Exam
Concurrent and Real-Time Programming

2010–12–18, 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.

To pass the exam (grade 3), you have to solve most of the first 6 problems (which are more of a theoretical type), and you have to develop an acceptable solution to problem 7 (programming). At least a partial solution to problem 8 is required for the highest grade (5).

---

1. When our OS or real-time kernel schedules a new thread for execution, there is the notion of a *context switch*.

   a) What is happening during a context switch?

   b) Why could context switching times be longer (compared to scheduling without pre-emption) in case of preemptive scheduling?

   c) When is the method `java.lang.Thread.yield()` useful?

   d) You are responsible for a product with embedded computing, and to improve performance you decided to use a faster CPU, which additionally provides floating-point computations in hardware. There are frequently changing numerical computations taking place in different threads and preemptive scheduling is used. Surprisingly, the same software runs slower on the faster CPU. How could that be?

(4p)

2. In the Alarm-Clock lab, and in many other systems, we have the need to keep track of the time without drift. Explain:

   a) How come (even if nextSecond only takes a few micro-seconds) there is a risk for drift (such that some seconds become for instance one or a few milliseconds longer and those extra delays add up) if you increment seconds by the following loop?

```
while (!isInterrupted()) {
  sleep(1000); display.nextSecond();
}
```

   b) Real-time Roger had the above implementation and tested his system for a week without any drift, while Critical Klas had a drift of a few seconds per day on his old laptop (running Windows XP)? How could that be?

   c) Rewrite the code snippet such that both Roger and Klas get happy (i.e., no drift as explained in the course booklet).

   d) With the new implementation, even if there is no drift, will each second (or each new day) start (according to the displayed time) exactly at the right millisecond? If not, what would be required to accomplish that?

(1+1+2+1 p)

3. a) Mutual exclusion between threads in the same program, as we normally have, can be accomplished by synchronized methods. In case of mutual exclusion between programs (or OS processes or threads running in different programs, possibly on different processors with shared memory) there is an algorithms according to the following pseudo code:

```
flag[0] := false
flag[1] := false
turn := 0   // or 1
```

p0:

```
    flag[0] := true
    while flag[1] = true {
        if turn ≠ 0 {
            flag[0] := false
            while turn ≠ 0 {
            }
            flag[0] := true
        }
    }

   // critical section
   ...
   turn := 1
   flag[0] := false
   // remainder section
```

p1:

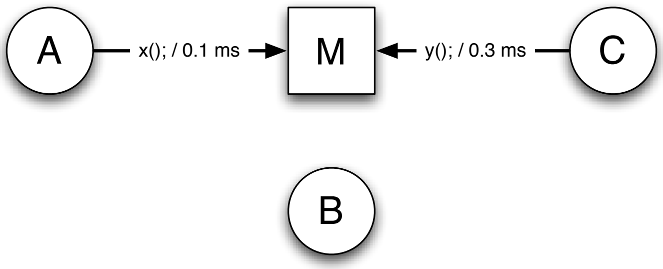
```
    flag[1] := true
    while flag[0] = true {
        if turn ≠ 1 {
            flag[1] := false
            while turn ≠ 1 {
            }
            flag[1] := true
        }
    }

    // critical section
    ...
    turn := 0
    flag[1] := false
    // remainder section
```

What is the name of this algorithm?

b) What disadvantage would there be if we use that algorithm for mutual exclusion between two threads in the same program/JVM?

(2p)

4. The three threads A, B, and C with highest priority in a real-time system communicate with each other using a monitor M according to the figure below (as can be seen, thread B is not sharing any resources with thread A or C):



The monitor operations x and y are called by thread A and C as illustrated in the figure. For each monitor operation, the maximum required execution time is shown in the figure. Thread A has the highest priority and C has the lowest priority among the three threads. The threads do not communicate or interact with lower priority threads other than as illustrated in the figure.

One of the threads in the system can experience something called *push-through blocking*.

a) Which of the three treads can experience push-through blocking? Briefly describe an execution scenario which illustrates the situation causing push-through blocking.

b) How long is the maximum blocking time caused by push-through blocking in the system?

(2p)

5. a) Consider a real-time system consisting of four independent, periodically executing, threads with characteristics according to the table below (C = worst-case execution time, T = period, C/T = CPU utilization). Rate monotonic scheduling is used to assign priorities to the threads.

Thread	C (ms)	T (ms)	C/T
A	2	8	0.25
B	3	15	0.20
C	4	20	0.20
D	?	30	

What is the *maximum possible execution time* (C) for thread D that still guarantees schedulability in the worst case (ignoring costs for context switches etc)?

b) For the same system as above, suppose we only know the CPU utilization for each of the threads A to C and the period of thread D:

Thread	C (ms)	T (ms)	C/T
A			0.25
B			0.20
C			0.20
D	?	30	

What is now the *maximum possible execution time* (C) for thread D that can always be safely assumed to guarantee schedulability (ignoring costs for context switches etc)? Answer with the number of *whole* milliseconds possible, i.e., round your answer downwards to the nearest lower integer.

n	1	2	3	4	5	...	∞
$n(2^{1/n}-1)$	1.000	0.828	0.780	0.757	0.743	...	0.693

(3p)

6. In a Java program we find two types of threads, T1, and T2, which in their `run()`-methods execute the following linear sequences of semaphore operations on the four mutex semaphores A, B, C and D. Intermediate code dependent on the semaphores are represented by function calls on the form "`useXY();`", where "`XY`" denotes that the semaphores X and Y must be taken when the code is executed. The methods on the form "`useXY();`" is mutually independent regarding synchronization.

T1	**T2**
A.take();	C.take();
C.take();	D.take();
useAC();	useCD();
D.take();	C.give();
useACD();	B.take();
D.give();	useBD();
C.give();	B.give();
A.give();	D.give();
B.take()	
A.take();	
useAB();	
A.give();	
B.give();	

a) Draw a resource allocation graph for the system.

b) Suppose there are only one instance of T1 and one instance of T2. Is there a risk for deadlock in the system? Motivate your answer!

(2p)

7. Timed Event Distribution

Introduction

When using event-based communication in a one-to-one manner, one thread calls `putEvent` of the thread that should receive the event, and that thread then asynchronously (with respect to the calling/providing thread) fetches the event from its mailbox for further processing. In the case of many-to-one, many different threads can call `putEvent` of the receiving thread.

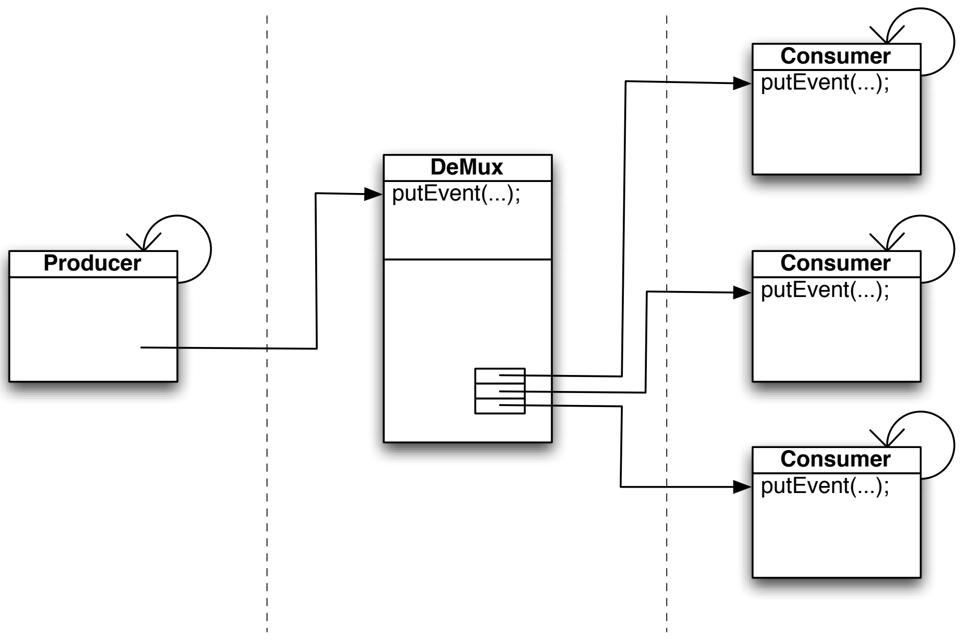
For one-to-many, which we here call event distribution or event demultiplexing, event can simply be put to many threads, one at a time from the providing thread. However, since it is references that are passed through the system, we need to construct a new copy for each consumer (unless no data can be changed and the the owner is set to null). In the following we ignore the issue of how the event copies are made. That is, the clone method is assumed to work, so we can do
`newEvent = existingEvent.clone();`

An existing software system, for which we do not even have the source code, contains a data-producing part that delivers results as (clonable subclasses of) `RTEvent` objects. The producer get (from it's constructor, not shown here) a reference to an `RTEventListener`, and then for each new result there is a call of putEvent (to a single consumer). The `RTEvenListener` interface is simply:

```
public interface RTEventListener {
  RTEvent putEvent(RTEvent e);
}
```

The consumer thread has been an instance of `JThread`, which implements the `RTEventListener` interface by the following method of that base-class:

```
/**
 * Put the time-stamped event in the input buffer of this thread.
 * If the buffer is full, the caller is blocked.
 *
 * @return null, but subclasses are free to return an event if desired
 *         to inform about blocking or results.
 */
public RTEvent putEvent(RTEvent ev) {
  mailbox.doPost(ev);
  return null;
}
```



Problem
The system was first extended with the `RTEventDispatcher` being an intermediate consumer that distributes the events to all registered listeners. That class is useful as a base-class for your solution, but note that an instance is a passive object that distributes all events in the context of the calling thread. That is, the producer thread of execution will run the `putEvent` code of the consumers.

The problem is that those `putEvent` methods can have any (for us unknown) implementation in the respective sub-classes, and the entire system fails if the run method of the producer (i.e. the producing thread) is delayed too much. For instance, if one producer thread performes CPU-time consuming preprocessing (or even hangs or crasches) in the `putEvent` method in the consumer object before posting to it's mailbox, the entire system will fail.

To overcome that problem, **your task is to extend the `RTEventDispatcher` class with a sub-class `DeMux` that separates the producer and thread from the consumer code**.

Specifications
A consumer registers itself for receiving events by calling `addListener` with itself as an argument. This code is available and hence the exact way the registration looks can be modified, and actually we want a simplification of the base class such that the priority does not need to be provided (see requirement 1 below).

The requirements to be fulfilled are:

1. The `addListener` method should be without a priority argument as outlined below. Hint: Class `Thread` has methods `currentThread` and `getPriority`, and you can make use of the available base-class method.
2. The distribution of one event to all listeners may take at the most 8ms. If it takes longer, as a worst case, we assume it is hanging forever and take measures for that.
3. Even if our example only includes one producer, the `DeMux.putEvent` method should work (be thread safe) also in the case of multiple producers.
4. The `DeMux.putEvent` method blocks until the current event is being sent to the last listener, or until the maximum delay of 8ms has passed (hanged call).
5. New threads may only be created (and started) upon first initialization, and after a hang according to the previous item.
6. If a `putEvent` consumer call returns but takes more than 2ms, that delaying listener should be automatically removed from the subscriber list.
7. Any calls to `addListener` and `removeListener` should be blocked during event distribution (of one event).
8. Listeners are to be sorted in priority order, as already accomplished by the base class. A hanging `putEvent` then only delays consumers with lower (listening) priority.
9. Before starting to distribute another event, all pending calls to `addListener` and `removeListener` should be permitted to complete.

As a simplification we only consider the real-time clock and not the consumed CPU time (which would require additional special system calls to obtain), and hence a (low-priority) consumer could be removed due to starvation by other threads.

Solution outline
Since the producer should be immune to consumer `putEvent` delays, and since we cannot add any exception handling to to those methods, we need another thread that calls consumer methods. that thread should get its event within the `DeMux` monitor, but could it simply stay in the monitor while calling the consumer `putEvent` methods?

No! For instance, if the consumer `putEvent` gets blocked on a full mailbox, there is a hold-wait situation that easily could lead to deadlock. The calling thread must therefore return from the monitor before the call, and to enable monitoring of the call progress it is best to return with data for one call at a time; see the `DeMux.next` method and the `DeMux.OnePutEvent` return type below.

Taking the above requirements into account, this leads to the need for a monitor with an additional thread as outlined by the following code:

```
public class DeMux extends RTEventDispatcher {

  // Add your attributes as needed...

  static class OnePutEvent {
    RTEvent message;
    RTEventListener destination;
    long before;
  }

  public DeMux() {
    // ...
  }

  /**
   * Adds the listener, with a registered priority being that of the
   * calling thread.
   * @param dest the thread for which putEvent should be called
   */
  public synchronized void addListener(RTEventListener dest) {
    // ...
    // add when there is no distribution going on
    // ...
  }

  /**
   * Provide event to all listening threads.
   * @param evt the event to be (cloned and) distributed.
   * @return null, or the provided event if not delivered on time.
   */
  public synchronized RTEvent putEvent(RTEvent evt) {
    // ...
  }

  /**
   * Obtain the next event to be distributed to the referenced listener.
   * A listener taking more than 2ms is removed (if not hanged).
   * @param tooSlow listener (with a delaying putEvent as detected
   *        from previous call) to be removed, null otherwise.
   * @return the listener, the event, and the deadline.
   */
  synchronized OnePutEvent next(RTEventListener tooSlow) {
    // ...
  }

  /**
   * Remove the listener.
   */
  public synchronized void removeListener(RTEventListener dest) {
    // ...
    // remove when there is no distribution going on
    // ...
  }

}
```

And the thread calling next can look like:

```
public class Distributer extends JThread {
  DeMux demux;
  DeMux.OnePutEvent todo;
  long t0, t;
  RTEventListener subscriber;

  Distributer(DeMux dispatcher) {demux = dispatcher;}

  // In base: run() {while (!isInterrupted()) {perform();}}

  public void perform() {
    todo = demux.next(subscriber);
    subscriber = todo.destination;
    t0 = System.currentTimeMillis();
    subscriber.putEvent(todo.message);
    t = System.currentTimeMillis();
    if (t<=t0+2 || t<todo.before) subscriber = null;
  }
}
```

Implement the outlined `DeMux` monitor according to the requirements.

(13p)

Problem appendix: source code of base class

```
/**
 * Event dispatching (by calling the listener's putEvent) starts with the
 * high priority listers. The priority is specified when adding a listener.
 */

public class RTEventDispatcher implements RTEventListener
{
  protected static class Subscription
  {
    RTEventListener listener;
    int priority;
    Subscription(RTEventListener listener, int priority)
    {
      this.listener = listener;
      this.priority = priority;
    }
  }

  protected final static Subscription[] NONE = new Subscription[0];

  protected Subscription[] listenerList = NONE;

  /**
   * Return the number of listeners
   */
  public synchronized int getListenerCount()
  {
    return listenerList.length;
  }

  /**
   * Add the listener as a listener of real-time events, with the specified
   * priority. Typically, this method is called directly from a thread
   * object, that is, <code>dest</code> typically is a thread object.
   *
   * @param dest the listener to be added.
   * @param priority the priority of the listener.
   */
  protected synchronized void addListener(RTEventListener dest, int priority)
  {
    if (dest == null) {
      throw new IllegalArgumentException("Listener is null!");
    }

    Subscription record = new Subscription(dest, priority);

    int i;
    for (i = 0; i < listenerList.length; i++) {
      if (priority >= listenerList[i].priority) {
        break;
      }
    }
    Subscription[] tmp = new Subscription[listenerList.length + 1];
    System.arraycopy(listenerList, 0, tmp, 0, i);
    tmp[i] = record;
    System.arraycopy(listenerList, i, tmp, i + 1, listenerList.length - i);
    listenerList = tmp;
  }
  /**
   * Remove the specified listener. If multiple copies
   * of the same listener are present, the one with the
   * highest priority is removed.
   *
   * @param dest the listener to be removed
   * @exception IllegalArgumentException if the arguments are null or
   *            incompatible.
   */
  public synchronized void removeListener(RTEventListener dest)
  {
    if (dest == null) {
      throw new IllegalArgumentException("Listener " + dest + " is null");
    }
    // Search...
    int i = 0;
    for (; i < listenerList.length; i++) {
      if (listenerList[i].listener == dest) {
        break;
      }
    }
    if (i < listenerList.length) { // found.
      Subscription[] tmp;
      if (listenerList.length == 1) {
        tmp = NONE;
      } else {
        tmp = new Subscription[listenerList.length - 1];
        System.arraycopy(listenerList, 0, tmp, 0, i);
        System.arraycopy(listenerList, i + 1, tmp, i, listenerList.length
            - (i + 1));
      }
      listenerList = tmp;
    }
  }

  /**
   * Dispatches the event by distributing it to the listeners.
   */
  public synchronized RTEvent putEvent(RTEvent evt)
  {
    for (int i = 0; i < listenerList.length; i++) {
      listenerList[i].listener.putEvent(evt.clone());
    }
    return null;
  }

}
```

8. **Networked Ad-hoc Event Distribution**

You need to design a networked system similar to the one of the previous problem, but with the following differences:

1. The timing is not specified, so distributing the events to all subscribers of the list according to the `RTEventDispatcher` base class would be fine.
2. The producer and the consumers are all situated at different computers on the network. Therefore, all interconnections including the subscription calls (e.g. to `addListener`) need to be accomplished by threads communication of the network.
3. For so called ad-hoc connection, new consumers should be able to connect at run time, but still it applies (like in previous problem) that an ongoing distribution of events (or dispatching of them; distribution can still be ongoing on the network) should complete before subscriptions are changed.
4. Networks can be unreliable. To prevent filling up of outgoing network connections, and to enable proper error handling to, it should be detected when a consumer looses contact with the distributing computer. Therefore, each consumer should initially tell a time period for checking the connection.
5. To avoid an extra thread (within the distributing computer) for pinging (checking the liveness), the consumer with the fastest liveness check period (known by the distributor) should be requested (by the distributor) to send ping messages to the distributing node (with the period it requested for itself). When receiving those messages in the distributor, it should check what consumers that should be pinged, do that, and evaluate previous ping responses.
6. Consumers that are not responding properly to the ping messages should be removed for the subscription list.

To simplify the networking itself, you can assume there are `send` and `receive` methods for socket objects. Those methods are blocking (e.g. when reading from a socket when there is no data yet) and you can assume there are object streams such that you simply send the event objects as messages. You can also assume that the connection to the fastest consumer is reliable (in the real system those pings are via network broadcasts, which is outside the scope here).

Design (**no code/implementation** required) such a system according to the specifications above. Special emphasis should be put on explaining how lost connections are detected and managed.

(5p)