CS-E4110 Concurrent Programming
Autumn 2016 - Tutorials
Semaphores and Monitors

2016-10-03
Task 1

Define the differences between weak general semaphores, strong general semaphores and binary semaphores.

Which of these three semaphore variants does the following Java implementation represent:

```java
public class Semaphore {
    private int value;

    public Semaphore(int v) {
        this.value = v;
    }

    public synchronized void release() {
        this.value++;
        this.notify();
    }

    public synchronized void acquire() throws InterruptedException {
        while(this.value <= 0) {
            wait();
        }
        this.value--;
    }
}
```

Task 2

Write a blocking stack implementation in Java using semaphores from java.util.concurrent instead of the Java concurrency primitives wait(), notify() or notifyAll(). You may choose to ignore interrupt handling.

The blocking stack is defined by the interface:

```java
public interface BlockingStack<T> {
    /**
     * Add a new object to the top of the stack. If there is no free space, the
     * call will block until space becomes available.
     * @param object
     * @throws InterruptedException
     */
    void push(T object) throws InterruptedException;

    /**
     * Remove the topmost object from the stack. If the stack is empty, the
     * call will block until objects are pushed onto the stack.
     * @return
     */
    T pop() throws InterruptedException;
}
```
Task 3

Write a Java implementation of Algorithm 7.4 listed below in Task 4 using the Java concurrency primitives wait(), notify() or notifyAll().

Does your implementation work exactly like the reference algorithm? Consider how the classic general monitors (as in Algorithm 7.4) differ from Java monitors.

Task 4 (Ben-Ari, 7.4)

<table>
<thead>
<tr>
<th>Algorithm 7.4: Readers and writers with a monitor</th>
</tr>
</thead>
<tbody>
<tr>
<td>monitor RW</td>
</tr>
<tr>
<td>integer readers ← 0</td>
</tr>
<tr>
<td>integer writers ← 0</td>
</tr>
<tr>
<td>condition OKtoRead, OKtoWrite</td>
</tr>
<tr>
<td>operation StartRead</td>
</tr>
<tr>
<td>if writers ≠ 0 or not empty(OKtoWrite)</td>
</tr>
<tr>
<td>waitC(OKtoRead)</td>
</tr>
<tr>
<td>readers ← readers + 1</td>
</tr>
<tr>
<td>signalC(OKtoRead)</td>
</tr>
<tr>
<td>operation EndRead</td>
</tr>
<tr>
<td>readers ← readers − 1</td>
</tr>
<tr>
<td>if readers = 0</td>
</tr>
<tr>
<td>signalC(OKtoWrite)</td>
</tr>
</tbody>
</table>
Modify the solution in Algorithm 7.4 to the problem of the readers and writers so as to implement each of the following rules (as a separate change to the original algorithm):

1. If there are reading processes, a new reader may commence reading even if there are waiting writers.
2. If there are waiting writers, they receive precedence over all waiting readers.
3. If there are waiting readers, no more than two writers will write before a process is allowed to read. In other words, ensure that, if a reader is blocked in StartRead, no more than two calls to StartWrite may complete (and will, if there are sufficient calls to StartWrite).