Tentamen
EDA698 – Realtidssystem (Helsingborg)

2015–10–30, 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. **Waiting, timeout and notification**
   Assume that a system with several threads makes use of a Java-monitor, i.e., the built-in mechanisms for mutual exclusion and signalling (modifier *synchronized* and method calls *wait(), wait(long time), notify(), and notifyAll()*) can be applied.
   a) Explain briefly when (how) a thread either calling *wait()* or *wait(long time)* on a monitor will be woken up for each of the method calls!
   b) Why is it not (always) possible to tell after the fact how a thread was woken up, when it previously had called *wait(long time)*?
   c) Discuss why it is not (and should not be) necessary to be able to know this!

   (1p+2p+1p)

2. **Semaphores**
   In the course mainly two types of semaphores were discussed and applied, *mutex semaphores* for mutual exclusion and *counting semaphores* for signalling.
   a) Use pseudocode to describe the counting semaphore principle.
   b) Describe one advantage with using a counting semaphore in a program that handles external events (button presses) instead of using a synchronized monitor-method that blocks the caller until the event data is available.
   c) Describe two reasons for having different semaphore classes for signalling and mutual exclusion semaphores respectively!

   (1p+1p+1p)

3. **Resource allocation**
   A control program written in Java features two thread classes. The threads perform the following sequences of operations on four global mutex semaphores A, B, C and D (only the semaphore operations are shown).

Thread 1	Thread 2
B.take();	A.take();
A.take();	C.take();
A.give();	C.give();
C.take();	A.give();
D.take();	D.take();
B.give();	B.take();
C.give();	B.give();
D.give();	D.give();

a) Draw a resource allocation graph of the system.

b) How many instances of each thread class can be alive with the system being deadlock free?

(2p+1p)

4. During the course two different (cyclic) scheduling strategies were discussed.

   a) Explain these two different strategies with about one sentence each!

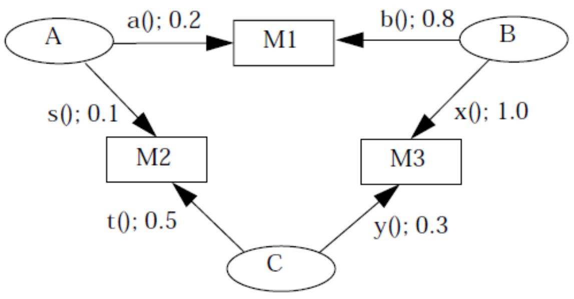
   b) Give one advantage and one disadvantage for each of the scheduling strategies.

(1p+2p)

5. **Schedulability and blocking**
   A real-time system is to be implemented in which three processes are to run. The T-values describe the period times, C the worst case execution times (WCET) for each of the threads respectively.

Thread	T [ms]	C [ms]
A	8	4
B	10	3
C	16	5

   a) Calculate maximum blocking time assuming RMS and the basic inheritance protocol being in effect. The threads A, B and C call monitor operations in the monitors M1, M2 and M3. The graph below shows monitor method WCETs in milliseconds.



   b) Is the system schedulable using RMS and basic inheritance protocol?

   c) What would the priorities of the threads be according to DMS? Assume the maximum allowed response time (deadline D) as equal to double WCET.

(2p+2p+1p)

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 – Back-drivable Robot Joint and Event Buffer**
The robot and graphics expert Mathias has developed a well-structured robot control system that he successfully used for virtual robots by visualising robot motions before carrying them out physically. The desired motion (path) of each robot joint is computed as a so called trajectory (time-based series of position references), buffered and finally executed stepwise on the robot arm.

Mathias assumes a thread class *TrajecHandler* to generate the trajectory data for each joint's desired motion, which is holding a reference to a respective external buffer, where it can place and a consumer (e.g., the *ServoHandler*, see below) can fetch their data. The events buffered are of type *PosEvent* extending *RTEvent* and each event hosts a position reference. The thread *ServoHandler*, which performs the feedback control of the joint motor, can then retrieve its next position reference (or setpoint) by calling fetch on the buffer.

The robot and mechatronics expert Klas happily started to use the virtual control system software as the embedded software of a new robot motion controller. He discovered rather quickly, that the simple design with only assuming one mode of operation did not suit his needs, as there can be different types of failures with the physical robot, the human user might want to teach robot motions by moving the actual robot arm and for a physical robot it must be possible to have an explicit STOP mode.

Klas decides that he needs a more complex buffer which he calls a *BackPropBuffer*, that can handle different modes of operation, including a TEACH mode. In total this means there now are the following modes of operation:

`PATH` Normal path following with each `PosEvent.pos` being a position reference to be reached at time `PosEvent.timestamp` (i.e., one step in the trajectory)

`JAM` The was too big a deviation in actual position from the planned references, no new reference should be retrieved and the joint should simply keep its current position until the mode is switched by the user

`TEACH` The `PosEvent` buffering should go the opposite direction (posting becomes fetching and vice versa)

`STOP` Motion (is or will very soon be) stopped, the buffer should be emptied, threads posting or fetching position references should be able to handle the mode switch

Apart from the threads *TrajecHandler* and *ServoHandler* mentioned above, there is a thread *Supervisor* that depending on operator input can change the operational mode by calling a respective method.

You are now supposed to develop a motion-buffering monitor class *BackPropBuffer* that supports the back-propagation of positions from the servo control upstreams to the trajectory generator, and can at least indicate other modes to event posting and fetching threads.

Implement the following methods according to their respective specification. A skeleton for the monitor and an explanation of the calling threads is found on pages 5 and following. You are allowed to change names and arguments (including return type) for the methods you implement. Make in such cases sure that the callers of the methods match your changes as well.

a) Method *updateMode(MoveMode mode)*: Should change the mode according to its parameter. In case the new mode is STOP, the caller should be blocked until all threads posting or fetching events have confirmed the STOP. Expected to be called by *Supervisor*.

b) Method *post(PosEvent nextPos)*: If in PATH mode, nextPos should be stored into the buffer. If in TEACH mode, the next available position reference should be returned. When in JAM or STOP mode, the buffer should indicate this with the help of the returned PosEvent respectively.

c) Method *fetch(PosEvent actPos)*: If in PATH mode, the next available position reference from the buffer should be returned. In case the retrieved position reference differs more than maxlag from the position referenced by actPos, the mode should change to JAM. Otherwise, current position and timestamp in the buffer should be updated according to actPos. If in TEACH mode, actPos should be buffered as if posted. When in JAM or STOP mode, the buffer should indicate this with the help of the returned PosEvent respectively.

(2p+3p+3p)

7. **Design - Supporting physical human-robot interaction**

Based on experiences from the system developed in the previous problem, the robot and cognition expert Elin wants to improve human-robot interaction in the case that the robot works in the same workspace as manual (human) work takes place, and then also utilize physical interaction such that the human can move the robot hand in order to adjust a not quite correct motion. Depending on available sensors, algorithms and system software, there are many ways to accomplish such physical human-robot interaction.

As a starting point, we assume a simple approach in which each joint is commanded according to the previous programming task. The idea is now to enable better handling of user involvement in the motion control of the robot according to the following specifications:

- Each robot joint is controlled by a servo-control thread, which internally handles all sensing and control (already existing). That thread class, which we can call ServoHandler, is accepting the setpoints via the `putEvent` method, i.e., we can now assume that there is an internal mailbox available. When a setpoint cannot be reached, as detected by the existing servo control, the respective ServoHandler object/thread sends a `JamEvent` back to the source of the setpoint event.
- When one joint is jammed it of course already has stopped or slowed down, but immediately thereafter all other (typically five or six) other joints should stop. Since some robots have the servo control distributed to a CPU that is embedded together with the mechanics/motor of the joint, this means that propagation of both setpoint events and jam events is over a network, with blocking read and non-blocking write.
- To better separate between a jammed motion due to unforeseen contact with some workpiece, and that of a human operator grabbing the hand (end-effector) of the robot for doing some corrections, there is a push button at the end-effector. Whenever that button is pushed or released, there should go a message in terms of an event being sent over the network, from the CPU managing one of the joints (typically the last one closest to the end-effector) to the central control computer.
- To make the robot safe, for instance if some hardware breaks, there must be fault tolerance/-detection. For communication to and from the (distributed embedded) ServoHandler threads there are double (redundant) network cables, which has to be reflected by redundant threads and error detection in the software.
- The central control should coordinate the robot joints by maintaining a back-drivable buffer (according to previous task) per joint, with a respective set of threads (like TrajecHandler). Since the application is data-flow oriented, there is the issue of managing the operational state, which is a key issue in the design of the central computer.

Provide a design (figures and descriptions) of a software system that would be a good basis for implementing the specified features.

(4p)

A Code skeleton for task 6

```
/*
 * The monitor class you are supposed to implement the required methods in
 */
public class BackPropBuffer extends PlainBuffer {

  double currentPos;        // The current joint position in radians
  long time;                // The timestamp in which currentPos was reached
  MoveMode mode;            // The current mode (PATH; TEACH; JAM; STOP)
  double maxlag = Math.PI;  // Deviation (lag) within one motor turn is OK
  /* add suitable attributes in case needed */

  public BackPropBuffer( double maxlag) {
    this.maxlag = maxlag;
  }

  /*
   * updates the mode according to Supervisor’s handling of external events
   * should indicate whether a mode change occurred
   * should block until incoming STOP mode has been confirmed for
   * posting and fetching threads
   */
  public synchronized MoveMode updateMode(MoveMode mode) {

  // TASK part a)

  }

  /*
   * post a message (event) or receive the latest posted one,
   * depending on mode. Called by TrajecHandler
   * returned PosEvent should reflect mode
   *
   * should block if full
   */
  public synchronized PosEvent post( PosEvent nextPos) {

  // TASK part b)

  }

  /*
   * receive a message (event) from the buffer or post actPos,
   * depending on mode. Called by ServoHandler
   * returned PosEvent should reflect mode
   *
   * should block if empty
   */
  public synchronized PosEvent fetch(PosEvent actPos) {

   // TASK part c)

  }
}
```

```
/*
 * The event class describing the timestamped position references
 */
public class PosEvent extends RTEvent {
  public double pos; // In radians for rotational joint, public for convenience
  public PosEvent( Object source, long timestamp, double pos) {
      super( source, timestamp);
      this.pos = pos;
  }
}

/*
 * The original PlainBuffer class (excerpt, relevant methods only)
 */
public class PlainBuffer extends RTEventBuffer {
  /* ... */

  /* blocking method for posting */
  public synchronized void doPost( PosEvent pe) { /*...*/}

  /* blocking method for fetching */
  public synchronized PosEvent doFetch() {  /*...*/}

  /* return true if empty */
  public synchronized boolean isEmpty() {  /*...*/}

  /* return true if full */
  public synchronized boolean isFull() {  /*...*/}

  /* empty the buffer (erase all buffered events) */
  public synchronized void flush() {  /*...*/}

  /* ... */
 }


/*
 * Thread class TrajecHandler
 */
public class TrajecHandler extends PeriodicThread {
  BackPropBuffer jointBuffer; // the reference to the buffer you should implement
  RobotPath pathRef;          // a reference to the full path
  PosEvent nextPos, retPos;

  public TrajecHandler( BackPropBuffer bpb, RobotPath pathRef, long period) {
      super( period);
      this.jointBuffer = bpb;
      this.pathRef = pathRef;
      nextPos = null;
      retPos = null;
  }

  public void perform() throws InterruptedException {
      nextPos = generateNextSetpoint();
      retPos = jointBuffer.post( nextPos);
```

```
        if( retPos != null){ // something is coming back
            if( retPos.source != null && retPos.source instanceof ServoHandler) { // TEACH mode
                addToPath( retPos);
            }  else {
                /* handle other situations */
            }
        }
    }

    /*
     * private help method to generate the next joint
     * position reference according to original path;
     * source is set to this
     * in case path is empty, the method
     * returns a default PosEvent
     */
    private PosEvent getNextSetpoint() { /*...*/}

    /*
     * private help method to add the next joint
     * position reference into a robot path, which can then be
     * replayed
     */
    private void addToPath( PosEvent pos) { /*...*/}
  } // end of class TrajecHandler

  /*
   * Thread class ServoHandler
   */
  public class ServoHandler extends PeriodicThread {
    BackPropBuffer jointBuffer; // the reference to the buffer you should implement
    RobotClient robRef;         // a reference to the robot hardware
    PosEvent actPos, retPos;

    public ServoHandler( BackPropBuffer bpb, RobotClient robRef, long period) {
        super( period);
        this.jointBuffer = bob;
        this.robRef = robRef;
        actPos = null;
        retPos = null;
    }

    public void perform() throws InterruptedException {
        actPos = new PosEvent( this, robRef.getJointPos(), System.currentTimeMillis());
        retPos = jointBuffer.fetch( nextPos);

        if( retPos != null) { // something comes back
            if( retPos.source == null) { // no useful source for an event, better stop the motor
                robRef.setMotorVelocity( 0.0);
            } else if( retPos.source instanceof TrajecHandler) {
                robRef.setMotorVelocity( generateVelocity( retPos.pos, retPos.timestamp));
            } else {
                /* handle other situations */
            }
        }
    }
  /*
   * private help method to generate the next joint
   * motion data (velocity)
   */
  private double generateVelocity() { /*...*/}
} // end of class ServoHandler

/*
 * Thread class Supervisor
 */
public class Supervisor extends Thread {
  BackPropBuffer jointBuffer; // the reference to the buffer you should implement
  ButtonClient buttonRef;     // a reference to the interface for control mode switches

  public Supervisor( BackPropBuffer bpb, ButtonClient buttonRef) {
      this.jointBuffer = bpb;
      this.buttonRef = buttonRef;
  }

  public void run() throws InterruptedException {

      MoveMode mode = STOP;
      jointBuffer.updateMode( mode);

      while( !isInterrupted()) {
          mode = buttonRef.awaitButtonPressed();
          jointBuffer.updateMode( mode);
      }
  }
} // end of class Supervisor
```

B Short, specific Java-reference with potentially appropriate methods

Tidshantering, trådhantering, väntelägen, etc

- klass Math och klass System:
  - public static double Math.random(): ger ett slump-värde mellan 0 och 1.
  - public static long System.currentTimeMillis(): aktuell systemtid i millisekunder
- klass Thread:
  - public static void sleep(long t): försätta en tråd i viloläge under t ms
  - public static void yield(): dra tråden tillbaka och sätt i kön för ny schemaläggning
  - public void join(): vänta att tråden dör
  - public void start(): starta upp tråden
  - public void run(): trådens arbete
- klass Object:
  - public final void wait(): försätta en tråd i vänteläge
  - public final void wait(long t): försätta en tråd i vänteläge under max t ms. Kan dock väckas ändå!
  - public final void notify(): meddela nästa tråden i vänte-kön att ett tillstånd har ändrats
  - public final void notifyAll(): meddela alla trådar i vänteläge att ett tillstånd har ändrats

Kortfattad lösning

1. a) wait(): woken up by notify(), when this is the first thread in waiting-queue, or notifyAll() no matter which position it has in queue. wait(long time): same as wait() plus - if not earlier woken up by notify() or notifyAll() - when time has passed, i.e., through timeout.

   b) If a call to notify() or notifyAll() wakes the thread close to its timeout, but it is not actually running before also the timeout has passed, it is not possible to tell which one occurred first due to notify having no state or information of who called it. The same issue arises, when the thread's timeout is passed, the thread is scheduled for running, but another thread modifies the monitor state and calls notify() or notifyAll(). This can involve both situations where the timeout is combined with another condition, or one, where it is the sole condition for waking up.

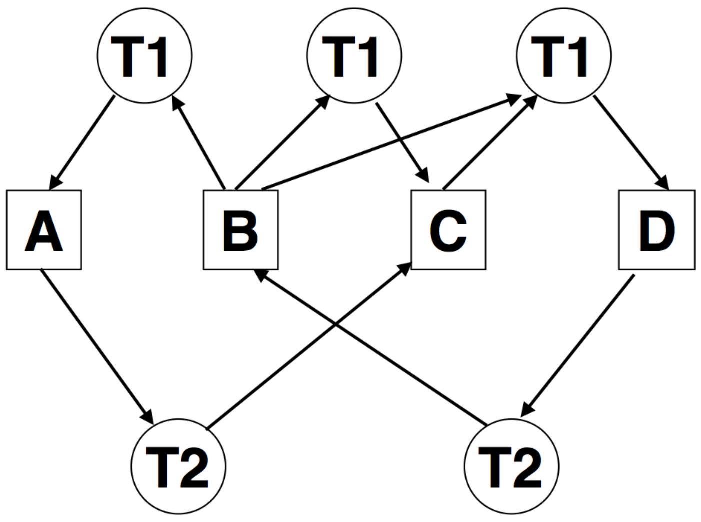
   c) The program logic should not make a difference - there should always be a loop that makes sure how to handle the fulfilment of different conditions, one of which can be the timeout. If in the combined case a condition that is not related to the timeout has become true and a certain action is required, this should also hold, when additionally the timeout occurred and vice versa. If on the other hand only the timeout or only the condition fulfilment occurred, this particular action might not be necessary to take. If, as a third option, an early wake-up is only a disturbance of a required time period to stay suspended, but the timeout is passed when the conditions are checked again, the thread should have been woken up anyway due to its timeout being passed. No need to know, and some issue in the logic, if there seems to be the need to know.

2. a)
```
int counter = 0;

method: take //atomic, hardware interrupts off
 if counter == 0
     suspend calling thread and put calling thread in queue
 counter--;

method: give (atomic, hardware interrupts off)
 if counter == 0 && thread(s) waiting
     wake up next thread in queue
 counter++;
```

   b) Less overhead: implementing a blocking method requires application controlled mutual exclusion (synchronized) plus the overhead for the waiting suspension and handling of (potentially immature) wake-ups (loop and call to notifyAll). The give and take methods should be implemented as atomic operations, and the scheduler's queue handling on the semaphore allows for effective code, as only threads waiting for give-calls on exactly this semaphore get in this queue. Also ok: Monitor would be blocked for other buttons when button press data is registered for one press.

   c) A signal should be possible to be sent and received or waited for many times and in random order. This requires a counter, which can even be initialized with a starting value greater than 0. If on the other hand mutual exclusion is needed, it should not be possible to signal availability more than once, hence the counter should be limited to the values 0 and 1, or be replaced by a boolean, to guarantee the expected effect. Additionally, signal sender and receiver do not have to be the same thread, but this must be the case for mutual exclusion, hence the mutual exclusion semaphore should provide a check of the caller of give being the expected earlier caller of take.

3. a) Resource allocation graph



b) It is not possible to run two threads (one of each class) at all.

4. a) Static scheduling is an off-line approach in which you manually generate a fixed execution table or calendar. The table states which thread should run at any given moment. The table repeats cyclically. Dynamic scheduling is applied ad-hoc, threads are scheduled on-line either according to priorities or following a round-robin approach.

   b) Static: Examples of advantages are that it is easy to implement on a bare computer, that it is easy to do scheduling analysis, and that inter-thread communication can be performed without complicated synchronization. Disadvantages include that it can be difficult to construct the schedule and that the schedule might have to be remade if the system changes. Dynamic: Examples for advantages are that it is flexible, that it adapts automatically to new threads in the system and that it can optimise the CPU usage to a certain extent. Examples for disadvantages are that it generates more overhead for the handling of priorities etc, and that there is no guarantee for every thread always meeting its deadlines, if the system is extended spontaneously.

5. a) Highest to lowest priority: B A C

   b) RMS blocking:
   $B_C = 0$
   $B_B = max(0.3[y], 0.5[t]) = 0.5$
   $B_A = 0.5[t] + 0.8[b] = 1.3$

   c) RMS response times:
   $R_A = 4 + 1.3 = 5.3$
   $R_B = 3 + 0.5 = 3.5$
   $R_B = 3 + 0.5 + \lceil \frac{3.5}{8} \rceil 4 = 7.5$
   $R_B = 3 + 0.5 + \lceil \frac{7.5}{8} \rceil 4 = 7.5$
   $R_C = 5$
   $R_C = 5 + \lceil \frac{5}{8} \rceil 4 + \lceil \frac{5}{10} \rceil 3 = 12$
   $R_C = 5 + \lceil \frac{12}{8} \rceil 4 + \lceil \frac{12}{10} \rceil 3 = 19$
   $R_C = 5 + \lceil \frac{19}{8} \rceil 4 + \lceil \frac{19}{10} \rceil 3 = 23$
   $R_C = 5 + \lceil \frac{23}{8} \rceil 4 + \lceil \frac{23}{10} \rceil 3 = 26$
   $R_C = 5 + \lceil \frac{26}{8} \rceil 4 + \lceil \frac{26}{10} \rceil 3 = 30$
   $R_C = 5 + \lceil \frac{30}{8} \rceil 4 + \lceil \frac{30}{10} \rceil 3 = 30$
   The system is not schedulable with RMS (C exceeds its deadline almost with factor 2).

6. OBS: this is a sketch.

```
package backdrivebuffer;
import se.lth.cs.realtime.event.*;

class BackpropBuffer extends PlainBuffer {

  double pos;
  long time;
  MoveMode mode;
  double maxlag = Math.PI; // Within one motor turn is OK.
  boolean postACK = false;
  boolean fetchACK = false;

  public synchronized MoveMode updateMode(MoveMode mode) {
    if (this.mode == mode) return mode;
    this.mode = mode;
    notifyAll();
    if( this.mode == MoveMode.STOP) {
         fetchACK = postACK = false;
    notifyAll();
    while ( !postAck && !fetchACK) wait();
    super.flush();
}
    return mode;
  }

  public synchronized PosEvent post(PosEvent nextPos) {
    PosEvent ans = null;
    switch (mode) {
    case MoveMode.PATH:      // Move along programmed path...
      while (mode == MoveMode.PATH && super.isFull()) wait(); //ok without this line,
                                                              //but line not without mode check
      if( mode == MoveMode.PATH ) doPost(nextPos);  // ok without mode check
      break;
    case TEACH:     // Accept where we are and report back.
      while (mode == MoveMode.TEACH && super.isEmpty()) wait();
      if( mode == MoveMode.TEACH ) ans = (PosEvent) doFetch();
      break;
    case JAM:       // Motion deviates from the path.
      ans = new PosEvent( this, time, currentPos);
      break;
    case STOP:      // Hold on, for standby or state change.
      postACK = true;
      notifyAll();
      ans = new PosEvent( null, time, currentPos);
      break;
    }
    return ans;
  }

  public synchronized PosEvent fetch(PosEvent actPos) {
    PosEvent ans = null;
    switch (mode) {
    case PATH:
      while (mode == MoveMode.PATH && super.isEmpty()) wait();
```

```
          if( mode == MoveMode.PATH ) ans = (PosEvent) doFetch();
          currentPos = ans.pos;
          if (Math.abs( actPos.pos-pos) > maxlag) {
            mode = MoveMode.JAM;
            ans = actPos;
            notifyAll();
          }
          break;
        case TEACH:
          while (mode == MoveMode.TEACH && super.isFull()) wait();
          if( mode == MoveMode.PATH ) doPost(actPos);
          break;
        case JAM:
          ans = actPos;
          break;
        case STOP:
          fetchACK = true;
          notifyAll();
          ans = new PosEvent( null, time, currentPos);
          break;
        }
        return ans;
      }
    }
```

7. The important parts that needed to become clear from drawing and description were roughly the following:
   1. There were servo handler threads (ServoH) for each joint connected to the respective hardware / motor controllers, which would communicate over a network with the central control computer (CentralController) and could hence spread "JAMEvents"if necessary. Communication to the ServoH-threads is via the mailbox of each thread. Covers item 1 and 2 of the list.
   2. There must be a button press handling thread connected to the respective buttons, communicating a respective press through the network to the central controller for mode switches.
   3. It must be clearly stated that there is redundancy in the system - doubled setups, double hardware (cables) etc, and some sketch of how to handle this.
   4. There must be a BackPropBuffer for each joint which is accessed from the TrajecH-threads according to the previous task. The buffers are used to reflect the operational state of the system.

   If there was a drawing, but no description at all, this led to some reduction of the overall points for this task.