Exam
Concurrent and Real-Time Programming

2014–01–07, 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.

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. When designing concurrent systems there are several concurrency-related errors that may occur. For each error below, briefly explain it and construct a situation where the error might occur.
   a) Priority inversion *(1p)*
   b) Starvation *(1p)*
   c) Live lock *(1p)*
   d) Dead lock *(1p)*
   e) Data corruption *(1p)*

2. Is the system below deadlock free? It is not known how many instances of the threads A, B and C are running. In case of deadlock, suggest program changes and show that the new solution is free of deadlocks. The methods are all independent of each other and may execute in different order. All monitors exist in one instance only. *(2p)*

```
public class M1 {
  private M2 m2;
  public synchronized void a1() { /* ... */ m2.a2_1(); /* ... */ }
  public synchronized void c1() { /* ... */ }
}

public class M2 {
  private M3 m3;
  public synchronized void a2_1() { /* ... */ }
  public synchronized void a2_2() { /* ... */ m3.a3(); /* ... */ }
}

public class M3 {
  private M1 m1; private M2 m2;
  public synchronized void a3() { /* ... */ }
  public synchronized void b3() { /* ... */ m4.b4(); /* ... */ }
  public synchronized void c3_1() { /* ... */ m1.c1(); /* ... */ }
  public synchronized void c3_2() { /* ... */ }
}
```

```
public class M4 {
  private M3 m3;
  public synchronized void b4() { /* ... */ }
  public synchronized void c4() { /* ... */ m3.c3_2(); /* ... */ }
}

public class A extends PeriodicThread {
  private M1 m1; private M2 m2;
  public void perform() { /* ... */ m1.a1(); /* ... */ m2.a2_2(); /* ... */ }
}

public class B extends PeriodicThread {
  private M3 m3;
  public void perform() { /* ... */ m3.b3(); /* ... */ }
}

public class C extends PeriodicThread {
  private M3 m3; private M4 m4;
  public void perform() { /* ... */ m3.c3_1(); /* ... */ m4.c4(); /* ... */ }
}
```

3. Most operating systems provide some support when programming concurrency. But what to do when no such support is available?

   a) Pre-emption is a common mechanism for context switching. How can context switching be done without pre-emption support? Mention one advantage and one disadvantage of your suggestion as compared to using pre-emption. *(2p)*

   b) There is support for threads but no synchronization primitives. As a workaround mutual exclusion has been implemented as shown below. What are the four requirements for a working mutual exclusion? Are they fulfilled (motivate your answer)? *(2p)*

```
volatile int turn = 1;

class M1 extends Thread {
  public void run() {
    while (true) {
      nonCriticalCode1();
      while (turn!=1);
      criticalSection1();
      turn = 2;
    }
  }
}

class M2 extends Thread {
  public void run() {
    while (true) {
      nonCriticalCode2();
      while (turn!=2);
      criticalSection2();
      turn = 1;
    }
  }
}
```

   c) In the Java code above, what might happen if *volatile* is left out (motivate your answer)? *(1p)*

4. When making use of the signalling mechanism in a Java monitor the threads are usually in one of three execution states.

   a) Name the three execution states. Explain each state. *(1p)*

b) Explain what transitions may occur between the states and who is initiating each transition (program or system). *(1p)*

5. 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 b1(){ /* 1 ms */ }
  public synchronized void d() { /* 2 ms */ }
}

public class M2 {
  public synchronized void a() { /* 1 ms */ }
  public synchronized void b2(){ /* 2 ms */ }
  public synchronized void c() { /* 3 ms */ }
}

public class A extends PeriodicThread {
  private M2 m2;
  public void perform() /* 10 ms period */ { m2.a(); /* + 2 ms */ }
}

public class B extends PeriodicThread {
  private M1 m1; private M2 m2;
  public void perform() /* 16 ms period */ { m1.b1(); /* + 1 ms + */ m2.b2(); }
}

public class C extends PeriodicThread {
  private M2 m2;
  public void perform() /* 23 ms period */ { /* 2 ms + */ m2.c(); }
}

public class D extends PeriodicThread {
  private M1 m1;
  public void perform() /* 31 ms period */ { m1.d(); /* + 4 ms */ }
}
```

Is the system schedulable. Motivate your answer. *(4p)*

Hint:

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

Programming problem: Hamburger simulation for optimization

6. The hamburger company *LU-smask* has contracted you (possibly again) to write software. Their marketing department wants to launch a new commercial, "Get your order within 2 minutes or eat for free". But being a bit cautious, they want you to implement a time-driven simulation of their hamburger place so they can investigate that the time limit is sensible for varying customer pressures. At the same time they do not want to hire extra personnel unless necessary. At the moment the hamburger place employs 7 hamburger makers (chefs) and 3 cashier assistants (sellers).

   The simulation should feature the chefs, sellers and customers. The chefs and sellers should be modelled with one thread per person. The customers should be handled a bit different, one thread should simulate customer arrival rate and put each new arrival in a customer queue. The sellers are then handling orders from the queueing customers while the chefs are manufacturing the burgers according to what is ordered. Several different burger types may be ordered. The threads should communicate through a monitor containing the logic of the simulation. An extra statistics thread should gather information about thrown away burgers, missed order deadlines, and number of laid orders on an hourly basis (simulated time).

A burger cabinet is used to store hamburgers after they have been manufactured. LU-smask wants to implement predictive burger manufacturing, allowing the chefs to manufacture a few burgers even though no customer has ordered them (yet). Of course, the burgers may only be stored for a few minutes before they are sold. If they are not sold within the time limit they have to be thrown away. LU-smask wants to optimize the number of stored burgers to minimize the number of thrown away burgers while still keeping the promised order time. For the simulation only one burger type per shelf is stored in the cabinet.

An utility class *Util* is provided for you to help implementing the simulation.

```
public class Util {
  // Return the number of shelfs in the burger cabinet.
  // Each shelf equals a burger type.
  public static int shelfs() { ... }

  // Return the number of burgers to store predictively on the shelf s
  public static int keepOnShelf(int s) { ... }

  // Return the max time a burger is allowed to be stored on a shelf
  // before it is too old. Time is returned in seconds.
  public static int maxTimeOnShelf() { ... }

  // Return the max allowed customer waiting time (2 mins) from the
  // moment the order is taken by a seller until the burger is delivered.
  // Time is returned in seconds.
  public static int maxWaitingTime() { ... }

  // Return the time it takes a chef to anufacture a burger of certain type.
  // Time is returned in seconds.
  public static int manufacturingTime(int burgerType) { ... }

  // Return current simulation time. To be used instead of System.currentTimeMillis.
  // Time is speeded up a scale factor as compared to real time.
  // Time is returned in seconds.
  public static int currentTime() { ... }

  // Return real time given simulation time. Time is scaled properly so
  // Thread.sleep, etc., waits an appropriate amount. Time is given
  // in seconds, but is returned in milliseconds.
  public static int waitTime(int t) { ... }

  // Blocks until a new customer has arrived. Returns the burger type
  // (modelled as an integer). The returned number corresponds directly
  // to a shelf in the burger cabinet.
  public static int newCustomer() { ... }
}
```

The threads should perform the following duties:

- The *customer* should wait for a new customer arrival, take the order and place it in queue.
- The *seller* should wait for a customer and then place the customer order with the chefs. While waiting for chefs to prepare the burger, old burgers should be cleaned from the cabinet. It is enough to clean the shelf for the ordered burger type. Time from given order to delivered burger should be measured (in simulation time) and missed deadlines should be counted.
- The *chef* should wait for an order or manufacture a predicted burger to put in the cabinet shelf if needed. Then the burger should be manufactured and finally put on the correct shelf (shelf number equals burger type). The manufacturing code section should wait for an appropriate amount of time (as given by *Util.manufacturingTime*) to simulate the preparation of the burger. While the burger is prepared it is important that no other chef is preparing the same order or starts manufacturing the same predicted burger.

- The *statistics* gathering thread should, once per simulated hour, gather the number of thrown away burgers, number of missed deadlines and the total number of ordered burgers during the hour. These numbers should be printed and reset for the next hour.

A skeleton monitor class has been prepared for you. It contains suggested simulation state, a constructor and one method per thread type. The *customer* attribute contains the customer queue, storing the customer order (burger type). The *orders* attribute contains the order list for the chefs, eeach entry being a burger type as ordered by a customer. The *shelfs* attribute represents the burger cabinet with one shelf per burger type. The *inProgress* attribute keeps track of burger currently being manufactured with one entry per burger type. Finally, the three attributes *thrownBurgers*, *missedDeadlines* and *orderedBurgers* counts the number of thrown away burgers, missed customer burger delivery (within 2 mins) and total number of ordered burgers per hour.

```
public class Monitor {

  // Suggested state
  private LinkedList<Integer> customers;// Customer list
  private LinkedList<Integer> orders;   // Order list
  private LinkedList<Integer>[] shelfs; // Shelfs
  private int[] inProgress;
  private int thrownBurgers;
  private int missedDeadlines;
  private int orderedBurgers;

  // Constructor
  public Monitor() { ... }

  // Simulate a chef manufacturing a burger
  public synchronized void manufactureBurger() { ... }

  // Simulate a seller handling an order
  public synchronized void handleOrder() { ... }

  // Placing a new customer in queue
  public synchronized void newCustomer(int burgerType) { ... }

  // Return a vector containing {thrownBurgers,missedDeadlines,orderedBurgers}
  public synchronized int[] gatherStatistics() { ... }
}
```

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

b) Implement the constructor and methods of the monitor. *(10p)*

Design problem: Throw away burgers automation

7. The employer was pleased with your performance in the last assignment. Therefore you have been awarded a new contract to improve handling of burgers to throw away in the burger cabinet. The employees have tired of constantly checking and throwing away old burgers. You are tasked with designing a system to automate the task of throwing away burgers. You have already talked to a friend of yours, a hardware engineer, about installation of sensors and actuators in the cabinet. You have decided upon installing a camera in the roof of the cabinet and a small oonveyor per shelf. The camera should detect when burgers are entered and removed from the cabinet. It is important that the camera keeps track of each individual burger so the time it has spent on the shelf can be accurately tracked. The conveyor is used to transport old burgers down into a waste bin. Sold burgers are still picked manually. A sensor is located in the waste bin to indicate when it is full. A beeper is sounding when the waste bin needs to be emptied.

The camera sensor and the conveyor is a “smart” sensor and actuator, respectively, meaning that they contain an onboard programmable computer. The computers are rather tiny but good for simple programs. Both communicate using TCP/IP and sockets. The waste bin sensor is not smart, giving a digital output indicating full/not full. You are hooking up the sensors and actuators to a computer to control the system.

Determine activities (threads) in the system. Locate each thread to a piece of hardware and determine responsibilities for each thread. Motivate your decisions. Draw a schematic of the system with threads and passive objects indicated. Show which threads are communicating, explain how they communicate and indicate what information is exchanged. *(6p)*

Short solution

1. a) Low priority thread preempts a high priority thread. Three threads L, M, H with low, medium and high priority is running in a system. L and H tries to access a common resource. L gets the resource stopping H from execution. Now M runs and preempts L. H then has to wait for M to finish execution before it can run. This is priority inversion.

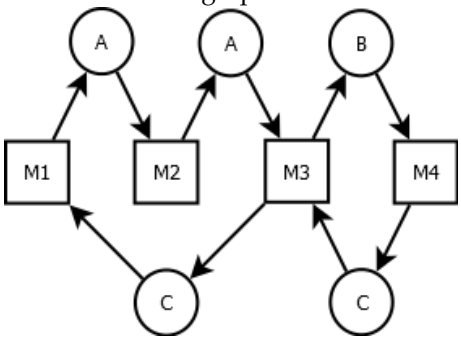
   b) Starvation is when a thread is continually denied access to resources and therefore cannot finish its task. Usually caused by simplistic scheduling algorithms. For instance CPU starvation, the system always switches between two tasks (for some reason) and a third task never gets to run.

   c) A live lock occurs when the involved threads are constantly changing state but not progressing. A real-world example of livelock occurs when two people meet in a narrow corridor, and each tries to be polite by moving aside to let the other pass, but they end up swaying from side to side without making any progress because they both repeatedly move the same way at the same time.

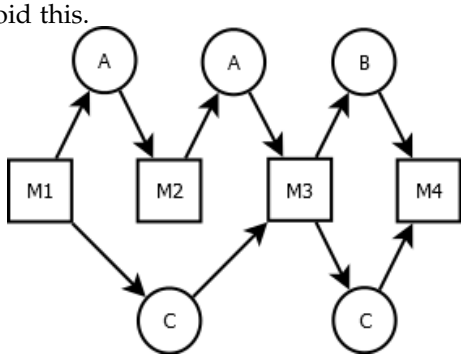
   d) A deadlock occurs when two or more threads are waiting for each other to finish, thus noone finishes. Two threads, A and B, tries to grab two semaphores S1 and S2. A grabs S1, then B grabs S2. Now A tries to grab S2 but cannot since B holds it. Now B tries to grab S1 but cannot since A holds it. Deadlock.

   e) Data corruption occurs when two or more threads compete about a resource that is not thread safe. For instance, if two threads try to write to a byte array at the same time the content of said array will with high probability be a mix of content from the two threads.

2. Resource allocation graph:



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



3. a) Non-preemtive scheduling, the running thread continues until it voluntarily releases the CPU. Advantage: it is possible to choose suitable points for context switching. Disadvantage: the thread must not misbehave.

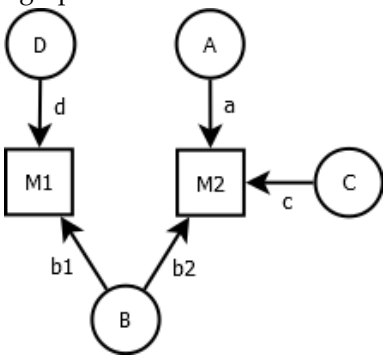
   b) Mutual exclusion, no deadlock, no starvation, efficiency. The two critical code segments are alternating in execution, so mutual exclusion is ok. One thread can always proceed, so no deadlock. No starvation since the code segments are alternating. The code uses busy waits and does not work for one thread only, so not very efficient.

c) If volatile is left out, the turn variable is not guaranteed to be shared between thread contexts. This is a necessary condition for the implementation to work.

4. a) Running, ready and blocked. A thread is ready when it may run but has not been selected by the scheduler. A thread is blocked when it may not run pending on an event to occur.

   b) Running->blocked happens when wait() is called on the monitor (program). Blocked->ready occurs when notify() is called on the monitor (program). Running<->ready is handled by the scheduler (system).

5. Call graph:



WCET:
$C_A = 1 + 2 = 3$
$C_B = 1 + 2 + 1 = 4$
$C_C = 3 + 2 = 5$
$C_D = 4 + 2 = 6$

Blocking times:
$B_D = 0$
$B_C = d = 2$ (indirect blocking)
$B_B = c + d = 5$ (indirect + direct)
$B_A = max(c, b2) = 3$ (direct)

Response times:
$R_A = 3 + 3 = 6$

$R_B = 4 + 5 = 9$
$R_B = 9 + \lceil \frac{9}{10} \rceil 3 = 12$
$R_B = 9 + \left\lceil \frac{12}{10} \right\rceil 3 = 15$
$R_B = 9 + \left\lceil \frac{15}{10} \right\rceil 3 = 15$

$R_C = 5 + 2 = 7$
$R_C = 7 + \lceil \frac{7}{10} \rceil 3 + \lceil \frac{7}{16} \rceil 4 = 14$
$R_C = 7 + \left\lceil \frac{14}{10} \right\rceil 3 + \left\lceil \frac{14}{16} \right\rceil 4 = 17$
$R_C = 7 + \left\lceil \frac{17}{10} \right\rceil 3 + \left\lceil \frac{17}{16} \right\rceil 4 = 21$
$R_C = 7 + \left\lceil \frac{21}{10} \right\rceil 3 + \left\lceil \frac{21}{16} \right\rceil 4 = 24$
$R_C = 7 + \left\lceil \frac{24}{10} \right\rceil 3 + \left\lceil \frac{24}{16} \right\rceil 4 = 24$

$R_D = 6$
$R_D = 6 + \lceil \frac{6}{10} \rceil 3 + \lceil \frac{6}{16} \rceil 4 + \lceil \frac{6}{23} \rceil 5 = 18$
$R_D = 6 + \left\lceil \frac{18}{10} \right\rceil 3 + \left\lceil \frac{18}{16} \right\rceil 4 + \left\lceil \frac{18}{23} \right\rceil 5 = 25$
$R_D = 6 + \lceil \frac{25}{10} \rceil 3 + \lceil \frac{25}{16} \rceil 4 + \lceil \frac{25}{23} \rceil 5 = 33$

$R_D = 6 + \lceil \frac{33}{10} \rceil 3 + \lceil \frac{33}{16} \rceil 4 + \lceil \frac{33}{23} \rceil 5 = 40$
$R_D = 6 + \left\lceil \frac{40}{10} \right\rceil 3 + \left\lceil \frac{40}{16} \right\rceil 4 + \left\lceil \frac{40}{23} \right\rceil 5 = 40$

Two response times are longer than their corresponding period time. The system is not schedulable.

6. a)

```
public class Main {
    private static final int nChefs = 7;
    private static final int nSellers = 3;

    public static void main(String[] args) {
        Monitor bc = new Monitor();
        for (int i=0; i<nChefs; i++) (new Chef(bc)).start();
        for (int i=0; i<nSellers; i++) (new Seller(bc)).start();
        (new Customers(bc)).start();
        (new Statistics(bc)).start();
    }

}


public class Chef extends Thread {
    private Monitor bc;

    public Chef(Monitor c) {
        bc = c;
    }

    public void run() {
        while (true) {
            bc.manufactureBurger();
        }
    }
}


public class Seller extends Thread {
    private Monitor bc;

    public Seller(Monitor c) {
        bc = c;
    }

    public void run() {
        while (true) {
            bc.handleOrder();
        }
    }
}


public class Customers extends Thread {
    private Monitor bc;

    public Customers(Monitor c) {
        bc = c;
    }

    public void run() {
        while (true) {
            bc.newCustomer(Util.newCustomer());
```

```
            }
        }
    }


    public class Statistics extends Thread {
        private Monitor bc;
        private int hour;

        public Statistics(Monitor c) {
            hour = 0;
            bc = c;
        }

        public void run() {
            System.out.println("Hour\tOrdered burgers\t\tOverdue orders\t\tThrown away burgers\t\t
            while (true) {
                hour += 1;
                try {
                    int diff = hour*3600 - Util.currentTime();
                    Thread.sleep(Util.waitTime(diff));
                } catch (InterruptedException e) {
                    e.printStackTrace();
                }

                int[] data = bc.gatherStatistics();

                int thrown = data[0];
                int overdue = data[1];
                int ordered = data[2];
                int customers = data[3];
                System.out.println(hour+"\t"+ordered+"\t\t\t"+overdue+"\t\t\t"+thrown+"\t\t\t\t"+c
            }
        }
    }
```

b)

```
    import java.util.LinkedList;

    public class Monitor {
        private LinkedList<Integer> customers;// Customer list
        private LinkedList<Integer> orders;   // Order list
        private LinkedList<Integer>[] shelfs; // Shelfs
        private int[] inProgress;
        private int thrownBurgers;
        private int missedDeadlines;
        private int orderedBurgers;

        public Monitor() {
            customers = new LinkedList<Integer>();
            orders = new LinkedList<Integer>();
            shelfs = new LinkedList[5];
            for (int i=0; i<Util.shelfs(); i++) shelfs[i] = new LinkedList<Integer>();
            inProgress = new int[Util.shelfs()];
            thrownBurgers = 0;
            missedDeadlines = 0;
            orderedBurgers = 0;
        }

        public synchronized void manufactureBurger() {
            try {
                // Wait for order or produce predicted burger
                while (orders.size() == 0 && missingOnShelf() < 0) wait(Util.waitTime(60));
```

```
            // Burger type
            int burgerType = -1;
            if (orders.size() > 0) {
                burgerType = orders.removeFirst();
            } else {
                burgerType = missingOnShelf();
            }

            // Manufacture burger
            inProgress[burgerType]++;
            int t1 = Util.currentTime() + Util.manufacturingTime(burgerType);
            while (Util.currentTime() < t1) wait(Util.waitTime(t1 - Util.currentTime()));

            // Put burger on shelf
            shelfs[burgerType].addLast(Util.currentTime());
            inProgress[burgerType]--;

            // Notify that burger is ready
            notifyAll();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }

    public synchronized void handleOrder() {
        try {
            // Wait for order
            while (customers.isEmpty()) wait();

            int burgerType = customers.removeFirst();

            // Order burger
            orders.addLast(burgerType);
            orderedBurgers++;
            notifyAll();

            // Remove old burgers
            cleanOldBurgers(burgerType);

            // Wait for burger
            int t0 = Util.currentTime();
            while (shelfs[burgerType].isEmpty()) wait();

            // Give burger to customer
            shelfs[burgerType].removeFirst();
            int t1 = Util.currentTime();
            int diff = t1 - t0;
            if (diff > Util.maxWaitingTime()) {
                //System.out.println("Miss");
                missedDeadlines++;
            }

            // Notify chefs to make more burgers
            notifyAll();
        } catch (InterruptedException e) {
            e.printStackTrace();
        }
    }


    public synchronized void newCustomer(int burgerType) {
```

```
            // Queue customer
            customers.addLast(burgerType);
            notifyAll();
        }


        public synchronized int[] gatherStatistics() {
            int a = thrownBurgers;
            int b = missedDeadlines;
            int c = orderedBurgers;
            thrownBurgers = 0;
            missedDeadlines = 0;
            orderedBurgers = 0;
            return new int[] {a, b, c, customers.size()};
        }

        private int missingOnShelf() {
            for (int i=0; i<Util.shelfs(); i++) {
                if (shelfs[i].size() + inProgress[i] < Util.keepOnShelf(i))
                    return i;
            }
            return -1;
        }

        private void cleanOldBurgers(int shelf) {
            if (shelfs[shelf].isEmpty()) return;
            while (Util.currentTime() - shelfs[shelf].getFirst() > Util.maxTimeOnShelf()) {
                shelfs[shelf].removeFirst();
                thrownBurgers++;
                if (shelfs[shelf].isEmpty()) return;
            }
        }
    }
```

7. No solution provided.