

## Solutions to Exercise session 1

### Exercises

1. A sequence of calls to Java's **System.out** needs to be in critical section. Import the package `se.lth.cs.realtime.semaphore`, and within your class you declare the attribute

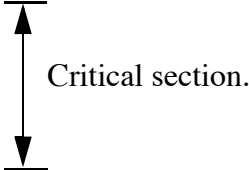
```
Semaphore sysoutLock;
```

which is instantiated and initialized to one (in the constructor of the class) by **one of** the following lines:

```
sysoutLock = new MutexSem();           // Preferred!  
sysoutLock = new CountingSem(1);       // Works in simple cases.  
(sysoutLock = new CountingSem()).give(); // Dito., less readable.
```

Then, on each place where print is called, the statements must be enclosed by take and give according to

```
sysoutLock.take();  
System.out.print(someStuff);  
System.out.print(moreStuff);  
// ....  
System.out.println(finalStuff);  
sysoutLock.give();
```



Of course, all functions using **System.out** have to follow this convention; **take** and **give** using the very same semaphore instance. Using a **MutexSem**, the run-time systems can check during **give** that the same thread first has called **take**.

2. With some additional lines in the `getLine` method, we obtain the following:

```
String getLine() {  
    avail.take();           // Wait for data available.  
    mutex.take();  
    String ans = buffData; // Get reference to data.  
    buffData = null;       // Extra care, not really needed here.  
    mutex.give();  
    free.give();  
    return ans;  
}
```

The semaphore `mutex` is actually not needed in this case since the other semaphores, which provide underflow and overflow protection, also provide mutual exclusion. Note, however, this will not be the case in Exercise 4.

3. When using semaphores as in Exercise 1, a large program will have many related pairs of take/give which can be hard to read, and errors may be very hard to find. Handling the semaphores locally in each class (without exposing them outside the class) is more structured. It agrees with the principles of object oriented programming and the program gets easier to read. We are then also free to use some other type of synchronization without changing the calling code.

4. To provide buffering of up to 8 lines without blocking the callers, we use an array as a ring buffer, the semaphore `free` should be initialized to the size of the array, and the methods `putLine` and `getLine` need to be changed. The buffer may then look according to the code below. The producer and the consumer remain unchanged.
5. If one swaps the order of the calls “`free.take(); mutex.take();`” to be “`mutex.take(); free.take();`”, then the producer and the consumer will both wait for each other. When a program hangs in this way, we call it a *deadlock*.

```

class Buffer {
    Semaphore mutex;           // For mutual exclusion blocking.
    Semaphore free;           // For buffer full blocking.
    Semaphore avail;         // For blocking when no data available.
    final int size=8;        // The number of buffered strings.
    String[] buffData;       // The actual buffer.
    int nextToPut;           // Writers index (init. to zero by Java).
    int nextToGet;          // Readers index.

    Buffer() {
        buffData = new String[size];
        mutex = new MutexSem(1);
        free = new CountingSem(size);
        avail = new CountingSem();
    }

    void putLine(String input) {
        free.take();           // Wait for buffer empty.
        mutex.take();         // Wait for exclusive access.
        buffData[nextToPut] = new String(input); // Store copy.
        if (++nextToPut >= size) nextToPut = 0; // Next index.
        mutex.give();         // Allow others to access.
        avail.give();         // Allow others to get line.
    }

    String getLine() {
        avail.take();
        mutex.take();
        String ans = buffData[nextToGet]; // Get ref. to data.
        buffData[nextToGet] = null;     // Extra care.
        if (++nextToGet >= size) nextToGet = 0; // Next index.
        mutex.give();
        free.give();
        return ans;
    }
}

```

Note, however, that it is better to explicitly declare each semaphore using the appropriate type directly. That is, declare a `MutexSem` or a `CountingSem`, and use the `Semaphore` interface only when an unknown type of semaphore is passed as an argument, or similar.

---