Exam
Concurrent and Real-Time Programming

2014–10–25, 14.00–19.00

You are allowed to use the Java quick reference and a calculator. Also dictionaries for English (and the native language for each student) are allowed. Answers in Swedish or English.

The exam has two parts; first the theory part that consists of a set of theory problems, and the final construction part that consists of two problems. The first construction problem is referred to as the programming problem, while the final problem is the design problem.

To pass the exam (grade 3), you have to solve most of the theory part, and you have to develop an acceptable solution to the programming problem. At least a partial solution to the design problem is required for the highest grade (5).

---

1. Robot-Klas has got a new work laptop, with Windows 8.1 and ClassicShell, and he experiences the following problems:
   1. Occasionally at boot time, during the initialization of various windows processes/threads, the computer 'hangs' with no progress and the CPU load being practically zero.
   2. With an earlier version of his email client, he experienced that the program and the OS both was doing something (processes running, using the CPU) but without progress from the user's point of view.
   3. When running a robot simulator that simulates robot dynamics in virtual time such that the generated trajectories (with time-stamped points as steps on the path to move along) later on should be useful for real-time execution (as input data when running the control on a real-time OS), it turns out that the time-stamps are set (in code from the robot provider) by using the real-time clock of the PC hardware.
   4. Several programs, depending on the timing of the execution (affecting how the code is interleaved at run-time), occasionally give the wrong user feedback/messages.

   a) What are the terms we use to refer to each of the situations described in item 1 and 2? *(2p)*

   b) What do we call the type of error that is reflected in each of item 3 and 4? *(2p)*

2. Java monitors have a built-in signaling mechanism that is utilized through the method calls wait, notify and notifyAll.

   a) Why do we write:

   ```
   while (!condition) wait();
   ```

   and not:

   ```
   if (!condition) wait();
   ```

   as, for instance, was done in the original definition by Hoare? *(1p)*

   b) Concerning state and execution order, mention one important difference between signaling via a counting semaphore and signaling in a monitor by means of wait and notify. *(1p)*

3. The program below takes two integers as input on the command line. These integers control how many threads are started in the main program. The threads are communicating through three monitors; MonitorA, MonitorB and MonitorC. For what values of the input parameters may the program deadlock? Motivate your answer! *(3p)*

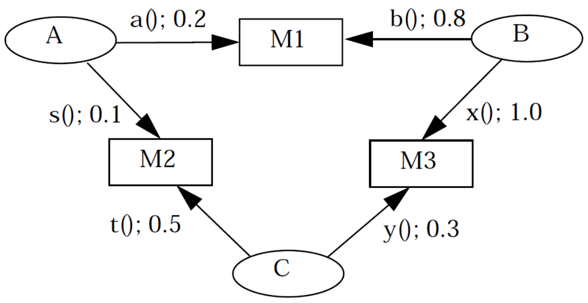
```
public class MainProgram {
  public static void main(String[] args) {
    Global g = new Global();
    g.a = new MonitorA(g);
    g.b = new MonitorB(g);
    g.c = new MonitorC(g);
    int noOfThreads =
      Integer.parseInt(args[0]);
    for(int i=0;i<noOfThreads;i++)
      new ThreadOne(g).start();
    noOfThreads =
      Integer.parseInt(args[1]);
    for(int i=0;i<noOfThreads;i++)
      new ThreadTwo(g).start();
  }
}

class Global {
  public MonitorA a;
  public MonitorB b;
  public MonitorC c;
}

class ThreadOne extends Thread {
  private Global g;
  public ThreadOne(Global g) {
    this.g = g;
  }
  public void run() {
    while(true) {
      g.a.first();
      //...
      g.c.seventh();
      //...
    }
  }
}

class ThreadTwo extends Thread {
  private Global g;
  public ThreadTwo(Global g) {
    this.g = g;
  }
  public void run() {
    while(true) {
      g.b.fifth();
      //...
      g.b.fourth();
      //...
    }
  }
}
```

```
class MonitorA {
  private Global g;
  public MonitorA(Global g) {
    this.g = g;
  }
  public synchronized void first() {
    //...
    g.b.third();
  }
  public synchronized void second() {
    //...
  }
}

class MonitorB {
  private Global g;
  public MonitorB(Global g) {
    this.g = g;
  }
  public synchronized void third() {
    //...
  }
  public synchronized void fourth() {
    //...
    g.c.sixth();
  }
  public synchronized void fifth() {
    //...
  }
}

class MonitorC {
  private Global g;
  public MonitorC(Global g) {
    this.g = g;
  }
  public synchronized void sixth() {
    //...
  }
  public synchronized void seventh() {
    //...
    g.a.second();
  }
}
```

4. You are implementing a realtime system running three periodic activities. The basic priority inheritance protocol is in use. The deadline D is shorter than the period for threads B and C, as shown in the following table:

Thread	T	C	D
A	8	4	8
B	10	2	4
C	16	3	6

The activities are communicating through three monitors. Each monitor method call is annotated with the call WCET given in milliseconds.



You are preparing for a schedulability analysis. Calculate blocking times assuming the system is scheduled by RMS. Then repeat the calculation assuming DMS scheduling is used. Which scheduling offer the lowest blocking times? Motivate your calculations! *(3p)*

5. a) Systems that are based on static scheduling for managing CPU time are typically also based on static memory allocation. That is, all objects ever needed are created when starting and object references are never changed, and hence there is no need for a deallocation procedure (called explicitly or by means of a GC). When is such a static approach applicable? When is such a static approach not suitable? *(1p)*

   b) In the dynamic case, with dynamic memory management, Java is assuming automatic management by means of a GC (Garbage Collector), which reclaims memory that is no longer used by any object. In some other languages without a GC, with manual deletion of objects, programming errors can have severe consequences. Name two such error types, and tell what they are results of. *(1p)*

   c) The use of a GC avoids the errors according to the previous item, but some concern is required in the case of real-time requirements. In the case of priority-based dynamic scheduling (of CPU time) as in most real-time operating systems, how can memory deallocation be organized such that deadlines for high-priority threads can be met? *(1p)*

6. Implement a class *FIFOSem* that implements the interface *Semaphore* according to

```
public interface Semaphore {
    /**
     * Increments by 1 the counter represented by this semaphore, and notify
     * at least one, if any, of the waiting threads.
     * Basically this means executing the following two lines, <br>
     * <code> ++count; </code><br>
     * <code> notify(); </code><br>
     * <em>atomically</em> (synchronized, with interrupts disabled, or in
     * hardware, as accomplished in a particular type of system).
     */
    void give();

    /**
     * Causes the calling thread to block until the counter that represents
     * this semaphore obtains a positive value. On return the counter, named
     * <code>count</code> below, is decremented by 1.
     * Basically this means executing<br>
     * <code> while (count<1) wait(); </code><br>
     * <code> --count; </code><br>
     * <em>atomically</em> (synchronized, with interrupts disabled, or in
     * hardware, as accomplished in each particular type of system). */
    void take();
}
```

but with the difference that threads calling take should get the semaphore in the same order as they called take. That is, take should work in a "first come first serve" manner (also called First In First Out; FIFO). The solution does not need to be efficient (concerning the number of context switches if many threads are waiting) but should be simple to understand. *(3p)*

PENALTY TRAFFIC LIGHT WITH CAMERA

7. The use of speed cameras to limit vehicle speed (with respect to speed limits) is nowadays common in most industrialized countries. In combination with a government-controlled database of collected data, this might be a dangerous development from a personal integrity and legal point of view:

   - Individuals can be tracked and traced without their knowledge.
   - Numerous faulty sensors have resulted in penalties to legal drivers. Since several penalties in a row is considered criminal in many countries, the justice system is potentially compromised due to faulty sensing.

   The situation is exceptionally bad in some countries, like in the UK, where average speed between speed cameras is monitored as well. However, some countries have developed alternatives. In Portugal, when entering a village there are *penalty red lights*. This is a traffic light, typically without any crossing, that turns red if a vehicle speed above the speed limit is detected. Not only does the speeder loose time, but the cars lining up behind get delayed too, imposing social pressure on the speeding driver.

   To further extend the idea, a red-light passage camera could capture those who do not stop at red light, or those who do not keep proper speed in the winter when the road is icy/slippery and hence might slide pass the red. *The main advantage is that sensor uncertainties do not result in legal uncertainty.* Images captured by a single camera snapshot of both the license plate and the traffic light are easily verified, opposite to speed penalties which require sensing of position change over time, which can go wrong in many ways.

Programming task

Your task is to develop part of a controller for a penalty red light with camera-based capturing of vehicles going against red. The hardware interfaces to control the traffic light, detecting vehicles passing the traffic light, capturing images and reporting abuse are already available and described below. Activities are also given as well as a specification of the monitor state and methods used by the activities. The monitor implementation is left to you.

Hardware interface classes (given)

Interface to the traffic light: Only one method, which immediately turns on the requested light.

```
class Signal {

    /**
     * Update traffic light signal.
     */
    static synchronized void
        show(boolean red, boolean yellow, boolean green) { /* ... */ }
}
```

Interface to the sensor systems: Two sensor systems are used. A (radar-based) speed detector system measures vehicle speed at some distance before the traffic light. The system is accessed through the methods *awaitSpeed* and *freeSpeed*. A passage detection system detects vehicles passing the traffic light position. The system is accessed through the methods *awaitPassage* and *freePassage*.

```
class Detection {

    /**
     * Block until next vehicle is detected.
     * @return speed of detected vehicle.
     */
    synchronized float awaitSpeed() { /* ... */ }

    /**
     * Disable speed-checking; not used outside Germany.
     * Unblocks/notifies all callers of awaitSpeed.
     */
    synchronized void freeSpeed() { /* ... */ }

    /**
     * Block until a vehicle is passing the traffic light,
     * or until freePassage unblocks.
     * @return true if vehicle was passing, false if freePassage was called.
     */
    synchronized boolean awaitPassage() { /* ... */ }

    /**
     * Stop monitoring passage (used when it is no longer red light).
     * Unblocks/notifies all callers of awaitPassage.
     */
    synchronized void freePassage() { /* ... */ }
}
```

Interface to the camera: One method for capturing an image of any vehicle passing the traffic light. The camera is positioned so that it captures the passing vehicle (but no other vehicles) with the traffic light within that (color) picture. The inner class *Image* holds the image data.

```
public class Camera {
    public static class Image {...}
    public static Image snapshot() {...}
}
```

Interface to the supervisory system: The supervisor may send a limited set of commands to the traffic light controller. These commands are read through the *getCommand* method. Currently only one command is specified, "status". Received commands are replied, posted through the *postReply* method. The *anyCommand* method may be used to check if any commands are available. Red light violations are posted through the *postReport* method. Violations may be posted at any time and they are not replies to commands. The timestamp parameter expects a time as returned by *System.currentTimeMillis()*.

```
class Network {

    /**
     * Obtain command from supervisory system.
     * Blocks caller until command is available.
     */
    synchronized String getCommand() { /* ... */ }

    /**
     * Peek if there is any command to be obtained. Not blocking.
     */
    synchronized boolean anyCommand() { /* ... */ }

    /**
     * Send a reply as an answer to an obtained command.
     * @param reply the result to be posted to supervisory system.
     */
    synchronized void postReply(String reply) { /* ... */ }

    /**
     * Send a speeding report including the photo of the offender.
     * @param img the snapshot taken by the camera.
     * @param timestamp the time when the picture was taken.
     */
    synchronized void postReport(Camera.Image img, long timestamp) { /* ... */ }
}
```

Threads and states (given, changes permitted)

The speed obtainer activity is responsible for registering speed on approaching vehicles with the monitor.

```
public class SpeedObtainer extends Thread {
    Detection sensing;
    Monitor mon;
    public SpeedObtainer(Detection sensing, Monitor mon) {
        this.sensing = sensing;    this.mon = mon;
    }

    public void run() {
        while (!isInterrupted()) {
            mon.registerSpeed(sensing.awaitSpeed());
        }
    }
}
```

The light controller activity is responsible for changing the color of the traffic light through the cyclic sequence green, yellow, red and back to green. It is also responsible for initiating the GYRG sequence upon speeding, which in turn enables the red light violation detection including the image capture.

```
public class LightController extends Thread {
    Monitor mon;
    public LightController(Monitor mon) {this.mon = mon;}

    public void run() {
        while (!isInterrupted()) {mon.color();}
    }
}
```

The state reporter activity is responsible for receiving supervisory commands and replying to them. After receiving a "status" command a traffic report is sent every 5 seconds, or immediately when the traffic situation changes until another command is received.

```
public class StateReporter extends Thread {
    Network net;
    Monitor mon;

    public StateReporter(Network net, Monitor mon) {
        this.net = net; this.mon = mon;
    }

    public void run() {
        String cmd = "";
        STATE state = STATE.EMPTY;
        while (!isInterrupted()) {
            long now = System.currentTimeMillis();
            if (cmd=="") {
                net.postReply("alive");
            } else if (cmd=="status") {
                while (!net.anyCommand()) {
                    state = mon.awaitStateChange(state, now+5*1000);
                    now = System.currentTimeMillis();
                    net.postReply(state.toString());
                }
            } else if (cmd=="assist-NSA") {
                net.postReply("NOT supported!");
            } else {
                net.postReply("Unknown command: "+cmd);
            }
            cmd = net.getCommand();
        }
    }
}
```

The penalty reporter activity is responsible for posting violation reports.

```
public class PenaltyReporter extends Thread {
    Monitor mon;
    Network net;
    Detection trigger;

    public PenaltyReporter(Monitor mon, Network net, Detection trigger) {
        this.mon = mon; this.net = net; this.trigger=trigger;
    }

    public void run() {
        while (!isInterrupted()) {
            mon.awaitRed();
            if (trigger.awaitPassage()) {
                System.out.println("You are busted!");
                net.postReport(Camera.snapshot(), System.currentTimeMillis());
            } else {
                System.out.println("Have a nice day!");
            }
        }
    }
}
```

The state class is not an activity. It enumerates the different traffic situations state the monitor may be in. Also, these traffic situations are reported to the supervisor.

```
enum STATE {
    EMPTY,      // No vehicles for a while; road is empty.
    TRAFFIC,    // Vehicles driving properly; traffic ok.
    SPEEDING,   // At least one car driving too fast.
    RED,        // Stop traffic, after SPEEDING, via yellow.
    REPORT      // Vehicle driven against red; reporting.
}
```

Monitor (to implement)

```
public class Monitor {
    /** Some attributes, to be used or to be changed... */
    private STATE state = STATE.EMPTY; // Assume no traffic until detected.
    private float maxspeed;            // [m/s] = [km/h]/3.6
    private Detection trigger;         // The up-road speed/vehicle detector.
    private long recentlyAny;          // Time of most recent vehicle.
    private long recentlyFast;         // Time of most recent speeding vehicle.
    private char c = ' ';              // Traffic light color state {R|Y|G}.

    /** Suggested contructor, assuming some static hardware interfaces.*/
    public Monitor(float limit, Detection trigger) {
        maxspeed = limit;    this.trigger = trigger;
    }

    /**
     * Inform system that a new vehicle is approaching,
     * @param spd the measured speed for this most recent vehicle.
     */
    synchronized void registerSpeed(float spd) {/*...*/}

    /**
     * Support states and colors regarding the traffic light and system status.
     * @return some color state, possibly after infinite time.
     */
    synchronized char color() {/*...*/}

    /**
     * Block until RED, i.e., until there is any speeding vehicle.
     */
    synchronized void awaitRed() {/*...*/}

    /**
     * Report state (but not vehicles) of road section to supervisory system.
     * If road is EMPTY the caller is blocked until there is anyONE coming.
     * Otherwise, wait for the change of state maximum until time is 'until'.
     * @param old is the previous state
     * @param until is the time to give up waiting unless in state EMPTY.
     * @return the present STATE
     */
    synchronized STATE awaitStateChange(STATE old, long until) {/*...*/}
}
```

If needed, adding further methods is permitted.

a) Implement the *awaitRed* monitor method. The method should block until the traffic light turns (or is about to turn) red. *(2p)*

b) Implement the *awaitStateChanged* monitor method. The method should block until state change occurs when the traffic situation changes, or until the time limit is reached. If the road is empty, and EMPTY has been reported, the method should ignore the time limit and block until the traffic situation changes. *(3p)*

c) Implement the *registerSpeed* monitor method. The method should properly update the traffic situation state to *TRAFFIC* or *SPEEDING* and other variables of the monitor. *(3p)*

d) Implement the *color* monitor method. The method is responsible for running the green-yellow-red-green cycle on the traffic light. During green light it should trigger a cycle when a speeding vehicle is approaching. Yellow light is shown for 2 seconds (fixed for now, in a leter version to be dependent on the speed limit), then a switch to red light occurs. Red light is kept for 15 seconds. If a speeding vehicle occurs during this period the red light time is simply reset to a further 15 seconds from the overspeed measurement time. Finally, if there has been no traffic (and no red light) for more than 30 seconds the traffic situation state should change to *EMPTY*. *(4p)*

Note: Clearly show and comment in the assigment any modifications to existing classes/threads (if any) that you do. Also, clearly explain additional states (that you might add) to the monitor. Changes and additions need to be stated as code, including comments as needed for understanding the solution.

ON THE DESIGN OF THE PENALTY TRAFFIC LIGHT

8. a) In the previous programming task there is the central monitor named `Monitor`, and some additional monitors for the IO classes. One such IO class is the `Detection`, which among other things has methods `awaitPassage` and `freePassage`. The `awaitPassage` is blocking (not shown) on a `wait` inside that monitor, and `freePassage` calls `notify` after having set some internal attribute accordingly when `awaitPassage` should give up waiting. Brifly describe two other ways of accomplishing such blocking and suspension of the blocking. *(2p)*

b) Although the proposed speed-checking system is designed for integrity of citizens, a change of thread `StateReporter` (for instance implementing the `assist-NSA` taking pictures without any speeding reason) could mean the opposite. Suggest two techniques for minimizing the risk for unauthorized/illegal changes. *(2p)*

c) If you would replace the given solution with a central Monitor (as a passive object holding the application state) with a solution based on message-passing for communication and synchronization, how would that be structured? Explain by a figure along with explanatory text including (at least) the involved threads and buffers therein. *(2p)*

Short solution

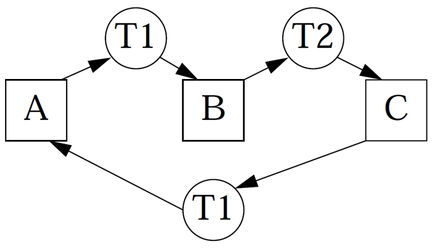
1. a) Deadlock, and livelock, respectively.

   b) Item 3 is the result of an real-time error, at least when considered as a system in which the software should be part of real-time correctness, and perhaps the critical threads where assigned real-time priority. Also concurrency-error is an acceptable answer, with a motivation that Windows is not a real-time OS, and hence the application only needs to be concurrency-correct in order to give a (given the system) correct answer (based on simulated time in this case). Item 4 describes the result of a concurrency error, since (despite timing dependency) there are interleaving that result in a concurrency fault.

2. a) A Hoare-monitor guarantees that the condition is fulfilled when the waiting thread starts after wait (this puts additional demands on both the starting thread (the one calling notify) and how the waiting thread is queued). A Java monitor does not fulfill these demands and the condition may no longer be valid when the waiting thread is started.

   b) The give operation on a semaphore has a memory, i.e. even if the queue of waiting threads is empty the information stored in the internal state variable of the semaphore remains. A notify / notifyAll is lost if the waiting queue is empty. The monitor case instead must use program state variables in the monitor as memory.

3. The two command line arguments control how many threads of type ThreadOne and ThreadTwo are instantiated and started, respectively. A resource allocation graph for the system indicates that two or more threads ThreadOne and one or more ThreadTwo are needed for a deadlock to be possible.



4. RMS thread priority order (highest first): A, B, C
   $B_C = 0$
   $B_B = max(M3.y, M2.t) = max(0.3, 0.5) = 0.5$ (push-through on M2 and direct blocking on M3)
   $B_A = M1.b + M2.t = 0.8 + 0.5 = 1.3$ (direct blocking on M1 and M2)

   DMS thread priority order (highest first): B, C, A
   $B_A = 0$
   $B_C = max(M1.a, M2.s) = max(0.2, 0.1) = 0.2$ (push-through on M1 and direct blocking on M2)
   $B_B = M1.a + M3.y = 0.2 + 0.3 = 0.5$ (direct blocking on M1 and M3)

   DMS offers the lowest blocking times.

5. a) Not applicable when dynamic data structures are used.

   b) Memory leak and dangling pointer.

   c) GC runs at lower priority.

6.
```
public class FIFOSem implements Semaphore {
    private long count = 0;
    private long turn = 0;
    private long ticket = 0;
    public synchronized void give() {
        count++;
```

```
            notifyAll();
        }
        public synchronized void take() throws Exception {
            long myTicket = ticket++;
            while (myTicket>turn || count<=0) wait();
            turn++;
            count--;
            notifyAll();
        }
    }
```

7. **Start system**

```
    public class Main {

      static Monitor mon;

      public static void main(String[] args) {
        Detection sens = new Detection();
        Network net = new Network();
        mon = new Monitor((float)(50/3.6), sens);
        (new StateReporter(net, mon)).start();
        (new LightController(mon)).start();
        (new PenaltyReporter(mon, net, sens)).start();
        (new SpeedObtainer(sens, mon)).start();
      }
    }
```

HW interfaces (with mockup code)

```
    class Detection {

      boolean isRed;
      int counter;

      /**
       * Block until next vehicle is detected.
       * @return speed of detected vehicle.
       */
      synchronized float awaitSpeed() {
        try {wait((long) (Math.random() * 10 * 1000));
        } catch (Throwable x) {}
        return (float) (Math.random() * 70.0 / 3.6);
      }

      /**
       * Disable speed-checking; not used outside Germany. Unblocks/notifies all
       * callers of awaitSpeed.
       */
      synchronized void freeSpeed() {
        return;
      }

      /**
       * Block until a vehicle is passing against/despite red light.
       * @return true if vehicle was passing, false if freePassage was called.
       */
      synchronized boolean awaitPassage() {
        isRed = true;
        try {
```

```
      wait(100 + (long) (1000 * Math.random()));
      if (isRed && Math.random() < 0.2) {
        return true;
      } else {
        while (isRed) wait();
      }
    } catch (Throwable x) {}
    return false;
  }

  /**
   * Stop monitoring red passage, since is is no longer red light.
   * Unblocks/notifies all callers of awaitPassage.
   */
  synchronized void freePassage() {
    notifyAll();
    isRed = false;
    return;
  }
}


class Network {

  /**
   * Obtain command from supervisory system.
   * Blocks caller until command is available.
   */
  synchronized String getCommand() {
    return "status";
  }

  /**
   * Peek if there is any command to be obtained.
   */
  synchronized boolean anyCommand() {
    return false;
  }

  /**
   * Send a reply as an answer ot an obtained command.
   * @param reply the result to be posted to supervisory system.
   */
  synchronized void postReply(String reply) {
    System.out.println("Sent over network: " + reply);
  }

  /**
   * Send a speeding report including the photo of the bastard.
   * @param img the snapshot taken by the camera.
   * @param timestamp the time when the picture was taken.
   */
  synchronized void postReport(Camera.Image img, long timestamp) {
    System.out.print("Report posted: " + img.picture);
    System.out.println(" at time " + timestamp);
  }
}


class Signal {
  static synchronized void show(boolean red, boolean yellow, boolean green) {
    if (red)    System.out.println("        -RED-");
```

```
        if (yellow) System.out.println("        YELLOW");
        if (green)  System.out.println("        GREEN");
      }
    }
```

Monitor

```
public class Monitor {

    private STATE state = STATE.EMPTY; // Assume no traffic until detected.
    private float maxspeed;            // [m/s] = [km/h]/3.6
    private Detection trigger;         // The up-road speed/vehicle detector.
    private long recentlyAny;          // Time of most recent vehicle.
    private long recentlyFast;         // Time of most recent speeding vehicle.
    private char c = ' ';              // Color state of the traffic light.
    private long red;                  // Last starting time for red light

    public Monitor(float limit, Detection trigger) {
        c = 'G';
        maxspeed = limit;
        this.trigger = trigger;
    }

    /**
     * Block until RED, i.e., until there is any speeding vehicle.
     */
    synchronized void awaitRed() {
        try {
            while (c != 'R') {
                wait();
            }
        } catch (InterruptedException exc) {
            return;
        }
        return;
    }

    /**
     * Report state (but not vehicles) of road section to supervisory system.
     * If road is EMPTY the caller is blocked until there is any vehicles coming.
     * Otherwise, wait for the change of state maximum until time is 'until'.
     * @param old is the previous state
     * @param until is the time to give up waiting unless in state EMPTY.
     * @return the present STATE
     */
    synchronized STATE awaitStateChange(STATE old, long until) {
        long time;
        try {
            if (state == STATE.EMPTY && old == STATE.EMPTY) {
                while (state == old) {
                    wait();
                }
            } else {
                while (state == old && (time=System.currentTimeMillis()) < until) {
                    wait(until-time);
                }
            }
        } catch (InterruptedException exc) {
            return null;
        }
        return state;
    }
    /**
     * Inform system that a new vehicle is approaching,
     * @param spd the measured speed for this most recent vehicle.
     */
    synchronized void registerSpeed(float spd) {
        recentlyAny = System.currentTimeMillis();
        if (state == STATE.EMPTY) {
            state = STATE.TRAFFIC;
        }
        if (spd > maxspeed) {
            if (state == STATE.TRAFFIC) {
                state = STATE.SPEEDING;
            }
            recentlyFast = System.currentTimeMillis();
        }
        notifyAll();
    }

    /**
     * Support states and colors regarding the traffic light and system status.
     */
    synchronized char color() {
        long t0, tf;
        try {
            switch (c) {
            case 'R':
                tf = (recentlyFast > red ? recentlyFast : red) + 15*1000;
                while ((t0=System.currentTimeMillis()) < tf) wait(tf-t0);
                if (recentlyFast <= tf-15*1000) { // No new speeding, change to green light
                    c = 'G';
                    state = STATE.TRAFFIC;
                } // Another speeder incoming, keep red for longer time.
                break;
            case 'Y':
                tf = System.currentTimeMillis() + 200*(long)maxspeed;
                while ((t0=System.currentTimeMillis()) < tf) wait(tf-t0);
                c = 'R';
                red = System.currentTimeMillis();
                break;
            case 'G':
                trigger.freePassage();
                while (state != STATE.SPEEDING) {
                    // Road with traffic
                    while (state != STATE.SPEEDING &&
                           (t0=System.currentTimeMillis()) < (tf=recentlyAny+30*1000))
                        wait(tf-t0);
                    if (state == STATE.SPEEDING) break;
                    // Empty road
                    state = STATE.EMPTY;
                    notifyAll();
                    while (state != STATE.SPEEDING && state == STATE.EMPTY) wait();
                }
                state = STATE.RED;
                c = 'Y';
                break;
            default:
                c = ' ';
            }
            notifyAll();
        } catch (InterruptedException exc) {
            return ' ';
            }
            return c;
        }

    }
```

8. No solution given.