Exam
Concurrent and Real-Time Programming

2009–12–16, 08.00–13.00

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.

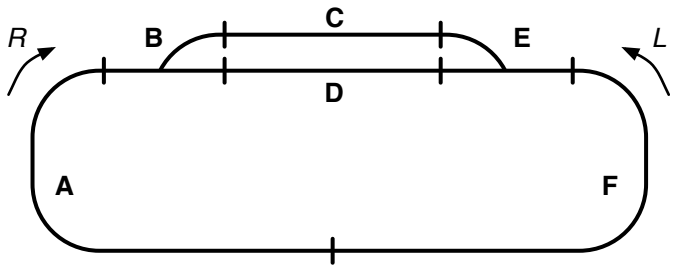
To pass the exam, grade 3, you have to solve most of the first 5 problems (which are more of a theoretical type), and you have to develop an acceptable solution to problem 6 (programming). At least a partial solution to problem 7 is required for grade 5.

---

1. To handle concurrent inputs and real-time computing of the resulting outputs, you have constructed a set of threads. Your real-time threads should run on an embedded computer that also runs other threads without real-time requirements (lower priority). Which of the following embedded computer/OS requirements should be fulfilled, and why is that needed?
   a) Processing pre-emption (pre-emptive scheduling of CPU time).
   b) Resource pre-emption.
   c) Strict priorities.
   d) Priority inheritance.

   (4p)

2. Little Lisa wishes to get model trains and tracks from Santa this Christmas, and in particular the new computerized model with configurable trains and mutual exclusion for tracks looks like a great option (at least here parents think so, but actually Lisa wants to crash trains into each other but that is another story). The starter-kit tracks look like the following, with each mutually exclusive section (shared resource) marked A to F. For a train to go into a new section, it must first (while holding the present one) wait for it to be free. Trains only go forward.



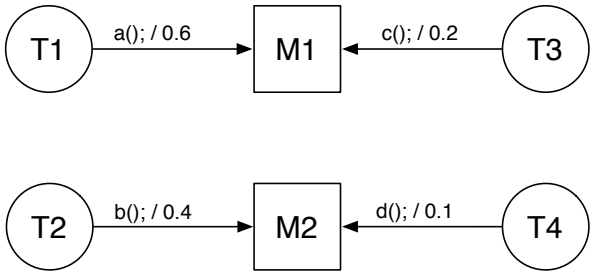
   a) Assume that trains go in the same direction (e.g. all clock-wise), and that each train is configured to always take either the left or the right track at the switches. (That is, each train is configured to always go on track C or D, and that is communicated with the switch control. The result would be the same if the selection could vary, but we simplify to fit our methods better.) Explain by referring to a resource allocation graph that more than four trains on the track will lead to a deadlock.
   b) Assume there are only two trains that go in opposite directions, one configured to always take the upper C branch and one that always takes the lower D branch. Update your resource allocation graph, and tell how many situations there are that result in deadlock with only these two trains.

---

c) Suggest an alternative allocation strategy with some merged resources (into a new resource X) such that two trains in opposite direction can run without risk for deadlock (and still without risking a collision). Supply the (very simple) resource allocation graph.

(3p)

3. A real-time system (with dynamic priority-based scheduling and priority inheritance) consists of four threads (T1, T2, T3, and T4) which communicate with each other by calling the monitor operations `a`, `b`, `c`, and `d` in the monitors M1 and M2 with maximum execution times (in milliseconds) according to the figure below. The threads call one monitor operation at a time (no nested calls) T1 has the highest priority, T2 the second highest, T3 the second lowest, and T4 has the lowest priority.



State for each of the threads (T1, T2, T3, and T4) the maximum time the thread can be blocked by lower priority threads during one invocation.

(3p)

4. A real-time system (with dynamic priority-based scheduling and priority inheritance) consists of four threads (A, B, C, and D) which communicate with each other through a system of monitors. The system is scheduled using rate monotonic scheduling. The threads have the following characteristics (C = worst-case execution time, B = worst-case blocking time, T = period).

Thread	C (ms)	B (ms)	T (ms)
A	2	1	10
B	1	0.5	4
C	2	0	20
D	1	1	6

The deadline (D) is equal to the period (T) for all threads.

The worst-case response time can be computed according to

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

Calculate *the worst-case response time* for each of the four threads, and determine for each thread if it meets it's deadline.

(4p)

5. a) Why should you not have thread objects with synchronized methods?
   b) Why should all public (or package-visible) monitor methods be synchronized?
   c) Why is it probably a good idea to declare sub-class methods synchronized whenever the base-class implementation was synchronized?
   d) Try to describe a risk with introducing synchronized sub-class methods that uses wait, for the case that the synchronised base-class method does not use wait? In other words, why could it be a good idea to declare a synchronized base-class method as final, if the logic of the monitor assumes there are no other threads manipulating monitor data when that method is used?

(4p)

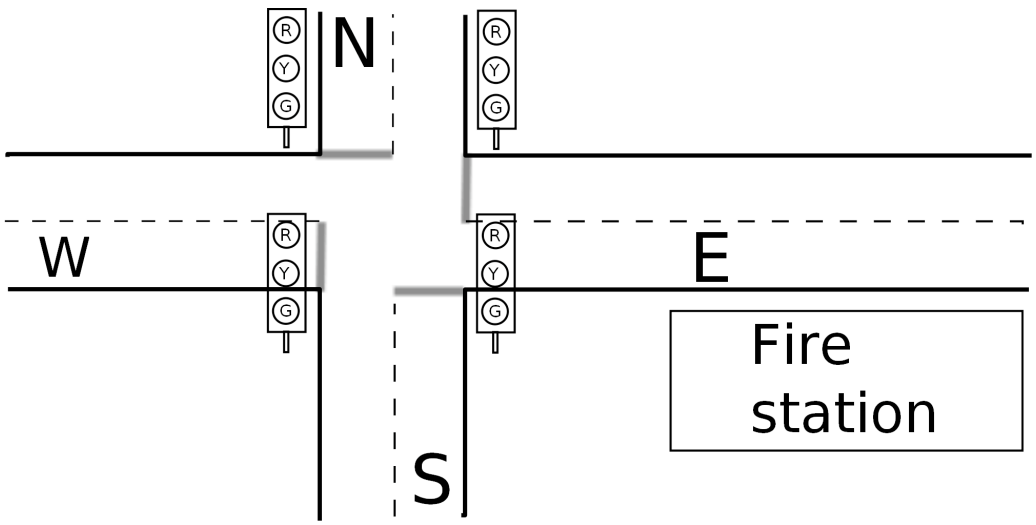
6. Synchronized Traffic Light Control

It is standard to have sensors in the street near crossings to detect if a vehicle is approaching the crossing, and then switch to green for that direction. Such an event-driven mode is used during certain times of the day/night when there is very little traffic. During other times, normal time-driven operation is used. While your county has a well working system since long, four new overall requirements have been added recently:

- There is a need to have a maximum waiting time for all directions when event-driven operation is active. The reasons is that new vehicles with less metal (and more composite materials such as carbon fibre) sometimes get stuck since the the sensing does not react.
- Even if there is some risk for missing a few vehicles, there is a desire to use the street sensors for measuring/counting the amount of traffic. The traffic light control should therefore be able to report the vehicle counting.
- The older traffic control implementations are either in hardware (relay logic) or in software without the source code (belonged to a company that no longer exists). The authorities therefore wants to have a first iteration towards an open-source implementation.
- A new fire station next to a crossing (the one you will consider) motivates control of the traffic lights such that the lights go green when fire fighters have to drive that way.

As a first step we want a principle implementation to see what issues that need to be further investigated, so we here simplify the problem to cover only car traffic (no pedestrians and no special lights for bikers) and a standard four-street crossing. The streets are assumed to have one lane each, there is no arrows for cars going/turning in a specific direction, and the streets are going in north-south and east-west directions (referred to as N, S, E and W in the following).

The street sensor input is obtained from the class `StreetInput`. The commands/requests from outside are obtained via the class `CommandInput`. The traffic lights are controlled via the class `Light`. Obtaining the inputs are by blocking calls, whereas the output is immediate without blocking. The standard crossing is depicted here:



It is of fundamental importance that there is green light for only one direction at a time (N-S or E-W, but never both), and it should be fully clear from the implementation that such a hazardous situation can never occur. Likewise, when switching from green in one direction, the specified time for yellow and red in one direction and then the time for yellow and green in the other direction are minimum times (just like for `Thread.sleep`); it does not matter if the duration gets a bit longer (CPU load and schedulability is not your problem).

In addition to common sense knowledge for street crossings (such that red means stop) and the above safety aspect, the requirements are the following (with R=red, Y=yellow and G=green):

1. When changing from R to G, there should be Y and R for two seconds.
2. When changing from G to R, there should be Y (and only Y) for two seconds.
3. After both directions (N&S or E&W) have got R, it should pass three seconds before the transition to green (starting with R&Y) takes place for the other direction.
4. When command 'w' (warning) has been obtained, there should be a change to a 1 Hz flashing Y in all directions. When that is requested, all four Y lights should first go on (not flashing) for three seconds, while all G and R are turned off.
5. When command 'a' (alarm) is obtained, the E-W direction should (by a normal sequence) go green and stay green for 90 sec (to let some fire brigade vehicle pass) after the last (of possibly several, for more vehicles) alarm commands.
6. When command 't' is obtained, time-driven operation should start (of course at the earliest 90 sec after an alarm). In this mode, a G stays on for 70 sec (or shorter if an alarm occurs).
7. When command 'e' is obtained, event-driven operation should start. In this mode, a detected vehicle triggers a change to G, it is by default R in all directions when no vehicle is detected, and G stays on for 15 sec. An extra feature, as mentioned above, is that after 2 min of R, change to G (as if a vehicle was detected) takes place.
8. The G period during time-driven operation should be interrupted (in a few seconds, including the transition via Y) when an 'a' command is obtained. The other sequences should complete as normal when any mode change takes place.
9. Setting a new mode should block until the light-controlling thread has obtained/acknowledged it (started to perform the change).

To ensure that there is no extra/undesired buffering, a monitor-based (`synchronized`) solution is required. The available classes for the IO are as follows:

```
class CommandInput {

   /**
    * Obtain next command for traffic-light control of this crossing,
    * blocking until a new command (or another alarm) is available.
    * @param cmd the previous command, which hereby is acknowledged.
    * @return the new command as a char according to problem documentation.
    */
   static char awaitCommand(char cmd) {...};
}

class StreetInput {

   static class Value {
       int N, S, W, E;  // Counters for each direction
   }

   /**
    * Get updated vehicle counts for the four streets,
    * blocking the caller until any value differs from the old one.
    * @param old the previous value to compare with.
    * @return the updated counters.
    */
   static Value awaitVehicle(Value old) {...}
}

class Light {

   /**
    * Directly and without blocking set the traffic-light color.
```

```
     * @param NSEW street specifier, with 'N' for north etc.
     * @param R turns the Red light on if 'r' or 'R', otherwise off.
     * @param Y turns the Yellow light on if 'y' or 'Y', otherwise off.
     * @param G turns the Green light on if 'g' or 'G', otherwise off.
     */
    static void setColor(char NSEW, char R, char Y, char G) {...}
}
```

Hints:

- As you know, `Object.wait` can be called with a timeout argument.
- A safe way to change mode is to turn Y on in all directions and then 2 sec later turn R on for all directions.
- Make your code more readable by defining private methods for frequent sequences and lengthy calls, such as the following ones (that you may refer to) or your own.

```
private void NS(String ryg) { // Set RYG by string for N-S street
    Light.setColor('N', ryg.charAt(0), ryg.charAt(1), ryg.charAt(2));
    Light.setColor('S', ryg.charAt(0), ryg.charAt(1), ryg.charAt(2));
}

private void EW(String ryg) { // E.g. ryg "RY " for red-yellow on E-W
    Light.setColor('E', ryg.charAt(0), ryg.charAt(1), ryg.charAt(2));
    Light.setColor('W', ryg.charAt(0), ryg.charAt(1), ryg.charAt(2));
}
```

Problems

a) As a preparation for your implementation, please answer the following:

- What treads and what shared data is appropriate for solving the problem?
- Show how these are interconnected by a clarifying figure. What methods are the threads calling, and with what arguments?
- What is better (as learned during lab 2), a few bigger monitor methods (possibly with several calls of `wait`), or many small methods (only using a minimal set of data not to block longer than necessary)?
- Why are none of the input threads well suited for calling `Light.setColor`?

b) Implement the monitor. Comment on your choice of where you call `Light.setColor` (from within the monitor, or outside the monitor based on returned data). Keep the state-machine clean/short by calling (your own private) methods to take care of the individual modes. Generally, lengthy sequential logic (but not the concurrency and synchronization) can be replaced by comments that tell what the sequence should do. Specifically, the logic for handling the event-driven mode does not need to be implemented.

(10p)

7. Asynchronous Traffic-Light Control

The traffic light control, as formulated for the previous problem, of course includes asynchronous activities in that the threads run their code asynchronously with respect to each other. In the previous problem, however, the interaction in terms of data between these threads were synchronized, whereas in this problem you should find a design that solves the problem by the use of message passing between the threads. That is, by buffering messages/events that are passed between the threads, the data storage (the event buffers) and the thread interaction is asynchronous also from a shared-data point of view.

a) Make an alternative design for the traffic-light control such that it becomes a pure message-based solution. Specifically, you should consider the following extra requirements:

1. Do not make use of the post and fetch methods with timeout. Such timeouts depend on a system timer (thread) and certain native methods, and in this case the platform properties and customer preferences resulted in that the timeout feature should not be used. You may of course need an additional thread to handle max-time situations.
2. Use the `BufferPlain` class for simplicity. You do not need the extras of the other buffer implementations so `BufferPlain` provides the most suitable and efficient buffer. The class documentation, with the timeout methods removed according to previous item, is attached to this exam. The `doPost` and `doFetch` methods should be the normal methods to use.

Assume the same `Input` and `Light` classes for interfacing with the physical system. If your figures and descriptions are sufficiently clear, you can illustrate your solution without code, showing what threads and what buffers that interact and how. It should also be clear where and how the application state (that in the previous problem was encapsulated by the monitor) is maintained.

(6p)

b) In the previous problem the so called "application state" (the history of happenings and their result within the application) was maintained in the monitor. In this problem the content of the buffers also comprises part of the application state, whereas some parts of the state is within the thread(s) that handle(s) the orders as specified by the buffered events.

To supervise the timing and the progress of each thread, and possibly safety requirements (such as ensuring that crossing roads do not have green light at the same time) if not inherently taken care of by the sequencing, supervisory software (in terms of data, subclasses and additional threads) is sometimes added to the application. Outline (text and/or figures, no code) how that can be done for the monitor-based and the message-based solutions, and explain the key differences from a programming point of view.

(2p)