Exam
EDAF55/EDA040 Concurrent & Real-Time Programming

2019–08–22, 14.00–19.00

You are allowed to use

- Java quick reference (Java snabbreferens)
- calculator
- dictionary from English to/from your language

The exam has two parts:

- a theory part (problems 1–5, 10 points in total), and
- a construction part (problems 6–7, 20 points in total).

The maximal score is 30 points. The preliminary requirement for grade 3 is 15 points, with at least 4 points from the theory part, and at least 8 points from the construction part.

---

1. Race condition is a term that is often used in connection with real-time systems. Explain in a few sentences what a race condition is. *(1p)*

2. Show, using schematic code, how to use a semaphore to...
   a) achieve mutual exclusion between two threads. *(1p)*
   b) send a signal from one thread to another. *(1p)*

3. A real-time system includes four independent periodic threads, A, B, C and D. The threads are scheduled using deadline monotonic scheduling, DMS. The table below shows the worst-case execution times (C), period (T), and deadline (D) for each thread.

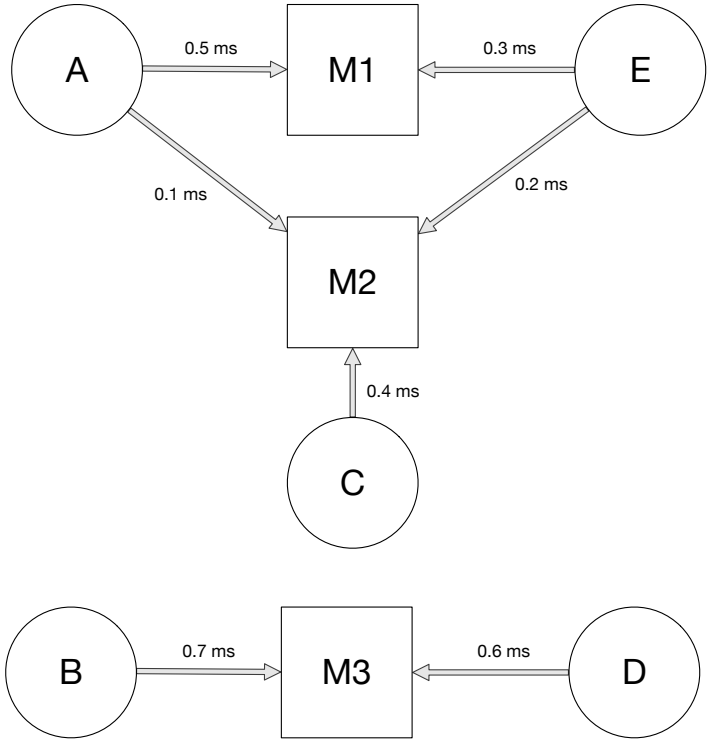
Thread	C [ms]	T [ms]	D [ms]
A	4	20	20
B	2	10	10
C	1	6	3
D	3	12	8

   What are the worst-case response times for each of the threads? *(2p)*

   Hint: $R_i = C_i + \sum_{j \in hp(i)} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$ (assuming no blocking)

4. There are four conditions that are necessary (and sufficient) for deadlock to occur in a system with concurrently executing threads. Which are these conditions? *(2p)*

5. A real-time system includes five periodic threads, A, B, C, D, and E. The threads are scheduled using RMS with the basic inheritance protocol. Thread A has the highest priority, B the next highest, and so on down to E that has the lowest priority. The threads communicate by calling methods from three monitors M1, M2, and M3 as described in the graph below (each method is only called once each period). Methods are annotated with their corresponding worst-case execution times (ms).



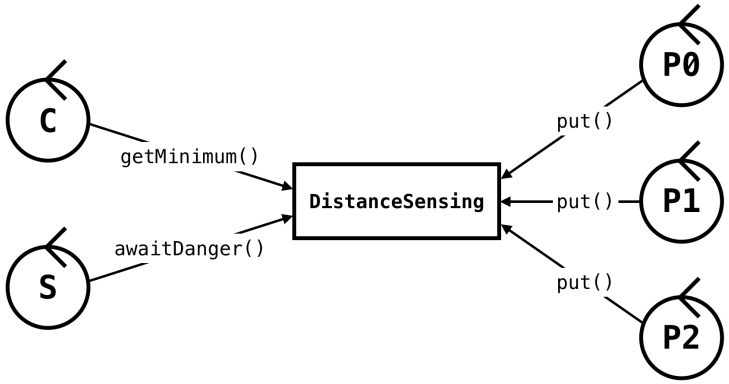
What are the worst-case blocking times (the time the threads can be blocked by lower-priority threads) for each one of the threads? *(3p)*

6. Filtering sensor input

In a control system for a mobile robot, three different sensors measure the distance to surrounding obstacles. The sensors are read by the three threads P0, P1, and P2. The sensor values are provided to a monitor, which you will implement.

A control thread C periodically fetches minimum distance by calling `getMinimum()`. Moreover, a supervisor thread S calls method `awaitDanger()`. This method determines when the robot is getting too close to another object, and emergency action is required (to avoid a collision).

The figure below summarizes the relations between threads and monitor methods.



Each sample is represented by a `Sample` object:

```
public class Sample {
  /** Returns the id (0, 1, 2) of the sensor thread that created the sample. */
  public int getId() { /* ... */ }

  /** Returns the sampled (distance) value. */
  public double getDistance() { /* ... */ }

  /** Returns the sample timestamp */
  public long getTimestamp() { /* ... */ }

  /* other stuff (not needed here) */
}
```

You will implement the monitor `DistanceSencing`:

```
public class DistanceSensing {

  private static final double DANGER_DIST = 0.8;

  /* attributes to be added */

  public DistanceSensing()                          {  /* to be implemented */  }

  public synchronized void put(Sample s)            {  /* to be implemented */  }

  public synchronized Sample getMinimum(long time)
    throws InterruptedException                     {  /* to be implemented */  }

  public synchronized Sample awaitDanger()
    throws InterruptedException                     {  /* to be implemented */  }
}
```

About the monitor operations

`void put(Sample s)` — Each sensor thread (P0, P1, P2) creates `Sample` objects and calls `put()` with these samples. Only the most recent `Sample` for each sensor is used: if `put()` is called multiple times for sensor Px without the previous value for sensor Px having been consumed, the old `Sample` is simply overwritten.

`Sample getMinimum(long time)` — This methods blocks until samples are available from all three sensors, such that each sample's timestamp is equal to or larger than `time`. The method then returns the `Sample` with the least (lowest) distance value.

`Sample awaitDanger()` — This method blocks until a dangerous situation occurs (when at least one sensor's distance value falls below `DANGER_DIST`).

This method is further described in task (b) below.

Your tasks

a) Implement the monitor class `DistanceSensing` as described above, *except the `AwaitDanger()` method*. That means you should decide what attributes you need and implement the methods `put()` and `getMinimum()`, as well as any private method you might desire. *(8p)*

b) Implement the `awaitDanger()` method. The `awaitDanger` method shall first block until the least distance value (out of three) is *larger than* `DANGER_DIST`, and then continue blocking until the least distance value (out of three) is *less than* `DANGER_DIST`.

In other words, the method blocks until a *transition* from a safe state to a dangerous state is detected.

In your implementation, you may assume that S is the only thread that calls `awaitDanger`, that there exists only one thread S, that priorities are strict, and that thread S has higher priority than any other thread calling the monitor. *(5p)*

7. **Design task: improving network performance**

In computer networking, sending small amounts of data can be inefficient. (Each transferred network packet includes a *header* of networking information. If the data payload in each packet is small, the network traffic will consist almost entirely of header information. This is clearly inefficent.)

For this reason, we wish to send a number of network messages at the same time, whenever possible. (They will then end up in the same packet, with a single header.)

In this task, our system has a low-level method for sending a single network packet:

```
public class LowLevelNetwork {
  /** Sends a network packet with the given messages. Up to 10 strings can be
      combined in a single packet (that is, the array size can be at most 10). */
  public static void sendPacket(String[] data) { ... }
}
```

We now wish to have a new method `BufferedNetwork.sendMessage`, as follows:

```
public class BufferedNetwork {
  /**
   * Submit a message (a string) to be sent over the network. Messages are NOT
   * always sent immediately. Instead, they are buffered, and sent when either
   *  (a) 10 messages are available (in the buffer) for sending, or
   *  (b) no message has been added (to the buffer) for the last 100 milliseconds.
   */
  public static void sendMessage(String m) { ... }
}
```

We expect `BufferedNetwork.sendMessage` to use additional classes to solve this problem.

Your task

Design the classes needed for `BufferedNetwork.sendMessage`. You are not required to provide a Java implementation (but you are allowed to). Your solution should include the following:

- A specification of the classes in your system. The specification includes the methods, their names, their parameters, and their return types.

  State clearly which methods are synchronized.
- For each method, provide a description of what it does (one or a few sentences). If a method blocks, specify under which conditions the calling thread is blocked, and under which conditions the thread is unblocked.
- For each class, indicate whether it is a thread, a monitor, an event, or something else.
- For each thread, specify which methods the thread calls.
- Specify which methods are called by `BufferedNetwork.sendMessage`.
- Specify what shared data is needed to coordinate your threads, and how that data is protected.
- Include a simple diagram of how your threads communicate.
  (UML notation is not required, but allowed.) *(7p)*