Computação Concorrente (DCC/UFRJ)

Aula 5: Comunicação entre threads via memória compartilhada e sincronização com bloqueio

Prof. Silvana Rossetto

3 de abril de 2012

1 Semáforos

2 Locks
- Locks na biblioteca Pthreads
- Locks em Java
- Thread safety

Conceito de semáforo

- Proposto por Dijkstra, em 1965, como um mecanismo para suporte à cooperação entre processos dentro de um sist. oper.
- Baseia-se no princípio de **troca de sinais** entre processos/threads: *uma thread bloqueia a sua execução em um ponto específico do código até que ela receba um sinal que a desbloqueie*

Definição de semáforo

Um **semáforo** é uma variável inteira com três operações associadas, todas elas executadas de forma **atômica**:

1. **inicialização**: inicia o semáforo com valor não negativo;
2. **decremento** (*P()* ou *down()* ou **sem_wait()**): "pode resultar no bloqueio da thread"
3. **incremento** (*V()* ou *up()* ou **sem_post()**): "pode resultar no desbloqueio de uma thread"

Troca de sinais via semáforo

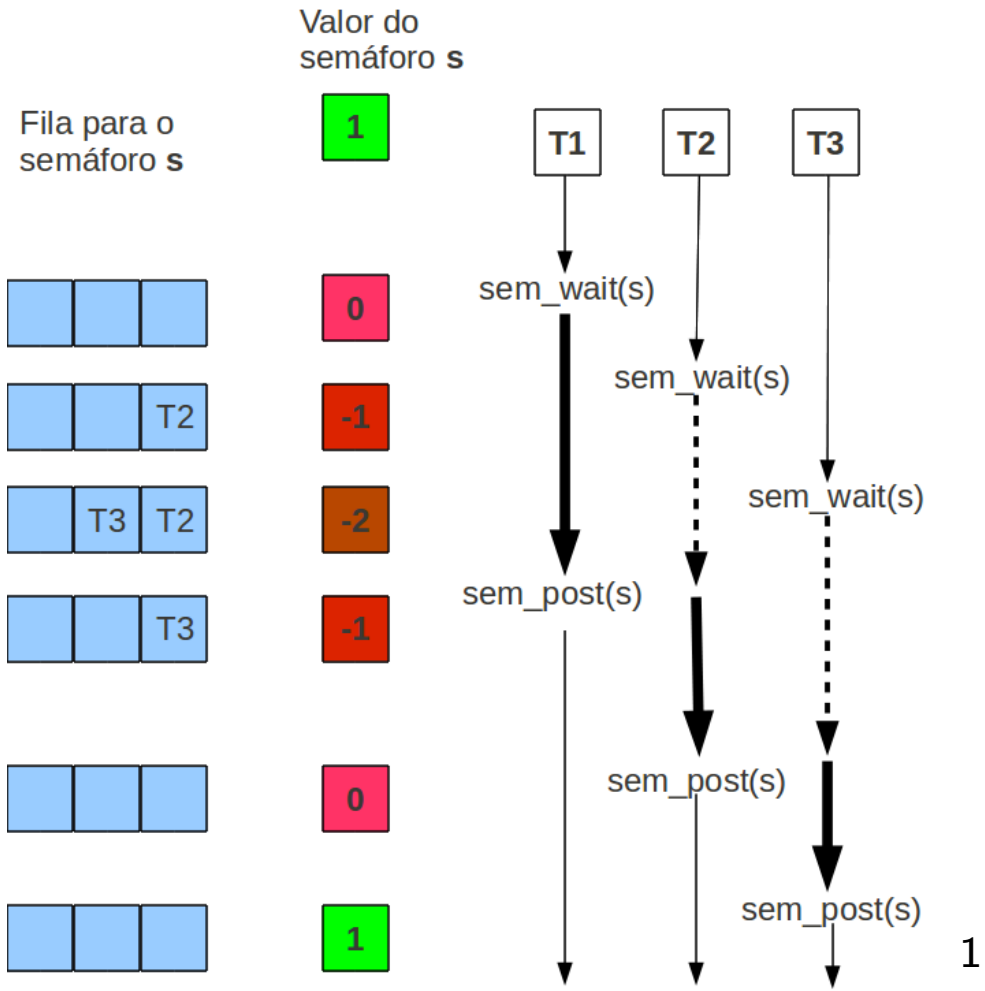
A operação **DOWN()** decrementa o semáforo:

- se o valor do semáforo ficar **menor que zero**, a thread é bloqueada!

A operação **UP()** incrementa o semáforo:

- se o valor do semáforo ficar ≤ 0, uma thread bloqueada na operação DOWN é desbloqueada!

Exemplo de execução com semáforos



[1]

[1] *Fonte: Stallings*

Semáforos para seção crítica

- O semáforo **sem** é inicializado com valor 1 (semáforo binário!)
- A entrada na seção crítica é implementada executando a operação **sem_wait(&sem)**
- A saída da seção crítica é implementada executando a operação **sem_post(&sem)**

```
while (true) {
      sem_wait(&sem);
      //executa a secao critica
      sem_post(&sem);
      //executa fora da secao critica
}
```

Exemplo de uso de semáforos

```
buffer<N>
sem vazio = 1; sem cheio = 0; int aux = 0;
```

Thread P

```
while(true) {
    DOWN(vazio);
    aux = N;
    UP(cheio);
}
```

Thread C

```
while(true) {
    DOWN(cheio);
    aux--;
    if (aux > 0)
      UP(cheio);
    else
      UP(vazio);
}
```

Definição de locks

- Um **lock** é um tipo de **objeto de sincronização** para resolver o problema de **exclusão mútua** no acesso a **variáveis/recursos compartilhados**

T1:	**T2:**	**T3:**
L.lock();	L.lock();	L.lock();
//seção crítica	//seção crítica	//seção crítica
L.unlock();	L.unlock();	L.unlock();

Propriedades dos locks

O **lock possui uma thread proprietária**:

1. Uma thread que executa **L.lock()** torna-se a proprietária do *lock* se nenhuma outra thread já possui o *lock*, caso contrário a thread é bloqueada
2. Uma thread libera sua posse sobre o *lock* executando **L.unlock** (se a thread não possui o lock a operação retorna com erro)
3. Uma thread que já possua o lock **L** e executa **L.lock()** novamente não é bloqueada (mas deve executar **L.unlock()** o mesmo número de vezes que **L.lock()** para o controle passar para outra thread) (requer propriedade de **lock recursivo**)

Exemplo de uso de locks

```
class ObjAlocavel {
  public void F() {
    L.lock();
    ...
    L.unlock();
  }
  public void G() {
    L.lock();
    ... F(); ... //método G chama o método F
    L.unlock();
  }
  private Lock L;
}
```

Locks versus semáforos binários

- Para um **semáforo binário**, se duas chamadas sem_wait() são feitas sem uma chamada sem_post() intermediária, a segunda chamada irá bloquear a thread
- Para o caso de **locks recursivos**, se a thread que está de posse do lock o requisita novamente, essa thread não é bloqueada

Locks versus semáforos binários

- Para o caso de **locks**, chamadas sucessivas das operações de **lock()** e **unlock()** devem ser feitas pela thread proprietária do lock
- Para o caso de **semáforos binários**, chamadas sucessivas de **sem_wait()** e **sem_post()** podem ser feitas por diferentes threads

Locks na biblioteca Pthreads

- A biblioteca Pthreads oferece o mecanismo de *locks* através de variáveis especiais chamadas **mutex**
- Por padrão, Pthreads implementa **locks não-recursivos** (uma thread não deve tentar alocar novamente um *lock* que já possui)
- Para tornar o *lock* recursivo é preciso mudar suas propriedades básicas

Locks em Java

A palavra reservada **synchronized** permite criar **blocos de código atômicos**, com duas partes:

1. uma referência para um objeto que servirá de *lock*
2. um bloco de código protegido (guardado) pelo lock

Exemplo de uso de synchronized em Java

```
//global variables
int x = 6;
int y = 0;
```

Thread *Foo*

```
void foo(){
  synchronized(this) {
    x++;
    y = x;
  }
}
```

Thread *Bar*

```
void bar(){
  synchronized(this) {
    y++;
    x += 3;
  }
}
```

Exemplo de lock em Java

Qualquer objeto Java pode agir como um *lock* para **sincronização por exclusão mútua**

Exemplo de lock

```
Object obj = new Object();
synchronized (obj) {
... //seção de código protegido
}
```

Aquisição e liberação do lock

- O **lock** é **adquirido automaticamente** pela thread quando seu fluxo de execução chega ao bloco synchronized
- ...e é **liberado automaticamente** quando seu fluxo de execução sai do bloco *synchronized*
- Quando o *lock* é atribuído a uma thread, as outras threads que tentarem alocá-lo serão bloqueadas: **blocos synchronized guardados pelo mesmo lock executam atomicamente entre si**

Reentrância

Locks em Java são adquiridos por threads — e não por invocação — e são **reentrantes**:

- Quando uma thread solicita um *lock* já alocado a ela mesma, essa thread não bloqueia
- A reentrância é implementada associando a cada *lock* um **contador de aquisições** e uma **thread proprietária**

- Quando o contador é igual a ZERO, o *lock* está disponível
- Quando a mesma thread adquire o *lock* novamente, o contador é incrementado
- Quando a thread deixa o bloco synchronized, o contador é decrementado

Exemplo de reentrância

```
class ObjAlocavel {
   public void F() {
      synchronized(this) {
         //faz algo
      }
   }
   public void G() {
      synchronized(this) {
        this.F(); //método G chama o método F
      }
   }
}
```

Thread safety

Uma classe é dita **thread-safe** se ela se "comporta" **corretamente** quando acessada a partir de várias threads, independentemente da ordem de execução dessas threads, e sem a necessidade de uso de mecanismos de sincronização por parte do código que chama a classe

Escrever **código thread-safe** implica em **cuidar do modo como o acesso ao estado do objeto é gerenciado**

Classes thread-safe

- Uma classe/objeto precisa ser **thread-safe** quando **ela pode ser acessada por várias threads de forma concorrente**
- Sempre que mais de uma thread acessa uma variável de estado e pelo menos uma delas escreve nessa variável, o acesso à variável (leitura/escrita) deve ser controlado usando mecanismos de sincronização

Classes thread-safe

Há três modos de garantir que uma classe é *thread-safe*:

1. **não compartilhar as variáveis de estado entre threads**
2. **tornar as variáveis de estado imutáveis**
3. **usar sincronização sempre que acessar as variáveis de estado**

Exemplo de classe NÃO thread-safe

```
@NotThreadSafe
public class A {
  private ExpensiveObject instance = null;
  public ExpensiveObject getInstance() {
    if(instance == null) {
      instance = new ExpensiveObject();
    }
    return instance;
  }
}
```

Referências bibliográficas

1. *Concurrent Programming — Principles and Practice*, Andrews, Addison-Wesley, 1991
2. *Modern Multithreading*, Carver e Tai, Wiley, 2006
3. *Java Concurrency in Practice*, Goetz et. al., Addison Wesley, 2006
4. *Synchronization Algorithms and Concurrent Programming*, G. Taubenfeld, Pearson/Prentice Hall, 2006