Exam
Concurrent and Real-Time Programming

2013–10–22, 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.

The exam has two parts; first the theory part that consist 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 so called 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 programming problem. At least a partial solution to the design problem is required for the highest grade (5).

---

1. Priorities are used to make sure important activities get to run before less important activities.
   a) Why should you not use priorities to ensure correct results of concurrent programs? *(1p)*
   b) Describe a situation when it is appropriate to use priorities. *(1p)*
   c) What is a *race condition*? *(1p)*
   d) Briefly explain *critical section*. *(1p)*

2. Is the system below deadlock free? In case of deadlock, suggest change and show that the new solution is free of deadlocks. The methods are all independent of each other and within a thread method execution order is irrelevant. All monitors exist in one instance only. *(2p)*

```
public class M1 {
    private M2 m2; private M3 m3;
    public synchronized void a1() { m2.a2(); }
    public synchronized void b2() { m3.b1(); }
    public synchronized void c1() { /* ... */ }
}

public class M2 {
    private M1 m1;
    public synchronized void a2() { /* ... */ }
    public synchronized void c3() { /* ... */ }
    public synchronized void c4() { m1.c1(); }
}

public class M3 {
    private M2 m2;
    public synchronized void b1() { /* ... */ }
    public synchronized void c2() { m2.c3(); }
}

public class A extends PeriodicThread {
    private M1 m1; private M2 m2;
    public void perform() { m1.a1(); m2.a2(); }
}
```

```
public class B extends PeriodicThread {
    private M1 m1;
    public void perform() { m1.b2(); }
}

public class C extends PeriodicThread {
    private M2 m2; private M3 m3;
    public void perform() { m2.c4(); m3.c2(); }
}
```

3. In the course we have used the following inequality:

$$\sum_{i=1}^{i=n} \frac{C_i}{T_i} \leq n(2^{1/n} - 1)$$

   a) What entities do the symbols C_i, T_i, and n represent? *(1p)*

   b) For the inequality to be applicable, the real-time system involved must have some specific properties. Give example of two such properties. *(1p)*

   c) What conclusions are we able to draw in each of the following cases: 1. The inequality is satisfied. 2. The inequality is *not* satisfied. *(1p)*

4. A Java program consists of two monitors and four threads. The program is running single-handed on a single-processor hard real-time system using RMS and the basic inheritance protocol.

```
public class M1 {
    public synchronized void a1(){ /* 1 ms */ }
    public synchronized void d() { /* 5 ms */ }
}

public class M2 {
    public synchronized void a2(){ /* 1 ms */ }
    public synchronized void b() { /* 3 ms */ }
    public synchronized void c() { /* 4 ms */ }
}

public class A extends PeriodicThread {
    private M1 m1; private M2 m2;
    public void perform() /* 16 ms period */ { m1.a1(); m2.a2(); }
}

public class B extends PeriodicThread {
    private M2 m2;
    public void perform() /* 17 ms period */ { m2.b(); }
}

public class C extends PeriodicThread {
    private M2 m2;
    public void perform() /* 18 ms period */ { m2.c(); }
}

public class D extends PeriodicThread {
    private M1 m1;
    public void perform() /* 19 ms period */ { m1.d(); }
}
```

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

   Hint:

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

5. When programming concurrency it is important to know if libraries and classes used by threads are *thread safe*.

   a) Explain briefly *thread safe*. *1p*

   b) How would you use a library that is not considered thread safe in a multi-threaded program? *1p*

   c) Which of the classes below are thread safe? Motivate with one sentence per class. *3p*

```
class A {
  final double s =
    Math.PI/1024;
  int sin(int x) {
    double a =
      Math.sin(s*x);
    return a/s;
  }
  int cos(int x) {
    double a =
      Math.cos(s*x);
    return a/s;
  }
}

class B {
  float square(float x){
    return x*x;
  }
  float cube(float x) {
    return x*x*x;
  }
}

class C {
  int tmp;
  int[] swap(int[] a) {
    int len = a.length;
    if (a==null)
      return null;
    if (len<2) return a;
    else {
      tmp = a[0];
      a[0] = a[len];
      a[len] = tmp;
    }
    return a;
  }
}
```

Programming problem: Hamburger "freshness" controller

6. The hamburger company *LU-smask* has contracted you to write the controller to a brand new heating cabinet for hamburgers. According to the company, hamburgers may only be stored in the cabinet for a certain time (10 minutes) before being sold to customer. Older burgers must be thrown away. The cabinet contains 7 shelfs (with a maximum of 10 burgers per shelf) to store 7 different hamburger types, and compared to the old cabinet, which used markers to indicate how long hamburgers had been on the shelf, the new cabinet features an innovative display for each shelf that shows the number of burgers on the shelf as well as the time remaining for the earliest stored hamburger before it needs to be thrown away.

The display is updated immediately when a hamburger is put on or removed from the shelf, but also periodically every 10 seconds to update time remaining. Each shelf also contains an input sensor and an output sensor, which, respectively, registers when a hamburger is added to the shelf and when a hamburger is removed from the shelf. The shelf is constructed so that hamburgers are stored, always, on a first-in-first-out basis, allowing your controller to keep track of the burgers. When a burger is stored on the shelf for too long time, an alarm is triggered and the display is updated with information that the hamburger should be thrown away. As an added feature there is also a check if there are more hamburgers close in time (within one minute) to be thrown away and those are indicated as well on the display, e.g. *throw away 5 hamburgers*. When the alarm is triggered it beeps once per second (done in hardware) until the requested number of hamburgers are removed. If the burgers are not removed for some time, the display is updated to reflect if more burgers get old and need to be removed. Since there is only one alarm loudspeaker for the entire cabinet, the alarm does not stop until conditions for all shelves are met.

The specification you have been given for the controller includes a skeleton design and a hardware interface specification. The hardware interface gives access to the cabinet electronics so you only need to worry about the controller as such. The class (except implementation) is shown below:

```
// Each method must be called from the same thread. But different
// methods may be called from different threads.
class HW {
  /** Returns the shelf number when a hamburger has been put on a shelf (blocking) */
  public static int inputDetected();

  /** Returns the shelf number when a hamburger has been removed from a shelf (blocking) */
  public static int removalDetected();

  /** Displays text txt on display i */
  public static void display(int i, String txt);

  /** Start alarm beeping (blocking until stopAlarm is called) */
  public static void alarm();

  /** Stop alarm beeping */
  public static void stopAlarm();

  /** Get current time (seconds since cabinet power on) */
  public static int time();
}
```

The required design is well-known to you, being based on a number of threads communicating through a monitor. The monitor class, named *BurgerCabinet*, contains the state of the entire cabinet. The constructor creates the monitor by giving a vector with one element for each shelf specifying max shelf time for burgers on that shelf (preparing for future use when storage times may differ), giving the number of burgers possible to store on a shelf and giving the number of shelfs in the cabinet. All shelves are equally sized. Four monitor methods are being specified for the monitor. The *putBurger* method is to be called when an input sensor is triggered, informing the monitor that a burger has been put on a shelf. In the same way, *removeBurger* is to be called when the output sensor is triggered, informing the monitor that a burger has been removed from a shelf. This method is also responsible for stopping the alarm as soon as possible after conditions are met. Both methods are responsible for signalling to immediately reflect the state change in the displays. Also, the sensors themselves do not contain any state, so it is important to quickly call the hardware sensor methods *HW.inputDetected* and *HW.removalDetected* again after a detection to avoid missing any sensor signal. The *alarm* method should block until the conditions for raising alarm are met. Finally, the *display* method should wait for signals from the *putBurger* and *removeBurger* methods to immediately display the current state (number of burgers on each shelf, time remaining for oldest burger on each shelf *or* number of burgers to remove *or* nothing if the shelf is empty). Also, the method updates the displays once every ten seconds even if no signals are received. The hardware is partially thread safe in the sense that each method is only allowed to be called by one thread, but different methods may be called by different threads.

```
class BurgerCabinet {

  /** Suggested partial state - for simplicity only primitive data types are used */
  private int[] max;        // Max minutes to store burger
  private int maxBurgers;   // Max burgers on shelf
  private int nShelfs;      // Max shelfs
  private int[][] timestamp;// nShelfs ringbuffers to hold timestamps of burgers per shelf
  private int[] inputIndex; // Input ringbuffer indices
  private int[] outputIndex;// Output ringbuffer indices
  private int[] size;       // Ringbuffer sizes
  private int[] remove;     // Number of burgers to remove per shelf
  /** Fill in with your own attributes as needed */


  /** Constructs a BurgerCabinet monitor */
  public BurgerCabinet(int[] maxMinutesOnShelf, int maxBurgers, int nShelfs) { ... }

  /** Put a burger on shelf number s */
  public synchronized void putBurger(int s) { ... }

  /** Remove a burger from shelf number s */
  public synchronized void removeBurger(int s) { ... }

  /** Blocks until alarm start condition is met */
  public synchronized void alarm() { ... }

  /** Update displays on all shelfs */
  public synchronized void display() { ... }
}
```

a) Implement a main program starting up the system and implement the needed four threads (one thread implementation and the run method of the rest). *(2p)*

b) Specify the full state and implement the constructor of the controller monitor. *(2p)*

c) Implement the methods of the monitor. *(8p)*

Design problem: Modernising the burger order system

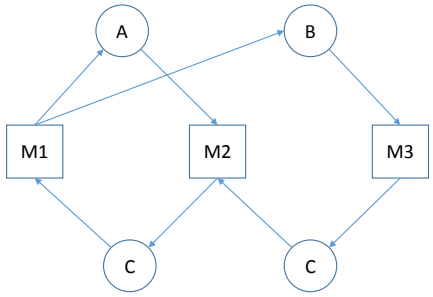
7. The employer was pleased with your performance in the last assignment. Therefore you have been awarded a new contract for modernising the order system in *LU-smask*. Customers should be able to place orders at counters as before, but also through their mobile phones. The phone ordering system is based on QR codes, with a menu with QR codes being at display in the entrance. Upon usage the customer directs the mobile camera of his/her mobile phone towards a QR code. The phone reads the code and opens a web page allowing the customer to make further customization and finally place the order. When finished the mobile display shows an order number and the price of the order. At the counter this number is shown for payment and receiving the order. The order system, including the web server and a database that keeps track of the orders from both the counters and the mobile phones is implemented in one controller. The web server is capable of handling requests from several customers at the same time. In the kitchen a display shows orders placed for the kitchen workers. The display is also controlled from the controller.

   Determine activities (threads) and responsibilities for each thread in the order controller. Answer by drawing a class diagram indicating threads and passive objects. List the responsibilities of each thread. *(6p)*

Short solution

1. a) The root cause of the real-time problem remains. There is no guarantee that it will not re-surface again.
   b) For instance: a high-priority control task combined with a low-priority logging task.
   c) A race condition occurs when the result of the execution of threads depends on the order the threads happen to be scheduled. The threads are thus "racing" against each other. This is usually a result of faulty synchronization.
   d) A critical section is a section of code that is protected against data corruption, race conditions, etc., usually by the means of a synchronization primitive allowing threads exclusive access to the code section (i.e. mutex semaphore).

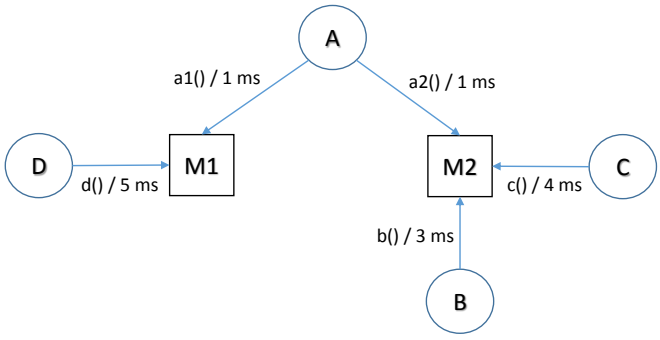
2. Resource allocation graph:



   The graph shows potential deadlock situations. Change the execution order of, for instance, A and B to avoid this.

3. a) C_i: worst case execution time (WCET) of thread i. T_i: period of thread i. n: number of threads in the system.
   b) Only periodic threads, no blocking, deadline = period, fixed priority scheduling (RMS).
   c) 1. The system is schedulable. 2. The system might be schedulable.

4. Call graph:



   WCET for A:
   $C_A = 1 + 1 = 2$

   Blocking times:
   $B_D = 0$
   $B_C = d = 5$ (indirect blocking)
   $B_B = d + c = 9$ (indirect + direct)
   $B_A = d + max(c, b) = 9$ (direct+direct)

   Response times:
   $R_A = 2 + 9 = 11$

$R_B = 3 + 9 = 12$
$R_B = 12 + \left\lceil \frac{12}{16} \right\rceil 2 = 14$
$R_B = 12 + \left\lceil \frac{14}{16} \right\rceil 2 = 14$

$R_C = 4 + 5 = 9$
$R_C = 9 + \left\lceil \frac{9}{16} \right\rceil 2 + \left\lceil \frac{9}{17} \right\rceil 3 = 14$
$R_C = 9 + \left\lceil \frac{14}{16} \right\rceil 2 + \left\lceil \frac{14}{17} \right\rceil 3 = 14$

$R_D = 5$
$R_D = 5 + \left\lceil \frac{5}{16} \right\rceil 2 + \left\lceil \frac{5}{17} \right\rceil 3 + \left\lceil \frac{5}{18} \right\rceil 4 = 14$
$R_D = 5 + \left\lceil \frac{14}{16} \right\rceil 2 + \left\lceil \frac{14}{17} \right\rceil 3 + \left\lceil \frac{14}{18} \right\rceil 4 = 14$

All response times are lower than their corresponding period time. The system is schedulable.

5. a) The software must function correctly with concurrent calls from different threads.
   b) Either the library must be called from one thread only, in which case one thread needs to be responsible for the library and other threads must defer execution to the responsible thread, or the library should simply be protected from concurrent access by the programmer.
   c) A is thread safe. There is one common attribute, but it cannot change during execution. B is thread safe since it only uses data that is local to each method. C is not thread safe since concurrent calls to swap will (not atomically) use the attribute tmp. If a call is interrupted by another with a higher priority between the assignments to and from tmp, the result will most likely be wrong.

6. a)

```
public class Main {
    private static int[] maxWait = {10, 10, 10, 10, 10, 10, 10};
    private static int nBurgers = 10;
    private static int nShelfs = 7;

    public static void main(String[] args) {
        BurgerCabinet c = new BurgerCabinet(maxWait, nBurgers, nShelfs);

        new DisplayThread(c).start();
        new InputThread(c).start();
        new OutputThread(c).start();
        new StartAlarmThread(c).start();
    }

}


public class InputThread extends Thread {
    private BurgerCabinet c;

    public InputThread(BurgerCabinet cabinet) {
        c = cabinet;
    }

    public void run() {
        while (true) {
            c.putBurger(HW.inputDetected());
        }
    }
}
```

```
public class OutputThread extends Thread {
    private BurgerCabinet c;

    public OutputThread(BurgerCabinet cabinet) {
        c = cabinet;
    }

    public void run() {
        while (true) {
            c.removeBurger(HW.removalDetected());
        }
    }
}


public class DisplayThread extends Thread {
    private BurgerCabinet c;

    public DisplayThread(BurgerCabinet cabinet) {
        c = cabinet;
    }

    public void run() {
        while (true) {
            c.display();
        }
    }
}


public class StartAlarmThread extends Thread {
    private BurgerCabinet c;

    public StartAlarmThread(BurgerCabinet cabinet) {
        c = cabinet;
    }

    public void run() {
        while (true) {
            c.alarm();
            HW.alarm();
        }
    }
}
```

b) See c.

c)

```
public class BurgerCabinet {
    private int[] max;        // Max minutes to store burger
    private int maxBurgers;   // Max burgers on shelf
    private int nShelfs;      // Max shelfs
    private int[][] timestamp;// nShelfs ringbuffers to hold timestamps of burgers per shelf
    private int[] inputIndex; // Input ringbuffer indices
    private int[] outputIndex;// Output ringbuffer indices
    private int[] size;       // Ringbuffer sizes
    private int[] remove;     // Number of burgers to remove per shelf
    private boolean[] update; // Update display immediately
    private int time;         // Current display time

    public BurgerCabinet(int[] maxMinutesOnShelf, int maxBurgers, int nShelfs) {
        max = maxMinutesOnShelf;
```

```
        this.maxBurgers = maxBurgers;
        this.nShelfs = nShelfs;
        timestamp = new int[nShelfs][maxBurgers];
        inputIndex = new int[nShelfs];
        outputIndex = new int[nShelfs];
        size = new int[nShelfs];
        remove = new int[nShelfs];
        update = new boolean[nShelfs];
        for (int i=0; i<nShelfs; i++) {
            update[i] = false;
        }
        time = HW.time();
    }

    public synchronized void putBurger(int s) {
        timestamp[s][inputIndex[s]] = HW.time();
        inputIndex[s] = (inputIndex[s]+1) % maxBurgers;
        size[s]++;
        update[s] = true;
        notifyAll();
    }

    public synchronized void removeBurger(int s) {
        outputIndex[s] = (outputIndex[s]+1) % maxBurgers;
        size[s]--;
        if (allShelfsOk()) HW.stopAlarm();
        update[s] = true;
        notifyAll();
    }

    public synchronized void alarm() {
        try {
            while (allShelfsOk()) wait();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public synchronized void display() {
        try {
            while (time + 10 > HW.time() && !immediateUpdate()) wait(1000);
            if (time + 10 <= HW.time()) time += 10;
            for (int s=0; s<nShelfs; s++) {
                update[s] = false;
                if (emptyShelf(s)) {
                    HW.display(s, "---");
                    continue;
                }
                remove[s] = toRemove(s);
                if (remove[s] == 0) {
                    int diff = timestamp[s][outputIndex[s]] + max[s]*60 - HW.time();
                    HW.display(s, size[s]+" burger(s). Time remaining: "+diff);
                    // No notify since removeBurger stops alarm
                } else {
                    HW.display(s, "Remove "+Integer.toString(remove[s])+" burgers");
                    notifyAll();
                }
            }
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }
```

```
    private boolean emptyShelf(int shelf) {
        return size[shelf] == 0;
    }

    private boolean allShelfsOk() {
        for (int i=0; i<nShelfs; i++) if (remove[i] > 0) return false;
        return true;
    }

    private boolean immediateUpdate() {
        for (int i=0; i<nShelfs; i++) if (update[i]) return true;
        return false;
    }

    private int toRemove(int s) {
        if (emptyShelf(s)) return 0;
        if (timestamp[s][outputIndex[s]] + max[s]*60 > HW.time()) return 0;
        int r = 0;
        for (int i=outputIndex[s]; i != inputIndex[s]; i = (i+1) % maxBurgers)
            if (timestamp[s][i] + max[s]*60 - 60 < HW.time()) r++;
        return r;
    }
}
```

7. No solution provided.