Exercise 1 - Semaphores

Basic knowledge of Threads, Semaphores and Java is assumed

Exercises

1. Sometimes, multiple threads in a program write to the same file concurrently. One example is a console window such as the Java console in Netscape. A common problem is that the interleaving of output from different threads is unpredictable and may result in a hardly readable result. Show, by using a *Semaphore*, how the output from a sequence of *print* or *println* calls of Javas *System.out* can be ensured to be printed without getting mixed with output from other threads.

2. 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 in Exercise 1, 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 in is to introduce an additional thread which is the only one that may use the output port. Review the program below and complete the code in the *getLine* method. The requirement is that only one thread at a time may write, and additional writers should be blocked by the semaphore.

3. In Exercise 1 the problem of mutual exclusion was solved by requiring the calling threads to use the semaphore in a proper way. In Exercise 2 we handled the semaphores internally in the class/object. Comment on the differences between these two approaches. Which is the easiest and safest way of programming? Why?

4. In the program provided for Exercise 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 then be changed? Hints: Consider the alternative constructor (taking an argument) of the class *CountingSem*. A so called ring-buffer is an efficient type of circular buffer that can easily be implemented by an ordinary array.

5. What will happen if you swap the order of the calls *free.take(); mutex.take();* so that you instead do *mutex.take(); free.take();*?

Program RTsemBuffer.java

```
1  import se.lth.cs.realtime.semaphore.*;
2  /**
3   * Simple producer/consumer example using semaphores.
4   * The complete example, including all classes, is put in one outer class.
5   * That lets us keep it all in one file. (Not good for larger programs.)
6   */
7  public class RTsemBuffer {
8    /**
9     * Static stuff which permits the class to be called as a program.
10    */
11   public static void main(String args[]) {
12     // Since this is a static function, we can only access static data.
13     // Therefore, create an instance of this class; run constructor:
14     new RTsemBuffer();
15   }
16
17   /**
18    * Constructor which in this example acts as the main program.
19    */
20   public RTsemBuffer () {
21     Buffer buff = new Buffer();
22     Producer p = new Producer(buff);
23     Consumer c = new Consumer(buff);
24     c.start();
25     p.start();
26     System.out.println("\n\n"+"RTsemBuffer: Threads are running ...");
27     try {
28       p.join();
29       // Give comsumer 10s to complete its work, then stop it.
30       Thread.sleep(10000);
31       c.interrupt(); // Tell consumer to stop.
32       c.join(); // Wait until really stopped.
33     }
34     catch (InterruptedException e) {/* Continue termination...*/};
35     System.out.println("\n"+"RTsemBuffer: Execution completet!");
36   }
37
```

```
38  /**
39   * The buffer.
40   */
41  class Buffer {
42    Semaphore mutex; // For mutual exclusion blocking.
43    Semaphore free; // For buffer full blocking.
44    Semaphore avail; // For blocking when no data is available.
45    String buffData; // The actual buffer.
46
47    Buffer() {
48      mutex = new MutexSem();
49      free = new CountingSem(1);
50      avail = new CountingSem();
51    }
52
53    void putLine(String input) {
54      free.take(); // Wait for buffer empty.
55      mutex.take(); // Wait for exclusive access.
56      buffData = new String(input); // Store copy of object.
57      mutex.give(); // Allow others to access.
58      avail.give(); // Allow others to get line.
59    }
60
61    String getLine() {
62      // Exercise 2 ...
63      // Here you should add code so that if the buffer is empty, the
64      // calling process is delayed until a line becomes available.
65      // A caller of putLine hanging on buffer full should be released.
66      // ...
67    }
68  }
69
70  /**
71   * The producer.
72   */
73  class Producer extends Thread {
74    Buffer theBuffer;
75    Producer(Buffer b) {
76      super(); // Construct the actual thread object.
77      theBuffer = b;
78    }
79    public void run() {
80      String producedData = "";
81      try {
82        while (true) {
83          if (producedData.length()>75) break;
84          producedData = new String("Hi! "+producedData);
85          sleep(1000); // It takes a second to obtain data.
86          theBuffer.putLine(producedData);
87        }
88      }
89      catch (Exception e) {
90        // Just let thread terminate (i.e., return from run).
91      }
92    } // run
93  } // Producer
94
```

```
95   /**
96    * The Consumer.
97    */
98   class Consumer extends Thread {
99     Buffer theBuffer;
100    Consumer(Buffer b) {
101      super();
102      theBuffer = b;
103    }
104    public void run() {
105    try {
106      sleep(10000); // 10s until work starts.
107      while (true) {
108        System.out.println(theBuffer.getLine());
109      }
110    }
111    catch (Exception e) {/* Let thread terminate. */};
112    } // run
113  } // Consumer
114 } // RTsemBuffer
```

Exercise 2 - Lab 1 Preparation

The programming task - AlarmClock

The control program for an alarm clock is to be developed. With the knowledge of threads and semaphores from exercise session 1, the basic design will be made during the exercise session. The software should be fully implemented and tested during the laboratory session. That requires substantial work to be carried out between the exercise and the lab. The testing is done by linking the control software to an applet that emulates the hardware. The physical hardware is under development but possibly not available during the lab.

Concerning the interface between the control software and hardware (i.e., the same interface as to the classes emulating the hardware), it mainly consists of three parts:

- Input: Obtaining input from buttons and time settings as done by the user.
- Output: Display of current clock time and giving alarm signal.
- Time: Obtaining the real-time clock, i.e., the time base.

There are specific available interfaces, in terms of a Java classes presented below, which model the functionality of the hardware. These classes are available in a package named *done*. Additionally, in the emulation case, there are some classes in the done package that implement the graphical user interface (GUI). Those GUI classes can, however, not be accessed from the control software. In this way, we ensure that the classes developed by you will actually run on the real hardware (without any changes of the source code). Synchronization and mutual exclusion are to be handled by using semaphores. Your task is to develop the classes in the *todo* package in such a way that the following specifications are fulfilled.

Specifications

1. The displayed clock time should be updated every second. When the user has selected *Set alarm* or *Set time*, the hardware/emulator shows the set time and an update of the clock time will have no effect on the clock. However, to better verify that the software keeps track of time,

and for convenience during testing, the clock time is displayed on an additional information panel below the actual clock panel. Therefore, update the clock time every second regardless of mode.

2. It should be possible to set the clock time and the alarm time. The hardware/emulator internally handles the actual setting/editing of the time value. When the user selects another mode, the set value is written to the *ClockInput* object and *give* is called for its semaphore. For example, the user selects *Set alarm* and edits (using number keys or arrows) the alarm time. Nothing needs to be done by the control software. Then the user selects *Time* and the time maintained according to previous item is automatically displayed. Also, the alarm time set by the user gets available in the *ClockInput* and give is called on the semaphore.

3. When the clock time is equal to the alarm time, and the *Alarm on* is selected, the alarm should *beep* once a second for 20 seconds[1]. The sound should be turned off if the user pushes any button while the alarm is sounding.

4. The program should be written in Java, using the *Thread* class. All synchronization and mutual exclusion shall be achieved by using semaphores.

Lab preparation

You will design a realtime system for the AlarmClock using threads, semaphores and the provided hardware interfaces. Express your design in class diagrams and in Java code where necessary. Your design need to answer the following questions:

1. Which parallel activities are needed, i.e., which are the thread objects you need?

2. What global data structures (abstract data types which we call passive objects) are needed. What operation should they support? Will these objects be accessed concurrently from different threads, that is, do you need to provide mutual exclusion?

[1]The *ClockOutput.doAlarm()* method only makes one beep.

3. Are there other situations where semaphores are needed for synchronization?

4. If you in a thread wait for the next second by simply calling sleep(1000), what will the effect on the clock time be? Is this ok for alarm clock time updating? Can it be improved?

Handout code

You will get a simulator for this assignment that emulates the AlarmClock hardware (display and buttons). The handout code consists of two Java packages called *done* and *todo*. The *done* package contains the simulator (as an applet) as well as hardware interfaces (*ClockInput* and *ClockOutput*). The *todo* package will contain your realtime system. The package right now contains one class, *AlarmClock*, that contains a sample implementation beeping upon key presses. You will modify and extend *todo* with classes as you deem necessary for your system.

AlarmClock class

```
1  package todo;
2  import done.*;
3  import se.lth.cs.realtime.semaphore.Semaphore;
4  import se.lth.cs.realtime.semaphore.MutexSem;
5
6  public class AlarmClock extends Thread {
7      private static ClockInput  input;
8      private static ClockOutput output;
9      private static Semaphore   sem;
10
11     public AlarmClock(ClockInput i, ClockOutput o) {
12         input = i;
13         output = o;
14         sem = input.getSemaphoreInstance();
15     }
16
17     // The AlarmClock thread is started by the simulator. No
18     // need to start it by yourself, if you do you will get
19     // an IllegalThreadStateException. The implementation
20     // below is a simple alarmclock thread that beeps upon
21     // each keypress. To be modified in the lab.
22     public void run() {
23         while (true) {
24             sem.take();
25             output.doAlarm();
26         }
27     }
28 }
```

Looking inside the simulator for a moment, what is happening at startup is that the simulator creates an instance of your *AlarmClock* class and starts a new thread on the resulting object. The new thread starts executing in the *AlarmClock.run()* method.

Simulator excerpt: AlarmClock startup code

```
1 ClockInput butt2ctrl; // Interface to user actions via hardware/software.
2 ClockOutput ctrl2disp; // Interface to display hardware/software.
3 AlarmClock control; // The actual alarm-clock software.
4 // ...
5 control = new AlarmClock(butt2ctrl, ctrl2disp);
6 control.start();
```

In the same manner, when the applet is stopped (corresponding to hardware reset), there is a call *control.terminate();* which you need to override, if you want to fulfil the optional specification item 5.

The following classes are the *ClockOutput* and the *ClockInput* that describe the interface between the control software and the clock hardware/emulator. Make sure you understand the comment on the *ClockInput.getValue* method.

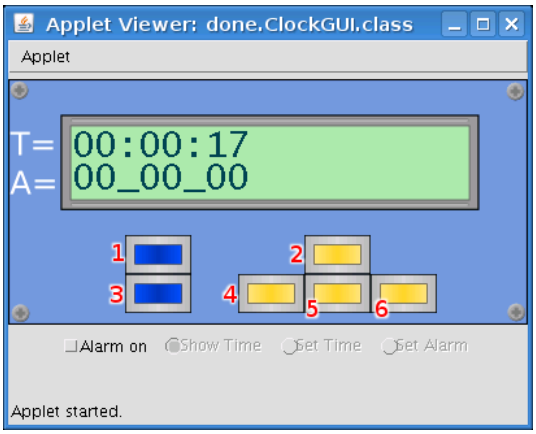
Excerpt from ClockOutput class

```
1  package done;
2
3  public class ClockOutput {
4
5      /**
6       * Wake-up clock user.
7       */
8      public void doAlarm() { ... }
9
10     /**
11      * If the display is currently used to display the time, update it.
12      * If user is using display for setting clock or alarm time, do
13      * nothing when with actual hardware or show info when simulating.
14      */
15     public void showTime(int hhmmss) { ... }
16 }
```

Excerpt from ClockInput class

```
1  package done;
2  import se.lth.cs.realtime.semaphore.*;
3
4  public class ClockInput {
5
6      /**
7       * Semaphore signaling when the user have changed any setting.
8       * Get-method to access the semaphore instance directly.
9       */
10     public Semaphore getSemaphoreInstance() { ... }
11
12     /* Package attributes, only used by the simulator/hardware. */
13     boolean alarmOn;     // Alarm activation according to checkbox.
14     int choice;          // The radio-button choice.
15     int lastValueSet;    // Value from last clock or alarm set op.
16
17     /**
18      * Get check-box state.
19      */
20     public boolean getAlarmFlag() { ... }
21
22     /**
23      * Return values for getChoice.
24      */
25     public static final int SHOW_TIME   = 0;
26     public static final int SET_ALARM   = 1;
27     public static final int SET_TIME    = 2;
28
29     /**
30      * Get radio-buttons choice.
31      */
32     public int getChoice() { ... }
33
34     /**
35      * When getChoice returns a new choice, and the previous choice
36      * was either SET_ALARM or SET_TIME, the set-value of the display
37      * is returned in the format hhmmss where h, m, and s denotes
38      * hours, minutes, and seconds digits respectively. This means,
39      * for example, that the hour value is obtained by dividing the
40      * return value by 10000.
41      */
42     public int getValue() { ... }
43 }
```

Using the simulator



First, you must click somewhere on the image of the clock to give the applet keyboard focus (light blue area). Clicking the check box or another window will steal keyboard focus again.

- Hold *Shift* to set clock time (button 1).
- Hold *Ctrl* to set alarm time (button 3).
- Hold *Shift+Ctrl* to toggle alarm on or off.
- Use direction keys for buttons 2-6.

The tow lines on the LCD display should be fairly obvious. The first line displays the current clock time. The second line displays the alarm time. When the alarm is set the separators in the alarm time turn into colons, otherwise they remain as underscores. When the alarm is beeping the separators will flash with the beeps of the alarm.

Below the buttons is a small status field which displays the alarm status with a check box and the current input mode with three radio buttons. The radio buttons may not be manipulated directly, but the check box can be used to modify the alarm status (on/off).

When either of the modes *Set Time* or *Set Alarm* is active the user may increase the digit under the cursor (indicated by an underscore) by pressing the *up* button (2), decrease the digit under the cursor with the *down* button (5), shift cursor left with the *left* button (4) and shift cursor right with the *right* button (6).