não são compartilhadas
  - declaradas nas pilhas de execução
  - cada thread possui sua pilha de execução (*ainda bem*)

Problema Fundamental

- Como lidar com threads que escrevem/leem valores de memória compartilhados?
- Como projetar código face a esta *funcionalidade* (evitando bugs)?

- Mecanismos de controle de acesso a recursos (memória) compartilhada
  - lock, mutex, semáforos, monitores, etc
- Coordenação e padrão de acesso a recursos compartilhados

***Sincronização* é o nome do jogo!**

Sincronização

- Threads cooperam na execução de um programa
  - para compartilhar recursos (ex. servidor web)
  - para coordenar suas ações (ex. Ping-pong)
- Para garantir corretude, execução precisa ser coordenada
  - execução das threads é arbitrário, não está sobre controle do programa (*a priori*)
- Cooperação precisa ser embutida explicitamente
  - sincronização → *domar* a execução das threads
- Também se aplica no contexto de processos
  - mas falaremos apenas sobre threads

Exemplo Clássico

- Considere um programa *multithreaded* que gerencia contas bancárias
  - uma thread para cada acesso (ex. online banking)
- Considere a seguinte função

```
retirada(conta, valor) {
   saldo = get_saldo(conta);
   saldo = saldo – valor;
   put_saldo(conta, saldo);
   retorna saldo;
}
```

- Considere que você tem uma conta com sua mãe com 1000 de saldo
- Você e sua mãe vão ao caixa eletrônico e fazem uma retirada de 100 simultaneamente

Exemplo Clássico

- Cada acesso é atendido por uma *thread*
  - cor abaixo indica thread do mesmo processo

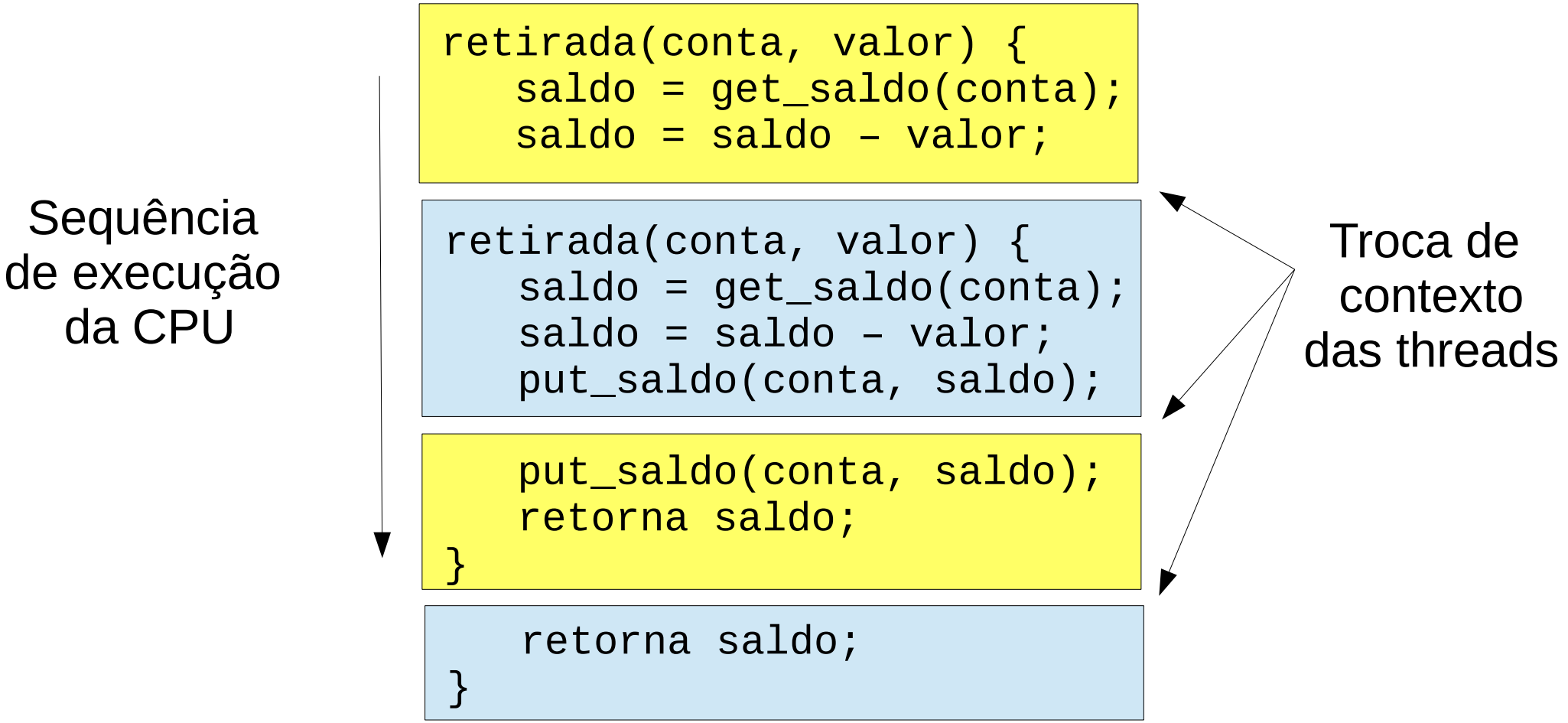
```
retirada(conta, valor) {
   saldo = get_saldo(conta);
   saldo = saldo – valor;
   put_saldo(conta, saldo);
   retorna saldo;
}
```

```
retirada(conta, valor) {
   saldo = get_saldo(conta);
   saldo = saldo – valor;
   put_saldo(conta, saldo);
   retorna saldo;
}
```

- O que pode acontecer? Pense no escalonamento (preemptivo)
- Quais são os possíveis resultados?

Escalonamento Alternado

- Cada acesso é atendido por uma *thread*
  - cor abaixo indica thread do mesmo processo



- Qual valor do saldo na conta?

Race Condition

- Situação onde resultado depende da sequência de eventos
  - *bug* quando resultado é indesejável
- Problema muito comum em software e hardware, quando há paralelismo
  - precisa ser explicitamente atacado

- Como resolver *condições de corrida*?

Exclusão mútua

Exclusão Mútua

- Exclusão mútua: sem concorrência, uma thread de cada vez (mesmo conceito em probabilidade)
- **região crítica**: parte do código que deve ser executada com exclusão mútua
  - somente uma thread por vez; outras thread precisam esperar sua vez; ao sair outra thread pode entrar

```
retirada(conta, valor) {
    saldo = get_saldo(conta);
    saldo = saldo – valor;
    put_saldo(conta, saldo);
    retorna saldo;
}
```

Região crítica!

Demandas da Região Crítica

- Exclusão mútua
  - se T está na região crítica (rc), nenhuma outra thread está
- Progresso
  - se thread T não está na rc, então T não pode impedir S de entrar na rc
- Espera limitada (*no starvation*)
  - se T aguarda entrar na rc, então T eventualmente entra na rc
- Sem demandas de desempenho
  - independente do número de CPUs (cores), e threads

Como implementar região crítica?

Locks

- Locks (*cadeados*): tranca a porta na entrada, destranca na saída
  - Forma simples de oferecer rc
- Lock é uma objeto em memória com duas operações
  - ***acquire()****:* chamada antes de entrar na rc
  - ***release()****:* chamada depois de sair da rc
- Cada thread faz o par de chamadas *acquire()/release()* ao entrar/sair da rc
  - entre *acquire()* e *release()* a thread tem o lock
  - *acquire()* bloqueia até que um *acquire()* anterior seja seguido por um *release()*

Usando Locks

```
retirada(conta, valor) {
    acquire(lock);
    saldo = get_saldo(conta);
    saldo = saldo – valor;
    put_saldo(conta, saldo);
    release(lock);
    retorna saldo;
}
```

- O que acontece no *aquire()* do azul?
- Por que o return está for a da região crítica?
- O que ocorre se uma terceira thread chama retirada()?

```
retirada(conta, valor) {
    acquire(lock);
    saldo = get_saldo(conta);
    saldo = saldo – valor;
```

```
retirada(conta, valor) {
    acquire(lock);
```

```
    put_saldo(conta, saldo);
    release(lock);
    retorna saldo;
}
```

```
    saldo = get_saldo(conta);
    saldo = saldo – valor;
    put_saldo(conta, saldo);
    release(lock);
    retorna saldo;
}
```

Como Implementar Locks?

- Primeira ideia:

```
struct lock {
   bool held = 0;
}

void acquire(lock) {
   while(lock->held);
   lock->held = 1;
}

void release(lock) {
   lock->held = 0;
}
```

Busy wait (spin wait): fica em loop esperando lock ser liberado
→ só sai do while quando held = 0

- Chamado de *spinlock*
  - thread gira sem parar até lock ser liberado
- **Funciona?**

Não Funciona!

```
struct lock {
   bool held = 0;
}

void acquire(lock) {
   while(lock->held);
   lock->held = 1;
}

void release(lock) {
   lock->held = 0;
}
```

- O que pode acontecer se mais de uma thread estiver no *spinlock*?

Race Condition!

- Troca de contexto pode deixar as duas threads entrarem na rc

Acesso Alternado?

- Considere apenas duas threads
- **Ideia:** alternar acessos ao lock
  - variável indica *quem* está no lock

```
struct lock {
   int turn = 0;
}

void acquire(lock) {
   while(lock->turn != this_thread);
}

void release(lock) {
   lock->turn = other_thread;
}
```

Variáveis locais da thread
- *this_thread*: identificador da thread em execução
- *other_thread*: identificador da outra thread

- Funciona? Qual é o problema?
- Sem progresso!

Indicando interesse

- Indicar interesse antes de pegar o lock
  - resolve o probema se outra não tiver interesse

```
struct lock {
   bool interested[2] = [0,0];
}

void acquire(lock) {
   lock->interested[this_thread] = 1
   while(lock->interested[other_thread]);
}

void release(lock) {
   lock->interested[this_thread] = 0;
}
```

- Funciona?
- Deadlock à vista!

Variáveis locais da thread
- *this_thread*: identificador desta thread
- *other_thread*: identificador da outra thread

Algoritmo de Peterson

- Alternar quando ambas tem interesse, ir em frente caso contrário

```
struct lock {
   int turn = 0;
   bool interested[2] = [0,0];
}

void acquire(lock) {
   lock->interested[this_thread] = 1
   lock->turn = other_thread;
   while(lock->interested[other_thread] &&
         lock->turn == other_thread);
}

void release(lock) {
   lock->interested[this_thread] = 0;
}
```

Dar a vez a outra thread!

Se a outra não estiver interessada, vou eu!

- Funciona?
- Finalmente!

- *this_thread*: identificador desta thread
- *other_thread*: identificador da outra thread

Problemas

- Algoritmo de Peterson limitado a duas threads
  - pode ser estendido para N threads, mas como ter N dinâmico?
- Atribuição de valor precisa ser atômico e sequencial
  - não funciona com processadores modernos, que permitem *out-of-order execution*
  - Por que? Pense em dois cores...

  - Como garantir instruções atômicas, sem surpresas?
    - apoio do hardware (instruções)
    - desabilitar troca de contexto (sinais)

Próxima aula!