Computação Concorrente (MAB-117)
Comunicação entre threads via memória compartilhada e mecanismos de sincronização (parte II)

Prof. Silvana Rossetto

[1]Departamento de Ciência da Computação (DCC)
Instituto de Matemática (IM)
Universidade Federal do Rio de Janeiro (UFRJ)
Março de 2012

1. Semáforos

Semáforo é um mecanismo de sincronização por escalonamento. Trata-se de uma solução cujo suporte deve ser oferecido pelo sistema operacional. (O conceito de semáforo foi proposto por Dijkstra, em 1965, como um mecanismo para dar suporte à cooperação entre processos dentro do próprio sistema operacional.)

A definição do conceito de semáforo baseia-se no princípio de **troca de sinais** entre processos/threads: *uma thread pode ser forçada a bloquear a sua execução em um ponto específico do código até que ela receba um sinal que a desbloqueie*. Usando esse princípio básico de troca de sinais de forma apropriada, qualquer requisito de sincronização entre threads pode ser satisfeito [1].

Para realizar a sinalização, variáveis especiais — chamadas **semáforos** — são usadas. 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 *wait*, ou *down*): pode resultar no bloqueio da thread;
3. **incremento** (*V*, ou *signal*, ou *up*): pode resultar no desbloqueio de uma thread.

Para receber um sinal via semáforo **s**, uma thread executa a operação `wait()` (ou *down*), decrementando o semáforo. Se o sinal ainda não foi transmitido, a thread é suspensa até a transmissão ocorrer. Para transmitir um sinal via semáforo **s**, uma thread executa a operação `signal(s)` (ou *up*), incrementando o semáforo. Se o valor do semáforo após o incremento for ≤ 0, um processo bloqueado em *wait()* é desbloqueado. O pseudocódigo abaixo ilustra o uso de um semáforo para implementar sincronização por exclusão mútua:

```
semaphore s=1;
void tarefa () {
   while(true) {
      wait(s);
      //executa a secao critica
      signal(s);
      //executa fora da secao critica
   }
}
```

A Figura 1 mostra um exemplo de execução de três threads que entram e saem das suas seções críticas usando um semáforo de controle.

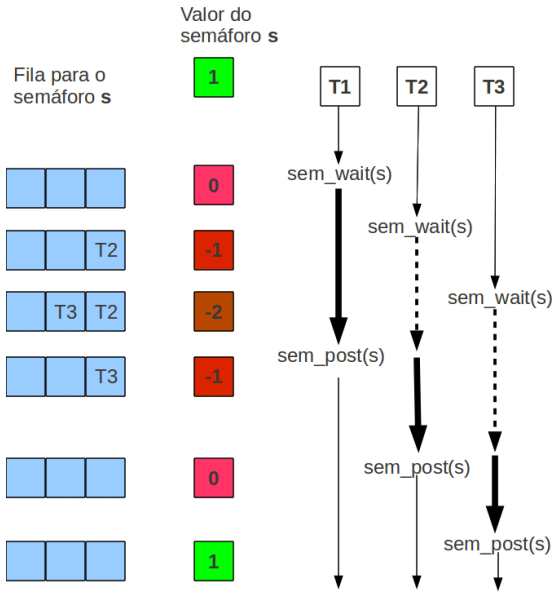

Figura 1. Exemplo de entrada/saída na seção crítica por três threads usando um semáforo. As setas normais indicam execução fora da seção crítica, as setas cheias indicam execução dentro da seção crítica e as setas pontilhadas indicam que a thread está suspensa.

Um semáforo no qual o **contador é iniciado com valor 1** é conhecido como **semáforo binário** e é normalmente usado para implementar **sincronização por exclusão mútua**. Um semáforo no qual o **contador é iniciado com valor N** é conhecido como **semáforo contador** (ou semáforo geral) e é normalmente usado para implementar **sincronização por condição**.

Um semáforo contador pode ser visto como um tipo abstrato de dado que mantém um ***pool* de permissões**. Uma thread chama a operação P() para requisitar uma permissão. Se o *pool* estiver vazio, a thread espera até uma permissão se tornar disponível. Uma thread chama a operação V() para devolver uma permissão para o *pool* [2].

2. *Locks*

Um *lock* é outro tipo de **objeto de sincronização** que pode ser usado para resolver o problema de seção crítica [2]. (*Locks* permitem implementar apenas exclusão mútua, e não sincronização por condição.) Diferente de semáforos, um ***lock* possui uma thread proprietária** e esta relação de posse determina características particulares das operações sobre *locks*:

- Uma thread requisita a posse de um *lock* L executando a operação `L.lock();`
- Uma thread que executa a operação `L.lock()` torna-se a proprietária do *lock* se nenhuma outra thread já possui o *lock*, caso contrário a thread é bloqueada;
- Um thread libera sua posse sobre o *lock* executando a operação `L.unlock` (se a thread não possui o *lock* a operação retorna com erro);
- 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 executou `L.lock()` antes que outra thread possa ganhar a posse de L;
- Um *lock* que permite a própria thread proprietária alocá-lo novamente é chamado **lock recursivo**.

O uso de *locks* para implementar seções críticas é similar ao uso de semáforos: a operação `L.lock()` implementa a entrada na seção crítica e a operação `L.unlock()` implementa a saída da seção crítica, como ilustrado abaixo:

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

As principais diferenças entre *locks* e semáforos binários são:

- Para um semáforo binário, se duas chamadas para a operação P() são feitas sem uma chamada intermediária para a operação V(), 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.
- Chamadas sucessivas das operações de `lock()` e `unlock()` devem ser feitas pela thread proprietária do *lock*. Para o caso de **semáforos binários**, chamadas sucessivas das operações P() e V() podem ser feitas por diferentes threads.

Locks são normalmente usados em métodos de classes para criar objetos alocáveis, como ilustrado no código abaixo:

```
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;
}
```

O *lock* L é usado para transformar os métodos `F` e `G` em seções críticas. Dessa forma apenas uma thread a cada instante pode executar dentro desses métodos. Quando uma thread chama o método `G`, o *lock* L é alocado. Quando o método `G` chama o método `F`, a operação `L.lock()` é executada em `F`, mas a thread não é bloqueada pois é a proprietária atual do *lock*. Se `L` fosse um semáforo binário ao invés de um *lock*, a chamada do método `F` dentro do método `G` iria bloquear a thread quando a operação `P()` (sobre o semáforo) fosse executada em `F`. Isso causaria um *deadlock* uma vez que nenhuma outra thread conseguiria executar dentro dos métodos `F` ou `G`.

2.1. Locks na biblioteca Pthreads

A biblioteca Pthreads oferece o mecanismo de sincronização por *locks* através de variáveis especiais chamadas ***mutex***. Por default, 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. O código abaixo ilustra o uso das operações de *lock* com a biblioteca Pthreads:

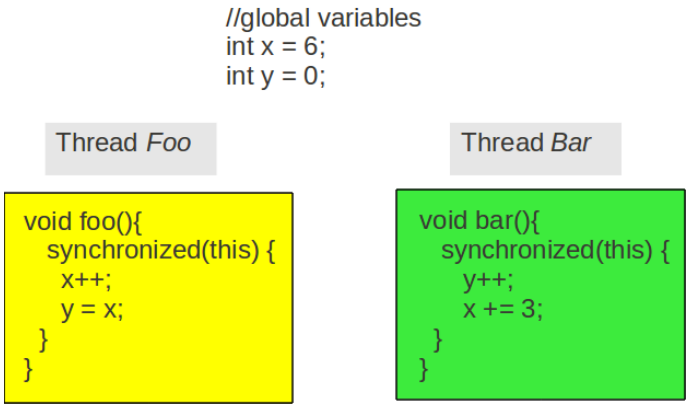

Figura 2. Exemplos de uso da palavra reservada `synchronized` em Java.

```
pthread_mutex_t mutex;
pthread_mutexattr_t mutexAttr;
pthread_mutexattr_init(&mutexAttr);
pthread_mutexattr_settype(&mutexAttr,PTHREAD_MUTEX_RECURSIVE);
pthread_mutex_init(&mutex, &mutexAttr);
pthread_mutex_lock(&mutex);
//seção crítica
pthread_mutex_unlock(&mutex);
```

2.2. Locks em Java

A palavra reservada `synchronized` em Java permite criar **blocos de código atômicos**. Um bloco synchronized tem duas partes [3]:

- uma referência para um objeto que servirá de *lock*; e
- um bloco de código protegido (guardado) pelo lock.

Um método precedido pela palavra `synchronized` é um **bloco synchronized que engloba o método inteiro** e seu objeto de lock é o próprio objeto sobre o qual o método é invocado (`this`). A Figura 2 ilusta essas duas formas de construção de blocos synchronized.

Qualquer objeto Java pode agir como um *lock* para **sincronização por exclusão mútua** (são chamados "intrinsic locks" ou "monitors locks"). Um exemplo é mostrado no trecho de código abaixo.

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

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. A saída de um bloco synchronized pode se dar de três formas básicas: pelo término da execução das sentenças dentro do bloco, pela ocorrência de uma exceção ou pelo bloqueio da thread.

Uma vez que o *lock* é atribuído a uma thread, outras threads que tentarem alocá-lo serão bloqueadas enquanto ele não for liberado pela primeira thread. Assim, **blocos synchronized guardados pelo mesmo *lock* executam atomicamente entre si**.

Reentrância Os *locks* em Java são adquiridos por threads, e não por invocação, e são **reentrantes** [3]. Isso significa que quando uma thread em Java 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 uma thread adquire um *lock*, a JVM registra a thread proprietária e seta o contador para UM. Se essa mesma thread adquire o *lock* novamente, apenas o contador é incrementado. Quando a thread deixa o bloco synchronized, o contador é decrementado. Um exemplo é mostrado abaixo.

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

Os métodos `F` e `G` estão guardados pelo mesmo objeto de *lock* (this). Dessa forma, apenas uma thread a cada instante pode executar dentro desses métodos. Quando uma thread chama o método `G`, o *lock* `this` é alocado. Quando o método `G` chama o método `F`, o mesmo *lock* é requisitado novamente, mas a thread não é bloqueada pois é a proprietária atual do *lock*. Se o *lock* não fosse reentrante, essa sequência de chamadas poderia causar um *deadlock*, uma vez que nenhuma outra thread conseguiria executar dentro dos métodos `F` ou `G`.

3. Thread safety

Uma classe é dita *thread-safe* se ela se "comporta" (ou executa) **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**. O **estado de um objeto** é definido pelos dados armazenados nas **variáveis de estado** (atributos estáticos ou de instância). *O estado de um objeto engloba qualquer dado que pode afetar seu comportamento visível externamente* [3].

Uma **classe/objeto precisa ser *thread-safe* quando ela pode ser acessada por várias threads de forma concorrente**. Para tornar uma classe *thread-safe* é necessário usar mecanismos de sincronização para coordenar o acesso ao seu estado. **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. 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.

O código abaixo ilustra uma classe que **não** é *thread-safe* [3].

```
@NotThreadSafe
```

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

O método `getInstance()` checa se uma instância da classe já foi criada. No caso da instância ainda ser vazia, uma instância é criada, armazenada para requisições futuras e devolvida pelo método. O exemplo contém uma sequência de operações que precisam ser **atômicas** com relação a elas mesmas e outras operações sobre o mesmo estado do objeto (**por que?**).

4. Exercícios

1. O que é o mecanismo de sincronização por semáforo? Quais operações ele oferece e como funcionam?
2. O que caracteriza "locks resursivos" e "locks não-recursivos"?
3. Quais são as diferenças entre locks recursivos e semáforos binários?
4. O que define uma classe *thread-safe*?

Referências

[1] W. Stallings. *Operating Systems – Internals and Design Principles*. Pearson - Prentice Hall, 6 edition, 2009.

[2] Richard H. Carver and Kuo-Chung Tai. *Modern multithreading*. Wiley, 2006.

[3] Brian Goetz et.al. *Java Concurrency in Practice*. Addison Wesley, 2006.