Guião N.º 4

Semáforos em Java

António Pinto
apinto@estg.ipp.pt

Outubro, 2016

1 Semáforos

Um semáforo é um mecanismo disponibilizado pela JVM que permite sincronizar a execução de múltiplas *threads*. Ou seja, permite resolver problemas decorrentes da partilha de recursos ou de situações de competição.

Semáforos em Java são disponilizados pela classe *Semaphore*[1] que dispõem de dois métodos principais:

- *acquire()*[2] - Reduz o número de autorizações. Esta operação é bloqueante quando o semáforo não dispõem de autorizações.
- *release()*[3] - Incrementa o número de autorizações.

O seu funcionamento baseia-se no conceito de quantidade de autorizações. O semáforo deve ser inicializado com um determinado número de autorizações. Cada chamada a *aquire()* reduz este número, cada chamada a *release()* aumenta. Assumindo que um semáforo foi inicializado com N autorizações, tal implica que apenas N *threads* podem chamar *aquire()*, no máximo, antes de ser chamado o *release()*.

[1] +info:`http://docs.oracle.com/javase/7/docs/api/java/util/concurrent/Semaphore.html`

[2] Também referenciado como *wait()* ou *down()*

[3] Também referenciado como *signal()* ou *up()*

1.1 Proteção de secções críticas

Os semáforos podem ser utilizados para proteger secções críticas de código, bastando para tal inicializar o semáforo com uma autorização. De seguida basta que se obtenha uma autorização do semáforo antes de se executar o código da secção crítica. Após terminar-se o código da secção critica, deve-se libertar a autorização para que outra *thread*, que esteja bloqueada no mesmo semáforo, possa continuar. A Listagem 1 exemplifica este comportamento.

```
Semaphore sem = new Semaphore(1);
...
sem.acquire();
//
// Codigo seccao critica
//     executado com
//    exclusao  mutua
//
sem.release();
...
```

Listagem 1: Utilização de semáforos em secções críticas

1.2 Outras situações de sincronização

Além de secções críticas, os semáforos, ao serem usados com um número de autorizações superior a um, podem permitir estratégias de sincronização avançadas. Pode-se inclusive utilizar vários semáforos, com diferentes valores iniciais de autorizações.

```
import java.util.concurrent.*;

public class Current implements Runnable {
  Semaphore s;
  int n;
  public Current (Semaphore sem,int i) {s=sem;n=i;}
    public void run() {
    String myname = Thread.currentThread().getName();
    System.out.println("["+myname+"] Inicio da thread");
    try{
      Thread.sleep(n*1000);
    } catch (InterruptedException iex){}
    s.release();
    System.out.println("["+myname+"] Fim da thread");
  }
    public static void main(String args[]) {
    Semaphore sem = new Semaphore(5);
```

```
    for (int i=0; i<10; i++) {
      try {
        sem.acquire();
      } catch (InterruptedException iex){}
      Thread th = new Thread(new Current(sem,i),"Th"+i);
      th.start();
    }
    }
}
```

Listagem 2: Outras situações de sincronização com semáforos

O excerto de código apresentado na Listagem 2 demonstra como se podem utilizar semáforos para limitar o número de *threads* concorrentes em cada instante de tempo. Neste caso serão criadas 10 *threads* no total, sendo que em cada instante de tempo apenas poderão estar a executar 5 *threads* no máximo.

1.3 Justiça

A implementação de semáforos do Java não dá garantias de justiça na sua utilização, exceto se tal for explicitamente solicitado. Ou seja, não há garantias de que a primeira *thread* a invocar o método *acquire()* seja a primeira *thread* a obter a permissão.

```
Semaphore sem = new Semaphore(1,true);
```

Listagem 3: Instanciação de semáforo justo

A justiça pode ser forçada com recurso a um segundo construtor desenvolvido particularmente para este efeito. Contudo, a utilização de semáforos com garantia de justiça implica um custo de performance. A sua utilização não é recomendada, a não ser quando realmente necessária. A Listagem 3 demonstra como se pode instanciar um semáforo justo.

2 Exercícios

1. Elabore um programa que lance a execução de 5 *threads*. Cada *thread* deve escrever 200 números num ficheiro de texto. Garanta a exclusão mútua nas operações de escrita. O nome do ficheiro deverá ser passado como argumento da linha de comandos.

2. Elabore um programa que lance a execução simultânea de 20 *threads*. Cada *thread* deve tentar escrever uma linha num ficheiro, aguardar um

segundo e que tentar escrever uma segunda linha no mesmo ficheiro. Garanta a exclusão mútua nas operações de escrita. O ficheiro só deverá conter no máximo 10 linhas. Utilize dois semáforos. O nome do ficheiro deverá ser passado como argumento da linha de comandos.

3. Elabore um programa que lance a execução de 2 *threads* em simultâneo. As *threads* devem ser implementadas como classes distintas. Uma *thread* deve escrever "Init"quando iniciar, esperar um número aleatório de segundos (entre 1 e 9) e escrever "End". A outra *thread* deve iniciar após a *thread* inicial escrever "End", escrevendo um número aleatório de linhas (entre 1 e 9).

4. Elabore um programa que lance 20 *threads* em simultâneo e que, infinitamente, solicite um número ao utilizador. O número será utilizado para identificar a *thread* a ativar até que seja inserido um novo número. As *threads* devem iniciar suspensas. Cada *thread* deve escrever uma mensagem por segundo para o ecrã sempre que estiver ativa.

3 Exercícios Extra

```
import java.util.concurrent.*;                                          1
public class SemaphoreTest{                                             2
    static Semaphore s = new Semaphore(0);                              3
                                                                        4
    public void fun(final char c, final int r) throws                   5
        Exception {
        new Thread(new Runnable(){                                      6
            public void run(){                                          7
                try{                                                    8
                    System.out.println("acquire "+r);                   9
                    s.acquire(r);                                       10
                    System.out.println(c+" "+r);                        11
                } catch(Exception e){ e.printStackTrace(); }            12
            }                                                           13
        }).start();                                                     14
        Thread.sleep(500);                                              15
    }                                                                   16
                                                                        17
    public static void main(String[] args) throws Exception{           18
        SemaphoreTest f = new SemaphoreTest();                          19
                                                                        20
        f.fun('B',2);                                                   21
        f.fun('F',6);                                                   22
        f.fun('A',1);                                                   23
```

```
            f.fun('C',3);                                                    24
            f.fun('D',4);                                                    25
            f.fun('E',5);                                                    26
                                                                             27
            while(s.hasQueuedThreads()){                                     28
                Thread.sleep(1000);                                          29
                System.out.println("release "+1+", available "+(              30
                    s.availablePermits()+1));
                s.release(1);                                                31
            }                                                                32
        }                                                                    33
}                                                                            34
```

Listagem 4: Classe *SemaphoreTest*

1. Considerando a classe *SemaphoreTest* apresentada na Listagem 4. Qual será o resultado esperado se se substituir as chamadas às funções *fun()* pelas seguintes:

```
f.fun('C',7);                                                                1
f.fun('B',2);                                                                2
f.fun('A',5);                                                                3
```