Seminar 1 - Semaphores

Answer the questions below:

1. Sometimes, multiple threads in a program write to the same file concurrently. A common problem is that the interleaving of output from different threads is unpredictable and may result in a non-readable result. Show, by using a *Semaphore*, how the output from a sequence of *print* or *println* calls of Java's System.out can be ensured to be printed without getting mixed with output from other threads[1].

2. Review the program *RTsemBuffer* below and complete the code in the *Buffer.getLine()* method. The requirement is that only one thread at a time may write or read the buffer data attribute. Additional threads should be blocked by a semaphore.

3. In task 1 above, the problem of mutual exclusion was solved by requiring the calling threads to use the semaphore in a proper way. In task 2 the semaphores were handled internally in a class responsible for the common data structure. Comment on the differences between these two approaches. Which might be the easier and safer way of programming? Why?

4. In the program provided for task 2, there is a semaphore *free*. Assume you want to provide buffering of up to 8 lines without blocking the callers. How should the program be changed?
   Hint: Consider the argument to the constructor of the class *Semaphore*.

5. What will happen if you swap the order of the calls *free.acquire(); mutex.acquire();* so that you instead do *mutex.acquire(); free.acquire();*? Hint: consider a full buffer.

Program RTsemBuffer

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

/**
 * Simple producer/consumer example using semaphores.
 * The complete example, including all classes, is put in one outer class.
 * That lets us keep it all in one file. (Not good for larger programs.)
 */
public class RTsemBuffer {
    /**
     * Static stuff which permits the class to be called as a program.
     */
    public static void main(String args[]) {
        // Since this is a static function, we can only access static data.
        // Therefore, create an instance of this class; run constructor:
        new RTsemBuffer();
    }
```

[1] Even if the printout from one thread only consists of a single *println* call, certain types of real-time systems do not ensure that a single line is not interrupted by more urgent printouts. We then have a problem similar to the one above, but for individual characters. It is also common that embedded systems (that is, systems with built-in computers) have such textual output on a simple serial port. That port is then a shared resource and we could use a semaphore to protect it just like in exercise 1. Another solution to the problem is to introduce an additional thread which is the only one that may use the output port.

```java
    /**
     * Constructor which in this example acts as the main program.
     */
    public RTsemBuffer () {
        Buffer buff = new Buffer();
        Producer p = new Producer(buff);
        Consumer c = new Consumer(buff);
        c.start();
        p.start();
        System.out.println("\n\n"+"RTsemBuffer: Threads are running ...");

        try {
            p.join();
            // Give consumer 10s to complete its work, then stop it.
            Thread.sleep(10000);
            c.interrupt(); // Tell consumer to stop.
            c.join(); // Wait until really stopped.
        }
        catch (InterruptedException e) {/* Continue termination...*/};
        System.out.println("\n"+"RTsemBuffer: Execution completed!");
    }
} // RTsemBuffer

/**
 * The producer.
 */
class Producer extends Thread {
    Buffer theBuffer;
    Producer(Buffer b) {
        super(); // Construct the actual thread object.
        theBuffer = b;
    }

    public void run() {
        String producedData = ””;
        try {
            while (true) {
                if (producedData.length()>75) break;
                producedData = new String("Hi! "+producedData);
                sleep(1000); // It takes a second to obtain data.
                theBuffer.putLine(producedData);
            }
        } catch (Exception e) { // Just let thread terminate (i.e. return from run). }
     } // run
} // Producer

/**
 * The consumer.
 */
class Consumer extends Thread {
    Buffer theBuffer;
    Consumer(Buffer b) {
        super();
        theBuffer = b;
    }

    public void run() {
        try {
            sleep(10000); // 10s until work starts.
            while (true) {
                System.out.println(theBuffer.getLine());
            }
        } catch (Exception e) {/* Let thread terminate. */};
    } // run
} // Consumer
```

```
/**
 * The buffer.
 */
class Buffer {
    Semaphore mutex; // For mutual exclusion blocking.
    Semaphore free;  // For buffer full blocking.
    Semaphore avail; // For blocking when no data is available.
    String buffData; // The actual buffer.

    Buffer() {
        mutex = new Semaphore(1);
        free = new Semaphore(1);
        avail = new Semaphore(0);
    }

    void putLine(String input) {
        free.acquire(); // Wait for buffer empty.
        mutex.acquire(); // Wait for exclusive access.
        buffData = new String(input); // Store copy of object.
        mutex.release(); // Allow others to access.
        avail.release(); // Allow others to get line.
    }

    String getLine() {
        // Task 2…
        // Here you should add code so that if the buffer is empty, the
        // calling thread is delayed until a line becomes available.
        // A caller of putLine hanging on buffer full should be released.
        // …
    }
}
```