Semaphores in Java

authors: Tuire Huhtamäki, Miikka Merikanto, Kai Nevalainen
course: Concurrent Programming
date: 14.12.2006

Main objective and application area for the method

Semaphores are an abstract data type which are used to restrict the usage of common shared resources in concurrent programming. Semaphores are implemented mainly in restricting the number of threads that can access some resource.

The basic idea of selected method

Java Semaphore keeps count of permits to access mutual resources. Threads ask permits from semaphore and release them when they are ready. The order priority for permits is defined in the constructor with the boolean fairness.

Similarities and differences of this method as compared to the similar basic method given in lectures

The similarities to the basic method:

- use the operations P (S) for wait and V (S) for signal to control concurrency. In Java Semaphores the operations are called acquire() and release().
- the semaphore operations are atomic - they cannot be interrupted. In Java this is implemented with encapsulation.
- create the semaphore as a non-negative integer and initialise it to some value. Java Semaphore is initialized with an integer to set the number of permits, negative integers are not allowed.

The variants that can be found

Binary semaphores can only compute one process at a time, thus no multiprocessing. The integer value can only be either 0 or 1, and it is initialised to be 1. It is somewhat similar to mutual exclusion algorithms. It decreases the value to 0 with the P(S) operation when the resource is in use, and then releases it with the V(S) operation by increasing it back to 1 when the resource is freed.

General semaphores differ as their integer value range goes from 0 higher than 1, ie 0,1,2,3,4.... and the value specifies how many advance permissions are available. They use schedulers to specify which process is executed when. The schedulers themselves can also differ in the way they prioritise - strong semaphores that strictly orderise threads or weak

semaphores that have no order.

A blocking semaphore is initialised to zero rather than one, working in a manner that any thread that does a P(S) operation will block until released by V(S) first. It can be used to control the order in which threads are executed when they need to be managed.

A busy-wait semaphore uses a busy-wait loop instead of placing the process in suspend. In Java thread scheduling controls when a thread is enabled or disabled, the thread lies dormant in the disabled state until the semaphore is released or the thread is interrupted.

Java Semaphores can be implemented as binary, general or blocking semaphore depending on how they are used. Fairness controls the priority in scheduling.

There are some variants that use negative integers for polling the number of awaiting processes but Java doesn't support this natively.

How to use Java semaphores

Semaphore can be initialized with constructors Semaphore(int permits) or Semaphore(int permits, boolean fair). Semaphore(int permits) creates a semaphore with given number of permits and unfair fairness setting and the other constructor creates a semaphore with given fairness setting. When fairness is set to true, the semaphore gives permits to access mutual resources in the order the threads have asked for it (FIFO) and when fairness is set false, semaphore can give permit to a thread asking for it before it gives permit to the already waiting thread in the queue. To avoid starving the fairness should be set true. Semaphore class is defined in java.util.concurrent package.

Threads can ask permits with the method acquire(), the thread is blocked until the permit is granted. The acquire() operation can throw exception InterruptedException and it must be caught or thrown. The permit is returned to the semaphore with the method release(). There are also variations of the methods described above. For example it is possible to acquire and release multiple permits simultaneously and acquire the permit bypassing the queue in the semaphore even when fairness is true.

Java program example:

```
import java.util.concurrent.Semaphore ;
class Example extends Thread {
        int id;
        static Semaphore semaphore = new Semaphore (1);  /*must be static so multiple threads have
                                                          only one semaphore to control CS!  The
                                                          created semaphore here allows only one
                                                          thread at a time execute the critical
                                                          section*/

        public Example(int id){
                this.id= id;
        }

        public void run(){      /*subclasses of Thread must override the method run()*/
                try {
                        semaphore.acquire();  /*thread stops here until it gets permit to go on*/
                }catch (InterruptedException e) {} /*exception must be caught or thrown*/

                //CRITICAL SECTION

                semaphore.release();
        }

        public static void main(String[] args){
                Example esim1= new Example(1);
                Example esim2= new Example(2);
                esim1.start(); /*start() is a method derived from class Thread, that calls the run()
                                method with the Java Virtual Machine. Thread esim1 is now
                                executed parallel with main() method*/
                esim2.start(); /*here another thread is started, therefore now being executed are main(),
                                esim1 and esim2 parallel*/
        }

}
```