Exercise 3 - Monitors, Wait/Notify

In this exercise you will write two monitors. You will re-implement the Buffer class from exercise 1 as a monitor and you will write the monitor for a queue handling system.

Buffer

Study the monitor *Buffer* below. Write the method *getLine* in such a way that it works together with the method *putLine*. The *getLine* should fetch one line from the internal buffer array. If there is no line available, the calling thread should wait for a line to be supplied by the method putLine. Compare with the buffer made in exercise session 1.

```
1  class Buffer {
2    int available;     // Number of lines that are available.
3    final int size=8;   // The max number of buffered lines.
4    String[] buffData;  // The actual buffer.
5    int nextToPut;     // Writers index.
6    int nextToGet;     // Readers index.
7
8    Buffer() {
9      buffData = new String[size];
10   }
11
12   synchronized void putLine(String inp) {
13     try {
14       while (available==size) wait();
15     } catch (InterruptedException exc) {
16       throw new RTError("Buffer.putLine interrupted: "+exc);
17     };
18     buffData[nextToPut] = new String(inp);
19     if (++nextToPut >= size) nextToPut = 0;
20     available++;
21     notifyAll(); // Only notify() could wake up another producer.
22   }
23
24   synchronized String getLine() {
25     // Write the code that implements this method ...
26   }
27 }
```

0

A note on InterruptedException

When we in a *synchronized* method wait for some condition to become true, that is, when we call *wait*, we are forced to (by the compiler) to take care of an *InterruptedException* by

- having the method declared to throw that exception, or
- catching the exception and completing the method under the actual situation, or
- catching the exception and terminating the program, or
- catching the exception and generate an unchecked exception like divide by zero, or
- catching the exception and throwing an instance of (a subclass of) *Error*.

Having examined these alternatives we recommend (in most cases) to throw an *Error*. The purpose of the standard *Error* class in Java is to throw unchecked exceptions that the ordinary programmer is not expected to handle, and when a monitor operation fails there is normally nothing else to do than to enter some type of fail-safe or recovery mode (or simply a thread exit in simple applications). To simply throw an *se.lth.cs.realtime.RTError* facilitates this.

Queue system

Your task

Develop the control program for a queue-handling system with queue tickets, like in the post office, in shops, etc. The thread classes for input and output have already been written. Your task is to develop the monitor class that the threads use. The hardware is encapsulated in an available static object HW.

To solve the task you need to answer some questions before implementation:

1. What data is needed in the monitor?
2. Which conditions must be fulfilled to permit each monitor operation to be completed?
3. Which events may imply that the conditions have been fulfilled?
4. An impatient clerk could possibly press its button several times. How do you handle that?

5. Write the monitor!

Specification of the system

The equipment consists of three parts:

1. A ticket-printing unit with a button which customers press to get a numbered queue ticket.
2. A button at each desk/counter. The clerk pushes this button when the next customer is to be served.
3. A display that shows queue number and counter number to the next customer that can be served.

Input to the control system consists of signals from the customer and clerk buttons. Output consists of the numbers to the display. Reading input is blocking which prevents use of polling. Instead, you need one thread for each of the inputs. This agrees with the rule to have one thread for each independent external sequence of events. It is also specified that the display may not be updated more often than each 10 seconds. Therefore, it is most convenient to have a dedicated thread for updating the display. Thus, we have the following threads:

- *CustomerHandler*: Waits for a customer to push the button, calls the monitor method *customerArrived* in the monitor, and then a queue ticket is printed for the customer.
- *ClerkHandler*: Waits for the clerk to push his/her button, then calls the monitor method *clerkFree*. Each instance handles one specified clerk.
- *DisplayHandler*: Calls the monitor method *UpdateDisplay* which returns values for display update. To give customers time to react, there must be at least 10 seconds between each update.

The monitor (to be implemented)

Specification of the monitor (for simplicity of the exercise, exceptions need not be caught):

```
1  class YourMonitor {
2    private int nCounters;
3    // Put your attributes here...
4
5    YourMonitor(int n) {
6      nCounters = n;
7      // Initialize your attributes here...
8    }
9
10   /**
11    * Return the next queue number in the intervall 0...99.
12    * There is never more than 100 customers waiting.
13    */
14   synchronized int customerArrived() {
15     // Implement this method...
16   }
17
18   /**
19    * Register the clerk at counter id as free. Send a customer if any.
20    */
21   synchronized void clerkFree(int id) {
22     // Implement this method...
23   }
24
25   /**
26    * Wait for there to be a free clerk and a waiting customer, then
27    * return the cueue number of next customer to serve and the counter
28    * number of the engaged clerk.
29    */
30   synchronized DispData getDisplayData() throws InterruptedException {
31     // Implement this method...
32   }
33 }
```

Below are given the implementations of the threads in the system. Note that most of the application logic is deferred to the monitor, very little is left in the thread code. This is in general a good design principle for embedded systems. Note also how all threads are sharing the same monitor.

Activity CustomerHandler

One thread to wait for the customer to push the ticket button.

```
1  class CustomerHandler extends Thread {
2    YourMonitor mon;
3
4    CustomerHandler(YourMonitor sharedData) { mon = sharedData; }
5
6    public void run() {
7      while (true) {
8        HW.waitCustomerButton();
9        int qNum = mon.customerArrived();
10       HW.printTicket(qNum);
11     }
12   }
13 }
```

Activity ClerkHandler

One activity (thread) for each clerk to wait for the clerk to push the free button at the clerk desk.

```
1  class ClerkHandler extends Thread {
2    YourMonitor mon;
3    int id;
4
5    ClerkHandler(YourMonitor sharedData, int id) { mon=sharedData; this.id=id; }
6
7    public void run() {
8      while (true) {
9        HW.waitClerkButton(id);
10       mon.clerkFree(id);
11     }
12   }
13 }
```

Activity DisplayHandler

One activity (thread) to display a customer waiting number and a free clerk desk periodically every 10 seconds. The pairing (customer, free clerk) is encoded in the class DispData.

```
1  class DispData {
2    int ticket;
3    int counter;
4  }
```

```
1  class DisplayHandler extends Thread {
2    YourMonitor mon;
3    DispData disp;
4
5    DisplayHandler(YourMonitor sharedData) { mon = sharedData; }
6
7    public void run() {
8      while (true) {
9        try {
10         disp = mon.getDisplayData();
11         HW.display(disp.ticket, disp.counter);
12         sleep(10000);
13       } catch (InterruptedException e) { break; }
14     }
15   }
16 }
```

Exercise 4a - Deadlock analysis

Semaphores

Consider the following two thread classes S and T referring to the shared resources A, B, C, D, and E which here are allocated by use of semaphores.

```
1  class S extends Thread {
2    public void run() {
3      // ....
4      A.take();
5      B.take();
6      useAB();
7      A.give();
8      C.take();
9      useBC();
10     B.give();
11     C.give();
12     // ....
13   }
14 }
```

```
1  class T extends Thread {
2    public void run() {
3      // ....
4      C.take();
5      D.take();
6      useCD();
7      A.take();
8      useACD();
9      A.give();
10     D.give();
11     E.take();
12     useCE();
13     C.give();
14     E.give();
15     // ....
16   }
17 }
```

1. Draw a resource allocation graph. Is the system safe concerning deadlock, that is, can we conclude that there is no risk for a deadlock?

2. Review the program and the graph from 1 again. Assume there is exactly one thread of type S and one thread of type T. Explain why the system cannot deadlock!

3. Assume that there are two threads of type S, and still one T thread. Can deadlock occur in this case?

4. Change one of the threads in such a way that there will be no risk for deadlock no matter how many threads there are of each type. Note that the same computations should be carried out and still with full synchronization. Mention a drawback with the new deadlock-free implementation.

Monitors

A Java program consists of three concurrently executing threads T1, T2, and T3. They communicate with each other using four shared objects A, B, C, and D. The shared objects contain methods declared *synchronized* as well as other, non-synchronized, methods (such as s() below). Relevant parts of the program and the design of the monitors are shown below (subset of the total program only). Each thread and shared object is assumed to have references to the (other) shared objects A, B, C, and D named a, b, c, and d, respectively.

```
1 class T1
2     extends Thread {
3     void run() {
4         b.z();
5         a.x();
6     }
7 }
```

```
1 class T2
2     extends Thread {
3     void run() {
4         b.r();
5         c.y();
6         b.w();
7     }
8 }
```

```
1 class T3
2     extends Thread {
3     void run() {
4         d.u();
5     }
6 }
```

```
1 class A {
2     synchronized void x() {
3         c.y();
4     }
5
6     void s() { // NOT SYNCHRONIZED
7         // ...
8     }
9
10    synchronized void t() {
11        // ...
12    }
13 }
```

```
1 class B {
2     synchronized void z() {
3         // ...
4     }
5
6     synchronized void r() {
7         a.s();
8     }
9
10    synchronized void w() {
11        d.v();
12    }
13 }
```

```
1 class C {
2     synchronized void y() {
3         b.z();
4     }
5 }
```

```
1 class D {
2     synchronized void u() {
3         a.t();
4     }
5
6     synchronized void v() {
7         // ...
8     }
9 }
```

Tip: Entering a *synchronized* method can be interpreted as *take*, leaving as *give*.

1. Draw a resource allocation graph for the system above.
2. Can the system experience deadlock?

Exercise 4b - Lab 2 preparation

Background

Concurrent programming, i.e. writing programs consisting of a number of cooperating threads executing in parallel, is a valuable technique in many situations. It is not only suitable for implementing embedded real-time systems such as a controller for an industrial process, but can also be used for example in interactive programs where it is desired that the user interface is still responding while the program is busy with some lengthy task. It is also a good technique when it comes to writing simulation software as we will see in this laboratory exercise.

Apart from illustrating how threads can be used in simulation software, the goal of the laboratory exercise is also to give you hands-on experience with how a monitor works and how to design one.

During the exercise

You will design a simulation using monitors and threads. Unlike exercise 3 you will have to decide upon activities (threads) and their implementation as well as the monitor. Remember, it is good design praxis to put most application logic in the monitor. Also, it is good praxis to minimize the number of methods in the monitor, avoid getters and setters. Your design should answer:

- Which monitor attributes are accessed/modified by which threads?
- What condition must be true for a person to be allowed to enter the lift?
- What condition must be true for a person to leave the lift?
- What condition must be true for the lift to start moving to another floor?
- Which monitor operations are necessary in order to guarantee proper synchronization of the various threads? What, and how much, must be performed as a part of each monitor operation?

Tip: The system should consist of a person thread class, a lift thread class, a monitor class according to the specification below, and a main() method which creates instances of the thread classes and the monitor class and starts the simulation.

Assume one thread for each person in the lift simulation, i.e. 20 persons equals 20 person threads. Also, assume that the person thread is re-used for another person after the current person has exited the lift.

Specification

A concurrent program simulating how a lift in an office building is used is to be written. The program shall contain a thread which controls the movement of the lift and a number of identical threads each simulating a person travelling with the lift. As a first approach to the problem, the following simple model is to be used:

Each person thread is activated after a randomly chosen delay (we suggest that the delay is in the range 0-45 seconds). When activated, the person thread determines between which floors he is to travel. The initial and target floors are to be chosen randomly. Make sure that the target floor is different from the initial floor! The thread then continues by performing the travel using the lift. After completing the travel, the thread is again to be delayed a randomly chosen time after which a new travel is performed.

The lift thread simulates the lift moving up and down through the lift shaft stopping at each floor to let travellers enter and exit the lift. It stops at each floor regardless of whether there are persons waiting for entering or exiting at the floor or not. The lift continuously moves between the bottom floor (floor 0) and the top floor (floor 6).

The lift is assumed to have room for a maximum of four (4) persons at any time.

The lift thread and the person threads need to synchronize their actions. For example, no person should enter the lift unless it has stopped on the same floor as the person is located. It is of course also necessary to verify that the lift is not already occupied, et cetera. Therefore, a shared monitor object containing shared attributes should be used. The attributes protected by your monitor should include the following:

Monitor state

Your monitor methods need to keep the state appropriately updated.

```
int here;    // If here!=next, here (floor number) tells from which floor
             // the lift is moving and next to which floor it is moving.
int next;    // If here==next, the lift is standing still on the floor
             // given by here.
int[] waitEntry;// The number of persons waiting to enter the lift at the
                // various floors.
int[] waitExit; // The number of persons (inside the lift) waiting to leave
                // the lift at the various floors.
int load;    // The number of people currently occupying the lift.
```

Visualization

Furthermore, in order to illustrate the simulation as it is running, the *LiftView* graphics class is available:

```
/** Class for opening a window displaying seven floors and a lift shaft.
 * methods for animating a lift basket and persons travelling using the lift.
 */
public class LiftView {
  /** Constructor. Displays the simulation window. */
  public LiftView();

  /** Draws a lift basket containing load persons on floor number floor. If
   * the number of persons in the lift is less than it was the last time
   * drawLift was called, an animation showing a person leaving the lift is
   * displayed. Should be called whenever the number of persons in the
   * lift is changed. */
  public void drawLift(int floor, int load);

  /** Draws persons number of persons waiting for the lift on floor floor.
   * Should be called whenever the number of waiting persons is changed. */
  public void drawLevel(int floor, int persons);

   /** Shows an animation moving the lift basket from floor number here to
    * floor number next. The number of persons drawed in the lift is the same
    * as the number of persons passed to drawLift the last time it was
    * called. */
  public void moveLift(int here, int next);
}
```

For the visualization to work there are some additional requirements on when the above methods may be called. The *drawLift* and *drawLevel* methods must be called *every* time the number of persons in the lift or waiting persons on a floor, respectively, are changed. It is not possible to follow the simulation on the screen otherwise.

The LiftView class is not entirely thread-safe, i.e. strange things can happen if two threads calls methods in the class simultaneously. You will

therefore be forced to consider a LiftView object a shared reseource to which access must be synchronized. The following rules apply:

- Two threads must not simultaneously call *drawLevel*.
- Two threads must not call *drawLift* simultaneously.

To speed up the simulation and make the animation more realistic you should allow *drawLevel* to be called at the same time as a call to *moveLift* is in progress. Calling *moveLift* should thus not block other threads.

Miscellaneous

In your design it is not allowed for you to share thread and monitor classes, i.e. you are not allowed to have thread classes that double as monitors. This is often a good design praxis when building embedded systems.

Random numbers can be generated using the Math.random() method which returns a random number between 0 and 1 (not including 1.000...). To generate a random integer between 0 and 45 you can for example write:

```
1 int delay = (int) (Math.random()*46.0);
```

Note that we have to multiply with 46 instead of 45 in order to make it possible to receive 45 as the result.

It is suggested that you make it possible for the user to specify the number of persons travelling using the lift. This can for example be done by requiring the user to specify this number on the command line when starting the simulation program. He could for example start the program with the command

```
java Lift 20
```

to run the program with 20 travellers. Your main() method can retrieve this number according to the following sample code:

```
1 public static void main(String[] args) {
2   int travellers = Integer.parseInt(args[0]);
3   ...
4 }
```

It is highly recommended that you write your program and attempt to remove any compilation errors before the laboratory session. Testing the program and removing errors can require a lot of time in some cases.