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:

```
1  public class Semaphore {
2      private int value;
3
4      public Semaphore(int v) {
5          this.value = v;
6      }
7
8      public synchronized void release() {
9          this.value++;
10         this.notify();
11     }
12
13     public synchronized void acquire() throws InterruptedException {
14         while(this.value <= 0) {
15             wait();
16         }
17         this.value--;
18     }
19 }
```

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:

```
1  public interface BlockingStack<T> {
2
3      /**
4       * Add a new object to the top of the stack. If there is no free space, the
5       * call will block until space becomes available.
6       * @param object
7       * @throws InterruptedException
8       */
9      void push(T object) throws InterruptedException;
10
11     /**
12      * Remove the topmost object from the stack. If the stack is empty, the
13      * call will block until objects are pushed onto the stack.
14      * @return
```

```
15          * @throws InterruptedException
16          */
17         T pop() throws InterruptedException;
18
19         /**
20          * Return the number of items in the stack: 0 <= size <= capacity
21          * @return
22          */
23         int size();
24
25     }
```

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)

Algorithm 7.4: Readers and writers with a monitor

```
monitor RW
    integer readers ← 0
    integer writers ← 0
    condition OKtoRead, OKtoWrite
    operation StartRead
        if writers ≠ 0 or not empty(OKtoWrite)
            waitC(OKtoRead)
        readers ← readers + 1
        signalC(OKtoRead)
    operation EndRead
        readers ← readers − 1
        if readers = 0
            signalC(OKtoWrite)
```

Algorithm 7.4: Readers and writers with a monitor (continued)

```
operation StartWrite
    if writers ≠ 0 or readers ≠ 0
        waitC(OKtoWrite)
    writers ← writers + 1

operation EndWrite
    writers ← writers − 1
    if empty(OKtoRead)
        then signalC(OKtoWrite)
        else signalC(OKtoRead)
```

reader	writer
p1: RW.StartRead	q1: RW.StartWrite
p2: read the database	q2: write to the database
p3: RW.EndRead	q3: RW.EndWrite

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`).