Tentamen
EDA698 – Realtidssystem (Helsingborg)

2016–10–25, 14.00–19.00

Det är tillåtet att använda Java snabbreferens och miniräknare, samt ordbok.
Det går bra att använda både engelska och svenska uttryck för svaren.

Den här tentamen består av två delar: *teori*, frågor 1-5 (18 poäng), och *programmering & design*, frågor 6-7 (12 poäng). För godkänd (betyg 3) krävs sammanlagt ca hälften av alla 30 möjliga poäng samt att det krävs ca en tredjedel av poängen i varje del för sig. För högsta betyg (5) krävs dessutom att en del av poängen uppnås med (del)lösningar till båda uppgifterna i *programmering & design* (preliminärt).

---

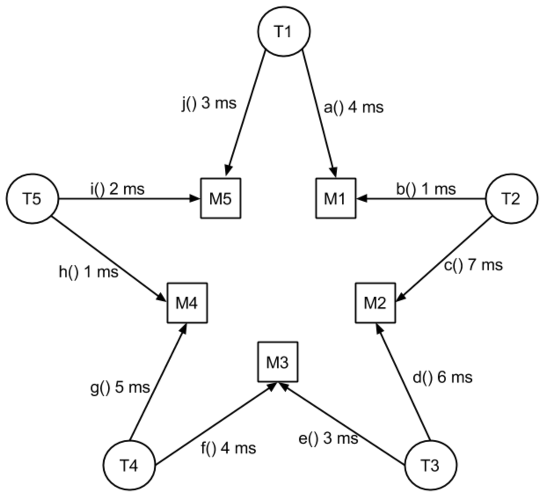
1. **Java monitors**
   In Java, we have two commonly used operations on monitors, *notify()* and *notifyAll()*.
   a) Give an example of a situation where you want to use *notify()* instead of *notifyAll()*. *(1p)*
   b) Give an example of a situation where you want to use *notifyAll()* instead of *notify()*. *(1p)*

2. **Deadlock analysis**
   A Java program has two types of threads, T1 and T2. They share a set of four resources A, B, C and D. In their run() methods, the threads T1 and T2 acquire and use the resources as shown below. The code segments where the threads do their actual work are replaced by *useXYZ()*, signifying which resources must be acquired before executing that code.
   a) Draw a resource allocation graph for the program. *(2p)*
   b) For which number of T1 and T2 thread instances is there a risk for deadlock?
   Motivate your answer using the graph! *(1p)*

T1	**T2**
c.take();	b.take();
a.take();	d.take();
useAC();	c.take();
c.give();	useBCD();
b.take();	b.give();
useAB();	d.give();
b.give();	c.give();
a.give();	
d.take();	
a.take();	
useAD();	
d.give();	
c.take();	
useAC();	
c.give();	
a.give();	

3. **Priority inversion and push-through blocking**
   Explain *priority inversion* and *push-through blocking* as well as how they are related. *(2p)*

4. **Schedulability**
The graph below describes a realtime system with rate monotonic scheduling (RMS) and with the basic priority inheritance protocol being in use. It consists of five periodic threads (T1 - T5) which are calling the monitors M1-M5 and there are no nested calls to the methods in the monitors.
The table below shows the worst case execution times (C) and period (T) for each thread.



Thread	C [ms]	T [ms]
T1	2	10
T2	4	20
T3	1	30
T4	2	40
T5	3	50

a) Which of the threads may experience push-through blocking? *(2p)*

b) Calculate the blocking factor for each one of the threads. *(2p)*

c) What is the worst-case response time for each of the threads? *(2p)*

d) Is the system schedulable? Motivate your answer! *(1p)*

Hint: $R_i = C_i + B_i + \sum\limits_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$

5. **Waiting, conditions, priorities**
Real-time Roger has the task of implementing a thread, P, in a real-time system and has concluded that the thread must wait for a new condition. However, the condition, B, can be evaluated without the need for a new critical section. Real-time Roger has suggested three possible solutions to implementing the wait for the condition to be true:

a) while (!B) ;

b) while (!B) yield(); (yield() causes immediate rescheduling)

c) while (!B) sleep(100);

The condition B can be affected (set to be true) by another thread, X. The relative priorities between P and X can be as follows:

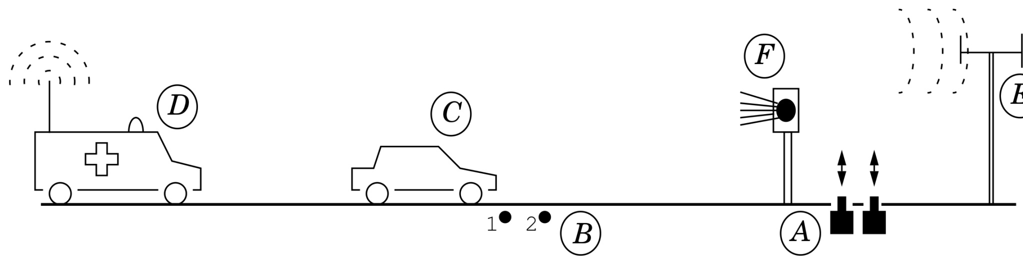
1) Thread X has lower priority than P.

2) Thread X has higher priority than P.

We assume that strict priority-based scheduling is employed. For each of the two cases (1, 2) Real-time Roger must consider what effect the three suggested implementation solutions (a-c) would have. Help Real-time Roger by describing what the effect would be in the 6 resulting cases, both regarding efficiency and functionality. *(4p)*

6. **Programming – Intelligent Speed Bump**
A new intelligent speed bump is being developed consisting of two steel beams which have been dug into the ground across the road, marked with A in the picture below. Using hydraulic motors, these beams can be elevated above the road surface - thus creating a speed bump. A certain distance before the beams, two cables are placed in the road (marked with B in the picture) which form two magnetic sensors that detect when a car (C in the picture) passes. By measuring the time it takes the car to pass between sensor 1 and sensor 2 we can calculate the speed of the car. If the car is speeding, the speed bump should be elevated.

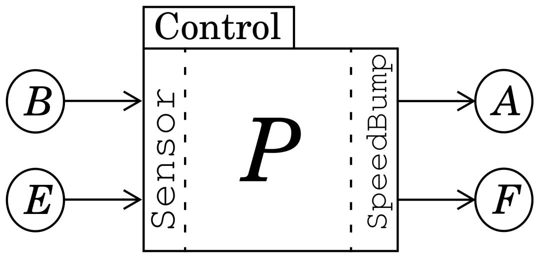
Ambulances (D) and other rescue vehicles can be equipped with a radio transmitter so that the bump system can be prevented from being activated. Consequently, the system is equipped with a radio receiver (E). There is also a signal light (F) which has the double function of warning speeding drivers of the speed bump being activated (fast blinking light) and to verify to drivers of rescue vehicles that the speed bump is inactivated (steady light).



Your task
Given a hardware interface for the speed bump including light and the sensors including the receiver (java classes *SpeedBump* and *Sensors* in the code skeleton on page A), you are supposed to develop the core components of the control software for the intelligent speed bump system according to the following specifications:

1. We assume that the speed bump will be located on a 30 km/h road outside a school and that it is to be activated if a car is driven at 35 km/h. The two magnetic sensors are located two meters apart, which means that a car is speeding if it activates the two sensors within an 0.21 seconds interval.
2. When a speeding car is detected, the speed bump shall start to raise immediately and stay elevated for 10 seconds from the time it started to rise. It should thereafter be lowered again. If a new speeder is detected while the bump is elevated, the bump shall stay elevated until 10 seconds have passed from the time the second speeder was detected, etc.
3. Separate hydraulic motors are used to raise and lower the speed bump. These two motors must never be running at the same time since that would cause an overload. Furthermore, the motors must be switched off explicitly as soon as the steel beams reach their upper or lower end positions. When the bump has reached one of its end positions, the corresponding motor must not be activated again.
4. When a rescue vehicle arrives, one signal is generated from the radio receiver unit (E). The speed bump must not be elevated during a time span of 30 seconds from this event (speeders should be ignored during this time). If the speed bump is elevated, or is being elevated, when the signal from the rescue vehicle arrives, it should immediately be lowered again. If a new rescue vehicle is detected while the speed bump is inactivated / lowered due to rescue vehicle, the down-time for the bump shall be extended with another 30 seconds from the time of detection. During the time the speed bump is inactivated (down) for a rescue vehicle, the signal light shall indicate this with a steady light.
5. The signal light shall start blinking when the bump starts raising and continue to do so until the bump has reached the low end position again.

Assume that the signals from the different parts of the system are connected directly to one central control component as shown in the figure below.



Implement / extend the central monitor class *SpeedBumpLogic* (corresponding to P in the figure), whose methods are called by the respective threads.

a) Implement the method *detectSpeeder*, called by two *CarHandler* threads, which evaluates the time passing between calls regarding reads from sensors 1 and 2 respectively according to the specification above and in case this is necessary (speeder detected and no rescue vehicle incoming) sets the timeout for speeders and triggers bumper (raising) and light (blinking). *(2p)*

b) Implement the method *detectRescueVehicle*, called by the *RescueVehicleHandler* thread, which handles the detection of rescue vehicles, i.e., sets the timeout for rescue vehicles and triggers continuous light as well as the bump to lower. *(2p)*

c) Implement the method *endpointReached*, called by the *BumpStopper* thread, that stops the motors and sets the internal state after the sensor signals reaching the end point of a motion. *(2p)*

d) Implement the method *handleTimeout*, called by the *TimeoutHandler* thread, that blocks until either a speeder or a rescue vehicle situation occurs and handles the respective timeout (which can be prolonged according to specifications). After a timeout, the bumps should be lowered if necessary (the timeout was set for a speeder) and the states and flags reset. Implement also the calling thread's run-method. *(3p)*

You may extend the attribute list of the monitor class and also change the parameters for the monitor methods, if you feel this is necessary. Please explain your thoughts in such a case. It might be useful to handle starting the motors for raising and lowering the bumps in respective private methods to guarantee correct handling (see specifications above).

7. **Design - Speed Bump Light with Controllable Frequency**
For the system you implemented in the previous task the assumption was that the hardware control interface (through the method setLightBlinking) would take care of handling the frequency the lamp would blink in, i.e., you did not have to control that yourself. Assume now, that the hardware interface only supports setting the lamp on or off through respective method calls that are forwarded directly to the respective power switch. This means, that you need to add respective functionality to control different light modes (off, steady, flashing). Explain the additions you would need in the control software, e.g., new software components (threads, monitors, etc) and how you would integrate them. Illustrate your suggestions with a respective diagram. (3p)

End of tasks, course specific Java reference and code skeleton follow

A Code skeleton for task 6

The monitor and thread classes which you should extend

```
/**
 * The SpeedBumpLogic monitor is the central logic of the system
 */
public class SpeedBumpLogic {
    // Suitable state values for the bump
    public static final int DOWN = 0;
    public static final int RAISING = 1;
    public static final int UP = 2;
    public static final int LOWERING = 3;

    // The speed bump
    private HWControlInterface hw;

    // the time to keep the bump up (speeder) or down (rescue vehicle)
    private long timeout;

    // speeder or rescue vehicle has been detected
    private boolean speederDetected;
    private boolean rescueVDetected;

    // Add suitable attributes here...


    /* constructor... (if you want to add some) */

    /**
     * Called when a car has been detected at sensor sensorNbr at time time.
     */
    public synchronized void detectSpeeder( int sensorNbr, long time) {
        // Your task!
    }

    /**
     * Called when a rescue vehicle has been detected at time time.
     */
    public synchronized void rescueVehicleDetected( long time) {
        // Your task!
    }

    /**
     * Stop the motors when the bump has reached an end point (up == true -> bump is up).
     * Set state accordingly.
     */
    public synchronized void endpointReached( boolean up) {
        // Your task!
    }

    /**
     * Wait for a timeout to be set and handle it. Lower
     * bumps after the timeout occurs (if they where UP) and reset state and
     * flags
     */
    public synchronized void handleTimeout() {
        // Your task!
    }
}
```

```
/**
 * TimeoutThread repeatedly waits in SpeedBumpLogic for a timeout
 * to occur and handles it in there.
 */
public class TimeoutHandler extends Thread {
    private SpeedBumpLogic logic;
    public TimeoutThread(SpeedBumpLogic logic) {/*...*/}

    public void run() {
        // Your task!
    }
}
```

The hardware interface for the bumps and light for you to use

```
public interface HWControlInterface {

    /**
     * Starts the hydraulic motor which causes the speed bump to raise.
     * The method is non-blocking.
     */
    public void startRaisingMotor();

    /**
     * Starts the hydraulic motor for lowering the bump.
     * The method is non-blocking.
     */
    public void startLoweringMotor();

    /**
     * Stops all hydraulic motors - both for raising and lowering the speed bump.
     * Calling the method with all motors already stopped has no effect. The
     * method is non-blocking.
     */
    public void stopAllMotors();

    /**
     * Set the lamp of the warning light to blinking light
     *
     */
    public void setLightBlinking();

    /**
     * Set the lamp of the warning light to continuous light
     */
    public void setLightOn();

    /**
     * Set the lamp of the warning light to "off"
     */
    public void setLightOff();
}
```

Other parts of the system (you may change things, but you should not need to)
The interface for the sensors

```
public interface HWSensorInterface {
    /**
     * Blocks the calling thread until a car triggers the magnetic sensor
     * indicated by the parameter.
     * @param sensor The number (1 or 2) of the sensor to wait for.
     */
    public void waitForCar(int sensor);

    /**
     * Blocks the calling thread until the arrival of an ambulance is detected.
     */
    public void waitForAmbulance();

    /**
     * Blocks the calling thread until the speed bump leaves and then again
     * reaches one of its extreme positions -
     * either fully raised or completely lowered.
     */
    public boolean waitForReachingEndPoint();
}
```

Other threads

```
/**
 * CarHandlers inform the SpeedBumpLogic when car sensors trigger.
 */
public class CarHandler extends Thread {
    private HWSensorInterface sensors;
    private int sensorNbr;
    private SpeedBumpLogic logic;
    public CarHandler(HWSensorInterface sensors, int sensorNbr,
                      SpeedBumpLogic logic) {/*...*/}

    public void run() {
        while( !isInterrupted()) {
            sensors.waitForCar( sensorNbr);
            logic.detectSpeeder( sensorNbr, System.currentTimeMillis());
        }
    }
}

/**
 * AmbulanceThread informs the SpeedBumpLogic when the ambulance sensor
 * triggers.
 */
public class RescueVehicleHandler extends Thread {
    private HWSensorInterface sensors;
    private SpeedBumpLogic logic;
    public RescueVehicleHandler(HWSensorInterface sensors,
                                SpeedBumpLogic logic) {/*...*/}

    public void run() {
        while (!isInterrupted()) {
            sensors.waitForAmbulance();
            logic.rescueVehicleDetected(System.currentTimeMillis());
        }
    }
}
```

```
/**
 * BumpController waits for the bump to move and reach an end point,
 * stops the motors, and starts over again.
 */
public class BumpStopper extends Thread {
    private HWSensorInterface sensors;
    private SpeedBumpLogic logic;
    public BumpStopper(HWSensorInterface sensors, SpeedBumpLogic logic) {/*...*/}

    public void run() {
        boolean up = false;
        while (!isInterrupted()) {
            up = sensors.waitForReachingEndPoint();
            logic.endpointReached( up);
        }
    }
}
```

B Short specific Java reference with potentially helpful methods

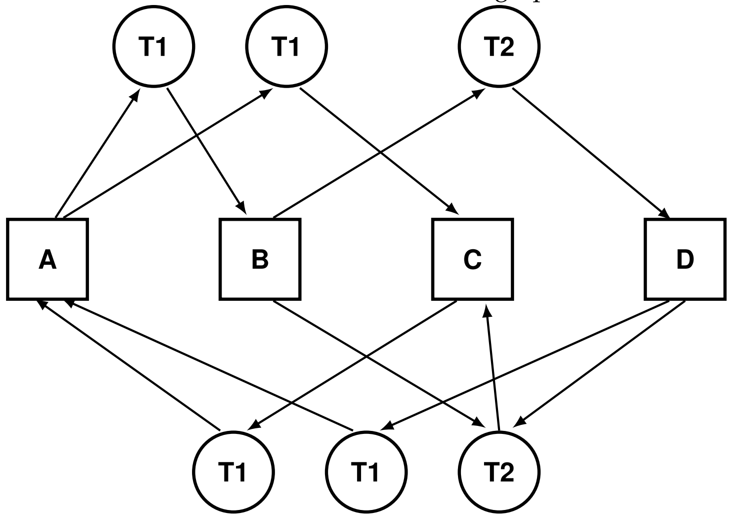
Timing, threads, waiting, math, etc

- class Math and class System:
  - public static double Math.random(): generates a random value between 0 and 1.
  - public static long System.currentTimeMillis(): current system time in milliseconds.
- klass Thread:
  - public static void sleep(long t): sets a (the current) thread to sleep for t milliseconds.
  - public static void yield(): withdraw running thread and put in queue for rescheduling.
  - public void join(): wait for the thread to die, i.e., terminate execution.
  - public void start(): start the thread.
  - public void run(): the actual work description for the thread.
- class Object:
  - public final void wait(): sets the calling thread in wait state until notified.
  - public final void wait(long t): sets the calling thread in wait state for t milliseconds. Can be woken up early!
  - public final void notify(): notify the thread first in queue of a change in respective monitor.
  - public final void notifyAll(): notify all waiting threads of a change in the respective monitor.

Kortfattad lösning

1. a) Thread pool
   b) Multiple signals in Java monitor

2. There are multiple possible loops, and thus risks of deadlocks, all of which require two instances of T1. So long as there is only a single instance of T1 there is no risk for deadlock, regardless of the number of T2 instances. Resource allocation graph:



3. Priority inversion happens when a high-priority thread is delayed by a lower-priority thread for a arbitrary amount of time. Example: P1-3 are threads. P1 has high priority, P2 has medium priority, and P3 has low priority. S is a mutual exclusion semaphore.

   1. P3 starts execution
   2. P3 does take on S and continues execution
   3. P1 starts execution, interrupting P3
   4. P1 does take in S. Since P3 is already holding S P1 is blocked and P3 continues execution
   5. P2 starts execution interrupting P3
   6. P1 must now wait for the lower-priority process P2 (and maybe even several medium priority threads) to finish execution.

   A solution to the priority inversion problem is priority inheritance. This means that while a higher-priority thread waits for a lower-priority thread to release a resource the priority of the lower-priority thread is temporarily raised until the resource is released. In the example above this would mean that P3 is not interrupted by P2. However, if priority inheritance is implemented, there can be push-through (indirect) blocking for P2, that gets blocked by P3 holding the resource P1 (but not P2 itself) requires and thus running with P1's (inherited) priority. However, in this case the timing is not a problem, as the blocking times in the resource can be considered in the respective analysis.

4. a) Blocking factors:
   $B_{T5} = 0$ Since there is no thread with lower priority than T5.
   $B_{T4} = max(1, 2) = 2$ (direct, indirect)
   $B_{T3} = 4 + 2 = 6$ (direct + indirect)
   $B_{T2} = (6 + 2) = 8$ (direct + indirect)
   $B_{T1} = (1 + 2) = 3$ (direct + direct)

   b) Thread T2, T3 and T4 may experience push through blocking (indirect blocking).

c) Calculate response times:
Observe: there was an issue with the execution times being lower than the times spent in respective monitors, i.e., C should have been much higher for each thread. We accepted both ways of calculating the response time, but give here the simpler one for the numbers in the table.
$R^0_{T1} = R^1_{T1} = 2 + 3 = 5$

$R^0_{T2} = 4 + 8 = 12$
$R^1_{T2} = 4 + 8 + \lceil 12/10 \rceil \cdot 2 = 16$
$R^2_{T2} = 4 + 8 + \lceil 16/10 \rceil \cdot 2 = 16$

$R^0_{T3} = 1 + 6 = 7$
$R^1_{T3} = 1 + 6 + \lceil 7/10 \rceil \cdot 2 + \lceil 7/20 \rceil \cdot 4 = 13$
$R^2_{T3} = 1 + 6 + \lceil 13/10 \rceil \cdot 2 + \lceil 13/20 \rceil \cdot 4 = 15$
$R^3_{T3} = 1 + 6 + \lceil 15/10 \rceil \cdot 2 + \lceil 15/20 \rceil \cdot 4 = 15$

$R^0_{T4} = 2 + 2 = 4$
$R^1_{T4} = 2 + 2 + \lceil 4/10 \rceil \cdot 2 + \lceil 4/20 \rceil \cdot 4 + \lceil 4/30 \rceil \cdot 1 = 11$
$R^2_{T4} = 2 + 2 + \lceil 11/10 \rceil \cdot 2 + \lceil 11/20 \rceil \cdot 4 + \lceil 11/30 \rceil \cdot 1 = 13$
$R^3_{T4} = 2 + 2 + \lceil 13/10 \rceil \cdot 2 + \lceil 13/20 \rceil \cdot 4 + \lceil 13/30 \rceil \cdot 1 = 13$

$R^0_{T5} = 3$
$R^1_{T5} = 3 + \lceil 3/10 \rceil \cdot 2 + \lceil 3/20 \rceil \cdot 4 + \lceil 3/30 \rceil \cdot 1 + \lceil 3/40 \rceil \cdot 2 = 12$
$R^2_{T5} = 3 + \lceil 12/10 \rceil \cdot 2 + \lceil 12/20 \rceil \cdot 4 + \lceil 12/30 \rceil \cdot 1 + \lceil 12/40 \rceil \cdot 2 = 14$
$R^3_{T5} = 3 + \lceil 14/10 \rceil \cdot 2 + \lceil 14/20 \rceil \cdot 4 + \lceil 14/30 \rceil \cdot 1 + \lceil 14/40 \rceil \cdot 2 = 14$

d) $R_C < T_C$ for all four threads, therefore is the system schedulable.

5. 1. a) P will use most CPU-time and will therefore starve X. If B is only changed by X there is also a live-lock.
   b) Like a) but X might be allowed to run eventually (depends how yield is handled).
   c) Sleeping will allow X to run (if CPU utilization is low enough). Even if P no longer requires all CPU time this is polling which is ineffective. There can be a delay of 300 ms or longer before P detects a change in B worsening the response time. In worst case, B has changed back before P is run again, having P miss the condition.
   2. a) P will get to run eventually, and at some point hit B being true. Still, somewhat ineffective, as it is unclear when things are happening.
   b) Like a) but X is allowed to run immediately when P requires B being true.
   c) Like 1-c). It works, but there might be unnecessary delays.

6. In this solution, the speed bump logic is to be implemented as a monitor that is concurrently called from the threads that are involved (and scheduled according to priority).
One implementation of the SpeedBumpLogic monitor is as follows:

```
/**
 * The SpeedBumpLogic monitor is the central logic of the system
 */
public class SpeedBumpLogic {
    // Suitable state values for the bump
    public static final int DOWN = 0;
    public static final int RAISING = 1;
    public static final int UP = 2;
    public static final int LOWERING = 3;

    // The speed bump
    private HWControlInterface hw;
```

```
// the time to keep the bump up (speeder) or down (rescue vehicle)
private long timeout;

// speeder or rescue vehicle has been detected
private boolean speederDetected;
private boolean rescueVDetected;

// Add suitable attributes here...
private long latestSensor1;


/* constructor... (if you want to add some) */

/**
 * Called when a car has been detected at sensor sensorNbr at time time.
 */
public synchronized void detectSpeeder( int sensorNbr, long time) {
    if (sensorNbr == 1) {
        latestSensor1 = time;
    } else if (sensorNbr == 2) {
        if (time - latestSensor1 < 210 && !rescueVDetected) {
            speederDetected = true;
            timeout = time + 10000;
            startRaising();
            hw.setLightBlinking();
            notifyAll();

        }
    }

}

/**
 * Called when a rescue vehicle has been detected at time time.
 */
public synchronized void rescueVehicleDetected( long time) {
    rescueVDetected = true;
    speederDetected = false;
    timeout = time + 30000;
    startLowering();
    hw.setLightOn();
    notifyAll();

}

/**
 * Stop the motors when the bump has reached an end point (up == true -> bump is up).
 * Set state accordingly.
 */
public synchronized void endpointReached( boolean up) {
    hw.stopAllMotors();
    if (state == RAISING) {
        state = UP;
    } else if (state == LOWERING) {
        state = DOWN;
        if (!rescueVDetected) {
            hw.setLightOff();
        }
    }
    notifyAll();
    }

    /**
     * Wait for a timeout to be set and handle it. Lower
     * bumps after the timeout occurs (if they where UP) and reset state and
     * flags
     */
    public synchronized void handleTimeout() {
        try {
            // Wait until there is a timeout
            while (!speederDetected && !rescueVDetected) {
                wait();
            }
            // Wait for the timeout
            // (timeout can be moved by new speeders or ambulances)
            long diff = timeout - System.currentTimeMillis();
            while (diff > 0) {
                wait(diff);
                diff = timeout - System.currentTimeMillis();
            }
            // Handle timeout
            if (speederDetected) {
                speederDetected = false;
                startLowering();
            } else if (rescueVDetected) {
                rescueVDetected = false;
                hw.setLightOff();
            }
        } catch (InterruptedException e) {
            throw new RTInterrupted(e);
        }
        notifyAll();
    }


    /**
     * Start raising the bump, if it is not already up.
     */
    public synchronized void startRaising() {
        if (state == DOWN || state == LOWERING) {
            hw.stopAllMotors();
            hw.startRaisingMotor();
            state = RAISING;
            notifyAll();
        }
    }


    /**
     * Start lowering the bump, if it is not already down.
     */
    public synchronized void startLowering() {
        if (state == UP || state == RAISING) {
            hw.stopAllMotors();
            hw.startLoweringMotorl();
            state = LOWERING;
            notifyAll();
        }
    }

}
```

The timeout handling thread:

```
/**
 * TimeoutThread repeatedly waits in SpeedBumpLogic for a timeout
 * to occur and handles it in there.
 */
public class TimeoutHandler extends Thread {
    private SpeedBumpLogic logic;
    public TimeoutThread(SpeedBumpLogic logic) {/*...*/}

    public void run() {
        while( !isInterrupted()) {
            logic.handleTimeout();
        }

    }
}
```

7. Add a LightController thread and maybe an extra monitor for the light to avoid too many threads locking the important motor control methods too often. The light monitor needs a three state variable (ON / OFF / BLINK) and a method to set this state. The state is set through the SpeedLogic monitor when necessary. A method for blinking blocks until the state is BLINK"(called by the looping LightController) and then sets the lights on / off according to some required frequency (can be handled in the thread as well. Lights can be switched on / off directly in the method for setting the mode, if not blinking is required.