Exam
Concurrent and Real-Time Programming

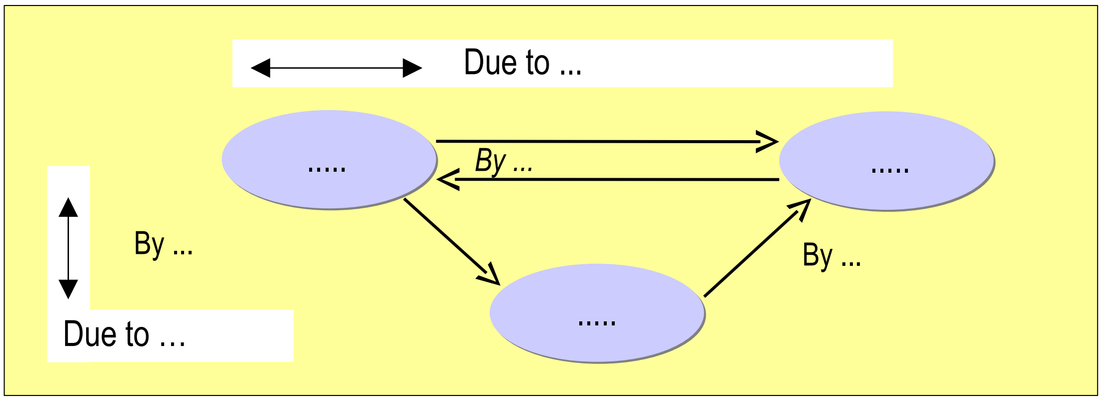
2012–10–27, 14.00–19.00

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

To pass the exam (grade 3), you have to solve most of the first 5 problems (which are more of a theoretical type), and you have to develop an acceptable solution to problem 6 (programming). At least a partial solution to problem 7 is required for the highest grade (5).

---

1. When threads are scheduled they can be in different execution states. Since the notion of state is about the data capturing the history, in this case of the thread, the execution state of a thread could (if extended with details) include locked resources, execution time, etc., and it may depend on the operating system. As a simplified and generic model that applies to threads in general, and the thread model in Java in particular, we have the following figure in the course:



   a) What are names of the three execution states; what do they mean? *(1p)*

   b) What are the transitions between states? Who or what part of the system is typically responsible for triggering/causing each transition? *(1p)*

   c) A so called *context switch* takes place during the transitioning between the execution states. What is happening during that switching? *(1p)*

   d) A so called natively pre-emptive scheduler has the advantage that a higher priority thread that gets ready can preempt a less important running thread, and that pre-emption works also in native (non-Java) code. A disadvantage is that some of the context switches get more expensive, since all CPU registers need to be stored when the switch is not issued by a call in the program. That is, upon a method call some CPU registers are scratch registers and do not need to be stored. Based on this information and your understanding of concurrent execution, between which execution states (in the figure above) are the context switching more expensive, and why? *(1p)*

2. A Java program consists of three monitors and three threads. The program is running on a real-time operating system, using the basic inheritance protocol, and the threads having priorities assigned according to RMS. The software runs without interference with any other programs. The time in ms after each method is the WCET of that method.

```
public class M1 {
    public synchronized void a1() { /* 20 ms */ }
    public synchronized void b1() { /* 30 ms */ }
}

public class M2 {
    public synchronized void b2() { /* 10 ms */ }
    public synchronized void c2() { /* 40 ms */ }
}

public class M3 {
    public synchronized void a3() { /* 60 ms */ }
    public synchronized void c3() { /* 120 ms */ }
}

public class A extends PeriodicThread {
    private M1 m1;
    private M3 m3;
    public void perform() /* 400ms period */ {
        m1.a1();
        // ... 15ms
        m3.a3();
    }
}

public class B extends PeriodicThread {
    private M1 m1;
    private M2 m2;
    public void perform() /* 600ms period */ {
        m1.b1();
        m2.b2();
        // ... 45ms
    }
}

public class C extends PeriodicThread {
    private M2 m2;
    private M3 m3;
    public void perform() /* 700ms period */ {
        m3.c3();
        // ... 35ms
        m2.c2();
    }
}
```

As known from the course, the response times including blocking time should fulfill

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

a) Show that the system is schedulable. *(4p)*

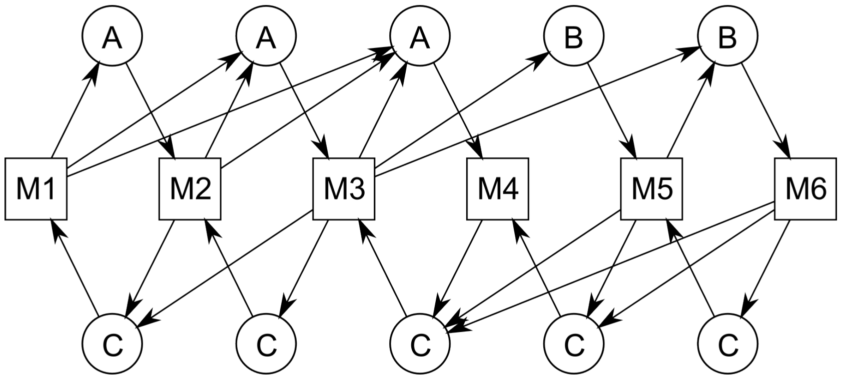
b) The system needs to be extended with a lowest priority data-logging thread. The thread needs an additional method in each monitor to read the state. The execution times of these methods are, however, very low, much below 1ms and within the margins of the stated WCETs; hence the blocking time due to this thread can be ignored. The estimated worst-case execution time in total for thread D is 100ms since a file system is involved. Determine (and show how you determine) the shortest possible period time for D? *(2p)*

3. For a programming language to be safe, i.e. fully type safe such as Java without usage native methods, memory deallocation needs to be automated. That activity is referred to as garbage collection (GC). To have GC working in a real-time system with priority-based preemptive scheduling, the GC algorithm must be implemented for performing incremental GC, but what about the scheduling of the GC work?

   Consider high priority (real-time) threads and low priority (concurrent) threads; how is the GC work to be scheduled such that response time for the real-time threads can be ensured? *(2p)*

4. What is priority inversion and how may it occur? *(2p)*

5. The system behind the following resource allocation graph is not deadlock-free, and the programmer allocated unnecessarily many rosources (just in case, and hence the graph is overly complex).



To fix the system, you look into the code, where use of each resource is coded as useResourses with the resources being monitors M1, M2, etc. Specifically:

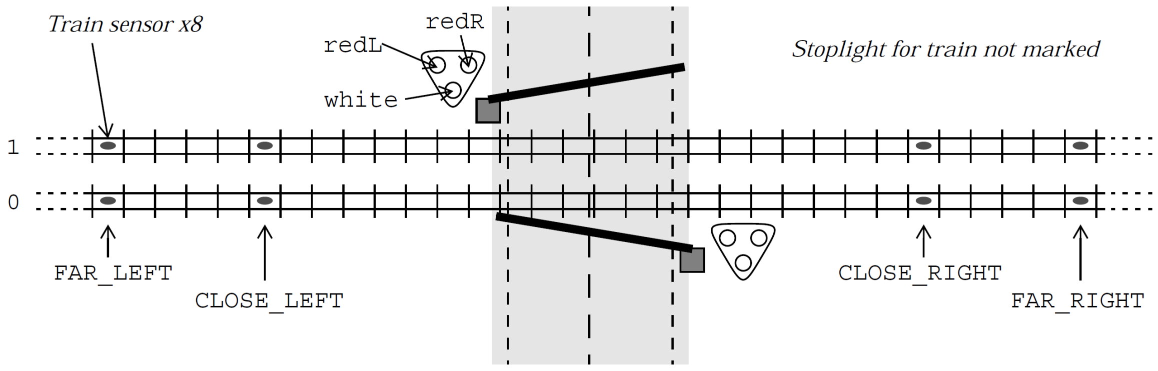
a) Thread A have critical sections *useM1M2* (needing resources M1 and M2) followed by *useM3M4*. B has critical section *useM3M5M6*. C have critical sections *useM1M2M3* and *useM3M4M6*, independent of each other. Redesign the system to be deadlock-free. Assume resources M1-6 are each protected by a mutex semaphore. Assume the critical sections are available as method calls in respective thread. Show the code for each thread. *(2p)*

b) Show that your new system is deadlock-free. *(2p)*

6. Simulated railroad crossing

Background: Some decision-makers in your county are quite annoyed by having to wait too long for trains at some crossings. The gate control might be made for high-speed trains, and then when long cargo trains pass the gates go down rather early and stay down so the queue of cars cannot pass before the next train is approaching. Given train schedules, the control of the gates of the crossing, and statistics for the car traffic, there is a request to develop a simulation of the crossing, which will later be used for enhancing the gate control or for motivating a budget for a tunnel/bridge under/over the tracks.

Control and monitoring of railroad traffic and railroad crossings are classical examples within concurrent and real-time software (including an old exam from Aug 2000). Here, instead of a control situation we have a simplified model of a railroad crossing, which we want to simulate to explore the average waiting time for cars crossing the tracks. The trains and their schedule, as well as the control of the gates, are provided. The following figure introduces some of the notions that are used internally by the gate and light control (also valid for the final problem of this exam):



We are interested in the passage of cars, and there are no sensors for cars. Therefore the cars are to be represented by threads that drive the simulation of them, somehow waiting the time they wait for the gates to get up. The cars threads therefore have to register their arrival as further explained below.

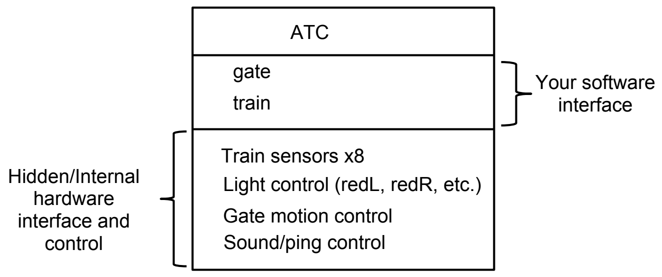
There is an available control system with train sensors and IO signals according to the figure above. This is part of the Automatic Train Control (ATC) that you might know from many countries (in Sweden from 1980). Now, we keep the ATC and just connect our car simulation system to the simulated ATC system. For your car simulation package we then have a simplified interface to the ATC system:

```
class ATC {
        enum GateState {IS_DOWN, IS_UP, GOING_DOWN, GOING_UP}
        GateState gate(GateState was){ // Block while state is as it 'was'
                // obtaining and returning changed state...
        }
        enum TrainState {NONE, APPROACH, DEPART, NEAR}
        TrainState train(TrainState was){
                // obtaining and returning changed state...
        }
}
```

where we only need some of the state constants. For example, cars may pass only when the `GateState` is `IS_UP`, and the other states are (from this point of view) equivalent.

However, here the purpose of the simulation is to be able to experiment with the timing of the system, so we also wan to capture the case that a car might be trapped on the tracks with `GateState IS_DOWN` or `TrainState NEAR`.

The following figure also shows the simple interface for the following, while the internals of the control in not part of your problem:



As can be understood from the ATC interface, we do not care about what track and in what directions trains are coming; we only consider the simplified states. Furthermore, since the two directions of the car traffic are independent of each other, we only consider one direction (the one with the most traffic).

In general, when simulating a concurrent system there are two choices to make:

- Should the simulated time be a scaled real time or should there be a separate simulated time that is managed completely independent of the real-time clock? In the following we base the simulation on scaled real time, with the virtual simulation time going some factor faster than real time, as suitable for an online visualization of an ongoing simulation. However, this is not a critical point for this problem (and you already did time scaling in lab 3), so we assume scaling 1.0 and you simply develop a real-time simulation.
- Should each new car and each new train be represented by a new thread instance each (as for lab3 washing programs), or should a fixed pool of threads be (re)used for the re-occurring activities (as the persons in lab2)? In the following we assume that a pool of threads represent (the maximum) number of cars with one thread per car.

Assignment: Your task is to develop the simulation of the crossing, with emphasis on the waiting times for the cars. For full score the cars should queue up in order of arrival, and they need about 2 seconds to advance one place in the queue (after the car in front of the queue starts to cross the tracks). The detailed timing on millisecond level is not important (the models are not that accurate anyway); you can select an overall meaningful implementation that captures the fundamentals of the simulation. You are free to add or change attributes, but basically the following is the monitor that needs to be implemented:

```
class Crossing {

    ATC.GateState gateState;    // The state of the gates
    ATC.TrainState trainState;  // The state fo the trains

    int carsOnTrack;            // n.o. cars crossing, usually 0 or 1.
    int n;                      // max n.o. cars in the system.
    Car[] q;                    // an array for cars waiting.
    long sumWaiting;            // sum of the waiting times.
    long maxWaiting;            // longest waiting time.
```

```
int numCars;                // how many cars did wait (for the average).

boolean trainReport;        // flag for cars on track when train is NEAR.
boolean gateReport;         // flag for cars on track when gates are down.

/**
 * Constructor, assuming no trains, no cars, and gates up to start with.
 *
 * @param nCars the maximum number of cars that might queue up.
 */
public Crossing(int nCars) {
        n = nCars;
        q = new Car[n];
        gateState = ATC.GateState.IS_UP;
        trainState = ATC.TrainState.NONE;
        // .. whatever more you need..
}

/**
 * Update the state of the gate, and check that there are no cars on the
 * tracks when gates are down.
 *
 * @param gateState is the new state to be assigned to the corresponding
 *           attribute.
 */
synchronized void setGate(ATC.GateState gateState) {
    // .. to be developed ..
}

/**
 * Update the state of train locations, and check that there are no cars on
 * the tracks when any train is NEAR.
 *
 * @param trainState is the new state to be assigned to the corresponding
 *           attribute.
 */
synchronized void setTrain(ATC.TrainState trainState) {
    // .. to be developed ..
}

/**
 * Method to be called by the Car threads to simulate a car crossing the
 * railroad. The waiting time is to be measured as the sum of the queing
 * time, which is the time from arrival (first in this method) until the car
 * has no other cars in front of it at the crossing and the gate being open.
 *
 * In case of a queue, cars will queue in the q array, which has the same
 * size as the number of cars in total. After being blocked in the queue, a
 * 2s waiting is added for each position. That is, a car arriving as the
```

```
     * third car waiting will need 2+2s to advance to the first position in the
     * queue. Also passing the tracks takes 2s but is not included in the
     * waiting time, but during the crossing the carsOnTrack is incremented and
     * checked in the setTrain a setGate methods so there are no carsOnTrack when
     * gates are down.
     */
    synchronized void carPassing() {
        // .. to be developed ..
    }

    /**
     * Block caller until results should be displayed, which is either every one
     * hour when the average waiting time per car and the maximum waiting time
     * is returned as a string. In case of cars still being on the tracks when a
     * train is near, or when the gates are down, that is reported immediately.
     *
     * @return a reporting string. (You may also just use System.out.print.)
     */
    synchronized String report() {
        // .. to be developed ..
    }
}
```

While the implementation of the monitor is the major part of the development, there are also the three (very simple) thread types needed, as clarified by the following main program that instantiates and connects the a `Crossing` monitor with the ATC environment.

```
class Main {
    public static void main(String[] args) {
        int n = 10;
        if (args.length>0) n = Integer.valueOf(args[0]);

        ATC atc = new ATC();

        Crossing x = new Crossing(n);

        Reporter rep = new Reporter(x);
        rep.start();

        Gate gate = new Gate(x, atc);
        gate.start();

        Train train = new Train(x, atc);
        train.start();

        for (int i=0; i<n; i++) {
            Car car = new Car(x);
            car.start();
        }
    }
}
```

The Car threads (available but not shown) simply call `carPassing` at times according to the statistical distribution, and when that method returns the car thread instance is available for next trevel. Hint: Inside `carPassing` the `Car` thread object can be obtained via a suitable static method of the `Thread` class.

a) Develop (the very short run methods of) the threads `Reporter`, `Gate` and `Train`. *(3p)*

b) Develop the monitor methods that are to be called by the `Reporter`, `Train` and `Gate` threads. *(3p)*

c) Develop the `carPassing` monitor method, and explain your design choices. Efficiency (e.g. by frequent `notifyAll`) is not an issue since the number of cars are small considering to the power of modern computers. *(6p)*

7. Distributed railroad crossing control

Following successful simulation, the task is now to design (but not implementing, but code can be used for clarifications) the real-time system for a real crossing. The crossing consists of a number of computational nodes:

- One node is attached to each sensor located on the tracks to detect incoming and passing trains. The sensor hardware needs to be polled.
- A node is attached to a stop signal for the train, changing from red to green when the crossing is ready for the train to pass. The default signal is red.
- One node controls the gate and is attached to two sensors indicating if the gate has reach upper or lower position. The motor control and the sensor hardware are non-blocking.
- A node controls the crossing lights, where it is suitable to use a thread for managing the blinking lights for the cars. The light should blink with a frequency of approximately 1 Hz. While the gate is about to (3s) starting to close, when it is closed, and until the gate is fully raised, the left and right red light should be blinking alternatively. When the gate is open and free for passage, the lower white light should be blinking. Show how a `Light` thread can be incorporated into your design.
- Explain any necessary revisions of methods and attributes of the `Crossing` monitor.

The nodes are controlled through a PLC (programmable logic controller), essentially a supervisor node, containing the logics for safe control of a crossing. All nodes communicate through a network. Special about network calls is that reading from a node is blocking, but writing is non-blocking.

Determine activities (threads) and responsibilities for each thread within each node. Make figures indicating threads as well as involved passive objects. List the responsibilities of each thread. *(6p)*