TITLE

Laboratory 6. Semaphores

PREREQUISITES

- OOP basic knowledge
- Java programming language basic knowledge
- Algorithms

RESOURCES

- Course slides
- Java Tutorial
- Java Concurrency

LABORATORY INSTRUCTIONS

Semaphore

Java provides an implementation of the mechanism of semaphore by the **Semaphore** class present in *java.util.concurrent* package.

The **Semaphore** class has two constructors:

- *Semaphore(int permits)* - Creates a Semaphore with the given number of permits and nonfair fairness setting
- *Semaphore(int permits, boolean fair)* - Creates a Semaphore with the given number of permits and the given fairness setting.

  "The constructor for this class optionally accepts a fairness parameter. When set false, this class makes no guarantees about the order in which threads acquire permits. In particular, barging is permitted, that is, a thread invoking acquire() can be allocated a permit ahead of a thread that has been waiting - logically the new thread places itself at the head of the queue of waiting threads. When fairness is set to true, the semaphore guarantees that threads invoking any of the acquire methods are selected to obtain permits in the order in which their invocation of those methods was processed (first-in-first-out; FIFO). Note that FIFO ordering necessarily applies to specific internal points of execution within these methods. So, it is possible for one thread to invoke acquire before another, but reach the ordering point after the other, and similarly upon return from the method. Also note that the untimed tryAcquire methods do not honor the fairness setting, but will take any permits that are available." Source: **Semaphore**

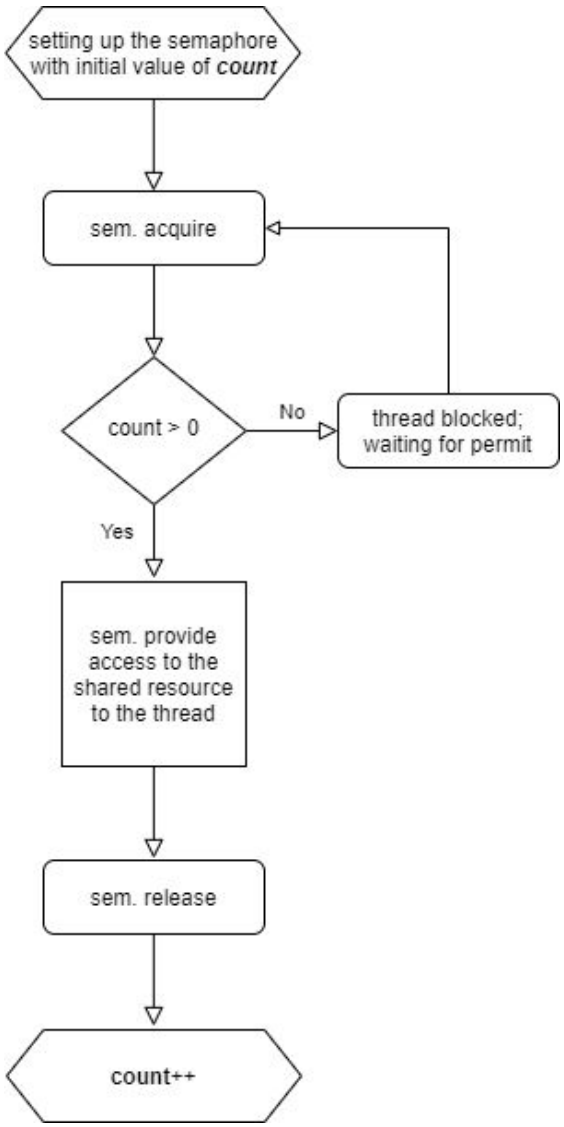

Fig. 1 - Semaphore Flow Chart

Acquire & Release methods:

public void acquire() throws InterruptedException

Acquires a permit from this semaphore, blocking until one is available, or the thread is interrupted.

Acquires a permit, if one is available and returns immediately, reducing the number of available permits by one.

If no permit is available then the current thread becomes disabled for thread scheduling purposes and lies dormant until one of two things happens:

- Some other thread invokes the *release()* method for this semaphore and the current thread is next to be assigned a permit; or

- Some other thread interrupts the current thread.

If the current thread:
- has its interrupted status set on entry to this method; or
- is interrupted while waiting for a permit,

then *InterruptedException* is thrown and the current thread's interrupted status is cleared.

public void release()
Releases a permit, returning it to the semaphore.

Releases a permit, increasing the number of available permits by one. If any threads are trying to acquire a permit, then one is selected and given the permit that was just released. That thread is (re)enabled for thread scheduling purposes.

There is no requirement that a thread that releases a permit must have acquired that permit by calling acquire(). Correct usage of a semaphore is established by programming convention in the application.

Concurrent counting with semaphores in Java:

```java
package ro.edu.ucv;

import java.util.concurrent.Semaphore;

public class CountSem extends Thread{
    static volatile int n = 0;
    static Semaphore s = new Semaphore( permits: 1);

    public int getN() {
        return n;
    }

    public void run() {
        int temp;

        for(int i = 0; i < 10000000; i++) {
            try {
                s.acquire();
                temp = n;
                n = temp + 1;
            } catch (InterruptedException e) {
                e.printStackTrace();
            } finally {
                s.release();
            }
        }
    }
}
```

Fig. 2 CountSem class

```java
package ro.edu.ucv;

public class Main {

    public static void main(String[] args) {
        CountSem p = new CountSem();
        CountSem q = new CountSem();
        p.start();
        q.start();
        try {
            p.join();
            q.join();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
        System.out.println("Final counter: " + p.getN());
    }
}
```

Fig. 3 Main class

CountDownLatch

Another Java synchronization mechanism is the ***CountDownLatch*** class. It allows one or more threads to wait until a set of operations being performed in other threads completes.
See the implementation in the lab slides.

CyclicBarrier

A synchronization aid that allows a set of threads to all wait for each other to reach a common barrier point. ***CyclicBarriers*** are useful in programs involving a fixed sized party of threads that must occasionally wait for each other. The barrier is called cyclic because it can be re-used after the waiting threads are released.
See the implementation in the lab slides.

LABORATORY TASKS

1. Implement the above Concurrent Countering example in Java.
2. Implement the CountDownLatch example in the slides.
3. Implement the CyclicBarrier example in the slides.
4. Check what happens in the above given examples if, for point 1, you do the following changes:
    a. The used “Semaphore s” variable is no longer static
    b. The “n” variable is no longer static
    c. You acquire before the for loop
    d. You don’t release or release after the for loop

    Is the program having the same expected output? Does it block?