Tentamen
EDAF85/EDA698 – Realtidssystem (Helsingborg)

2017–10–26, 14.00–19.00

Det är tillåtet att använda Java snabbreferens och miniräknare, samt ordbok.
Det går bra att använda både engelska och svenska uttryck för svaren.

Den här tentamen består av två delar: *teori*, frågor 1-5 (18 poäng), och *programmering & design*, frågor 6-7 (12 poäng). För godkänd (betyg 3) krävs sammanlagt ca hälften av alla 30 möjliga poäng samt att det krävs ca en tredjedel av poängen i varje del för sig. För högsta betyg (5) krävs dessutom att en del av poängen uppnås med (del)lösningar till båda uppgifterna i *programmering & design* (preliminärt).

---

1. **Terms**
   Define and briefly discuss the following terms:
   a) *Critical section*, as used in concurrent programming.
   b) *Critical instance*, as used in response time analysis.

   (1p+1p)

2. **Deadlock**
   Below are three threads that are present in a system and share resources A, B, C, D and E.
   a) Is there a risk for deadlock in the system? Motivate your answer.

```
class R extends Thread {
  void run() {
    E.take();
    D.take();
    useED();
    D.give();
    E.give();
    A.take();
    useA();
    B.take();
    useAB();
    B.give();
    A.give();
  }
}
```

```
class S extends Thread {
  void run() {
    while (true) {
      B.take();
      useB();
      D.take();
      useBD();
      B.give();
      E.take();
      useDE();
      E.give();
      D.give();
    }
  }
}
```

```
class T extends Thread {
  void run() {
    C.take();
    D.take();
    A.take();
    useCDA();
    A.give();
    D.give();
    C.give();
  }
}
```

   b) Change the allocation order to avoid deadlock, but note that the operations used in useA (in thread R) and useB (in thread S) take long time, a thread calling them should not have allocated any unnecessary resource.

   (2p+1p)

3. **Priority inversion and inheritance**
A system scheduled strictly according to priorities can experience *priority inversion*, which can be handled by different versions of *priority inheritance protocols*.

   a) Explain the phenomenon *priority inversion*! Why is its occurrence problematic for a real-time system?

   b) *Dynamic (basic) priority inheritance* can be used to avoid priority inversion. How does this work?

   c) When applying the dynamic priority inheritance protocol, there might be *push-through blocking*. What does that mean and why is it not as bad as priority inversion?

   d) Name and explain two other inheritance protocols! What are their advantages and disadvantages in comparison to the basic inheritance protocol?

   (2p+1p+2p+2p)

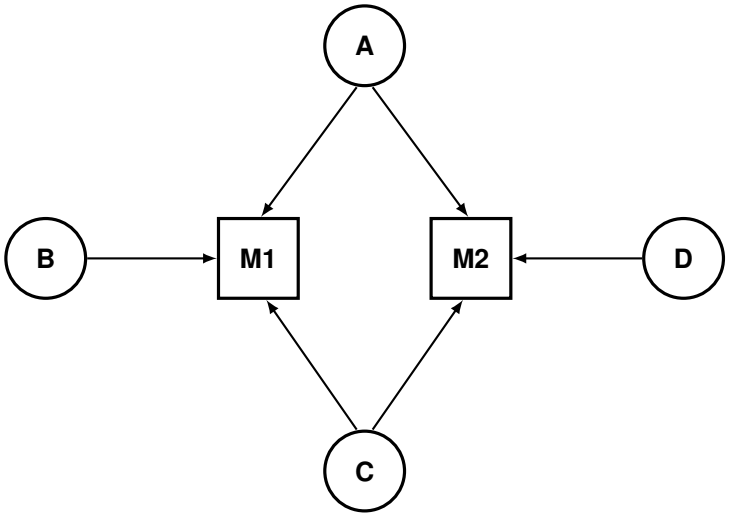
4. **Concurrency**
Your friend, Basil, working at an online banking software developer, has some knowledge of concurrent programming and decided to implement the accounts with the following Java class:

```
public class Account {
  private long balance = 0;
  public synchronized add(long money) {
    balance = balance + money;
  }
  public synchronized transfer_funds(long money, Account B) {
    this.add(-money);
    B.add(money);
  }
}
```

   However, Basil's code does not seem to work properly under heavy load conditions and he asks you for your help: What is the problem with this class? Motivate your answer.

   (1p)

5. **Schedulability**
A Java program consists of two monitors and four threads. The relevant part of this program is given below as a code listing. The program is running single-handedly on a single-processor hard real-time system using RMS and the basic inheritance protocol. The call structure from threads to monitors gives the following blocking diagram structure:



```
public class M1 {
  public synchronized void a1(){ /* 0.5 ms */ }
  public synchronized void b(){ /* 0.7 ms */ }
  public synchronized void c1() { /* 2.0 ms */ }
}

public class M2 {
  public synchronized void a2() { /* 1.0 ms */ }
  public synchronized void c2(){ /* 1.0 ms */ }
  public synchronized void d() { /* 2.1 ms */ }
}

 /* Thread A: T= 7.0 ms; C = 2.0 ms */
public class A extends PeriodicThread {
  private M1 m1;
  private M2 m2;
  public void perform() { m1.a1(); /* do other stuff */ m2.a2(); }
}

/* Thread B: T= 12.0 ms; C = 3.0 ms */
public class B extends PeriodicThread {
  private M1 m1;
  public void perform() { m1.b(); /* do other stuff */}
}

/* Thread C: T= 16.0 ms; C = 4.0 ms */
public class C extends PeriodicThread {
  private M1 m1;
  private M2 m2;
  public void perform() { m1.c1(); /* do other stuff */ m2.c2(); }
}

/* Thread D: T= 25.0 ms; C = 7.0 ms */
public class D extends PeriodicThread {
  private M2 m2;
  public void perform() { m2.d(); /* do other stuff */  }
}
```

Is the system schedulable? Motivate your answer, i.e. analyse the system by first computing the blocking times for each thread based on the call structure shown in the diagram and then applying a suitable analysis method to answer the question. Execution times for the monitor methods called by the threads are given in the code excerpt, as well as periods (T) and overall WCETs (C) for each thread.

(5p)

Hint:

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

6. **Programming: Alarm clock redundancy monitor**
Klas Kodare has started to work with the AlarmClock lab to once and for all ensure it is entirely thread safe, as he has found that the provided code skeleton has some work-around built in that does not guarantee this, but usually works under lab conditions. He finds that the device driver (hardware emulator) simply *signals* the application thread (usually the button listener thread) via a counting semaphore that new data are available, after storing these data in the package visible variables in the ClockInput interface (see the code excerpts and skeleton on pages 6 and following).

Under normal course lab circumstances this is sufficient to test the provided solutions against the conditions and expectations given in the task. However, considering high CPU load due to other processes running, it cannot be ensured, that *getChoice()* and *getValue()* return coherent data after the device driver and the button listening thread called give and take on the signal semaphore respectively. It is actually possible that *getChoice()* returns SET_ALARM but *getValue()* immediately thereafter returns a value belonging to SHOW_TIME.

Klas decides to first build a lock-free solution for the signalling / data acquisition. Then he also wants to have the possibility to run duplicate implementations based on the exact same input. The time handling (ticking each second) is straightforward to duplicate (directly via classes Thread and System), but the button input needs to be distributed to two solutions at the same time. He wants here not only to provide data and assume they are consumed on time, but make sure they are actually taken care of. That is, a thread that is reading data according to the new lock-free solution should provide them to two different alarm-clock application/solutions, but providing here means waiting for confirmation that each solution has got its own copy of the data to work on.

Your task
Help Klas to implement the lock-free data acquisition and implement the necessary monitor methods to make the duplication of data for two applications work. Note: You can work on a) and b) independently, i.e. it is safe to assume a correctly working method from a) that can be used in the solution to b).

a) Klas has started on an alternative *ClockInputNew* class (see code skeleton), where the main change is a new attribute *pending* that can be used to make sure that data have been read before they are overwritten. Also, he introduced a simple class with public attributes, *InputData*, to store all relevant data in one go. However, the implementation of the method *getData()* has not been finished. Implement the method *getData()* to make the *ClockInput* completely thread safe. You can specify in text/comments how the device driver should use the variable *pending*. Some hints:
   - As the device driver reacts to a hardware interrupt routine, it will always get CPU time.
   - The device driver should in any case not be blocked.
   - The receiving thread (button listener) that calls *getData()* will otherwise have the highest priority. If somehow its deadline is missed (the previous signal has not been taken care of before a new button press should be signalled), the device driver (providing the button press information) should be able to detect that (for instance by using the variable *pending*).

b) Implement the two monitor methods *provideData(...)* and *collectData(...)* in the *DuplicateClockInput* class. Add / change class attributes and return values / parameters to the methods as you find necessary. The methods need to assure that each new data item gets read (consumed) by two application threads before new data can be provided. This means that the method *provideData(...)* should block the caller in case the previously provided data have not been consumed by both consuming application threads. If the data have not been consumed twice within 1s however, this should be indicated with an error message stating which thread(s) did not consume the data. In this case, the currently stored data should be discarded and new data can be provided. The method *collectData(...)* that is called from the original button handler threads should be blocking until new data are available, and should be taking care of handling the double consumption. You may assume that the calling consumer threads (essentially looking like the button listener of your AlarmClock lab solution) provide an 'id-number' or unique name when calling the method.

(3p+6p)

7. **Design: Alarm clock solution based on message handling**
   Klas is rather happy with the new AlarmClock lab classes, but he thinks it would be nice to see whether the usually monitor-based solution can be transferred into a message-based solution, like the Washing Machine lab. Design a solution for this, i.e. find a way to handle the mix of periodic and sporadic (event based) activities in the AlarmClock lab with threads that can communicate via messages. Explain your ideas preferably with some combination of class and activity diagrams, like you did in the Washing Machine lab.

   (3p)

End of tasks, a short Java reference and code skeleton follow

---

A Short, specific Java-reference with potentially appropriate methods

Time handling, thread handling, waiting, etc

- class Math och class System:
  - public static double Math.random(): generates random value between 0 and 1
  - public static long System.currentTimeMillis(): current system time in milliseconds
- class Thread:
  - public static void sleep(long t): put a thread to sleep for t ms
  - public static void yield(): thread withdraws from running to be scheduled again
  - public void join(): wait for a thread to die
  - public void start(): start up the thread
  - public void run(): the method that describes a thread's actual task
- class Object:
  - public final void wait(): send a thread to wait (block)
  - public final void wait(long t): wait for at least t ms; note: can be woken up early
  - public final void notify(): notify first thread in line that a change has occurred
  - public final void notifyAll(): notify all waiting threads
- specific classes of the Lund Java Real-Time (ljrt) package:
  - RTEvent: a message to send information about a certain event
  - RTEventBuffer: a buffer for RTEvents
  - RTThread: thread with a mailbox of type RTEventBuffer
  - PeriodicThread: an RTThread with a given period. Implement perform instead of run

B Code skeleton for task 6

The original ClockInput class

```
package done;
import se.lth.cs.realtime.semaphore.*;

public class ClockInput {
    /**
     * Return values for getChoice.
     */
    public static final int SHOW_TIME   = 0;
    public static final int SET_ALARM   = 1;
    public static final int SET_TIME    = 2;

    /**
     * Construct the interface with semaphore reset.
     */
    public ClockInput() {
        anyButtonChanged = new CountingSem();
    }

    /**
     * Semaphore signaling when the user has changed any setting.
     */
    private Semaphore anyButtonChanged;

    /**
     * Get-method to access the semaphore instance directly.
     */
    public Semaphore getSemaphoreInstance() {
        return anyButtonChanged;
    }

    /* Package attributes, only used by the simulator/hardware. */
    boolean alarmOn;    // Alarm activation according to checkbox.
    int choice;         // The radio-button choice.
    int lastValueSet;   // Value from last clock or alarm set op.

    /**
     * Get check-box state.
     */
    public boolean getAlarmFlag() {
        return alarmOn;
    }

    /**
     * Get radio-buttons choice.
     */
    public int getChoice() {
        return choice;
    }

    /**
     * The set-value of the display is returned in the format hhmmss
     * where h, m, and s denote hours, minutes, and seconds digits respectively.
     */
    public int getValue() {
        return lastValueSet;
    }
}
```

The new InputData class

```
public class InputData {
  public boolean alarmOn;
  public int choice;
  public int lastValueSet;
}
```

The revised ClockInputNew class (excerpt)

```
public class ClockInputNew {

  /* mainly the same as ClockInput */

  /* Package attributes, only used by the simulator/hardware. */
  boolean alarmOn;    // Alarm activation according to checkbox.
  int choice;         // The radio-button choice.
  int lastValueSet;   // Value from last clock or alarm set op.

  /**
   * For use by device driver; as you explain...
   */
  boolean pending; // Suggestion ...

  /**
   * Method to be developed for task 6 a)
   * @return consistent input state, or null for error.
   */
  public InputData getData() {
    /* Task 6a): Your code goes here */
  }
}
```

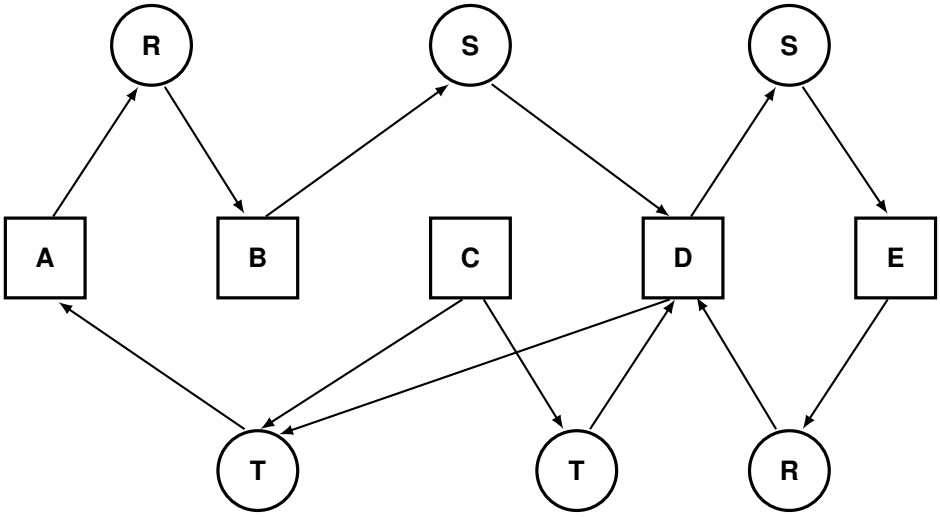
The new monitor class DuplicateClockInput

```
public class DuplicateClockInput {
  private InputData in;

  /* add suitable attributes for the handling of two consumers, timing, etc */
  /* add a constructor if you find it necessary,
   * otherwise explain your choices in text
   */

  /*
   * Task 6 b): add thread-safe methods provideData(...) and collectData(...)
   * assure they have suitable return values and parameters
   */
}
```

Kortfattad lösning

1. - *critical section*: a sequence of code that needs to execute in an atomic fashion (un-interruptible), or the concurrency breaks.
   - *critical instance*: the worst possible case for jobs wanting to execute in a system. In particular, when they all arrive at the same time.

2. a) Draw a resource allocation graph. There are multiple cycles, hence possible dead-locks. Shortest cycle requires only one each of threads R and S. Another cycle requires one each of R, S, T.



   b) Reverse allocation order of E, D in thread R. Change allocation order of C, D, A in thread T to A, C, D. This reverses the direction of the edges, breaking the cycles. Do NOT reverse order of A, B in thread R or B, D in thread S, to not lock resources for long periods unnecessarily.

3. a) To be found in the lecture slides. The time that hp-thread might be blocked by one or even several mp-thread(s) is not computable, hence, time guarantees might not be possible to keep.
   b) To be found in lecture slides.
   c) The blocking time caused by push-through blocking is computable and contained, hence, the analysis can consider it and guarantees can be given.
   d) Priority ceiling and direct inheritance. Explanation in the lecture slides, as well as pros (avoid multiple blockings) and cons (more overhead).

4. Two threads operating on the same two accounts A and B, calling A.transfer_funds(x, B) and B.transfer_funds(y, A) may lead to deadlock, since they both wait for the other object lock to get freed.

5. Call graph:
   Worst case execution times:
   $C_A = 0.5 + 0.5 + 1.0 = 2.0ms$
   $C_B = 0.7 + 2.3 = 3.0ms$
   $C_C = 2.0 + 1.0 + 1.0 = 4.0ms$
   $C_D = 2.1 + 4.9 = 7.0ms$

   RMS blocking:
   $B_D = 0ms$
   $B_C = 2.1[d] = 2.1ms(direct)$
   $B_B = 2[c1] + 2.1[d] = 4.1ms(direct + indirect)$
   $B_A = max(0.7[b], 2.0[c1]) + max(1.0[c2], 2.1[d]) = 4.1ms(direct)$

RMS response times:
$R_A = 2.0 + 4.1 = 6.1 \leq 7.0$

$R_B = 3 + 4.1 = 7.1$
$R_B = 7.1 + \lceil \frac{5.1}{7} \rceil 2 = 9.1$
$R_B = 7.1 + \lceil \frac{7.1}{7} \rceil 2 = 11.1$
$R_B = 7.1 + \lceil \frac{9.1}{7} \rceil 2 = 11.1 \leq 12.0$

$R_C = 4.0 + 2.1 = 6.1$
$R_C = 6.1 + \lceil \frac{6.1}{7} \rceil 2 + \lceil \frac{6.1}{12} \rceil 3 = 12.1$
$R_C = 6.1 + \lceil \frac{9.1}{7} \rceil 2 + \lceil \frac{9.1}{12} \rceil 3 = 13.1$
$R_C = 6.1 + \lceil \frac{13.1}{7} \rceil 2 + \lceil \frac{13.1}{12} \rceil 3 = 16.1 > 16.0$

$R_D = 7$
$R_D = 7 + \lceil \frac{7}{7} \rceil 2 + \lceil \frac{7}{12} \rceil 3 + \lceil \frac{7}{16} \rceil 4 = 16$
$R_D = 7 + \lceil \frac{16}{7} \rceil 2 + \lceil \frac{16}{12} \rceil 3 + \lceil \frac{16}{16} \rceil 4 = 23$
$R_D = 7 + \lceil \frac{23}{7} \rceil 2 + \lceil \frac{23}{12} \rceil 3 + \lceil \frac{19}{16} \rceil 4 = 29 > 25.0$

The system is not schedulable since the response times for C and D are greater than the corresponding periods (ok to stop after C).

6. a) The variable pending should be set to *true* when the button for alarm_set or time_set is released (the choice switches back from setting time or alarm time), i.e., the complete data chunk is consistent and it can be assumed that the value actually belongs to the correct choice. If already true, the device driver should not write any new data into the choice / value fields. Pending is then set back to *false* when a complete data chunk is read.

```
/**
 * Method to be developed for task 6 a).
 * @return consistent input state, or null for error.
 */
public InputData getData() {
  InputData ans = new InputData();
  anyButtonChanged.take();
  if (pending) {
    ans.alarmOn = alarmOn;
    ans.choice = choice;
    ans.lastValueSet = lastValueSet;
    pending = false;
    return ans;
  } else {
    return null;
  }
}
```

b) The class DuplicateClockInput with methods provideData() and collectData() should look (roughly) like this:

```
public class DuplicateClockData {
  long dataInTime;
  InputData inData;
  boolean[] read = {false, false};
  // ...

  /**
   * Methods to be developed for task 6 b).
   */
  public synchronized void provideData( InputData data) {
  long time = System.currentTimeMillis();
```

```
    long tDiff;

    while( (!read[0] || !read[1])
           && ( tDiff = System.currentTimeMillis() - dataInTime) < 1000)
           wait( tDiff);

      if( !read[0] || !read[1]) {
        System.out.println( "discarding data set with timestamp" + dataInTime);
        if( !read[0]) System.out.println( "Thread with id 0 did not read");
        if( !read[1]) System.out.println( "Thread with id 1 did not read");
      }

      dataInTime = time;
      inData = data;
      read[0] = false;
      read[1] = false;

      notifyAll();
    }

    public synchronized InputData collectData( int threadID) {
      while( read[threadID]) wait();
      InputData res = inData;
      read[threadID] = true;
      notifyAll();
      return res;
    }
  }
```

7. There should be a periodic thread (like any of the controllers in the WashingMachine lab), i.e., a clock-controller, that takes care of the clock tick and the output to the display. It can receive timeData-messages containing an älarm flag on/off", as well as a timeänd an älarm time"entry. Another (looping) thread takes care of reacting to the button presses in the hardware. If new data is available, it generates a timeData-message, fills it with the relevant data and sends it directly to the clock-controller, instead of into the data-buffer. The design should explain the messages and these two threads.